XRL

Release 3.0.0

October 29, 2016

Contents

XRL (short for XML-RPC Library) is a collection of PHP classes that ease the creation of XML-RPC clients or servers.

CHAPTER 1

Overview

- **Simple setup**—grab the PHAR archive or add `fpoirotte/xrl` as a dependency in your `composer.json` and you're good to go.
- **Very intuitive syntax**—write XML-RPC clients & servers like you would any other piece of code.
- **Automatic type conversions**—use native PHP types without worrying about XML-RPC quirks.
- **Support for many extensions**—want capabilities? introspection? multicalls?... yep, *we support them*!

CHAPTER 2

Setup & Usage

2.1 Setup

Before you install XRL, make sure you have a working PHP installation.

XRL requires PHP 5.3.4 or later and the following PHP extensions:

- XMLReader
- XMLWriter
- libxml
- GMP
- PCRE
- SPL
- Reflection

Note: Use `php -v` and `php -m` to retrieve information about your PHP version and available extensions.

XRL can be installed using a *PHAR archive*, *Composer* or from *Sources*. The PHAR approach is recommended.

2.1.1 PHAR archive

Download the latest PHAR available on https://github.com/fpoirotte/XRL/releases and save it to your computer.

(For Unix/Linux users) Optionally, make the file executable.

2.1.2 Composer

XRL can be installed using the Composer dependency manager. Just add `fpoirotte/xrl` to the dependencies in your `composer.json`:

```
$ php composer.phar require fpoirotte/xrl
```

2.1.3 Sources

To install XRL from sources, use **git** to clone the repository:

```
$ git clone https://github.com/fpoirotte/XRL.git /new/path/for/XRL
```

2.2 Quick start

Assuming XRL is correctly *installed* on your computer, you can now write XML-RPC clients and servers.

2.2.1 Writing an XML-RPC client

1. Load and register the autoloader [1]

```
require_once('/path/to/XRL/src/Autoload.php');
\fpoirotte\XRL\Autoload::register();
```

2. Create a new client configured to query the remote XML-RPC server

```
$client = new \fpoirotte\XRL\Client("http://xmlrpc.example.com/server");
```

3. Call a method provided by that server, like it were just any other code

```
// Call the remote procedure named "hello",
// with "world" as its parameter.
$result = $client->hello('world');

// $result now contains the remote procedure's result,
// as a regular PHP type (integer, string, double, array, etc.)
var_dump($result); // string(12) "hello world!"

// Methods with names that are not valid PHP identifiers
// can still be called!
var_dump($client->{'string.up'}('game over')); // string(9) "GAME OVER"
```

2.2.2 Writing an XML-RPC server

1. Load and register the autoloader [1]

```
require_once('/path/to/XRL/src/Autoload.php');
\fpoirotte\XRL\Autoload::register();
```

2. Create a new server instance

```
$server = new \fpoirotte\XRL\Server();
```

3. Attach some methods to that server

   - You can register anonymous functions, closures, global functions, public methods on objects, etc. using the attribute access operator `->`. You may even use invokable objects!

[1] Users of the Composer dependency manager should load the regular autoloader found in `vendor/autoload.php` instead.

```
class Simpson
{
    private $speech = array(
        'Homer'     => 'Doh!',
        'Marge'     => 'Hmm...',
        'Bart'      => 'Aie, caramba!',
        'Lisa'      => M_PI,
        'Maggie'    => null,
    );
    private $character;

    public function __construct($character)
    {
        if (!array_key_exists($character, $this->speech)) {
            throw new InvalidArgumentException("Who's that?");
        }
        $this->character = $character;
    }

    public function __invoke()
    {
        return $this->speech[$this->character];
    }
}
$server->homer  = new Simpson('Homer');
$server->marge  = new Simpson('Marge');
$server->bart   = new Simpson('Bart');
$server->lisa   = new Simpson('Lisa');
$server->maggie = new Simpson('Maggie');
```

- Alternatively, you can use the array syntax `[]` instead. This is recommended as it avoids potential conflicts with XRL's own attributes and it makes things easier when the method's name is not a valid PHP identifier.

```
$server['hello'] = function ($s) { return "Hello $s!"; };
$server['string.up'] = 'strtoupper';
```

4. Handle incoming XML-RPC requests and publish the results

```
$server->handle()->publish();
```

2.3 HipHop Virtual Machine

> HipHop Virtual Machine (HHVM) is a process virtual machine based on just-in-time (JIT) compilation, serving as an execution engine for PHP and Hack programming languages.
>
> Source: http://en.wikipedia.org/wiki/HipHop_Virtual_Machine

XRL should be compatible with HHVM. To ensure that, we actively test against HHVM in our Continuous Integration process.

CHAPTER 3

Features

3.1 XML-RPC Types

3.1.1 Supported types

XRL supports all the datatypes defined in the official XML-RPC specification, namely:

- `int` and `i4`: 32-bit signed integer value
- `boolean`: your usual boolean type
- `string`: regular string
- `double`: double-precision signed floating point number
- `dateTime.iso8601`: date/time (without milliseconds/timezone information)
- `base64`: base64-encoded binary string
- `struct`: associative array
- `array`: numeric array

It also accepts the following types, which are pretty common, despite them not being part of the official specification:

- `nil`: null value
- `i8`: 64-bit signed integer value

Last but not least, it supports the following namespaced types, defined by the Apache Foundation. Please note that in this particular case, the types must belong to the namespace URI http://ws.apache.org/xmlrpc/namespaces/extensions to be correctly interpreted.

- `nil`: null value (same as the non-namespaced type)
- `i1`: 8-bit signed integer value
- `i2`: 16-bit signed integer value
- `i8`: 64-bits signed integer value (same as the non-namespaced type)
- `biginteger`: arbitrary-length integer
- `dom`: a DOM node, transmitted as an XML fragment
- `dateTime`: date/time with milliseconds and timezone information

When transmitting non-standard types, XRL always uses namespaced types. See the next chapter for more information.

3.1.2 Type conversions

By default, XRL automatically converts values between PHP & XML-RPC types where appropriate.

The following table shows how XRL converts PHP types to XML-RPC types.

Table 3.1: PHP to XML-RPC conversion

PHP type	XML-RPC type
`null`	namespaced `nil`
`boolean`	`boolean`
`integer`	`i4` if it fits into 32 bits, namespaced `i8` [1] otherwise
`double`	`double`
`string`	`string` if it is a valid UTF-8 string, `base64` otherwise
`array`	`array` for numeric arrays, `struct` for associative arrays
`GMP integer` resource (PHP < 5.6.0)	`i4` if it fits into 32 bits, namespaced `i8` [1] if it fits into 64 bits, namespaced `biginteger` [1] otherwise
`\fpoirotte\XRL\Types\AbstractType` object	XML-RPC type it represents
`\GMP` object (PHP >= 5.6.0)	`i4` if it fits into 32 bits, namespaced `i8` [1] if it fits into 64 bits, namespaced `biginteger` [1] otherwise
`\DateTime` object	`dateTime.iso8601` (using local timezone information by default)
`\DOMNode` object	namespaced `dom` [1]
`\XMLWriter` object	namespaced `dom` [1]
`\SimpleXMLElement` object	namespaced `dom` [1]
`\Exception` object	XML-RPC fault (derived from `struct`)

The following table shows how XRL converts XML-RPC types to PHP types.

[1] Using the namespace URI http://ws.apache.org/xmlrpc/namespaces/extensions for compatibility with other implementations.

Table 3.2: XML-RPC to PHP conversion

XML-RPC type	PHP type
`boolean`	`boolean`
`i4`	`integer`
`int`	`integer`
`double`	`double`
`string`	`string`
`base64`	`string`
`array`	numeric array
`struct`	`\Exception` object if the structure represents a fault [2], associative array otherwise
`dateTime.iso8601`	`\DateTime` (using local timezone information by default)
`nil`	`null`
namespaced `nil` [1]	`null`
namespaced `i1` [1]	`integer`
namespaced `i2` [1]	`integer`
`i8`	`GMP integer` resource (PHP < 5.6.0) or `\GMP` object (PHP >= 5.6.0)
namespaced `i8` [1]	`GMP integer` resource (PHP < 5.6.0) or `\GMP` object (PHP >= 5.6.0)
namespaced `biginteger` [1]	`GMP integer` resource (PHP < 5.6.0) or `\GMP` object (PHP >= 5.6.0)
namespaced `dom` [1]	`\SimpleXMLElement` object
namespaced `datetime` [1]	`\DateTime` object

3.1.3 Under the hood

Type conversions are handled by the `\fpoirotte\XRL\NativeEncoder` class (for PHP to XML-RPC conversions) and `\fpoirotte\XRL\NativeDecoder` class (for XML-RPC to PHP conversions), with support from classes in the `\fpoirotte\XRL\Types\` namespace.

You may override or disable the conversion by passing another encoder/decoder to the XML-RPC client or server constructor.

Note: If you change the default encoder/decoder, you will then be responsible for handling conversions to/from the `\fpoirotte\XRL\Types\AbstractType` instances XRL uses internally.

Warning: XML-RPC faults are handled specially and will always turn into `\fpoirotte\XRL\Exception` objects that get raised automatically, no matter what decoder has been passed to the client/server's constructor.

3.2 Extensions

XRL supports several extensions to the original XML-RPC specification. These extensions are known to be widely supported by other implementations and generally do not conflict with the original specification.

[2] An XML-RPC `struct` representing a fault (ie. an error condition) gets converted to an exception that is automatically thrown.

3.2.1 Supported extensions

getCapabilities

The getCapabilities extension has been designed for two reasons:

- To let XML-RPC servers announce (non-standard) features they support.
- To provide an easy way for XML-RPC clients to adapt their behaviour depending on the non-standard features supported by a server.

XRL servers implement the following additional methods when this extension is enabled:

- `system.getCapabilities`

introspection

The introspection extension makes it possible for a client to retrieve information about a remote method by querying the XML-RPC server providing it.

XRL servers implement the following additional methods when this extension is enabled:

- `system.listMethods`
- `system.methodSignature`
- `system.methodHelp`

multicall

The multicall extension has been designed to avoid the latency incurred by HTTP round-trips when making several method calls against the same XML-RPC server.

XRL servers implement the following additional methods when this extension is enabled:

- `system.multicall`

faults_interop

The faults_interop extension contains specifications for a set of standard error conditions (faults), to promote interoperability between XML-RPC implementations.

This extension is always enabled and does not add any additional methods to an XML-RPC server. A developer willing to use the interoperability faults defined in this extension can throw the associated exception from the `\fpoirotte\XRL\Faults` namespace.

```
$server->error = function () {
    throw new \fpoirotte\XRL\Faults\SystemErrorException();
};
```

The following exceptions can be used for interoperability faults:

- `ApplicationErrorException`
- `InternalErrorException`
- `InvalidCharacterException`
- `InvalidParameterException`

- `InvalidXmlRpcException`
- `MethodNotFoundException`
- `NotWellFormedException`
- `SystemErrorException`
- `TransportErrorException`
- `UnsupportedEncodingException`

Also, the `ImplementationDefinedErrorException` exception can be used for implementation-defined errors, but please note than an error code conforming to the specification must be passed explicitly when creating such an error:

```
$server->error = function () {
    throw new \fpoirotte\XRL\Faults\ImplementationDefinedErrorException(
        -32000,                 // Implementation-defined error code
        "You're out of memory"  // Implementation-defined error message
    );
};
```

Apache types

The Apache types extension is kind of special. It does not define any additional methods, but instead focuses on defining additional XML-RPC types.

This extension is always enabled. See also the documentation on *supported XML-RPC types* for more information on these types and how they are used in XRL.

3.2.2 Enabling the extensions

By default, XRL enables only a few extensions (namely, the `faults_interop` and `Apache types` extensions).

To enable the rest of the extensions, you must call `\fpoirotte\XRL\CapableServer::enable()` on the server:

```
// Create a regular XML-RPC server.
$server = new \fpoirotte\XRL\Server();

// Enable additional extensions (capabilities) for that server.
\fpoirotte\XRL\CapableServer::enable($server);
```

Note: It is not currently possible to enable each extension separately when using `\fpoirotte\XRL\CapableServer::enable()`. It's an all-or-nothing kind of situation.

3.3 Command-line XML-RPC client

XRL provides a command-line XML-RPC client that makes it easy to test a remote XML-RPC server.

For PHAR installations, this client is embedded with the PHAR archive itself. To use it, just call the PHP interpreter on the PHAR archive:

```
$ php XRL.phar
Usage: XRL.phar [options] <server URL> <procedure> [args...]

Options:
 -h               Show this program's help.
 [...]
```

For other types of installations, call the PHP interpreter on `bin/xrl`:

```
$ php ./bin/xrl
Usage: ./bin/xrl [options] <server URL> <procedure> [args...]

Options:
 -h               Show this program's help.
 [...]
```

CHAPTER 4

Contributing

4.1 Contributing to XRL

There are several ways by which you may contribute to XRL.

4.1.1 Try it!

The more people use it, the better, because it means bugs and regressions can be detected more quickly.

4.1.2 Report Bugs / Suggest Features

If you use XRL and find issues with it, please let us know on GitHub. Try to provide as much detail as possible on how to reproduce your issue. As a rule of thumb, the easier it is to reproduce an issue, the quicker it gets fixed.

4.1.3 Improve the Documentation

We try to document XRL as much as we can, but acting both as developers and documentation writers, we tend to be biased as to what needs documentation.

So if you feel like some parts could be clearer, send us a pull request with your modifications and we'll try to review them as soon as possible. Any help to improve the documentation will be greatly appreciated!

4.1.4 Fork It! And Improve the Code

If you find a bug and have some PHP knowledge and time to spare, grab a copy of the code and have a go at fixing it.

If you already have a GitHub account, it's quite simple:

1. Fork the code
2. Hack away
3. Send a pull request for review
4. Repeat again starting at step 2

4.2 Coding style

The XRL team closely follows the PSR-2 coding style.

Also, when developing, the following command can be used to check various aspects of the code quality:

```
$ vendor/bin/phing qa
```

It runs the following tools on XRL's code to detect possible issues:

- PHP lint (`php -l`): checks PHP syntax
- PHP_CodeSniffer: checks code compliance with coding standard
- PDepend: identifies tight coupling between two pieces of code
- PHPMD (PHP Mess Detector): detects high-risk code structures
- PHPCPD (PHP Copy-Paste Detector): detects copy/paste abuse
- PHPUnit: checks unit tests

4.3 Credits

Specials thanks to:

- Thibaud Rohmer for being the first user of XRL and the very reason this whole project exists

Thanks to:

- David Goodger for Docutils and reStructuredText
- Georg Brandl for Sphinx
- Whoever made the `haiku` theme for Sphinx

whose work made it possible to write and publish this documentation.

CHAPTER 5

Licence

XRL is released under the 3-clause BSD licence.

CHAPTER 6

Other resources

- XRL on GitHub (source code and issue tracker)
- XRL on Packagist (Composer repository)
- XRL on Travis-CI (continuous integration)
- XRL on Read The Docs (online documentation)
- Full API documentation (hosted on Read The Docs)