Jaxl Documentation

Release 3.1.0

Abhinav Singh

Sep 27, 2017

Contents

Jaxl v3.x is a successor of v2.x (and is NOT backward compatible), carrying a lot of code from v2.x while throwing away the ugly parts. A lot of components have been re-written keeping in mind the feedback from the developer community over the last 4 years. Also Jaxl now shares a few philosophies from my experience with erlang and python languages.

Jaxl is an asynchronous, non-blocking I/O, event based PHP library for writing custom TCP/IP client and server implementations. From it's previous versions, library inherits a full blown stable support for XMPP protocol stack. In v3.0, support for HTTP protocol stack was also added.

At the heart of every protocol stack sits a Core stack. It contains all the building blocks for everything that we aim to do with Jaxl library. Both XMPP and HTTP protocol stacks are written on top of the Core stack. Infact the source code of protocol implementations knows nothing about the standard (inbuilt) PHP socket and stream methods.

CHAPTER 1

Users Guide

Getting Started

Requirements

No external component or library is required. You simply need a standard PHP installation to work with Jaxl.

Library has been developed and tested extensively on linux operating systems. But there is no reason why it should not work on other OS. File an issue if you face any glitches.

Install

Use Composer to install:

php composer.phar require "jaxl/jaxl=^3.1.0"

Then use autoloader in your application:

require dirname(__FILE__) . '/vendor/autoload.php';

Library Structure

Jaxl library comprises of following packages:

- `jaxl-core`

  contains generic networking and eventing components

- `jaxl-xmpp`

  contains xmpp rfc implementation

- `jaxl-xmpp-xep`

  contains various xmpp xep implementation

- `jaxl-http`

  contains http rfc implementation

- `jaxl-docs`

  this documentation comes from this package

- `jaxl-tests`

  test suites for all the above packages

Inside Jaxl everything that you will interact with will be an object which will emit events and callbacks which we will be able to catch in our applications for custom processing and routing. Listed below are a few main objects:

1. Core Stack

- `JAXLLoop`

  main select loop

- `JAXLClock`

  timed job/callback dispatcher

- `JAXLEvent`

  event registry and emitter

- `JAXLFsm`

  generic finite state machine

- `JAXLSocketClient`

  generic tcp/udp client

- `JAXLSocketServer`

  generic tcp/udp server

- `JAXLXmlStream`

  streaming XML parser

- `JAXLXml`

  custom XML object implementation

- `JAXLLogger`

  logging facility

1. XMPP Stack

- `XMPPStream`

  base xmpp rfc implementation

- `XMPPStanza`

  provides easy access patterns over xmpp stanza (wraps `JAXLXml`)

- `XMPPIq`

  xmpp iq stanza object (extends `XMPPStanza`)

- `XMPPMsg`

  xmpp msg stanza object (extends `XMPPStanza`)

- `XMPPPres`

  xmpp pres stanza object (extends `XMPPStanza`)

- `XMPPXep`

  abstract xmpp extension (extended by XEP implementations)

- `XMPPJid`

  xmpp jid object

1. HTTP Stack

- `HTTPServer`

  http server implementation

- `HTTPClient`

  http client implementation

- `HTTPRequest`

  http request object

- `HTTPResponse`

  http response object

Questions, Bugs and Issues

If you have any questions kindly post them on google groups. Groups are the quickest way to get an answer to your questions which is actively monitored by core developers.

If you are facing a bug or issue, please report that it on github issue tracker. You can even contribute to the library if you already have fixed the bug.

JAXL Instance

`JAXL` instance configure/manage other *sub-packages*. It provides an event based callback methodology on various underlying object. Whenever required `JAXL` instance will itself perform the configured defaults.

Constructor options

1. `jid`
2. `pass`
3. `resource`

   If not passed Jaxl will use a random resource value

4. `auth_type`

   DIGEST-MD5, PLAIN (default), CRAM-MD5, ANONYMOUS

5. `host`
6. `port`
7. `bosh_url`

8. `log_path`
9. `log_level`

   `ERROR`, `WARNING`, `NOTICE`, `INFO` (default), `DEBUG`
10. `force_tls`
11. `stream_context`
12. `priv_dir`

    Jaxl creates 4 directories names `log`, `tmp`, `run` and `sock` inside a private directory which defaults to `getcwd().'/.jaxl'`. If this option is passed, it will overwrite default private directory.

    Note: Jaxl currently doesn't check upon the permissions of passed `priv_dir`. Make sure Jaxl library have sufficient permissions to create above mentioned directories.

Available Event Callbacks

Following `$ev` are available on `JAXL` lifecycle for registering callbacks:

1. `on_connect`

   `JAXL` instance has connected successfully
2. `on_connect_error`

   `JAXL` instance failed to connect
3. `on_stream_start`

   `JAXL` instance has successfully initiated XMPP stream with the jabber server
4. `on_stream_features`

   `JAXL` instance has received supported stream features
5. `on_auth_success`

   authentication successful
6. `on_auth_failure`

   authentication failed
7. `on_presence_stanza`

   `JAXL` instance has received a presence stanza
8. `on_{$type}_message`

   `JAXL` instance has received a message stanza. `$type` can be `chat`, `groupchat`, `headline`, `normal`, `error`
9. `on_stanza_id_{$id}`

   Useful when dealing with iq stanza. This event is fired when `JAXL` instance has received response to a particular xmpp stanza id
10. `on_{$name}_stanza`

    Useful when dealing with custom xmpp stanza
11. `on_disconnect`

JAXL instance has disconnected from the jabber server

Available Methods

Following methods are available on initialized JAXL instance object:

1. `get_pid_file_path()`

   returns path of JAXL instance pid file

2. `get_sock_file_path()`

   returns path to JAXL ipc unix socket domain

3. `require_xep($xeps = array())`

   autoload and initialize passed XEP's

4. `add_cb($ev, $cb, $priority = 1)`

   add a callback to function `$cb` on event `$ev`, returns a reference of added callback

5. `del_cb($ref)`

   delete previously registered event callback

6. `set_status($status, $show, $priority)`

   send a presence status stanza

7. `send_chat_msg($to, $body, $thread = null, $subject = null)`

   send a message stanza of type chat

8. `get_vcard($jid = null, $cb = null)`

   fetch vcard for bare `$jid`, passed `$cb` will be called with received vcard stanza

9. `get_roster($cb = null)`

   fetch roster list of connected jabber client, passed `$cb` will be called with received roster stanza

10. `start($opts = array())`

    start configured JAXL instance, optionally accepts two options specified below:

    (a) `--with-debug-shell`

        start JAXL instance and enter an interactive console

    (b) `--with-unix-sock`

        start JAXL instance with support for IPC and remote debugging

11. `send($stanza)`

    send an instance of JAXLXml packet over connected socket

12. `send_raw($data)`

    send raw payload over connected socket

13. `get_msg_pkt($attrs, $body = null, $thread = null, $subject = null, $payload = null)`

14. `get_pres_pkt($attrs, $status = null, $show = null, $priority = null, $payload = null)`

15. `get_iq_pkt($attrs, $payload)`

./jaxlctl

Usage: `./jaxlctl command [options...]`

`jaxlctl` is a control script that can be seen as a useful tool for:

- debugging daemons running in the background
- customize daemons on the fly
- monitoring daemons
- as a playground for learning XMPP/HTTP and Jaxl library itself

Type `./jaxlctl help` to see list of available commands.

Note: Various commands are still experimental. Know what you are doing before using them in production. You have been warned !!!

Interactive Shell

```
>>> ./jaxlctl shell
jaxl 1>
jaxl 1> // create a test message object
jaxl 1> $msg = new XMPPMsg(array('to' => 'friend@gmail.com'), 'Hello World!');
jaxl 2>
jaxl 2> // object to string conversion
jaxl 2> print_r($msg->to_string());
<message to="friend@gmail.com"><body>Hello World!</body></message>
jaxl 3>
```

Debug Running Instances

```
>>> ./jaxlctl attach XXXXX
jaxl 1>
jaxl 1> // create a message to be sent
jaxl 1> $msg = new XMPPMsg(array('to' => 'friend@gmail.com'), 'Hello World!');
jaxl 2>
jaxl 2> // this client is from the echo bot example
jaxl 2> global $client;
jaxl 3>
jaxl 3> // send the message packet
jaxl 3> $client->send($msg);
jaxl 4>
jaxl 4> // or we can directly do
jaxl 4> $client->send_chat_msg('friend@gmail.com', 'Hello World!');
jaxl 5>
```

Where `XXXXX` is the pid of running `JAXL` instance.

Logging Interface

`JAXLLogger` provides all the logging facilities that we will ever require. When logging to `STDOUT` it also colorizes the log message depending upon its severity level. When logging to a file it can also do periodic log rotation.

log levels

- ERROR (red)
- WARNING (blue)
- NOTICE (yellow)
- INFO (green)
- DEBUG (white)

global logging methods

Following global methods for logging are available:

- `error($msg)`
- `warning($msg)`
- `notice($msg)`
- `info($msg)`
- `debug($msg)`

log/2

All the above global logging methods internally use `log($msg, $verbosity)` to output colored log message on the terminal.

Cron Jobs

`JAXLClock` maintains a global clock which is updated after every iteration of the *main select loop*. During the clock tick phase, `JAXLClock` also dispatches any scheduled cron jobs.

Lets try some cron job scheduling using Jaxl interactive shell:

```
>>> ./jaxlctl shell
jaxl 1>
jaxl 1> function do_job($params)
....... {
.......     echo "cron job called";
....... }
jaxl 2>
jaxl 2> $ref = JAXLLoop::$clock->call_fun_after(
.......     4000,
.......     'do_job',
.......     'some_parameters'
....... );
```

```
jaxl 3> echo $ref;
1
jaxl 4>
cron job called
jaxl 5> quit
>>>
```

We just saw a live example of a cron job. Using `JAXLClock::call_fun_after/3` we were able to call our `do_job` function after 4000 microseconds.

Note: Since cron jobs are called inside main select loop, do not execute long running cron jobs using `JAXLClock` else the main select loop will not be able to detect any new activity on watched file descriptors. In short, these cron job callbacks are blocking.

In future, cron jobs might get executed in a seperate process space, overcoming the above limitation. Until then know what your jobs are doing and for how long or execute them in a seperate process space yourself. You have been warned !!!

---

one time jobs

`call_fun_after($time, $callback, $args)`

schedules $callback with $args after $time microseconds

periodic jobs

`call_fun_periodic($time, $callback, $args)`

schedules periodic $callback with $args after $time microseconds

cancel a job

`cancel_fun_call($ref)`

cancels a previously scheduled $callback

detecting bottlenecks

`tc($callback, $args)`

calculate execution time of a $callback with $args

CHAPTER 2

XMPP Users Guide:

XMPP Examples

Echo Bot Client

include `jaxl.php` and initialize a new `JAXL` instance:

```
$client = new JAXL(array(
    'jid' => 'user@domain.tld',
    'pass' => 'password'
));
```

We just initialized a new `JAXL` instance by passing our jabber client `jid` and `pass`.

View list of *available options* that can be passed to `JAXL` constructor.

Next we need to register callbacks on events of interest using `JAXL::add_cb/2` method as shown below:

```
function on_auth_success_callback()
{
    global $client;
    $client->set_status("available!");  // set your status
    $client->get_vcard();               // fetch your vcard
    $client->get_roster();              // fetch your roster list
}
$client->add_cb('on_auth_success', 'on_auth_success_callback');

function on_chat_message_callback($stanza)
{
    global $client;

    // echo back
    $stanza->to = $stanza->from;
    $stanza->from = $client->full_jid->to_string();
    $client->send($stanza);
```

```
}
$client->add_cb('on_chat_message', 'on_chat_message_callback');

function on_disconnect_callback()
{
    JAXLLogger::debug("got on_disconnect cb");
}
$client->add_cb('on_disconnect', 'on_disconnect_callback');
```

We just registered callbacks on `on_auth_success`, `on_chat_message` and `on_disconnect` events that will occur inside our configured `JAXL` instance lifecycle. We also passed a method that will be called (with parameters if any) when the event has been detected.

See list of *available event callbacks* that we can hook to inside `JAXL` instance lifecycle.

Received `$msg` parameter with `on_chat_message` event callback above, will be an instance of `XMPPMsg` which extends `XMPPStanza` class, that allows us easy to use access patterns to common XMPP stanza attributes like `to`, `from`, `type`, `id` to name a few.

We were also able to access our xmpp client full jabber id by calling `$client->full_jid`. This attribute of `JAXL` instance is available from `on_auth_success` event. `full_jid` attribute is an instance of `XMPPJid`.

To send our echo back `$msg` packet we called `JAXL::send/1` which accepts a single parameter which MUST be an instance of `JAXLXml`. Since `XMPPStanza` is a wrapper upon `JAXLXml` we can very well pass our modified `$msg` object to the send method.

Read more about various *XML Objects* and how they make writing XMPP applications fun and easy. You can also *add custom access patterns* upon received `XMPPStanza` objects. Since all access patterns are evaluated upon first access and cached for later usage, adding hundreds of custom access patterns that retrieves information from 100th child of received XML packet will not be an issue.

We will finally start our xmpp client by calling:

```
$client->start();
```

See list of *available options* that can be passed to the `JAXL::start/2` method. These options are particularly useful for debugging and monitoring.

Echo Bot BOSH Client

Everything goes same for a cli BOSH client. To run above echo bot client example as a bosh client simply pass additional parameters to `JAXL` constructor:

```
$client = new JAXL(array(
    'jid' => 'user@domain.tld',
    'pass' => 'password',
    'bosh_url' => 'http://localhost:5280/http-bind'
));
```

You can even pass custom values for `hold`, `wait` and other attributes.

View list of *available options* that can be passed to `JAXL` constructor.

Echo Bot External Component

Again almost everything goes same for an external component except a few custom `JAXL` constructor parameter as shown below:

```
$comp = new JAXL(array(
    // (required) component host and secret
    'jid' => $argv[1],
    'pass' => $argv[2],

    // (required) destination socket
    'host' => $argv[3],
    'port' => $argv[4]
));
```

We will also need to include `XEP0114` which implements Jabber Component XMPP Extension.

```
// (required)
$comp->require_xep(array(
    '0114' // jabber component protocol
));
```

`JAXL::require_xep/1` accepts an array of XEP numbers passed as strings.

Xml Objects

Jaxl library works with custom XML implementation which is similar to inbuild PHP XML functions but is lightweight and easy to work with.

JAXLXml

`JAXLXml` is the base XML object. Open up *Jaxl interactive shell* and try some xml object creation/manipulation:

```
>>> ./jaxlctl shell
jaxl 1>
jaxl 1> $xml = new JAXLXml(
.......     'dummy',
.......     'dummy:packet',
.......     array('attr1' => 'friend@gmail.com', 'attr2' => ''),
.......     'Hello World!'
....... );
jaxl 2> echo $xml->to_string();
<dummy xmlns="dummy:packet" attr1="friend@gmail.com" attr2="">Hello World!</dummy>
jaxl 3>
```

`JAXLXml` constructor instance accepts following parameters:

- `JAXLXml($name, $ns, $attrs, $text)`
- `JAXLXml($name, $ns, $attrs)`
- `JAXLXml($name, $ns, $text)`
- `JAXLXml($name, $attrs, $text)`
- `JAXLXml($name, $attrs)`
- `JAXLXml($name, $ns)`
- `JAXLXml($name)`

JAXLXml draws inspiration from StropheJS XML Builder class. Below are available methods for modifying and manipulating an JAXLXml object:

- t($text, $append = false)

  update text of current rover

- c($name, $ns = null, $attrs = array(), $text = null)

  append a child node at current rover

- cnode($node)

  append a JAXLXml child node at current rover

- up()

  move rover to one step up the xml tree

- top()

  move rover back to top element in the xml tree

- exists($name, $ns = null, $attrs = array())

  checks if a child with $name exists, return child JAXLXml if found otherwise false. This function returns at first matching child.

- update($name, $ns = null, $attrs = array(), $text = null)

  update specified child element

- attrs($attrs)

  merge new attrs with attributes of current rover

- match_attrs($attrs)

  pass a kv pair of $attrs, return bool if all passed keys matches their respective values in the xml packet

- to_string()

  get string representation of the object

JAXLXml maintains a rover which points to the current level down the XML tree where manipulation is performed.

XMPPStanza

In the world of XMPP where everything incoming and outgoing payload is an JAXLXml instance code can become nasty, developers can get lost in dirty XML manipulations spreaded all over the application code base and what not. XML structures are not only unreadable for humans but even for machine.

While an instance of JAXLXml provide direct access to XML name, ns and text, it can become painful and time consuming when trying to retrieve or modify a particular attrs or children. I was fed up of doing getAttributeByName, setAttributeByName, getChild etc everytime i had to access common XMPP Stanza attributes.

XMPPStanza is a wrapper on top of JAXLXml objects. Preserving all the functionalities of base JAXLXml instance it also provide direct access to most common XMPP Stanza attributes like to, from, id, type etc. It also provides a framework for adding custom access patterns.

XMPPMsg, XMPPPres and XMPPIq extends XMPPStanza and also add a few custom access patterns like body, thread, subject, status, show etc.

Here is a list of default access patterns:

1. name
2. ns
3. text
4. attrs
5. children
6. to
7. from
8. id
9. type
10. to_node
11. to_domain
12. to_resource
13. from_node
14. from_domain
15. from_resource
16. status
17. show
18. priority
19. body
20. thread
21. subject

XMPP Extensions (XEP)

Writing Custom XMPP Extension

Entity Discovery

Multi-User Chat

Direct MUC Invitation

Publish-Subscribe

In-Band User Registration

External Jabber Component

Entity Capabilities

Delayed Delivery

XMPP Over HTTP (Bosh)

CHAPTER 3

HTTP Users Guide:

HTTP Examples

Writing HTTP Server

Intialize an `HTTPServer` instance

```
$http = new HTTPServer();
```

By default `HTTPServer` will listen on port 9699. You can pass a port number as first parameter to change this.

Define a callback method that will accept all incoming `HTTPRequest` objects

```
function on_request($request)
{
    if ($request->method == 'GET') {
        $body = json_encode($request);
        $request->ok($body, array('Content-Type' => 'application:json'));
    } else {
        $request->not_found();
    }
}
```

`on_request` callback method will receive a `HTTPRequest` object instance. For this example, we will simply echo back json encoded `$request` object for every http GET request.

Start http server:

```
$http->start('on_request');
```

We pass `on_request` method as first parameter to `HTTPServer::start/1`. If nothing is passed, requests will fail with a default 404 not found error message

Writing REST API Server

Intialize an `HTTPServer` instance

```
$http = new HTTPServer();
```

By default `HTTPServer` will listen on port 9699. You can pass a port number as first parameter to change this.

Define our REST resources callback methods:

```
function index($request)
{
    $request->send_response(
        200, array('Content-Type' => 'text/html'),
        '<html><head/><body><h1>Jaxl Http Server</h1><a href="/upload">upload a file</a></body></html>'
    );
    $request->close();
}

function upload($request)
{
    if ($request->method == 'GET') {
        $request->send_response(
            200, array('Content-Type' => 'text/html'),
            '<html><head/><body><h1>Jaxl Http Server</h1><form enctype="multipart/form-data" method="POST" acti
        );
    } elseif ($request->method == 'POST') {
        if ($request->body === null && $request->expect) {
            $request->recv_body();
        } else {
            // got upload body, save it
            JAXLLogger::debug("file upload complete, got ".strlen($request->body).
" bytes of data");
            $request->close();
        }
    }
}
```

Next we need to register dispatch rules for our callbacks above:

```
$index = array('index', '^/$');
$upload = array('upload', '^/upload', array('GET', 'POST'));
$rules = array($index, $upload);
$http->dispatch($rules);
```

Start REST api server:

```
$http->start();
```

Make an HTTP request

HTTP Extensions

Dispatch Rules

Dispatch rules are convinient way of redirecting callback for a specific request pattern to a custom methods inside your application. A dispatch rule consists of following 4 match information:

- `$callback`

  reference to a method that will be callback'd when a matching request is received

- `$pattern`

  a regular expression for matching on url path

- `$methods`

  (optional) if not specified rule will match for all HTTP Methods. if specified, must be an array of HTTP Method in uppercase.

- `$extra`

  (reserved) this is for future where we will allow matching on headers, session, cookies etc.

Below are a few examples of dispatch rules:

```
$index = array('serve_index_page', '^/');
$upload_form = array('serve_upload_form', '^/upload', array('GET'));
$upload_handler = array('handle_upload_form', '^/upload', array('POST'));
```

Some REST CRUD dispatch rules:

```
$event_create = array('create_event', '^/event/create/$', array('PUT'));
$event_read = array('read_event', '^/event/(?P<pk>\d+)/$', array('GET', 'HEAD'));
$event_update = array('update_event', '^/event/(?P<pk>\d+)/$', array('POST'));
$event_delete = array('delete_event', '^/event/(?P<pk>\d+)/$', array('DELETE'));
```

Finally don't forget to active these dispatch rules by doing:

```
$rules = array($index, $upload_form, $upload_handler, $event_create, $event_read,
↪$event_update, $event_delete);
$http->dispatch($rules);
```