RingPHP

Release

January 13, 2017

Contents

Provides a simple API and specification that abstracts away the details of HTTP into a single PHP function. RingPHP be used to power HTTP clients and servers through a PHP function that accepts a request hash and returns a response hash that is fulfilled using a promise, allowing RingPHP to support both synchronous and asynchronous workflows.

By abstracting the implementation details of different HTTP clients and servers, RingPHP allows you to utilize pluggable HTTP clients and servers without tying your application to a specific implementation.

CHAPTER 1

Specification

RingPHP applications consist of handlers, requests, responses, and middleware.

1.1 Handlers

Handlers are implemented as a PHP `callable` that accept a request array and return a response array (`GuzzleHttp\Ring\Future\FutureArrayInterface`).

For example:

```
use GuzzleHttp\Ring\Future\CompletedFutureArray;

$mockHandler = function (array $request) {
    return new CompletedFutureArray([
        'status'  => 200,
        'headers' => ['X-Foo' => ['Bar']],
        'body'    => 'Hello!'
     ]);
};
```

This handler returns the same response each time it is invoked. All RingPHP handlers must return a `GuzzleHttp\Ring\Future\FutureArrayInterface`. Use `GuzzleHttp\Ring\Future\CompletedFutureArray` when returning a response that has already completed.

1.2 Requests

A request array is a PHP associative array that contains the configuration settings need to send a request.

```
$request = [
    'http_method' => 'GET',
    'scheme'      => 'http',
    'uri'         => '/',
    'body'        => 'hello!',
    'client'      => ['timeout' => 1.0],
    'headers'     => [
        'host'  => ['httpbin.org'],
        'X-Foo' => ['baz', 'bar']
    ]
];
```

The request array contains the following key value pairs:

request_method (string, required) The HTTP request method, must be all caps corresponding to a HTTP request method, such as `GET` or `POST`.

scheme (string) The transport protocol, must be one of `http` or `https`. Defaults to `http`.

uri (string, required) The request URI excluding the query string. Must start with "/".

query_string (string) The query string, if present (e.g., `foo=bar`).

version (string) HTTP protocol version. Defaults to `1.1`.

headers (required, array) Associative array of headers. Each key represents the header name. Each value contains an array of strings where each entry of the array SHOULD be sent over the wire on a separate header line.

body (string, fopen resource, `Iterator`, `GuzzleHttp\Stream\StreamInterface`) The body of the request, if present. Can be a string, resource returned from fopen, an `Iterator` that yields chunks of data, an object that implemented `__toString`, or a `GuzzleHttp\Stream\StreamInterface`.

future (bool, string) Controls the asynchronous behavior of a response.

Set to `true` or omit the `future` option to *request* that a request will be completed asynchronously. Keep in mind that your request might not necessarily be completed asynchronously based on the handler you are using. Set the `future` option to `false` to request that a synchronous response be provided.

You can provide a string value to specify fine-tuned future behaviors that may be specific to the underlying handlers you are using. There are, however, some common future options that handlers should implement if possible.

lazy Requests that the handler does not open and send the request immediately, but rather only opens and sends the request once the future is dereferenced. This option is often useful for sending a large number of requests concurrently to allow handlers to take better advantage of non-blocking transfers by first building up a pool of requests.

If an handler does not implement or understand a provided string value, then the request MUST be treated as if the user provided `true` rather than the string value.

Future responses created by asynchronous handlers MUST attempt to complete any outstanding future responses when they are destructed. Asynchronous handlers MAY choose to automatically complete responses when the number of outstanding requests reaches an handler-specific threshold.

1.2.1 Client Specific Options

The following options are only used in ring client handlers.

client (array) Associative array of client specific transfer options. The `client` request key value pair can contain the following keys:

cert (string, array) Set to a string to specify the path to a file containing a PEM formatted SSL client side certificate. If a password is required, then set `cert` to an array containing the path to the PEM file in the first array element followed by the certificate password in the second array element.

connect_timeout (float) Float describing the number of seconds to wait while trying to connect to a server. Use `0` to wait indefinitely (the default behavior).

debug (bool, fopen() resource) Set to true or set to a PHP stream returned by fopen() to enable debug output with the handler used to send a request. If set to `true`, the output is written to PHP's STDOUT. If a PHP `fopen` resource handle is provided, the output is written to the stream.

"Debug output" is handler specific: different handlers will yield different output and various various level of detail. For example, when using cURL to transfer requests, cURL's CURLOPT_VERBOSE will be used. When using the PHP stream wrapper, stream notifications will be emitted.

decode_content (bool) Specify whether or not `Content-Encoding` responses (gzip, deflate, etc.) are automatically decoded. Set to `true` to automatically decode encoded responses. Set to `false` to not decode responses. By default, content is *not* decoded automatically.

delay (int) The number of milliseconds to delay before sending the request. This is often used for delaying before retrying a request. Handlers SHOULD implement this if possible, but it is not a strict requirement.

progress (function) Defines a function to invoke when transfer progress is made. The function accepts the following arguments:

1. The total number of bytes expected to be downloaded
2. The number of bytes downloaded so far
3. The number of bytes expected to be uploaded
4. The number of bytes uploaded so far

proxy (string, array) Pass a string to specify an HTTP proxy, or an associative array to specify different proxies for different protocols where the scheme is the key and the value is the proxy address.

```
$request = [
    'http_method' => 'GET',
    'headers'     => ['host' => ['httpbin.org']],
    'client'      => [
        // Use different proxies for different URI schemes.
        'proxy' => [
            'http'  => 'http://proxy.example.com:5100',
            'https' => 'https://proxy.example.com:6100'
        ]
    ]
];
```

ssl_key (string, array) Specify the path to a file containing a private SSL key in PEM format. If a password is required, then set to an array containing the path to the SSL key in the first array element followed by the password required for the certificate in the second element.

save_to (string, fopen resource, `GuzzleHttp\Stream\StreamInterface`) Specifies where the body of the response is downloaded. Pass a string to open a local file on disk and save the output to the file. Pass an fopen resource to save the output to a PHP stream resource. Pass a `GuzzleHttp\Stream\StreamInterface` to save the output to a Guzzle StreamInterface. Omitting this option will typically save the body of a response to a PHP temp stream.

stream (bool) Set to true to stream a response rather than download it all up-front. This option will only be utilized when the corresponding handler supports it.

timeout (float) Float describing the timeout of the request in seconds. Use 0 to wait indefinitely (the default behavior).

verify (bool, string) Describes the SSL certificate verification behavior of a request. Set to true to enable SSL certificate verification using the system CA bundle when available (the default). Set to false to disable certificate verification (this is insecure!). Set to a string to provide the path to a CA bundle on disk to enable verification using a custom certificate.

version (string) HTTP protocol version to use with the request.

1.2.2 Server Specific Options

The following options are only used in ring server handlers.

server_port (integer) The port on which the request is being handled. This is only used with ring servers, and is required.

server_name (string) The resolved server name, or the server IP address. Required when using a Ring server.

remote_addr (string) The IP address of the client or the last proxy that sent the request. Required when using a Ring server.

1.3 Responses

A response is an array-like object that implements `GuzzleHttp\Ring\Future\FutureArrayInterface`. Responses contain the following key value pairs:

body (string, fopen resource, `Iterator`, `GuzzleHttp\Stream\StreamInterface`) The body of the response, if present. Can be a string, resource returned from fopen, an `Iterator` that yields chunks of data, an object that implemented `__toString`, or a `GuzzleHttp\Stream\StreamInterface`.

effective_url (string) The URL that returned the resulting response.

error (`\Exception`) Contains an exception describing any errors that were encountered during the transfer.

headers (Required, array) Associative array of headers. Each key represents the header name. Each value contains an array of strings where each entry of the array is a header line. The headers array MAY be an empty array in the event an error occurred before a response was received.

reason (string) Optional reason phrase. This option should be provided when the reason phrase does not match the typical reason phrase associated with the `status` code. See RFC 7231 for a list of HTTP reason phrases mapped to status codes.

status (Required, integer) The HTTP status code. The status code MAY be set to `null` in the event an error occurred before a response was received (e.g., a networking error).

transfer_stats (array) Provides an associative array of arbitrary transfer statistics if provided by the underlying handler.

version (string) HTTP protocol version. Defaults to `1.1`.

1.4 Middleware

Ring middleware augments the functionality of handlers by invoking them in the process of generating responses. Middleware is typically implemented as a higher-order function that takes one or more handlers as arguments followed by an optional associative array of options as the last argument, returning a new handler with the desired compound behavior.

Here's an example of a middleware that adds a Content-Type header to each request.

```
use GuzzleHttp\Ring\Client\CurlHandler;
use GuzzleHttp\Ring\Core;

$contentTypeHandler = function(callable $handler, $contentType) {
    return function (array $request) use ($handler, $contentType) {
        return $handler(Core::setHeader('Content-Type', $contentType));
    };
```

```
};

$baseHandler = new CurlHandler();
$wrappedHandler = $contentTypeHandler($baseHandler, 'text/html');
$response = $wrappedHandler([/** request hash **/]);
```

CHAPTER 2

Futures

Futures represent a computation that may have not yet completed. RingPHP uses hybrid of futures and promises to provide a consistent API that can be used for both blocking and non-blocking consumers.

2.1 Promises

You can get the result of a future when it is ready using the promise interface of a future. Futures expose a promise API via a `then()` method that utilizes React's promise library. You should use this API when you do not wish to block.

```
use GuzzleHttp\Ring\Client\CurlMultiHandler;

$request = [
    'http_method' => 'GET',
    'uri'         => '/',
    'headers'     => ['host' => ['httpbin.org']]
];

$response = $handler($request);

// Use the then() method to use the promise API of the future.
$response->then(function ($response) {
    echo $response['status'];
});
```

You can get the promise used by a future, an instance of `React\Promise\PromiseInterface`, by calling the `promise()` method.

```
$response = $handler($request);
$promise = $response->promise();
$promise->then(function ($response) {
    echo $response['status'];
});
```

This promise value can be used with React's aggregate promise functions.

2.2 Waiting

You can wait on a future to complete and retrieve the value, or *dereference* the future, using the `wait()` method. Calling the `wait()` method of a future will block until the result is available. The result is then returned or an

exception is thrown if and exception was encountered while waiting on the the result. Subsequent calls to dereference a future will return the previously completed result or throw the previously encountered exception. Futures can be cancelled, which stops the computation if possible.

```
use GuzzleHttp\Ring\Client\CurlMultiHandler;

$response = $handler([
    'http_method' => 'GET',
    'uri'         => '/',
    'headers'     => ['host' => ['httpbin.org']]
]);

// You can explicitly call block to wait on a result.
$realizedResponse = $response->wait();

// Future responses can be used like a regular PHP array.
echo $response['status'];
```

In addition to explicitly calling the `wait()` function, using a future like a normal value will implicitly trigger the `wait()` function.

2.3 Future Responses

RingPHP uses futures to return asynchronous responses immediately. Client handlers always return future responses that implement `GuzzleHttp\Ring\Future\ArrayFutureInterface`. These future responses act just like normal PHP associative arrays for blocking access and provide a promise interface for non-blocking access.

```
use GuzzleHttp\Ring\Client\CurlMultiHandler;

$handler = new CurlMultiHandler();

$request = [
    'http_method'  => 'GET',
    'uri'          => '/',
    'headers'      => ['Host' => ['www.google.com']]
];

$response = $handler($request);

// Use the promise API for non-blocking access to the response. The actual
// response value will be delivered to the promise.
$response->then(function ($response) {
    echo $response['status'];
});

// You can wait (block) until the future is completed.
$response->wait();

// This will implicitly call wait(), and will block too!
$response['status'];
```

Important: Futures that are not completed by the time the underlying handler is destructed will be completed when the handler is shutting down.

2.4 Cancelling

Futures can be cancelled if they have not already been dereferenced.

RingPHP futures are typically implemented with the `GuzzleHttp\Ring\Future\BaseFutureTrait`. This trait provides the cancellation functionality that should be common to most implementations. Cancelling a future response will try to prevent the request from sending over the wire.

When a future is cancelled, the cancellation function is invoked and performs the actual work needed to cancel the request from sending if possible (e.g., telling an event loop to stop sending a request or to close a socket). If no cancellation function is provided, then a request cannot be cancelled. If a cancel function is provided, then it should accept the future as an argument and return true if the future was successfully cancelled or false if it could not be cancelled.

2.5 Wrapping an existing Promise

You can easily create a future from any existing promise using the `GuzzleHttp\Ring\Future\FutureValue` class. This class's constructor accepts a promise as the first argument, a wait function as the second argument, and a cancellation function as the third argument. The dereference function is used to force the promise to resolve (for example, manually ticking an event loop). The cancel function is optional and is used to tell the thing that created the promise that it can stop computing the result (for example, telling an event loop to stop transferring a request).

```
use GuzzleHttp\Ring\Future\FutureValue;
use React\Promise\Deferred;

$deferred = new Deferred();
$promise = $deferred->promise();

$f = new FutureValue(
    $promise,
    function () use ($deferred) {
        // This function is responsible for blocking and resolving the
        // promise. Here we pass in a reference to the deferred so that
        // it can be resolved or rejected.
        $deferred->resolve('foo');
    }
);
```

CHAPTER 3

Client Middleware

Middleware intercepts requests before they are sent over the wire and can be used to add functionality to handlers.

3.1 Modifying Requests

Let's say you wanted to modify requests before they are sent over the wire so that they always add specific headers. This can be accomplished by creating a function that accepts a handler and returns a new function that adds the composed behavior.

```
use GuzzleHttp\Ring\Client\CurlHandler;

$handler = new CurlHandler();

$addHeaderHandler = function (callable $handler, array $headers = []) {
    return function (array $request) use ($handler, $headers) {
        // Add our custom headers
        foreach ($headers as $key => $value) {
            $request['headers'][$key] = $value;
        }

        // Send the request using the handler and return the response.
        return $handler($request);
    }
};

// Create a new handler that adds headers to each request.
$handler = $addHeaderHandler($handler, [
    'X-AddMe'       => 'hello',
    'Authorization' => 'Basic xyz'
]);

$response = $handler([
    'http_method' => 'GET',
    'headers'     => ['Host' => ['httpbin.org']]
]);
```

3.2 Modifying Responses

You can change a response as it's returned from a middleware. Remember that responses returned from an handler (including middleware) must implement `GuzzleHttp\Ring\Future\FutureArrayInterface`. In order to be a good citizen, you should not expect that the responses returned through your middleware will be completed synchronously. Instead, you should use the `GuzzleHttp\Ring\Core::proxy()` function to modify the response when the underlying promise is resolved. This function is a helper function that makes it easy to create a new instance of `FutureArrayInterface` that wraps an existing `FutureArrayInterface` object.

Let's say you wanted to add headers to a response as they are returned from your middleware, but you want to make sure you aren't causing future responses to be dereferenced right away. You can achieve this by modifying the incoming request and using the `Core::proxy` function.

```
use GuzzleHttp\Ring\Core;
use GuzzleHttp\Ring\Client\CurlHandler;

$handler = new CurlHandler();

$responseHeaderHandler = function (callable $handler, array $headers) {
    return function (array $request) use ($handler, $headers) {
        // Send the request using the wrapped handler.
        return Core::proxy($handler($request), function ($response) use ($headers) {
            // Add the headers to the response when it is available.
            foreach ($headers as $key => $value) {
                $response['headers'][$key] = (array) $value;
            }
            // Note that you can return a regular response array when using
            // the proxy method.
            return $response;
        });
    }
};

// Create a new handler that adds headers to each response.
$handler = $responseHeaderHandler($handler, ['X-Header' => 'hello!']);

$response = $handler([
    'http_method' => 'GET',
    'headers'     => ['Host' => ['httpbin.org']]
]);

assert($response['headers']['X-Header'] == 'hello!');
```

3.3 Built-In Middleware

RingPHP comes with a few basic client middlewares that modify requests and responses.

3.3.1 Streaming Middleware

If you want to send all requests with the `streaming` option to a specific handler but other requests to a different handler, then use the streaming middleware.

```
use GuzzleHttp\Ring\Client\CurlHandler;
use GuzzleHttp\Ring\Client\StreamHandler;
```

```php
use GuzzleHttp\Ring\Client\Middleware;

$defaultHandler = new CurlHandler();
$streamingHandler = new StreamHandler();
$streamingHandler = Middleware::wrapStreaming(
    $defaultHandler,
    $streamingHandler
);

// Send the request using the streaming handler.
$response = $streamingHandler([
    'http_method' => 'GET',
    'headers'     => ['Host' => ['www.google.com']],
    'stream'      => true
]);

// Send the request using the default handler.
$response = $streamingHandler([
    'http_method' => 'GET',
    'headers'     => ['Host' => ['www.google.com']]
]);
```

3.3.2 Future Middleware

If you want to send all requests with the `future` option to a specific handler but other requests to a different handler, then use the future middleware.

```php
use GuzzleHttp\Ring\Client\CurlHandler;
use GuzzleHttp\Ring\Client\CurlMultiHandler;
use GuzzleHttp\Ring\Client\Middleware;

$defaultHandler = new CurlHandler();
$futureHandler = new CurlMultiHandler();
$futureHandler = Middleware::wrapFuture(
    $defaultHandler,
    $futureHandler
);

// Send the request using the blocking CurlHandler.
$response = $futureHandler([
    'http_method' => 'GET',
    'headers'     => ['Host' => ['www.google.com']]
]);

// Send the request using the non-blocking CurlMultiHandler.
$response = $futureHandler([
    'http_method' => 'GET',
    'headers'     => ['Host' => ['www.google.com']],
    'future'      => true
]);
```

CHAPTER 4

Client Handlers

Client handlers accept a request array and return a future response array that can be used synchronously as an array or asynchronously using a promise.

4.1 Built-In Handlers

RingPHP comes with three built-in client handlers.

4.1.1 Stream Handler

The `GuzzleHttp\Ring\Client\StreamHandler` uses PHP's http stream wrapper to send requests.

Note: This handler cannot send requests concurrently.

You can provide an associative array of custom stream context options to the StreamHandler using the `stream_context` key of the `client` request option.

```
use GuzzleHttp\Ring\Client\StreamHandler;

$response = $handler([
    'http_method' => 'GET',
    'uri' => '/',
    'headers' => ['host' => ['httpbin.org']],
    'client' => [
        'stream_context' => [
            'http' => [
                'request_fulluri' => true,
                'method' => 'HEAD'
            ],
            'socket' => [
                'bindto' => '127.0.0.1:0'
            ],
            'ssl' => [
                'verify_peer' => false
            ]
        ]
    ]
]);
```

```
// Even though it's already completed, you can still use a promise
$response->then(function ($response) {
    echo $response['status']; // 200
});

// Or access the response using the future interface
echo $response['status']; // 200
```

4.1.2 cURL Handler

The `GuzzleHttp\Ring\Client\CurlHandler` can be used with PHP 5.5+ to send requests using cURL easy handles. This handler is great for sending requests one at a time because the execute and select loop is implemented in C code which executes faster and consumes less memory than using PHP's `curl_multi_*` interface.

Note: This handler cannot send requests concurrently.

When using the CurlHandler, custom curl options can be specified as an associative array of cURL option constants mapping to values in the `client` option of a requst using the **curl** key.

```
use GuzzleHttp\Ring\Client\CurlHandler;

$handler = new CurlHandler();

$request = [
    'http_method' => 'GET',
    'headers'     => ['host' => [Server::$host]],
    'client'      => ['curl' => [CURLOPT_LOW_SPEED_LIMIT => 10]]
];

$response = $handler($request);

// The response can be used directly as an array.
echo $response['status']; // 200

// Or, it can be used as a promise (that has already fulfilled).
$response->then(function ($response) {
    echo $response['status']; // 200
});
```

4.1.3 cURL Multi Handler

The `GuzzleHttp\Ring\Client\CurlMultiHandler` transfers requests using cURL's multi API. The `CurlMultiHandler` is great for sending requests concurrently.

```
use GuzzleHttp\Ring\Client\CurlMultiHandler;

$handler = new CurlMultiHandler();

$request = [
    'http_method' => 'GET',
    'headers'     => ['host' => [Server::$host]]
];

// this call returns a future array immediately.
```

```php
$response = $handler($request);

// Ideally, you should use the promise API to not block.
$response
    ->then(function ($response) {
        // Got the response at some point in the future
        echo $response['status']; // 200
        // Don't break the chain
        return $response;
    })->then(function ($response) {
        // ...
    });

// If you really need to block, then you can use the response as an
// associative array. This will block until it has completed.
echo $response['status']; // 200
```

Just like the `CurlHandler`, the `CurlMultiHandler` accepts custom curl option in the `curl` key of the `client` request option.

4.1.4 Mock Handler

The `GuzzleHttp\Ring\Client\MockHandler` is used to return mock responses. When constructed, the handler can be configured to return the same response array over and over, a future response, or a the evaluation of a callback function.

```php
use GuzzleHttp\Ring\Client\MockHandler;

// Return a canned response.
$mock = new MockHandler(['status' => 200]);
$response = $mock([]);
assert(200 == $response['status']);
assert([] == $response['headers']);
```

4.2 Implementing Handlers

Client handlers are just PHP callables (functions or classes that have the `__invoke` magic method). The callable accepts a request array and MUST return an instance of `GuzzleHttp\Ring\Future\FutureArrayInterface` so that the response can be used by both blocking and non-blocking consumers.

Handlers need to follow a few simple rules:

1. Do not throw exceptions. If an error is encountered, return an array that contains the `error` key that maps to an `\Exception` value.
2. If the request has a `delay` client option, then the handler should only send the request after the specified delay time in seconds. Blocking handlers may find it convenient to just let the `GuzzleHttp\Ring\Core::doSleep($request)` function handle this for them.
3. Always return an instance of `GuzzleHttp\Ring\Future\FutureArrayInterface`.
4. Complete any outstanding requests when the handler is destructed.

CHAPTER 5

Testing

RingPHP tests client handlers using PHPUnit and a built-in node.js web server.

5.1 Running Tests

First, install the dependencies using Composer.

```
composer.phar install
```

Next, run the unit tests using `Make`.

```
make test
```

The tests are also run on Travis-CI on each commit: https://travis-ci.org/guzzle/guzzle-ring

5.2 Test Server

Testing client handlers usually involves actually sending HTTP requests. RingPHP provides a node.js web server that returns canned responses and keep a list of the requests that have been received. The server can then be queried to get a list of the requests that were sent by the client so that you can ensure that the client serialized and transferred requests as intended.

The server keeps a list of queued responses and returns responses that are popped off of the queue as HTTP requests are received. When there are not more responses to serve, the server returns a 500 error response.

The test server uses the `GuzzleHttp\Tests\Ring\Client\Server` class to control the server.

```
use GuzzleHttp\Ring\Client\StreamHandler;
use GuzzleHttp\Tests\Ring\Client\Server;

// First return a 200 followed by a 404 response.
Server::enqueue([
    ['status' => 200],
    ['status' => 404]
]);

$handler = new StreamHandler();

$response = $handler([
    'http_method' => 'GET',
    'headers'     => ['host' => [Server::$host]],
```

```
    'uri'          => '/'
]);

assert(200 == $response['status']);

$response = $handler([
    'http_method' => 'HEAD',
    'headers'     => ['host' => [Server::$host]],
    'uri'         => '/'
]);

assert(404 == $response['status']);
```

After requests have been sent, you can get a list of the requests as they were sent over the wire to ensure they were sent correctly.

```
$received = Server::received();

assert('GET' == $received[0]['http_method']);
assert('HEAD' == $received[1]['http_method']);
```

```
<?php
require 'vendor/autoload.php';

use GuzzleHttp\Ring\Client\CurlHandler;

$handler = new CurlHandler();
$response = $handler([
    'http_method' => 'GET',
    'uri'         => '/',
    'headers'     => [
        'host'  => ['www.google.com'],
        'x-foo' => ['baz']
    ]
]);

$response->then(function (array $response) {
    echo $response['status'];
});

$response->wait();
```

RingPHP is inspired by Clojure's Ring, which, in turn, was inspired by Python's WSGI and Ruby's Rack. RingPHP is utilized as the handler layer in Guzzle 5.0+ to send HTTP requests.