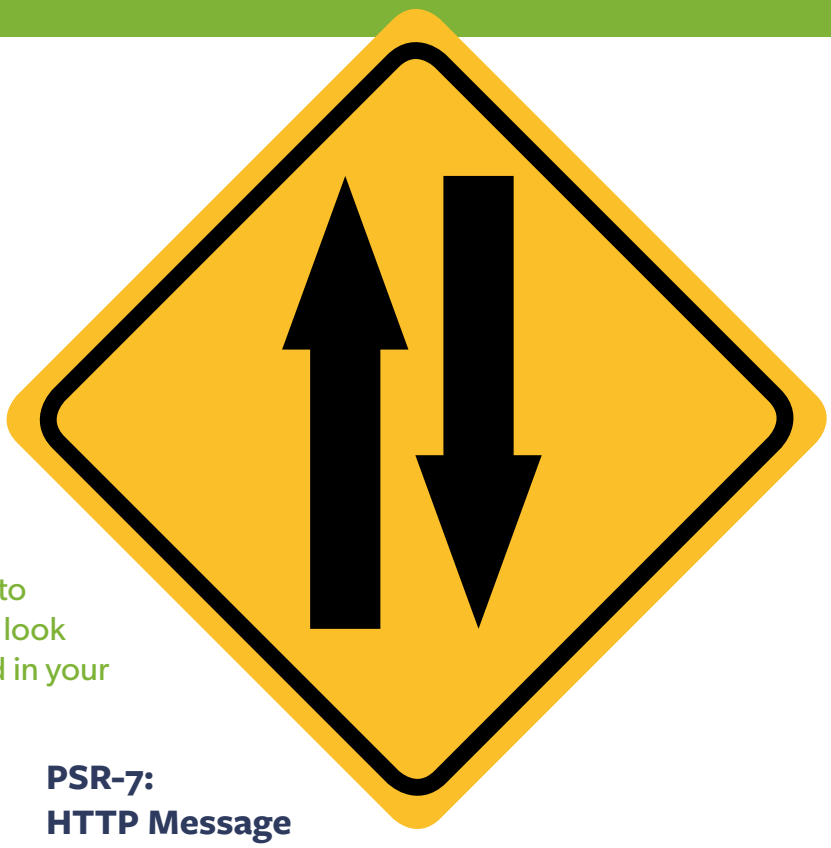
Abstracting HTTP Clients in PHP

David Buchmann



Many PHP applications are designed as server applications which receive a request and return a response to the client. However, PHP is also used to write API clients. This can be CLI tools written in PHP or server applications which need to talk to another system over HTTP. In this article, we'll look at a library to abstract out the HTTP client used in your code to keep it flexible and future-proof.

The PSR-7 standard[1] published by the Framework Interoperability Group (FIG) defines domain model for HTTP request and response. A standard for request and response is enough for server applications and middleware[2]. However, when we write code which acts as an HTTP *client*, we need to send a request. PSR-7 requests are value objects and can't send themselves—which is good, as they are also used to represent the request in a server application. There is no PSR for HTTP clients (yet).

HTTPlug[3] defines interfaces and behaviors for interoperable clients. Until we have a PSR defining HTTP clients, it's a good first step. Instead of writing code hard coupled to a specific client library like Guzzle[4], it can be written against HTTPlug and used with any supported client. You can use one client today with the knowledge if a better client comes along or your current one is no longer supported, you'll be able to easily switch it. This is valuable for all applications, and particularly useful for reusable libraries like general API clients which are meant to be integrated into applications. The HTTP client implementation should not be forced by a reusable library, but a choice of the integrator. HTTPlug is provided by the PHP HTTP group[5], along with additional tools. The organization consists of individuals interested in improving the state of HTTP support in PHP. At the time of writing, there are HTTPlug adapters for Guzzle 5 and 6, Buzz and React, as well as native cURL and socket clients.

1 *PSR-7 standard: http://www.php-fig.org/psr/psr-7/*

2 *Middleware is code which alters a request or response object and then passes it to the next layer.*

3 *HTTPlug: http://httplug.io*

4 *Guzzle: http://docs.guzzlephp.org*

5 *the PHP HTTP group: https://github.com/php-http*

PSR-7: HTTP Message Interfaces

PSR-7 defines interfaces for HTTP messages. Both requests and responses can have headers and a body. Requests additionally specify the requested URL and an HTTP method, like `GET`, `POST`, `PUT`, and so on. Responses have a status code. In addition to the messages themselves, PSR-7 also defines the `StreamInterface` to represent the (potentially huge) body, and the `UriInterface` to handle request URI information.

An important design decision is request and response objects are immutable. Methods to alter headers or add a body return an altered copy of the message, leaving the original message unchanged.

The full specification is available at http://www.php-fig.org/psr/psr-7/. For example, in the code below the `with·` methods return a copy of the original request.

```
$request = new GuzzleHttp\Psr7\Request();
$request = $request
   ->withMethod('GET')
   ->withUri($uri)
;

echo $request->getMethod(); // prints "GET"
```

Why Abstract Further?

PSR-7 by itself is a big help in interoperability. However, applications acting as HTTP clients need to create requests and send them with a client. There should be no hard coupling to specific PSR-7 and HTTP client implementations. This is even more important for reusable libraries. PHP does not support different parts of an application using different versions of the same library. Reusable libraries require a

specific major version of an HTTP client implementation or risk running into conflicts. Ideally, an application should only need to use one HTTP client implementation.

Most libraries should not need to know what client they are using. They need to send HTTP requests and receive responses. With the HTTPlug client interface, the library can state it needs an HTTP client without tying directly to a specific implementation. The only time specifics about the client configuration—like timeouts or special headers—have to be present on each request (e.g. API token) should be when creating the HTTP client. This should be handled in the bootstrapping part of the application and when the client is injected into the library. Bootstrapping is discussed in detail later in this article.

The HTTPlug Client

When we write PHP server applications, we see HTTP as:

1. Receive a request,
2. return a response.

Client applications, on the other hand, perceive HTTP as:

1. Send a request,
2. get a response back.

Clients could be CLI tools written in PHP or part of a server application that needs to send requests to other systems.

To send a request, we need a client. Instead of hard coupling our code to a specific client implementation or even raw cURL functions, we can use the HTTPlug interface. The HTTPlug interface consists of a single method:

```
sendRequest(RequestInterface \$request)
```

This method accepts any PSR-7 request and returns a PSR-7 response. It does not get much simpler!

Support for Asynchronous Requests

There is also an interface for asynchronous clients. Its method `sendAsyncRequest` returns a `Promise` which will eventually contain the response or an exception. This behavior is defined specifically for HTTPlug as PHP has no built-in support for promises and the PSR has not yet released their standard.

The HTTPlug promise has a `then()` method which accepts two callbacks. One of them will be executed at some point in the future. This allows your application to continue while a request is being sent, for example to send several asynchronous requests or perform other operations.

The first callback is called with the response, once it arrives. The second callback is called with an exception if and when the client throws an exception instead of returning a response.

It is important to note if the request has not yet been sent when the PHP process terminates, the requests will never be sent and the callbacks will not be called. To avoid issues, you should call the `Promise::wait()` method that blocks until the promise resolves with either success or an error. When the wait method returns, you can be sure one of the two callbacks has been executed. If it is important that the requests are handled, the strategy is to collect all promises and wait for each promise before exiting the application. On the console, that would be at the very end of the script; in web applications it is ideally *after* the response has been sent to the client. In Symfony, it would be an event listener which triggers on kernel.terminate. Listing 1 shows sending a request asynchronously and specifying the callbacks.

LISTING 1

```
01. $promise = $httpAsyncClient->sendAsyncRequest($request);
02. $promise->then(function (ResponseInterface $response) {
03.     // onFulfilled callback
04.     echo 'The response is available';
05.
06.     return $response;
07. }, function (Exception $e) {
08.     // onRejected callback
09.     echo 'An error happens';
10.
11.     throw $e;
12. });
13. // ...
14. $promise->wait();
```

Factories for PSR-7

If we do not want to bind the client application to a specific message implementation, we need a message factory which can create requests. The PHP-HTTP organization defines factories for messages, streams, and URIs. The package PHP-HTTP/message[6] provides factories for the Guzzle and Zend Diactoros PSR-7 implementations. Bootstrapping aside, the implementation is no longer coupled to a specific implementation. Listing 2 is an example usage of the PSR-7 factories.

LISTING 2

```
01. <?php
02. use Http\Message\MessageFactory\DiactorosMessageFactory;
03. use Http\Message\StreamFactory\GuzzleStreamFactory;
04.
05. $messageFactory = new DiactorosMessageFactory();
06. $streamFactory = new GuzzleStreamFactory();
07. // ...
08. $request = $messageFactory->createRequest(
09.     'GET', 'http://example.com'
10. );
11. $stream = $streamFactory->createStream('stream content');
12. $request = $request->withBody($stream);
```

Middleware

With PSR-7 setting a standard for requests and responses, middleware becomes easy to implement. From an HTTP client perspective, Middleware can change a request before it is sent out, or alter the response before it goes back to the application. It is usually implemented as a chain of logic which passes the request on and returns the response. From

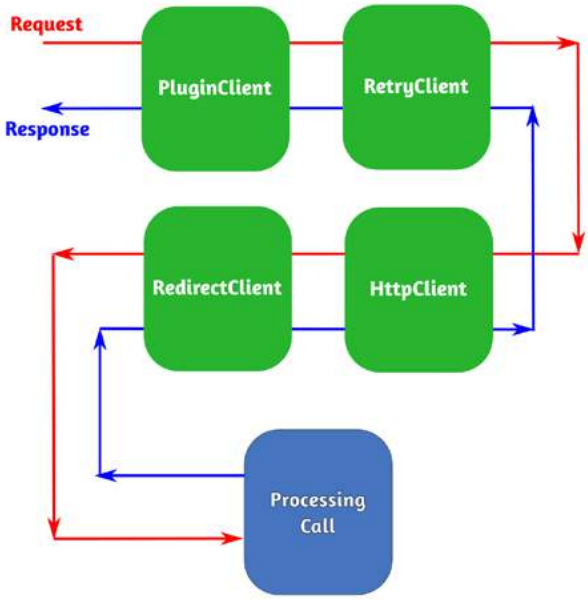
6 *PHP-HTTP/message:* http://docs.php-http.org/en/latest/message.html

the consumer point of view, middleware and the actual client do the same; they transform a request into a response. The difference is in the implementation—a middleware acts in the chain and forwards the request, while a client actually knows how to send the request to the server.

> *"From the consumer point of view, middleware and HTTP client transform a request into a response."*

A middleware could, for example, implement HTTP caching and check if we have a valid cached response for a request, and return the cached response instead of continuing the middleware chain. If no cache is found the middleware continues the chain and looks at the response to decide if the response can be added to the cache for the future.

Figure 1



As this example illustrates, the order of middlewares matters: those coming before the cache will already have altered the request. The changes they do to the response will be applied to cache hits as well. Middleware which comes after the cache is not executed on a cache hit. On the other hand, changes to the response done by middleware between the cache and the actual client get stored in the cache. Imagine a middleware which transforms HTTP status codes into domain exceptions—if this middleware is put at the start of the chain, the error could be stored in the cache and cache hits on the error response would still be transformed into exceptions. If the exception is thrown after the caching plugin, caching will not be attempted as the exception aborts the chain and will go straight to the library.

In HTTPlug, middleware is used with the PluginClient class that decorates a client and applies the plugins before forwarding to the client. As the order of plugins is relevant, plugins can only be set in the constructor of the PluginClient.

Using HTTPlug in Your Application

Let us build a simple application which uses HTTPlug to load the homepage of www.phparch.com.

Installation

HTTPlug is an abstraction from client implementation, but in the end your application will need a concrete client. In our example, we're using Guzzle 6. In an empty folder, run the following commands and choose to not select dependencies interactively:

```
composer init
composer require php-http/guzzle6-adapter php-http/message
```

Using HTTPlug

Once Composer has installed our dependencies, the smallest possible application looks as follows:

LISTING 3

```
01. #!/usr/bin/env php
02. <?php
03.
04. use Http\Adapter\Guzzle6\Client;
05. use Http\Message\MessageFactory\GuzzleMessageFactory;
06. require 'vendor/autoload.php';
07.
08. // HTTP implementation specific bootstrap code
09. $httpClient = new Client();
10. $messageFactory = new GuzzleMessageFactory();
11. // HTTP client agnostic application
12. $request = $messageFactory->createRequest(
13.     'GET', 'https://www.phparch.com/'
14. );
15. $response = $httpClient->sendRequest($request);
16. echo substr($response->getBody(), 0, 600) . '...';
```

If you invoke the script at the command line, you'll see output similar to:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en-US">

<head profile="http://gmpg.org/xfn/11">
    <meta http-equiv="Content-Type"
          content="text/html; charset=UTF-8" />
    <meta name="viewport" content="initial-scale=1,
          maximum-scale=1">
    <title>php[architect] &#8211; Magazine, Training,
           Books, Conferences</title>

    <link rel="shortcut icon" ...
```

Only the bootstrapping code needs to know which implementation is used. The rest of the application doesn't change when we need to adjust the client configuration or switch to a different implementation.

Let's add a plugin to the client to send an API key. First, we need to install the plugin decorator:

```
composer require php-http/client-common
```

With the code installed, we can adjust our bootstrap from the above code example to use the plugin:

```
$httpClient = new Http\Adapter\Guzzle6\Client();
$httpClient = new Http\Client\Common\PluginClient(
   $httpClient, [
       new Http\Client\Common\Plugin\HeaderSetPlugin(
          ['API-Key' => 'my-api-key']
       ),
]);
```

The application itself does not need to change; it will continue to talk to an object which implements `HttpClient`.

As a last and slightly more complicated example, we can cache responses. The cache plugin is in a separate package because it has an additional dependency on PSR-6[7]—the caching standard—and we also need a cache implementation:

```
composer require php-http/cache-plugin cache/filesys-
tem-adapter
```

The bootstrap for caching consists in setting up Flysystem and telling it the root folder of the cache, see Listing 4.

LISTING 4

```
01. use Cache\Adapter\Filesystem\FilesystemCachePool;
02. use Http\Client\Common\PluginClient;
03. use Http\Client\Common\Plugin\HeaderSetPlugin;
04. use Http\Client\Common\Plugin\CachePlugin;
05.
06. $filesystem = new FilesystemCachePool(
07.     new League\Flysystem\Filesystem(
08.         new League\Flysystem\Adapter\Local(
09.           sys_get_temp_dir() . DIRECTORY_SEPARATOR . 'phparch'
10.         )
11.     )
12. );
13. $cacheOpts = [
14.     'respect_cache_headers' => false,
15.     'default_ttl' => 60,
16. ];
17.
18. $httpClient = new PluginClient($httpClient, [
19.     new HeaderSetPlugin(['API-Key' => 'my-api-key']),
20.     new CachePlugin($filesystem, $streamFactory, $cacheOpts),
21. ]);
```

By default, the cache plugin respects the Cache-Control headers sent by the server. But the homepage of phparch.com has headers that forbid to cache. For the sake of the example, we configured the plugin to ignore the caching instructions from the server, and cache everything for one minute (60 seconds).

Writing an HTTP Client Agnostic Library

When writing a reusable library which does HTTP requests, the bootstrapping should not be in the library, to be completely implementation agnostic. This means that you also don't require a specific client implementation in your composer file, but only HTTPlug and the virtual package php-http/client-implementation:

Listing 5 is a `composer.json` for a HTTP client agnostic library. It is then up to the application consuming your library to chose which HTTPlug client implementation to use.

LISTING 5

```
01. {
02.     "require": {
03.         "php-http/client-implementation": "^1.0",
04.         "php-http/httplug": "^1.0",
05.         "php-http/message-factory": "^1.0"
06.     },
07.     "require-dev": {
08.         "php-http/mock-client": "^0.3",
09.         "guzzlehttp/psr7": "^1.0"
10.     }
11. }
```

Bootstrapping a Shared Library

In recent years, Inversion of Control (e.g. with Symfony 2 using the Dependency Injection pattern) has become popular in the PHP world. HTTPlug and the PSR-7 factories work very well with code built for Dependency Injection (DI). To use inversion of control, have your classes require the HTTP client and factories in their constructor. Then your code does not need to know which factories to instantiate or how to build the client. See Listing 6 for a constructor of a class which needs to send HTTP messages.

LISTING 6

```
01. <?php
02.
03. use Http\Client\HttpClient;
04. use Http\Message\MessageFactory;
05.
06. class ApiClient
07. {
08.     public function __construct(
09.         HttpClient $httpClient,
10.         MessageFactory $messageFactory
11.     ) {
12.         $this->httpClient = $httpClient;
13.         $this->messageFactory = $messageFactory;
14.     }
15.
16.     // ...
17. }
```

7 PSR-6: http://www.php-fig.org/psr/psr-6/

Discovery

In addition to DI, the PHP-HTTP/discovery[8] package provides static discovery functions to find a currently available implementation of the client or, of factories. This is mainly useful for a simple start with minimal configuration. It is recommended to only use discovery as a fallback, and always allow the user of the library to explicitly inject the required client and factories. Listing 7 shows a constructor with optional zero-config thanks to discovery.

LISTING 7

```
01. <?php
02. use Http\Client\HttpClient;
03. use Http\Discovery\HttpClientDiscovery;
04. use Http\Discovery\MessageFactoryDiscovery;
05. use Http\Message\MessageFactory;
06.
07. class ApiClient
08. {
09.     public function __construct(
10.         HttpClient $httpClient = null,
11.         MessageFactory $messageFactory = null
12.     ) {
13.         $this->httpClient = $httpClient ?:
14.                             HttpClientDiscovery::find();
15.         $this->messageFactory = $messageFactory ?:
16.                                 MessageFactoryDiscovery::find();
17.     }
18. }
```

Provide an HTTP Client Factory

If a library needs a specifically set up client, e.g. with plugins, the best way is to provide a factory. Together with discovery, you can keep the base client an optional argument allowing your users to fine tune the client to their needs if needed. Refer to Listing 8 for an example client factory which sets the host and accepts additional plugins.

When using an HTTP client factory, make the `$httpClient` constructor argument of the API client class required. This allows your users to wire factory and API client together.

Outlook

This article discussed decoupling code from HTTP client implementations. Ideally, these interfaces would not be provided by a particular group, but exist as PSR standards. There is hope; PSR-15[9], currently in draft status, attempts to standardize middleware on the server side. Once a PSR for promises is implemented, a PSR for client side middleware can be implemented to replace the PHP-HTTP Plugin interface. For message factories, there is PSR-17, also currently a draft. Message implementations will likely provide factories that implement the PSR-17 interfaces and the message factory part of PHP-HTTP can be retired.

LISTING 8

```
01. <?php
02.
03. class HttpClientFactory
04. {
05.     public static function createClient(
06.         string $host,
07.         array $plugins = [],
08.         HttpClient $httpClient = null
09.     ) {
10.         $host = UriFactoryDiscovery::find()->createUri($host);
11.         if (!$host->getHost()) {
12.             throw new \InvalidArgumentException(sprintf(
13.                 'server uri must specify the host: "%s"',
14.                 $host
15.             ));
16.         }
17.
18.         $plugins[] = new AddHostPlugin($host);
19.
20.         if (!$httpClient) {
21.             $httpClient = HttpClientDiscovery::find();
22.         }
23.
24.         return new PluginClient($httpClient, $plugins);
25.     }
26. }
```

For the client interface, the PHP-HTTP group is preparing to propose a PSR. This would enable clients to implement the interface directly instead of the adapters PHP-HTTP now provides.

Even without these PSRs, there are early adopters using PHP-HTTP to decouple their libraries from the HTTP client: Payum, Geocoder, Mailgun, the KNPLabs GitHub client, the Happyr LinkedIn client and others.

If you take away one thing from this article, it should be that API clients should not be coupled to a specific HTTP client implementation. HTTPlug is the best we currently have to decouple the client. Once the whole topic is covered by PSRs, migrating to the PSR standards should be easier if you start with HTTPlug instead of coupling to a specific client implementation.

David Buchmann works at Liip AG as Symfony expert. He is maintaining the Symfony Content Management Framework, co-author of the FOSHttpCacheBundle and active with the PHP-HTTP HTTPlug client abstraction. When he is not coding, he enjoys travelling with his girlfriend, music and boardgames. @dbu

8 *PHP-HTTP/discovery: http://docs.php-http.org/en/latest/discovery.html*

9 *PSR-15: http://www.php-fig.org/psr/*