PHP-HTTP Documentation

Release 1.0.0

The PHP-HTTP Team

Mar 16, 2024

CONTENTS

PHP-HTTP is the next step in standardizing HTTP interaction for PHP packages.

It builds on top of PSR-7, which defines interfaces for HTTP requests and responses. The HTTPlug HTTP client interface has been standardized in PSR-18 to define synchronous HTTP requests. When using a client that implements PSR-18, we recommend directly using PSR-18 and not HTTPlug nor our adapters.

However, PSR-18 does not define asynchronous requests. HTTPlug provides interfaces for those, but to do that also needs to define how *promises* are implemented.

PHP-HTTP has three goals:

1. Encourage package developers to depend on the simple HTTPlug interface instead of concrete HTTP clients.
2. Provide good quality HTTP-related packages to the PHP community.
3. Now that PSR-18 exists, we miss a PSR for asynchronous requests. This is blocked by not having a PSR for promises.

CHAPTER

ONE

HTTPLUG

HTTPlug abstracts from HTTP clients written in PHP, offering a simple interface. It also provides an implementation-independent plugin system to build pipelines regardless of the HTTP client implementation used.

Read more about *HTTPlug*.

1.1 They use us

Geocoder PHP

Fos Bundles

CHAPTER

TWO

PACKAGES

PHP-HTTP offers several packages:

Type	Description	Namespace
Clients	HTTP clients: Socket, cURL and others	`Http\Client\[Name]`
Client adapters	Adapters for other clients: Guzzle, React and others	`Http\Adapter\[Name]`
Plugins	Implementation-independent authentication, cookies and more	`Http\Client\Common\Plugin\[Name]`

Read more about *clients and adapters* and *plugins*.

CHAPTER

THREE

THE FUTURE

HTTPlug, as a working example of an HTTP client interface, can serve as a basis for discussion around a future HTTP client PSR.

CHAPTER

FOUR

COMMUNITY

If you want to get in touch, feel free to ask questions on our Slack channel or ping @httplug on Twitter.

4.1 HTTPlug: HTTP client abstraction

HTTPlug allows you to write reusable libraries and applications that need an HTTP client without binding to a specific implementation. When all packages used in an application only specify HTTPlug, the application developers can choose the client that best fits their project and use the same client with all packages.

4.1.1 Client Interfaces

HTTPlug defines two HTTP client interfaces that we kept as simple as possible:

- PSR-18 defines the `ClientInterface` with a `sendRequest` method that accepts a PSR-7 `RequestInterface` and either returns a PSR-7 `ResponseInterface` or throws an exception that implements `Psr\Http\Client\ClientExceptionInterface`.

  HTTPlug has the compatible interface `HttpClient` which now extends the PSR-18 interface to allow migrating to PSR-18.

- `HttpAsyncClient` defines a `sendAsyncRequest` method that sends a PSR-7 request asynchronously and always returns a `Http\Client\Promise`. See *Promise* for more information.

4.1.2 Implementations

PHP-HTTP offers two types of clients that implement the above interfaces:

1. Standalone clients that directly implement the interfaces.

   Examples: *cURL Client* and *Socket Client (deprecated)*.

2. Adapters that wrap existing HTTP clients, such as Guzzle. These adapters act as a bridge between the HTTPlug interfaces and the clients that do not (yet) implement these interfaces.

   More and more clients implement PSR-18 directly. If that is all you need, we recommend not using HTTPlug as it would only add overhead. However, as there is no PSR for asynchronous requests yet, you can use the adapters to do such requests without binding yourself to a specific implementation.

   Examples: *Guzzle 7 Adapter* and *React Adapter*.

Note: Ideally, there will be a PSR for asynchronous requests and all HTTP client libraries out there will implement PSR-18 and the not yet existing PSR. At that point, our adapters will no longer be necessary.

4.1.3 Usage

There are two main use cases for HTTPlug:

- Usage in an application that executes HTTP requests (see *HTTPlug Tutorial* and *Framework Integrations*);
- Usage in a reusable package that executes HTTP requests (see *HTTPlug for Library Developers*).

4.1.4 History

This project has been started by Eric Geloen as Ivory Http Adapter. It never made it to a stable release, but it relied on PSR-7 which was not stable either that time. Because of the constantly changing PSR-7, Eric had to rewrite the library over and over again (at least the message handling part, which in most cases affected every adapter as well).

In 2015, a decision has been made to move the library to its own organization, so PHP-HTTP was born.

See *Migrating to HTTPlug* for a guide how to migrate your code from the Ivory adapter.

4.2 HTTPlug usage

HTTPlug is relevant for three groups of users:

4.2.1 HTTPlug for library users

This page explains how to set up a library that depends on HTTPlug.

TL;DR

For the impatient: Require the following packages before requiring the library you plan to use:

```
composer require php-http/curl-client guzzlehttp/psr7 php-http/message
```

If you use a framework, check the *integrations* overview to see if there is a plugin for your framework.

Details

If a library depends on HTTPlug, it requires the virtual package php-http/client-implementation. A virtual package is used to declare that the library needs *an* implementation of the HTTPlug interfaces, but does not care which implementation specifically.

When using such a library, you need to choose a HTTPlug client and include that in your project explicitly. Lets say you want to use `some/awesome-library` that depends on `php-http/client implementation`. In the example we are using cURL:

```
$ composer require php-http/curl-client some/awesome-library
```

You can pick any of the clients or adapters *provided by PHP-HTTP*. Popular choices are `php-http/curl-client` and `php-http/guzzle6-adapter`.

Many libraries also need a PSR-7 implementation and the PHP-HTTP message factories to create messages. The PSR-7 implementations are Laminas Diactoros (also still supports the abandoned Zend Diactoros), Guzzle's PSR-7 and Slim Framework's PSR-7 messages. Do one of the following:

```
$ composer require php-http/message laminas/laminas-diactoros
```

```
$ composer require php-http/message guzzlehttp/psr7
```

```
$ composer require php-http/message slim/psr7
```

Troubleshooting

Composer fails

If you try to include the HTTPlug dependent library before you have included a HTTP client in your project, Composer will throw an error:

```
Loading composer repositories with package information
Updating dependencies (including require-dev)
Your requirements could not be resolved to an installable set of packages.

  Problem 1
    - The requested package php-http/client-implementation could not be found in any␣
↪version,
    there may be a typo in the package name.
```

You can solve this by including a HTTP client or adapter, as described above.

No Message Factories

You may get an exception telling you that "No message factories found", this means that either you have not installed a PSR-7 implementation or that there are no factories installed to create HTTP messages.

```
No message factories found. To use Guzzle or Diactoros factories install
php-http/message and the chosen message implementation.
```

You can solve this by including `php-http/message` and Zend Diactoros or Guzzle PSR-7, as described above.

Background

Reusable libraries do not depend on a concrete implementation but only on the virtual package `php-http/client-implementation`. This is to avoid hard coupling and allows the user of the library to choose the implementation. You can think of this as an "interface" or "contract" for packages.

The reusable libraries have no hard coupling to the PSR-7 implementation either, which gives you the flexibility to choose an implementation yourself.

4.2.2 HTTPlug Tutorial

This tutorial should give you an idea how to use HTTPlug in your project. HTTPlug has two main use cases:

1. Usage in your project;
2. Usage in a reusable package.

This tutorial will start with the first use case and then explain the special considerations to take into account when building a reusable package.

We use Composer for dependency management. Install it if you don't have it yet.

Note: If you are using a framework, check the *Framework Integrations* to see if there is an integration available. Framework integrations will simplify the way you set up clients, letting you focus on handling the requests.

Setting up the Project

```
mkdir httplug-tutorial
cd httplug-tutorial
composer init
# specify your information as you want. say no to defining the dependencies interactively
composer require php-http/guzzle6-adapter
```

The last command will install Guzzle as well as the Guzzle HTTPlug adapter and the required interface repositories. We are now ready to start coding.

Writing Some Simple Code

Create a file `demo.php` in the root folder and write the following code:

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

use Http\Discovery\HttpClientDiscovery;
use Http\Discovery\MessageFactoryDiscovery;

$client = HttpClientDiscovery::find();
$messageFactory = MessageFactoryDiscovery::find();
$homeResponse = $client->sendRequest(
    $messageFactory->createRequest('GET', 'http://httplug.io')
);

var_dump($homeResponse->getStatusCode()); // 200, hopefully

$missingPageResponse = $client->sendRequest(
    $messageFactory->createRequest('GET', 'http://httplug.io/missingPage')
);

var_dump($missingPageResponse->getStatusCode()); // 404
```

Using an Asynchronous Client

Asynchronous client accepts a PSR-7 `RequestInterface` and returns a `Http\Promise\Promise`:

```
use Http\Discovery\HttpAsyncClientDiscovery;

$httpAsyncClient = HttpAsyncClientDiscovery::find();
$promise = $httpAsyncClient->sendAsyncRequest($request);
```

Using the Callback System

The promise allows you to add callbacks for when the response is available or an errors happens by using the then method:

```
$promise->then(function (ResponseInterface $response) {
    // onFulfilled callback
    echo 'The response is available';

    return $response;
}, function (Exception $e) {
    // onRejected callback
    echo 'An error happens';

    throw $e;
});
```

This method will return another promise so you can manipulate the response and/or exception and still provide a way to interact with this object for your users:

```
$promise->then(function (ResponseInterface $response) {
    // onFulfilled callback
    echo 'The response is available';

    return $response;
}, function (Exception $e) {
    // onRejected callback
    echo 'An error happens';

    throw $e;
})->then(function (ResponseInterface $response) {
    echo 'Response still available';

    return $response;
}, function (Exception $e) {
     throw $e
});
```

In order to achieve the chain callback, if you read previous examples carefully, callbacks provided to the `then` method *must* return a PSR-7 `ResponseInterface` or throw a `Http\Client\Exception`. For both of the callbacks, if it returns a PSR-7 `ResponseInterface` it will call the `onFulfilled` callback for the next element in the chain, if it throws a `Http\Client\Exception` it will call the `onRejected` callback.

i.e. you can inverse the behavior of a call:

```
$promise->then(function (ResponseInterface $response) use($request) {
    // onFulfilled callback
    echo 'The response is available, but it\'s not ok...';

    throw new HttpException('My error message', $request, $response);
}, function (Exception $e) {
    // onRejected callback
    echo 'An error happens, but it\'s ok...';

    return $exception->getResponse();
});
```

Calling the `wait` method on the promise will wait for the response or exception to be available and invoke callback provided in the `then` method.

Using the promise directly

If you don't want to use the callback system, you can also get the state of the promise with `$promise->getState()` will return of one `Promise::PENDING`, `Promise::FULFILLED` or `Promise::REJECTED`.

Then you can get the response of the promise if it's in `FULFILLED` state or trigger the exception of the promise if it's in `REJECTED` state with `$promise->wait(true)` call.

Note: Read *Promise* for more information about promises.

Example

Here is a full example of a classic usage when using the `sendAsyncRequest` method:

```
use Http\Client\Exception;
use Http\Discovery\HttpAsyncClientDiscovery;

$httpAsyncClient = HttpAsyncClientDiscovery::find();

$promise = $httpAsyncClient->sendAsyncRequest($request);
$promise->then(function (ResponseInterface $response) {
   echo 'The response is available';

   return $response;
}, function (Exception $e) {
   echo 'An error happens';

   throw $e;
});

// Do some stuff not depending on the response, calling another request, etc ..
...

try {
    // We need now the response for our final treatment...
```

(continues on next page)

(continued from previous page)

```
    $response = $promise->wait(true);
} catch (Exception $e) {
    // ...or catch the thrown exception
}

// Do your stuff with the response
...
```

Handling Errors

TODO: explain how to handle exceptions, distinction between network exception and HttpException.

4.2.3 HTTPlug for Library Developers

If you're developing a library or framework that performs HTTP requests, you should not be dependent on concrete HTTP client libraries (such as Guzzle). Instead, you should only make sure that *some* HTTP client is available. It is then up to your users to decide which HTTP client they want to include in their projects. This complies with the dependency inversion principle.

Manage Dependencies

To depend on *some* HTTP client, specify either `psr/http-client-implementation` for PSR-18 synchronous requests or `php-http/async-client-implementation` for asynchronous requests in your library's `composer.json`. These are virtual Composer packages that will throw an error if no concrete client was found:

```
{
    "name": "you/and-your-awesome-library",
    "require": {
        "psr/http-client-implementation": "^1.0"
    }
}
```

Your users then include your project alongside with a HTTP client of their choosing in their project's `composer.json`. In this case, the user decided to include the Socket client:

```
{
    "name": "some-user/nice-project",
    "require": {
        "you/and-your-awesome-library": "^1.2",
        "php-http/socket-client": "^1.0"
    }
}
```

Testing your library

When you install your library on a CI-server (like Travis) you need to include a client. So specify any concrete client in the `require-dev` section in your library's `composer.json`. You could use any client but the *Mock Client* will make it easier to write good tests.

```
{
    "name": "you/and-your-awesome-library",
    "require": {
        "psr/http-client-implementation": "^1.0"
    },
    "require-dev": {
        "php-http/mock-client": "^1.0"
    }
}
```

Messages

When you construct HTTP message objects in your library, you should not depend on a concrete PSR-7 message implementation. Instead, use the *HTTP factories*.

Discovery

To make it as convenient as possible for your users you should use the *Discovery* component. It will help you find factories to create `Request`, `Streams` etc. That component is light weight and has no hard dependencies.

Plugins

If your library relies on specific plugins, the recommended way is to provide a factory method for your users, so they can create the correct client from a base HttpClient. See *Libraries that Require Plugins* for a concrete example.

User Documentation

To explain to your users that they need to install a concrete HTTP client, you can point them to *HTTPlug for library users*.

Your Final `composer.json`

Putting it all together your final `composer.json` is much likely to look similar to this:

```
{
    "name": "you/and-your-awesome-library",
    "require": {
        "psr/http-message": "^1.0",
        "psr/http-client-implementation": "^1.0",
        "php-http/httplug": "^2.0",
        "php-http/message-factory": "^1.0",
        "php-http/discovery": "^1.0"
    },
    "require-dev": {
```

(continues on next page)

(continued from previous page)

```
        "php-http/mock-client": "^1.0",
        "php-http/message": "^1.0",
        "guzzlehttp/psr7": "^1.0"
    }
}
```

4.3 Exceptions

HTTPlug defines a common interface for all exceptions thrown by HTTPlug implementations. Every exception thrown by a HTTP client must implement `Http\Client\Exception`.

`HttpClient::sendRequest()` can throw one of the following exceptions.

Exception	Thrown when	Methods available
TransferException	something unexpected happened	-
└ RequestException	the request is invalid	`getRequest()`
└ NetworkException	no response received due to network issues	`getRequest()`
└ HttpException	error response	`getRequest()` `getResponse()`

Note: By default clients will always return a PSR-7 response instead of throwing a `HttpException`. Write your application to check the response status or use the *Error Plugin* to make sure the `HttpException` is thrown.

Note: The `sendAsyncRequest` should never throw an exception but always return a *Promise*. The exception classes used in `Promise::wait` and the `then` callback are however the same as explained here.

4.4 Migrating to HTTPlug

If you currently use a concrete HTTP client implementation or the Ivory HTTP Adapter, migrating to HTTPlug should not be very hard.

4.4.1 Migrating from Ivory HTTP Adapter

The monolithic Ivory package has been separated into several smaller, more specific packages.

Instead of `Ivory\HttpAdapter\PsrHttpAdapter`, use `Http\Client\HttpClient`. The HttpClient simply has a method to send requests.

If you used the `Ivory\HttpAdapter\HttpAdapter`, have a look at *Client Common* and use the `Http\Client\Utils\HttpMethodsClient` which wraps any HttpClient and provides the convenience methods to send requests without creating RequestInterface instances.

4.4.2 Migrating from Guzzle

Replace usage of `GuzzleHttp\ClientInterface` with `Http\Client\HttpClient`. The `send` method is called `sendRequest`. Instead of the `$options` argument, configure the client appropriately during set up. If you need different settings, create different instances of the client. You can use *Plugins* to further tune your client.

If you used the `request` method, have a look at *Client Common* and use the `Http\Client\Utils\HttpMethodsClient` which wraps any HttpClient and provides the convenience methods to send requests without creating RequestInterface instances.

4.5 Clients & Adapters

There are two types of libraries you can use to send HTTP messages; clients and adapters. A client implements the `HttpClient` and/or the `HttpAsyncClient` interfaces directly. A client adapter is a class implementing the interface and forwarding the calls to an HTTP client not implementing the interface. (See Adapter pattern on Wikipedia).

Hint: Modern PHP clients implement the `PSR-18 HTTP Client` standard. If you want to do synchronous requests, you don't need a PHP-HTTP adapter anymore. We keep providing the the curl client and a mock client for testing.

The adapters are still useful if you need the PHP-HTTP `HttpAsyncClient`.

Note: All clients and adapters comply with Liskov substitution principle which means that you can easily change one for another without any side effects.

4.5.1 cURL Client

This client uses cURL PHP extension.

Installation

To install the cURL client, run:

```
$ composer require php-http/curl-client
```

Usage

The cURL client needs a PSR-17 message factory and stream factory to work. You can either specify the factory explicitly:

```
use Http\Client\Curl\Client;
use Http\Message\MessageFactory\DiactorosMessageFactory;
use Http\Message\StreamFactory\DiactorosStreamFactory;

$client = new Client(new DiactorosMessageFactory(), new DiactorosStreamFactory());
```

Or you can omit the parameters and let the client use *Discovery* to determine a suitable factory:

```
use Http\Client\Curl\Client;
use Http\Discovery\MessageFactoryDiscovery;
use Http\Discovery\StreamFactoryDiscovery;

$client = new Client();
```

Configuring Client

You can use cURL options to configure Client:

```
use Http\Client\Curl\Client;
use Http\Discovery\MessageFactoryDiscovery;
use Http\Discovery\StreamFactoryDiscovery;

$options = [
    CURLOPT_CONNECTTIMEOUT => 10, // The number of seconds to wait while trying to␣
↪connect.
];
$client = new Client(null, null, $options);
```

The following options can not be changed in the set up. Most of them are to be provided with the request instead:

- CURLOPT_CUSTOMREQUEST
- CURLOPT_FOLLOWLOCATION
- CURLOPT_HEADER
- CURLOPT_HTTP_VERSION
- CURLOPT_HTTPHEADER
- CURLOPT_NOBODY
- CURLOPT_POSTFIELDS
- CURLOPT_RETURNTRANSFER
- CURLOPT_URL
- CURLOPT_USERPWD

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.2 Mock Client

The mock client is a special type of client. It is a test double that does not send the requests that you pass to it, but collects them instead. You can then retrieve those request objects and make assertions about them. Additionally, you can fake HTTP server responses and exceptions to validate how your code handles them. This behavior is most useful in tests.

To install the Mock client, run:

```
$ composer require php-http/mock-client
```

Collect Requests

To make assertions:

```
use Http\Mock\Client;

class YourTest extends \PHPUnit_Framework_TestCase
{
    public function testRequests()
    {
        // $firstRequest and $secondRequest are Psr\Http\Message\RequestInterface
        // objects

        $client = new Client();
        $client->sendRequest($firstRequest);
        $client->sendRequest($secondRequest);

        $bothRequests = $client->getRequests();

        // Do your assertions
        $this->assertEquals('GET', $bothRequests[0]->getMethod());
        // ...
    }
}
```

Fake Responses and Exceptions

By default, the mock client returns an empty response with status 200. You can set responses and exceptions the mock client should return / throw. You can set several exceptions and responses, to have the client first throw each exception once and then each response once on subsequent calls to send(). Additionally you can set a default response or a default exception to be used instead of the empty response.

Test how your code behaves when the HTTP client throws exceptions or returns certain responses:

```
use Http\Mock\Client;

class YourTest extends \PHPUnit_Framework_TestCase
{
    public function testClientReturnsResponse()
    {
        $client = new Client();
```

(continues on next page)

(continued from previous page)

```
        $response = $this->createMock('Psr\Http\Message\ResponseInterface');
        $client->addResponse($response);

        // $request is an instance of Psr\Http\Message\RequestInterface
        $returnedResponse = $client->sendRequest($request);
        $this->assertSame($response, $returnedResponse);
        $this->assertSame($request, $client->getLastRequest());
    }
}
```

Or set a default response:

```
use Http\Mock\Client;

class YourTest extends \PHPUnit_Framework_TestCase
{
    public function testClientReturnsResponse()
    {
        $client = new Client();

        $response = $this->createMock('Psr\Http\Message\ResponseInterface');
        $client->setDefaultResponse($response);

        // $firstRequest and $secondRequest are instances of Psr\Http\Message\
↪RequestInterface
        $firstReturnedResponse = $client->sendRequest($firstRequest);
        $secondReturnedResponse = $client->sendRequest($secondRequest);
        $this->assertSame($response, $firstReturnedResponse);
        $this->assertSame($response, $secondReturnedResponse);
    }
}
```

To fake an exception being thrown:

```
use Http\Mock\Client;

class YourTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @expectedException \Exception
     */
    public function testClientThrowsException()
    {
        $client = new Client();

        $exception = new \Exception('Whoops!');
        $client->addException($exception);

        // $request is an instance of Psr\Http\Message\RequestInterface
        $returnedResponse = $client->sendRequest($request);
    }
}
```

Or set a default exception:

```
use Http\Mock\Client;

class YourTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @expectedException \Exception
     */
    public function testClientThrowsException()
    {
        $client = new Client();

        $exception = new \Exception('Whoops!');
        $client->setDefaultException($exception);

        $response = $this->createMock('Psr\Http\Message\ResponseInterface');
        $client->addResponse($response);

        // $firstRequest and $secondRequest are instances of Psr\Http\Message\
↪RequestInterface
        // The first request will returns the added response.
        $firstReturnedResponse = $client->sendRequest($firstRequest);
        // There is no more added response, the default exception will be thrown.
        $secondReturnedResponse = $client->sendRequest($secondRequest);
    }
}
```

Mocking request-dependent responses

You can mock responses and exceptions which are only used in certain requests or act differently depending on the request.

To conditionally return a response when a request is matched:

```
use Http\Mock\Client;
use Psr\Http\Message\ResponseInterface;

class YourTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @expectedException \Exception
     */
    public function testClientThrowsException()
    {
        $client = new Client();

        // $requestMatcher is an instance of Http\Message\RequestMatcher

        $response = $this->createMock(ResponseInterface::class);
        $client->on($requestMatcher, $response);

        // $request is an instance of Psr\Http\Message\RequestInterface
```

(continues on next page)

(continued from previous page)

```
        $returnedResponse = $client->sendRequest($request);
    }
}
```

Or for an exception:

```
use Http\Mock\Client;
use Exception;

class YourTest extends \PHPUnit_Framework_TestCase
{
    public function testClientThrowsException()
    {
        $client = new Client();

        // $requestMatcher is an instance of Http\Message\RequestMatcher

        $exception = new \Exception('Whoops!');
        $client->on($requestMatcher, $exception);

        // $request is an instance of Psr\Http\Message\RequestInterface

        $this->expectException(\Exception::class);
        $returnedResponse = $client->sendRequest($request);
    }
}
```

Or pass a callable, and return a response or exception based on the request:

```
use Http\Mock\Client;
use Psr\Http\Message\RequestInterface;

class YourTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @expectedException \Exception
     */
    public function testClientThrowsException()
    {
        $client = new Client();

        // $requestMatcher is an instance of Http\Message\RequestMatcher

        $client->on($requestMatcher, function (RequestInterface $request) {

            // return a response
            return $this->createMock('Psr\Http\Message\ResponseInterface');

            // or an exception
            return new \Exception('Whoops!');
        });
```

(continues on next page)

(continued from previous page)

```
        // $request is an instance of Psr\Http\Message\RequestInterface
        $returnedResponse = $client->sendRequest($request);
    }
}
```

Hint: If you're using the *Symfony Bundle*, the mock client is available as a service with `httplug.client.mock` id.

See *Mock Responses In Functional Tests* for more on how to use the mock client in Symfony.

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.3 Symfony Client

The Symfony HTTP client provides a `HttplugClient` class that implements the `Http\Client\HttpAsyncClient`. Until Symfony 5.4, it also implemented the `Http\Client\HttpClient`, newer versions implement the PSR-18 `HttpClientInterface` instead.

Installation

The Symfony client does not depend on HTTPlug, but the `HttplugClient` does. To use the Symfony client with HTTPlug, you need to install both the client and HTTPlug with:

```
$ composer require symfony/http-client php-http/httplug
```

This client does not come with a PSR-7 implementation out of the box. If you do not require one, *discovery <../discovery>* will install Nyholm PSR-7. If you do not allow the composer plugin of the `php-http/discovery` component, you need to install a PSR-7 implementation manually:

```
$ composer require nyholm/psr7
```

Usage

```
use Symfony\Component\HttpClient\HttplugClient;

$symfonyClient = new HttplugClient();
```

Note: Check the official Symfony HttpClient documentation for more details.

4.5.4 Artax Adapter

An HTTPlug adapter for the Artax HTTP client.

Installation

To install the Artax adapter, which will also install Artax itself (if it was not yet included in your project), run:

```
$ composer require php-http/artax-adapter
```

Usage

Begin by creating a Artax adapter:

```
use Amp\Artax\DefaultClient;
use Http\Adapter\Artax\Client as ArtaxAdapter;
use Http\Message\MessageFactory\GuzzleMessageFactory;

$adapter = new ArtaxAdapter(new DefaultClient(), new GuzzleMessageFactory());
```

Or relying on *discovery*:

```
use Http\Adapter\Artax\Client as ArtaxAdapter;

$adapter = new ArtaxAdapter();
```

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.5 Guzzle 7 Adapter

An HTTPlug adapter for the Guzzle 7 HTTP client. Guzzle 7 supports PSR-18 out of the box. This adapter makes sense if you want to use HTTPlug async interface or to use Guzzle 7 with a library that did not upgrade to PSR-18 yet and depends on `php-http/client-implementation`.

Installation

To install the Guzzle adapter, which will also install Guzzle itself (if it was not yet included in your project), run:

```
$ composer require php-http/guzzle7-adapter
```

Usage

To create a Guzzle7 adapter you should use the *createWithConfig()* function. It will let you to pass Guzzle configuration to the client:

```
use Http\Adapter\Guzzle7\Client as GuzzleAdapter;

$config = [
    'timeout' => 2,
    'handler' => //...
    // ...
];
$adapter = GuzzleAdapter::createWithConfig($config);
```

Note: If you want even more control over your Guzzle object, you may give a Guzzle client as first argument to the adapter's constructor:

```
use GuzzleHttp\Client as GuzzleClient;
use Http\Adapter\Guzzle7\Client as GuzzleAdapter;

$config = ['timeout' => 5];
// ...
$guzzle = new GuzzleClient($config);
// ...
$adapter = new GuzzleAdapter($guzzle);
```

If you pass a Guzzle instance to the adapter, make sure to configure Guzzle to not throw exceptions on HTTP error status codes, or this adapter will violate PSR-18.

And use it to send synchronous requests:

```
use GuzzleHttp\Psr7\Request;

$request = new Request('GET', 'http://httpbin.org');

// Returns a Psr\Http\Message\ResponseInterface
$response = $adapter->sendRequest($request);
```

Or send asynchronous ones:

```
use GuzzleHttp\Psr7\Request;

$request = new Request('GET', 'http://httpbin.org');

// Returns a Http\Promise\Promise
$promise = $adapter->sendAsyncRequest(request);
```

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.6 React Adapter

An HTTPlug adapter for the React Http client.

Installation

Use Composer to install the React adapter. It will also install React as it's a dependency.

```
$ composer require php-http/react-adapter
```

Usage

If you need control on the React instances, you can inject them during initialization:

```php
use Http\Adapter\React\Client;

$systemDnsConfig = React\Dns\Config\Config::loadSystemConfigBlocking();
if (!$config->nameservers) {
    $config->nameservers[] = '8.8.8.8';
}

$dnsResolverFactory = new React\Dns\Resolver\Factory();
$dnsResolver = $factory->create($config);

$connector = new React\Socket\Connector([
    'dns' => $dnsResolver,
]);
$browser = new React\Http\Browser($connector);

$adapter = new Client($browser);
```

You can also use a `ReactFactory` in order to initialize React instances:

```php
use Http\Adapter\React\ReactFactory;

$reactHttp = ReactFactory::buildHttpClient();
```

Then you can use the adapter to send synchronous requests:

```php
use GuzzleHttp\Psr7\Request;

$request = new Request('GET', 'http://httpbin.org');
```

(continues on next page)

(continued from previous page)

```
// Returns a Psr\Http\Message\ResponseInterface
$response = $adapter->sendRequest($request);
```

Or send asynchronous ones:

```
use GuzzleHttp\Psr7\Request;

$request = new Request('GET', 'http://httpbin.org');

// Returns a Http\Promise\Promise
$promise = $adapter->sendAsyncRequest(request);
```

Note that since v4 calling *wait* on *HttpPromisePromise* is expected to run inside a fiber:

```
use function React\Async\async;

async(static function () {
    // Returns a Http\Promise\Promise
    $promise = $adapter->sendAsyncRequest(request);

    // Returns a Psr\Http\Message\ResponseInterface
    $response = $promise->wait();
})();
```

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.7 Buzz Adapter (deprecated)

This adapter only implements the PHP-HTTP synchronous interface. This interface has been superseded by PSR-18, which the Buzz HTTP client implements directly.

Installation

To install the Buzz adapter, which will also install Buzz itself (if it was not yet included in your project), run:

```
$ composer require php-http/buzz-adapter
```

Usage

Begin by creating a Buzz client, you may pass any listener or configuration parameters to it like:

```
use Buzz\Browser;
use Buzz\Client\Curl;
use Buzz\Listener\CookieListener;

$browser = new Browser();

$client = new Curl();
$client->setMaxRedirects(0);
$browser->setClient($client);

// Create CookieListener
$listener = new CookieListener();
$browser->addListener($listener);
```

Then create the adapter:

```
use Http\Adapter\Buzz\Client as BuzzAdapter;
use Http\Message\MessageFactory\GuzzleMessageFactory;

$adapter = new BuzzAdapter($browser, new GuzzleMessageFactory());
```

Or relying on *discovery*:

```
use Http\Adapter\Buzz\Client as BuzzAdapter;

$adapter = new BuzzAdapter($browser);
```

Be Aware

This adapter violates the Liskov substitution principle in a rare edge case. When the adapter is configured to use Buzz' Curl client, it does not send request bodies for request methods such as GET, HEAD and TRACE. A `RequestException` will be thrown if this ever happens.

If you need GET request with a body (e.g. for Elasticsearch) you need to use the Buzz FileGetContents client or choose a different HTTPlug client like Guzzle 6.

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.

4.5.8 CakePHP Adapter (deprecated)

This adapter only implements the PHP-HTTP synchronous interface. This interface has been superseded by PSR-18, which the CakePHP HTTP client implements directly.

Installation

To install the CakePHP adapter, which will also install CakePHP itself (if it was not yet included in your project), run:

```
$ composer require php-http/cakephp-adapter
```

Usage

Begin by creating a CakePHP HTTP client, passing any configuration parameters you like:

```
use Cake\Http\Client as CakeClient;

$config = [
    // Config params
];
$cakeClient = new CakeClient($config);
```

Then create the adapter:

```
use Http\Adapter\Cake\Client as CakeAdapter;

$adapter = new CakeAdapter($cakeClient);
```

Note: The client parameter is optional; if you do not supply it (or set it to `null`) the adapter will create a default CakePHP HTTP client without any options.

Or if you installed the *discovery* layer:

```
use Http\Adapter\Cake\Client as CakeAdapter;

$adapter = new CakeAdapter($cakeClient);
```

Warning: The message factory parameter is mandatory if the discovery layer is not installed.

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.9 Guzzle5 Adapter (deprecated)

An HTTPlug adapter for the Guzzle 5 HTTP client.

This adapter only implements the PHP-HTTP synchronous interface. This interface has been superseded by PSR-18.

Guzzle 5 is very old and not maintained anymore. We recommend to upgrade to Guzzle version 7.

Installation

To install the Guzzle adapter, which will also install Guzzle itself (if it was not yet included in your project), run:

```
$ composer require php-http/guzzle5-adapter
```

Usage

Begin by creating a Guzzle client, passing any configuration parameters you like:

```
use GuzzleHttp\Client as GuzzleClient;

$config = [
    // Config params
];
$guzzle = new GuzzleClient($config);
```

Then create the adapter:

```
use Http\Adapter\Guzzle5\Client as GuzzleAdapter;
use Http\Message\MessageFactory\GuzzleMessageFactory;

$adapter = new GuzzleAdapter($guzzle, new GuzzleMessageFactory());
```

Or if you installed the *discovery* layer:

```
use Http\Adapter\Guzzle5\Client as GuzzleAdapter;

$adapter = new GuzzleAdapter($guzzle);
```

Warning: The message factory parameter is mandatory if the discovery layer is not installed.

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.

4.5.10 Guzzle 6 Adapter (deprecated)

An HTTPlug adapter for the Guzzle 6 HTTP client.

Guzzle 5 is not maintained anymore. We recommend to upgrade to Guzzle version 7.

Installation

To install the Guzzle adapter, which will also install Guzzle itself (if it was not yet included in your project), run:

```
$ composer require php-http/guzzle6-adapter
```

Usage

To create a Guzzle6 adapter you should use the *createWithConfig()* function. It will let you to pass Guzzle configuration to the client:

```
use Http\Adapter\Guzzle6\Client as GuzzleAdapter;

$config = [
    'timeout' => 2,
    'handler' => //...
    // ...
];
$adapter = GuzzleAdapter::createWithConfig($config);
```

Note: If you want even more control over your Guzzle object, you may give a Guzzle client as first argument to the adapter's constructor:

```
use GuzzleHttp\Client as GuzzleClient;
use Http\Adapter\Guzzle6\Client as GuzzleAdapter;

$config = ['timeout' => 5];
// ...
$guzzle = new GuzzleClient($config);
// ...
$adapter = new GuzzleAdapter($guzzle);
```

If you pass a Guzzle instance to the adapter, make sure to configure Guzzle to not throw exceptions on HTTP error status codes, or this adapter will violate PSR-18.

And use it to send synchronous requests:

```
use GuzzleHttp\Psr7\Request;

$request = new Request('GET', 'http://httpbin.org');

// Returns a Psr\Http\Message\ResponseInterface
$response = $adapter->sendRequest($request);
```

Or send asynchronous ones:

```php
use GuzzleHttp\Psr7\Request;

$request = new Request('GET', 'http://httpbin.org');

// Returns a Http\Promise\Promise
$promise = $adapter->sendAsyncRequest(request);
```

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.
- Read more about *promises* when using asynchronous requests.

4.5.11 Socket Client (deprecated)

The socket client uses the stream extension from PHP, which is integrated into the core.

This client only implements the PHP-HTTP synchronous interface, which has been superseded by PSR-18. Use one of the PSR-18 clients instead.

Features

- TCP Socket Domain (`tcp://hostname:port`)
- UNIX Socket Domain (`unix:///path/to/socket.sock`)
- TLS / SSL encryption
- Client Certificate (only for PHP > 5.6)

Installation

To install the Socket client, run:

```
$ composer require php-http/socket-client
```

This client does not come with a PSR-7 implementation out of the box, so you have to install one as well (for example Guzzle PSR-7):

```
$ composer require guzzlehttp/psr7
```

In order to provide full interoperability, message implementations are accessed through *factories*. Message factories for Laminas Diactoros (and its abandoned predecessor Zend Diactoros), Guzzle PSR-7 and Slim PSR-7 are available in the *message* component:

```
$ composer require php-http/message
```

Usage

The Socket client needs a *message factory* in order to to work:

```php
use Http\Client\Socket\Client;

$options = [];
$client = new Client($messageFactory, $options);
```

The available options are:

remote_socket
Specify the remote socket where the library should send the request to

- Can be a TCP remote: `tcp://hostname:port`
- Can be a UNIX remote: `unix:///path/to/remote.sock`
- Do not use a TLS/SSL scheme, this is handle by the SSL option.
- If not set, the client will try to determine it from the request URI or host header.

timeout
Timeout in milliseconds for writing request and reading response on the remote

ssl
Activate or deactivate SSL/TLS encryption

stream_context_options
Custom options for the context of the stream. See PHP stream context options.

stream_context_params
Custom parameters for the context of the stream. See PHP stream context parameters.

write_buffer_size
When sending the request we need to buffer the body, this option specify the size of this buffer, default is 8192, if you are sending big file with your client it may be interesting to have a bigger value in order to increase performance.

As an example someone may want to pass a client certificate when using the ssl, a valid configuration for this use case would be:

```php
use Http\Client\Socket\Client;

$options = [
    'stream_context_options' => [
        'ssl' => [
            'local_cert' => '/path/to/my/client-certificate.pem'
        ]
    ]
];
$client = new Client($messageFactory, $options);
```

Warning: This client assumes that the request is compliant with HTTP 2.0, 1.1 or 1.0 standard. So a request without a `Host` header, or with a body but without a `Content-Length` will certainly fail. To make sure all requests will be sent out correctly, we recommend to use the `PluginClient` with the following plugins:

- `ContentLengthPlugin` sets the correct `Content-Length` header, or decorate the stream to use chunked encoding

- `DecoderPlugin` decodes encoding coming from the response (chunked, gzip, deflate and compress)

Read more on plugins

Further reading

- Use *plugins* to customize the way HTTP requests are sent and responses processed by following redirects, adding Authentication or Cookie headers and more.
- Learn how you can decouple your code from any PSR-7 implementation by using the *HTTP factories*.

4.5.12 Zend Adapter (deprecated)

An HTTPlug adapter for the Zend HTTP client.

Zend framework meanwhile has been renamed to Laminas, and the client is no longer maintained.

This adapter only implements the PHP-HTTP synchronous interface. This interface has been superseded by PSR-18, which the Laminas Diactoros implements directly.

Installation

To install the Zend adapter, which will also install Zend itself (if it was not yet included in your project), run:

```
$ composer require php-http/zend-adapter
```

4.5.13 Current Clients and Adapters

Name	Type	Links	Stats
`php-http/curl-client`	Client	*Docs*, Repo	
`php-http/mock-client`	Client	*Docs*, Repo	
`symfony/http-client`	Client	*Docs*, Repo	
`php-http/artax-adapter`	Adapter	*Docs*, Repo	
`php-http/guzzle7-adapter`	Adapter	*Docs*, Repo	
`php-http/react-adapter`	Adapter	*Docs*, Repo	

4.5.14 Legacy Clients and Adapters

These are not maintained anymore, but we keep documentation around for now. Please upgrade your applications to use a maintained client or adapter.

Name	Type	Links	Stats
`php-http/socket-client`	Client	*Docs*, Repo	
`php-http/buzz-adapter`	Adapter	*Docs*, Repo	
`php-http/cakephp-adapter`	Adapter	*Docs*, Repo	
`php-http/guzzle5-adapter`	Adapter	*Docs*, Repo	
`php-http/guzzle6-adapter`	Adapter	*Docs*, Repo	
`php-http/zend-adapter`	Adapter	*Docs*, Repo	

4.5.15 Composer Virtual Packages

Virtual packages are a way to specify the dependency on an implementation of an interface-only repository without forcing a specific implementation. For HTTPlug, the virtual packages are called php-http/client-implementation (though you should be using psr/http-client-implementation to use PSR-18) and php-http/async-client-implementation.

There is no library registered with those names. However, all client implementations (including client adapters) for HTTPlug use the `provide` section to tell composer that they do provide the client-implementation.

4.6 Plugins

The plugin system allows to wrap a Client and add some processing logic prior to and/or after sending the actual request or you can even start a completely new request. This gives you full control over what happens in your workflow.

4.6.1 Introduction

Install

The plugin client and the core plugins are available in the php-http/client-common package:

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

New in version 1.1: The plugins were moved to the clients-common package in version 1.1. If you work with version 1.0, you need to require the separate *php-http/plugins* package and the namespace is `Http\Client\Plugin` instead of `Http\Client\Common`

How it works

In the plugin package, you can find the following content:

- the `PluginClient` itself which acts as a wrapper around any kind of HTTP client (sync/async);
- the `Plugin` interface;
- a set of core plugins (see the full list in the left side navigation).

The `PluginClient` accepts an HTTP client implementation and an array of plugins. Let's see an example:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\RetryPlugin;
use Http\Client\Common\Plugin\RedirectPlugin;

$retryPlugin = new RetryPlugin();
$redirectPlugin = new RedirectPlugin();

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [
        $retryPlugin,
        $redirectPlugin,
    ]
);
```

The `PluginClient` accepts and implements both `Http\Client\HttpClient` and `Http\Client\HttpAsyncClient`, so you can use both ways to send a request. In case the passed client implements only one of these interfaces, the `PluginClient` "emulates" the other behavior as a fallback.

It is important to note that the order of plugins matters. During the request, plugins are executed in the order they have been specified in the constructor, from first to last. Once a response has been received, the plugins are called again in reversed order, from last to first.

For our previous example, the execution chain will look like this:

```
Request  ---> PluginClient ---> RetryPlugin ---> RedirectPlugin ---> HttpClient ----
                                                                                    |␣
↪(processing call)
Response <--- PluginClient <--- RetryPlugin <--- RedirectPlugin <--- HttpClient <---
```

In order to achieve the intended behavior in the global process, you need to pay attention to what each plugin does and define the correct order accordingly.

For example, the `RetryPlugin` should probably be at the end of the chain to keep the retry process as short as possible. However, if one of the other plugins is doing a fragile operation that might need a retry, place the retry plugin before that.

The recommended way to order plugins is the following:

1. Plugins that modify the request should be at the beginning (like Authentication or Cookie Plugin);
2. Plugins which intervene in the workflow should be in the "middle" (like Retry or Redirect Plugin);
3. Plugins which log information should be last (like Logger or History Plugin).

Note: There can be exceptions to these rules. For example, for security reasons you might not want to log the authentication information (like `Authorization` header) and choose to put the *Authentication Plugin* after the *Logger Plugin*.

Configuration Options

The `PluginClient` accepts an array of configuration options to tweak its behavior.

`max_restarts`: int (default 10)

To prevent issues with faulty plugins or endless redirects, the `PluginClient` injects a security check to the start of the plugin chain. If the same request is restarted more than specified by that value, execution is aborted and an error is raised.

`debug_plugins`: array of Plugin

The debug plugins are injected between each normal plugin. This can be used to log the changes each plugin does on the request and response objects.

Libraries that Require Plugins

When *writing a library based on HTTPlug*, you might require specific plugins to be active. The recommended way for doing this is to provide a factory method for the `PluginClient` that library users should use. This allows them to inject their own plugins or configure a different client. For example:

```php
$myApiClient = new My\Api\Client('https://api.example.org', My\Api\
↪HttpClientFactory::create('john', 's3cr3t'));

use Http\Client\HttpClient;
use Http\Client\Common\Plugin;
use Http\Client\Common\Plugin\AuthenticationPlugin;
use Http\Client\Common\Plugin\ErrorPlugin;
use Http\Discovery\HttpClientDiscovery;

class HttpClientFactory
{
    /**
     * Build the HTTP client to talk with the API.
     *
     * @param string     $user    Username for the application on the API
     * @param string     $pass    Password for the application on the API
     * @param Plugin[]   $plugins List of additional plugins to use
     * @param HttpClient $client  Base HTTP client
     *
     * @return HttpClient
     */
    public static function create($user, $pass, array $plugins = [], HttpClient $client␣
↪= null)
    {
        if (!$client) {
            $client = HttpClientDiscovery::find();
        }
        $plugins[] = new ErrorPlugin();
        $plugins[] = new AuthenticationPlugin(
             // This API has it own authentication algorithm
            new ApiAuthentication(Client::AUTH_OAUTH_TOKEN, $user, $pass)
        );
        return new PluginClient($client, $plugins);
    }
}
```

4.6.2 Building Custom Plugins

When writing your own Plugin, you need to be aware that the Plugin Client is async first. This means that every plugin must be written with Promises. More about this later.

Each plugin must implement the `Http\Client\Common\Plugin` interface.

New in version 1.1: The plugins were moved to the *client-common* package in version 1.1. If you work with version 1.0, the interface is called `Http\Client\Plugin\Plugin` and you need to require the separate *php-http/plugins* package. The old interface will keep extending `Http\Client\Common\Plugin`, but relying on it is deprecated.

This interface defines the `handleRequest` method that allows to modify behavior of the call:

```
/**
 * Handles the request and returns the response coming from the next callable.
 *
 * @param RequestInterface $request Request to use.
 * @param callable         $next    Callback to call to have the request, it muse have␣
↪the request as it first argument.
 * @param callable         $first   First element in the plugin chain, used to to␣
↪restart a request from the beginning.
 *
 * @return Promise
 */
public function handleRequest(RequestInterface $request, callable $next, callable
↪$first);
```

The `$request` comes from an upstream plugin or `PluginClient` itself. You can replace it and pass a new version downstream if you need.

The `$next` callable is the next plugin in the execution chain. When you need to call it, you must pass the `$request` as the first argument of this callable.

For example a simple plugin setting a header would look like this:

```
public function handleRequest(RequestInterface $request, callable $next, callable $first)
{
    $newRequest = $request->withHeader('MyHeader', 'MyValue');

    return $next($newRequest);
}
```

The `$first` callable is the first plugin in the chain. It allows you to completely reboot the execution chain, or send another request if needed, while still going through all the defined plugins. Like in case of the `$next` callable, you must pass the `$request` as the first argument:

```
public function handleRequest(RequestInterface $request, callable $next, callable $first)
{
    if ($someCondition) {
        $newRequest = new Request();
        $promise = $first($newRequest);

        // Use the promise to do some jobs ...
    }

    return $next($request);
}
```

Warning: In this example the condition is not superfluous: you need to have some way to not call the `$first` callable each time or you will end up in an infinite execution loop.

The `$next` and `$first` callables will return a *Promise*. You can manipulate the `Psr\Http\Message\ResponseInterface` or the `Http\Client\Exception` by using the `then` method of the promise:

```
public function handleRequest(RequestInterface $request, callable $next, callable $first)
{
    $newRequest = $request->withHeader('MyHeader', 'MyValue');

    return $next($request)->then(function (ResponseInterface $response) {
        return $response->withHeader('MyResponseHeader', 'value');
    }, function (\Http\Client\Exception $exception) {
        echo $exception->getMessage();

        throw $exception;
    });
}
```

Warning: Contract for the `Http\Promise\Promise` is temporary until a PSR is released. Once it is out, we will use this PSR in HTTPlug and deprecate the old contract.

Warning: If a plugin throws an exception that does not implement `Http\Client\Exception` it will break the plugin chain.

To better understand the whole process check existing implementations in the client-common package.

Contributing Your Plugins to PHP-HTTP

We are open to contributions. If the plugin is of general interest, not too complex and does not have dependencies, the best is to do a Pull Request to `php-http/client-common`. Please see the *contribution guide*. We don't promise that every plugin gets merged into the core. We need to keep the core as small as possible with only the most widely used plugins to keep it maintainable.

The alternative is providing your plugins in your own repository. Please let us know when you do, we would like to add a list of existing third party plugins to the list of plugins.

4.6.3 Seekable Body Plugins

`RequestSeekableBodyPlugin` and `ResponseSeekableBodyPlugin` ensure that body used in request and response is always seekable. Use this plugin if you want plugins to read the stream and then be able to rewind it:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\RequestSeekableBodyPlugin;
use Http\Client\Common\Plugin\ResponseSeekableBodyPlugin;

$options = [
    'use_file_buffer' => true,
    'memory_buffer_size' => 2097152,
];
$requestSeekableBodyPlugin = new RequestSeekableBodyPlugin($options);
$responseSeekableBodyPlugin = new ResponseSeekableBodyPlugin($options);
```

(continues on next page)

(continued from previous page)

```
$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$requestSeekableBodyPlugin, $responseSeekableBodyPlugin]
);
```

Those plugins support the following options (which are passed to the `BufferedStream` class):

- `use_file_buffer`: Whether it should use a temporary file to buffer the body of a stream if it's too big
- `memory_buffer_size`: Maximum memory to use for buffering the stream before it switch to a file

`RequestSeekableBodyPlugin` should be the first of your plugins, then the following plugins can seek in the request body (i.e. for logging purpose). `ResponseSeekableBodyPlugin` should be the last plugin, then previous plugins can seek response body.

4.6.4 Authentication Plugin

This plugin uses the *authentication component* from `php-http/message` to authenticate requests sent through the client:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Message\Authentication\BasicAuth;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\AuthenticationPlugin;

$authentication = new BasicAuth('username', 'password');
$authenticationPlugin = new AuthenticationPlugin($authentication);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$authenticationPlugin]
);
```

Check the *authentication component documentation* for the list of available authentication methods.

4.6.5 Cache Plugin

Install

```
$ composer require php-http/cache-plugin
```

Usage

The `CachePlugin` allows you to cache responses from the server. It can use any PSR-6 compatible caching engine. By default, the plugin respects the cache control headers from the server as specified in **RFC 7234**. It needs a PSR-17 StreamFactoryInterface and a PSR-6 implementation:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\CachePlugin;
```

(continues on next page)

(continued from previous page)

```
/** @var \Psr\Cache\CacheItemPoolInterface $pool */
$pool = ...
/** @var \Psr\Http\Message\StreamFactoryInterface $streamFactory */
$streamFactory = ...

$options = [];
$cachePlugin = new CachePlugin($pool, $streamFactory, $options);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$cachePlugin]
);
```

The `CachePlugin` has also 2 factory methods to easily set up the plugin by caching type. See the example below.

```
// This will allow caching responses with the 'private' and/or 'no-store' cache directives
$cachePlugin = CachePlugin::clientCache($pool, $streamFactory, $options);

// Returns a cache plugin with the current default behavior
$cachePlugin = CachePlugin::serverCache($pool, $streamFactory, $options);
```

Note: The two factory methods have been added in version 1.3.0.

Options

The third parameter to the `CachePlugin` constructor takes an array of options. The available options are:

Name	Default value	Description
`default_ttl`	`0`	The default max age of a Response
`respect_cache_hea`	`true`	Whether we should care about cache headers or not * This option is deprecated. Use *respect_response_cache_directives*
`hash_algo`	`sha1`	The hashing algorithm to use when generating cache keys
`cache_lifetime`	30 days	The minimum time we should store a cache item
`methods`	`['GET', 'HEAD']`	Which request methods to cache
`respect_response_`	`['no-cache', 'private', 'max-age', 'no-store']`	A list of cache directives to respect when caching responses
`cache_key_generat`	`new SimpleGenerator()`	A class implementing `CacheKeyGenerator` to generate a PSR-6 cache key.
`cache_listeners`	`[]`	A array of classes implementing `CacheListener` to act on a response with information on its cache status.
`blacklisted_paths`	`[]`	A array of regular expressions to defined paths, that shall not be cached.

Note: A HTTP response may have expired but it is still in cache. If so, headers like `If-Modified-Since` and `If-None-Match` are added to the HTTP request to allow the server answer with 304 status code. When a 304 response is received we update the CacheItem and save it again for at least `cache_lifetime`.

Using these options together you can control how your responses should be cached. By default, responses with no cache control headers are not cached. If you want a default cache lifetime if the server specifies no `max-age`, use:

```
$options = [
    'default_ttl' => 42, // cache lifetime time in seconds
];
```

You can tell the plugin to completely ignore the cache control headers from the server and force caching the response for the default time to live. The options below will cache all responses for one hour:

```
$options = [
    'default_ttl' => 3600, // cache for one hour
    'respect_response_cache_directives' => [],
];
```

Generating a cache key

You may define a method how the PSR-6 cache key should be generated. The default generator is `SimpleGenerator` which is using the request method, URI and body of the request. The cache plugin does also include a `HeaderCacheKeyGenerator` which allow you to specify what HTTP header you want include in the cache key.

Controlling cache listeners

One or more classes implementing `CacheListener` can be added through `cache_listeners`. These classes receive a notification on whether a request was a cache hit or miss, and can optionally mutate the response based on those signals. As an example, adding the provided `AddHeaderCacheListener` will mutate the response, adding an `X-Cache` header with a value `HIT` or `MISS`, which can be useful in debugging.

Semantics of null values

Setting null to the options `cache_lifetime` or `default_ttl` means "Store this as long as you can (forever)". This could be a great thing when you requesting a pay-per-request API (e.g. GoogleTranslate).

Store a response as long the cache implementation allows:

```
$options = [
    'default_ttl' => null,
    'respect_response_cache_directives' => [],
    'cache_lifetime' => null,
];
```

Ask the server if the response is valid at most ever hour. Store the cache item forever:

```
$options = [
    'default_ttl' => 3600,
    'respect_response_cache_directives' => [],
    'cache_lifetime' => null,
];
```

Ask the server if the response is valid at most ever hour. If the response has not been used within one year it will be removed from the cache:

```
$options = [
    'default_ttl' => 3600,
    'respect_response_cache_directives' => [],
    'cache_lifetime' => 86400*365, // one year
];
```

Caching of different request methods

Most of the time you should not change the `methods` option. However if you are working for example with HTTPlug based SOAP client you might want to additionally enable caching of `POST` requests:

```
$options = [
    'methods' => ['GET', 'HEAD', 'POST'],
];
```

The `methods` setting overrides the defaults. If you want to keep caching `GET` and `HEAD` requests, you need to include them. You can specify any uppercase request method which conforms to **RFC 7230**.

Note: If your system has both normal and SOAP clients you need to use two different `PluginClient` instances. SOAP client should use `PluginClient` with POST caching enabled and normal client with POST caching disabled.

Cache Control Handling

By default this plugin does not cache responses with `no-store`, `no-cache` or `private` instructions. Use `CachePlugin::clientCache($pool, $streamFactory, $options);` to cache `no-store` or `private` responses or change the `respect_response_cache_directives` option to your needs.

It does store responses with cookies or a `Set-Cookie` header. Be careful with the order of your plugins.

4.6.6 Content-Length Plugin

The `ContentLengthPlugin` sets the correct `Content-Length` header value based on the size of the body stream of the request. This helps HTTP servers to handle the request:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\ContentLengthPlugin;

$contentLengthPlugin = new ContentLengthPlugin();

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$contentLengthPlugin]
);
```

If the size of the stream can not be determined, the plugin sets the Encoding header to `chunked`, as defined in **RFC 7230#section-4.1**

This is useful when you want to transfer data of unknown size to an HTTP application without consuming memory.

As an example, let's say you want to send a tar archive of the current directory to an API. Normally you would end up doing this in 2 steps, first saving the result of the tar archive into a file or into the memory of PHP with a variable, then sending this content with an HTTP Request.

With this plugin you can achieve this behavior without doing the first step:

```
proc_open("/usr/bin/env tar c .", [["pipe", "r"], ["pipe", "w"], ["pipe", "w"]], $pipes,
↪"/path/to/directory");
$tarResource  = $pipes[1];

$request = MessageFactoryDiscovery::find()->createRequest('POST', '/url/to/api/endpoint',
↪ [], $tarResource);
$response = $pluginClient->sendRequest($request);
```

In this case the tar output is directly streamed to the server without using memory on the PHP side.

4.6.7 Content-Type Plugin

The `ContentTypePlugin` sets the correct `Content-Type` header value based on the content of the body stream of the request. This helps HTTP servers to handle the request:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\ContentTypePlugin;

$contentTypePlugin = new ContentTypePlugin();

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$contentTypePlugin]
);
```

For now, the plugin can only detect JSON or XML content. If the content of the stream can not be determined, the plugin does nothing.

Options

`skip_detection`: boolean (default: false)

When set to `true`, content type detection will be performed only if the body request content size is under the size_limit parameter value.

`size_limit`: int (default: a little bit over 15Mb)

Determine the size stream limit for which the detection as to be skipped if `skip_detection` is `true`.

4.6.8 Cookie Plugin

The CookiePlugin allow you to store cookies in a CookieJar and reuse them on consequent requests according to **RFC 6265#section-4** specification:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Message\CookieJar;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\CookiePlugin;

$cookiePlugin = new CookiePlugin(new CookieJar());

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$cookiePlugin]
);
```

4.6.9 Decoder Plugin

The DecoderPlugin decodes the body of the response with filters coming from the Transfer-Encoding or Content-Encoding headers:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\DecoderPlugin;

$decoderPlugin = new DecoderPlugin();

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$decoderPlugin]
);
```

The plugin can handle the following encodings:

- chunked: Decode a stream with a chunked encoding (no size in the Content-Length header of the response)
- compress: Decode a stream encoded with the compress encoding according to **RFC 1950**
- deflate: Decode a stream encoded with the inflate encoding according to **RFC 1951**
- gzip: Decode a stream encoded with the gzip encoding according to **RFC 1952**

You can also use the decoder plugin to decode only the Transfer-Encoding header and not the Content-Encoding one by setting the use_content_encoding configuration option to false:

```
$decoderPlugin = new DecoderPlugin(['use_content_encoding' => false]);
```

Not decoding content is useful when you don't want to get the encoded response body, or acting as a proxy but still be able to decode message from the Transfer-Encoding header value.

4.6.10 Error Plugin

The `ErrorPlugin` transforms responses with HTTP error status codes into exceptions:

- 400-499 status code are transformed into `Http\Client\Common\Exception\ClientErrorException`;
- 500-599 status code are transformed into `Http\Client\Common\Exception\ServerErrorException`

> **Warning:** Throwing an exception on a valid response violates the PSR-18 specification. This plugin is provided as a convenience when writing a small application. When providing a client to a third party library, this plugin must not be included, or the third party library will have problems with error handling.

Both exceptions extend the `Http\Client\Exception\HttpException` class, so you can fetch the request and the response coming from them:

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\ErrorPlugin;
use Http\Client\Common\Exception\ClientErrorException;

$errorPlugin = new ErrorPlugin();

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$errorPlugin]
);

...

try {
    $response = $pluginClient->sendRequest($request);
} catch (ClientErrorException $e) {
    if ($e->getResponse()->getStatusCode() == 404) {
        // Something has not been found
    }
}
```

The error plugin is intended for when an application operates with the client directly. When writing a library around an API, the best practice is to have the client convert responses into domain objects, and transform HTTP errors into meaningful domain exceptions. In that scenario, the ErrorPlugin is not needed. It is more efficient to check the HTTP status codes yourself than throwing and catching exceptions.

If your application handles responses with 4xx status codes, but needs exceptions for 5xx status codes only, you can set the option `only_server_exception` to `true`:

```php
$errorPlugin = new ErrorPlugin(['only_server_exception' => true]);
```

4.6.11 Header Plugins

Header plugins are useful to manage request headers. Many operations are possible with the provided plugins.

Default Headers Values

The plugin `HeaderDefaultsPlugin` allows you to define default values for given headers. If a header is not set, it will be added. However, if the header is already present, the request is left unchanged:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\HeaderDefaultsPlugin;

$defaultUserAgent = 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/
↪40.1';

$headerDefaultsPlugin = new HeaderDefaultsPlugin([
    'User-Agent' => $defaultUserAgent
]);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$headerDefaultsPlugin]
);
```

Mandatory Headers Values

The plugin `HeaderSetPlugin` allows you to fix values of given headers. That means that any request passing through this plugin will be set to the specified value. Existing values of the header will be overwritten.

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\HeaderSetPlugin;

$userAgent = 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/40.1';

$headerSetPlugin = new HeaderSetPlugin([
    'User-Agent' => $userAgent,
    'Accept' => 'application/json'
]);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$headerSetPlugin]
);
```

Removing Headers

The plugin `HeaderRemovePlugin` allows you to remove headers from the request.

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\HeaderRemovePlugin;

$headerRemovePlugin = new HeaderRemovePlugin([
    'User-Agent'
]);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$headerRemovePlugin]
);
```

Appending Header Values

The plugin `HeaderAppendPlugin` allows you to add headers. The header will be created if not existing yet. If the header already exists, the value will be appended to the list of values for this header.

Note: The use cases for this plugin are limited. One real world example of headers that can have multiple values is "Forwarded".

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\HeaderAppendPlugin;

$myIp = '100.100.100.100';

$headerAppendPlugin = new HeaderAppendPlugin([
    'Forwarded' => 'for=' . $myIp
]);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$headerAppendPlugin]
);
```

Mixing operations

Different header plugins can be mixed together to achieve different behaviors and you can use the same plugin for identical operations.

The following example will force the `User-Agent` and the `Accept` header values while removing the `Cookie` header:

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\HeaderSetPlugin;
```

(continues on next page)

(continued from previous page)

```php
use Http\Client\Common\Plugin\HeaderRemovePlugin;

$userAgent = 'Mozilla/5.0 (Windows NT 6.1; WOW64; rv:40.0) Gecko/20100101 Firefox/40.1';

$headerSetPlugin = new HeaderSetPlugin([
    'User-Agent' => $userAgent,
    'Accept' => 'application/json'
]);

$headerRemovePlugin = new HeaderRemovePlugin([
    'Cookie'
]);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [
        $headerSetPlugin,
        $headerRemovePlugin
    ]
);
```

4.6.12 History Plugin

The `HistoryPlugin` notifies a `Http\Client\Common\Plugin\Journal` of all successful and failed calls:

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\HistoryPlugin;

$historyPlugin = new HistoryPlugin(new \My\Journal\Implementation());

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$historyPlugin]
);
```

As an example, HttplugBundle uses this plugin to collect responses and exceptions associated with requests for the debug toolbar.

This plugin only collects data after resolution. For logging purposes, it is recommended to use the *LoggerPlugin* instead, which logs events as they occur.

4.6.13 Logger Plugin

Install

```
$ composer require php-http/logger-plugin
```

Usage

The `LoggerPlugin` converts requests, responses and exceptions to strings and logs them with a PSR3 compliant logger:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\LoggerPlugin;
use Monolog\Logger;

$loggerPlugin = new LoggerPlugin(new Logger('http'));

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$loggerPlugin]
);
```

The log level for exceptions is `error`, the request and responses without exceptions are logged at level `info`. Request and response/errors can be correlated by looking at the `uid` of the log context. If you don't want to normally log requests, you can set the logger to normally only log `error` but use the `Fingerscrossed` logger of Monolog to also log the request in case an exception is encountered.

By default it uses `Http\Message\Formatter\SimpleFormatter` to format the request or the response into a string. You can use any formatter implementing the `Http\Message\Formatter` interface:

```
$formatter = new \My\Formatter\Implementation();

$loggerPlugin = new LoggerPlugin(new Logger('http'), $formatter);
```

4.6.14 Query plugin

Default Query parameters

The plugin `QueryDefaultsPlugin` allows you to define default values for query parameters. If a query parameter is not set, it will be added. However, if the query parameter is already present, the request is left unchanged. Names and values must not be URL encoded as this plugin will encode them:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\QueryDefaultsPlugin;

$queryDefaultsPlugin = new QueryDefaultsPlugin([
    'locale' => 'en'
]);

$pluginClient = new PluginClient(
```

(continues on next page)

(continued from previous page)

```
    HttpClientDiscovery::find(),
    [$queryDefaultsPlugin]
);
```

4.6.15 Redirect Plugin

The `RedirectPlugin` automatically follows redirection answers from a server. If the plugin detects a redirection, it creates a request to the target URL and restarts the plugin chain.

The plugin attempts to detect circular redirects and will abort when such a redirect is encountered. Note that a faulty server appending something on each request is not detected. This situation is caught by the plugin client itself and can be controlled through the *max_restarts: int (default 10)* setting.

Initiate the redirect plugin as follows:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\RedirectPlugin;

$redirectPlugin = new RedirectPlugin();

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$redirectPlugin]
);
```

Warning: Following redirects can increase the robustness of your application. But if you build some sort of API client, you want to at least keep an eye on the log files. Having your application follow redirects instead of going to the right end point directly makes your application slower and increases the load on both server and client.

Note: Depending on the status code, redirecting should change POST/PUT requests to GET requests. This plugin implements this behavior - except if you set the `strict` option to true, as explained below. It removes the request body if the method changes, see `stream_factory` below.

To understand the exact semantics of which HTTP status changes the method and which not, have a look at the configuration in the source code of the RedirectPlugin class.

Options

`preserve_header`: boolean|string[] (default: true)

When set to `true`, all headers are kept for the next request. `false` means all headers are removed. An array of strings is treated as a whitelist of header names to keep from the original request.

`use_default_for_multiple`: bool (default: true)

Whether to follow the default direction on the multiple redirection status code 300. If set to false, a status of 300 will raise the `Http\Client\Common\Exception\MultipleRedirectionException`.

`strict`: bool (default: false)

When set to `true`, 300, 301 and 302 status codes will not modify original request's method and body on consecutive requests. E. g. POST redirect requests are sent as POST requests instead of POST redirect requests are sent as GET requests.

`stream_factory`: StreamFactoryInterface (default: auto discovered)

The PSR-17 stream factory is used to create an empty stream for removing the body of the request on redirection. To keep the body on all redirections, set `stream_factory` to null. The stream factory is discovered if either `php-http/discovery` is installed and provides a factory, or `nyholm/psr7` or a new enough version of `guzzlehttp/psr7` are installed. If you only have other implementations, you need to provide the factory in `stream_factory`.

If no factory is found, the redirect plugin does not remove the body on redirection.

4.6.16 Request URI Manipulations

Request URI manipulations can be done thanks to several plugins:

- `AddHostPlugin`: Set host, scheme and port. Depending on configuration, the host is overwritten in every request or only set if not yet defined in the request.
- `AddPathPlugin`: Prefix the request path with a path, leaving the host information untouched.
- `BaseUriPlugin`: It's a combination of `AddHostPlugin` and `AddPathPlugin`.

Each plugin uses the `UriInterface` to build the base request:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Discovery\UriFactoryDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\BaseUriPlugin;

$plugin = new BaseUriPlugin(UriFactoryDiscovery::find()->createUri('https://domain.
↪com:8000/api'), [
    // Always replace the host, even if this one is provided on the sent request.␣
↪Available for AddHostPlugin.
    'replace' => true,
]);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$plugin]
);
```

The `AddPathPlugin` will check if the path prefix is already present on the URI. This will break for the edge case when the prefix is repeated. For example, if `https://example.com/api/api/foo` is a valid URI on the server and the configured prefix is `/api`, the request to `/api/foo` is not rewritten.

For further details, please see the phpdoc on the `AddPathPlugin` source code.

No solution fits all use cases. This implementation works fine for the common use cases. If you have a specific situation where this is not the right thing, you can build a custom plugin that does exactly what you need.

4.6.17 Retry Plugin

The `RetryPlugin` can automatically attempt to re-send a request that failed, to work around unreliable connections and servers. It re-sends the request when an exception is thrown, unless the exception is a HttpException for a status code in the 5xx server error range. Since version 2.0, responses with status codes in the 5xx range are also retried. Each retry attempt is delayed by an exponential backoff time.

See below for how to configure that behavior.

Warning: You should keep an eye on retried requests, as they add overhead. If a request fails due to a client side mistake, retrying is only a waste of time and resources.

Contrary to the *Redirect Plugin*, the retry plugin does not restart the chain but simply tries again from the current position.

Async

This plugin is not fully compatible with asynchronous behavior, as the wait between retries is done with a blocking call to a sleep function.

Options

`retries`: int (default: 1)

Number of retry attempts to make before giving up.

`error_response_decider`: callable (default behavior: retry if status code is in 5xx range)

A callback function that receives the request and response to decide whether the request should be retried.

`exception_decider`: callable (default behavior: retry if the exception is not an HttpException or status code is in 5xx range)

A callback function that receives a request and an exception to decide after a failure whether the request should be retried.

`error_response_delay`: callable (default behavior: exponential backoff)

A callback that receives a request, a response, the current number of retries and returns how many microseconds we should wait before trying again.

`exception_delay`: callable (default behavior: exponential backoff)

A callback that receives a request, an exception, the current number of retries and returns how many microseconds we should wait before trying again.

Interaction with Exceptions

If you use the *ErrorPlugin*, you should place it after the RetryPlugin in the plugin chain:

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\ErrorPlugin;
use Http\Client\Common\Plugin\RetryPlugin;

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [
        new RetryPlugin(),
        new ErrorPlugin(),
    ]
);
```

4.6.18 Stopwatch Plugin

Install

```
$ composer require php-http/stopwatch-plugin
```

Usage

The `StopwatchPlugin` records the duration of HTTP requests with a `Symfony\Component\Stopwatch\Stopwatch` instance:

```php
use Http\Discovery\HttpClientDiscovery;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\StopwatchPlugin;
use Symfony\Component\Stopwatch\Stopwatch;

$stopwatch = new Stopwatch();

$stopwatchPlugin = new StopwatchPlugin($stopwatch);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$stopwatchPlugin]
);

// ...

foreach ($stopwatch->getSections() as $section) {
    foreach ($section->getEvents() as $name => $event) {
        echo sprintf('Request %s took %s ms and used %s bytes of memory', $name, $event->
↪getDuration(), $event->getMemory());
    }
}
```

Warning: The results of the stop watch will be unreliable when using an asynchronous client. Execution time can be longer than it really was, depending on when the status was checked again, and memory consumption will be mixed up with other code that was executed while waiting for the response.

4.6.19 VCR Plugin - Record and Replay Responses

The VCR plugins allow you to record & replay HTTP responses. It's very useful for test purpose (using production-like predictable fixtures and avoid making actual HTTP request). You can also use it during your development cycle, when the endpoint you're contacting is not ready yet.

Unlike the *php-http/mock-client*, where you have to manually define responses, the responses are **automatically** generated from the previously recorded ones.

Install

```
$ composer require --dev php-http/vcr-plugin
```

Usage

To record or replay a response, you will need two components, a **naming strategy** and a **recorder**.

The naming strategy

The naming strategy turn a request into a deterministic and unique identifier. The identifier must be safe to use with a file system. The plugin provide a default naming strategy, the `PathNamingStrategy`. You can define two options:

- **hash_headers**: the list of header(s) that make the request unique (Ex: 'Authorization'). The name & content of the header will be hashed to generate a unique signature. By default no header is used.
- **hash_body_methods**: indicate for which request methods the body makes requests distinct. (Default: PUT, POST, PATCH)

This naming strategy will turn a GET request to https://example.org/my-path to the `example.org_GET_my-path` name, and optionally add hashes if the request contain a header defined in the options, or if the method is not idempotent.

To create your own strategy, you need to create a class implementing `Http\Client\Plugin\Vcr\NamingStrategy\NamingStrategyInterface`.

The recorder

The recorder records and replays responses. The plugin provides two recorders:

- `FilesystemRecorder`: Saves the response on your file system using Symfony's filesystem component and Guzzle PSR7 library.
- `InMemoryRecorder`: Saves the response in memory. **Response will be lost at the end of the running process**

To create your own recorder, you need to create a class implementing the following interfaces:

- `Http\Client\Plugin\Vcr\Recorder\RecorderInterface` used by the RecordPlugin.
- `Http\Client\Plugin\Vcr\Recorder\PlayerInterface` used by the ReplayPlugin.

The plugins

There are two plugins, one to record responses, the other to replay them.

- `Http\Client\Plugin\Vcr\ReplayPlugin`, use a `PlayerInterface` to replay previously recorded responses.
- `Http\Client\Plugin\Vcr\RecordPlugin`, use a `RecorderInterface` instance to record the responses,

Both plugins add a response header to indicate either under which name the response has been stored (RecordPlugin, `X-VCR-RECORD` header), or which response name has been used to replay the request (ReplayPlugin, `X-VCR-REPLAYED` header).

If you plan on using both plugins at the same time (Replay or Record), the `ReplayPlugin` **must always** come first. Please also note that by default, the `ReplayPlugin` throws an exception when it cannot replay a request. If you want the plugin to continue the request (possibly to the actual server), set the third constructor argument to `false` (See example below).

Example

```php
<?php

use Http\Client\Common\PluginClient;
use Http\Client\Plugin\Vcr\NamingStrategy\PathNamingStrategy;
use Http\Client\Plugin\Vcr\Recorder\FilesystemRecorder;
use Http\Client\Plugin\Vcr\RecordPlugin;
use Http\Client\Plugin\Vcr\ReplayPlugin;
use Http\Discovery\HttpClientDiscovery;

$namingStrategy = new PathNamingStrategy([
    'hash_headers' => ['X-Custom-Header'], // None by default
    'hash_body_methods' => ['POST'], // Default: PUT, POST, PATCH
]);
$recorder = new FilesystemRecorder('some/dir/in/vcs'); // You can use InMemoryRecorder␣
↪as well

// To record responses:
$record = new RecordPlugin($namingStrategy, $recorder);

// To replay responses:
// Third argument prevent the plugin from throwing an exception when a request cannot be␣
↪replayed
$replay = new ReplayPlugin($namingStrategy, $recorder, false);

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [$replay, $record] // Replay should always go first
);

/** @var \Psr\Http\Message\RequestInterface $request */
$request = new MyRequest('GET', 'https://httplug.io');

// Will be recorded in "some/dir/in/vcs"
$client->sendRequest($request);
```

(continues on next page)

(continued from previous page)

```
// Will be replayed from "some/dir/in/vcs"
$client->sendRequest($request);
```

4.7 Framework Integrations

HTTPlug provides the following framework integrations:

4.7.1 Symfony Bundle

This bundle integrates HTTPlug with the Symfony framework. The bundle helps to register services for all your clients and makes sure all the configuration is in one place. The bundle also features a profiling plugin with information about your requests.

This guide explains how to configure HTTPlug in the Symfony framework. See the *HTTPlug Tutorial* for examples how to use HTTPlug in general.

Installation

HTTPlug works with any HTTP client implementation that provides PSR-18 or a HTTPlug adapter. The flex recipe installs the *php-http* curl client. See *Clients & Adapters* for a list of clients known to work with the bundle.

You can find all available configuration at the *full configuration* page.

Using Symfony Flex

HttplugBundle has a Symfony Flex recipe that will set it up with default configuration:

```
$ composer require php-http/httplug-bundle
```

Without Symfony Flex

Install the HTTPlug bundle with composer and enable it in your AppKernel.php.

```
$ composer require php-http/httplug-bundle [some-adapter?]
```

If you already added the HTTPlug client requirement to your project, then you only need to add `php-http/httplug-bundle`. Otherwise, you also need to specify an HTTP client to use - see *Clients & Adapters* for a list of available clients.

Activate Bundle in Symfony 4 and newer

```
 // config/bundles.php
 return [
     ...
     Http\HttplugBundle\HttplugBundle::class => ['all' => true],
];
```

Activate Bundle in Symfony 3

```
// app/AppKernel.php
public function registerBundles()
{
    $bundles = [
        // ...
        new Http\HttplugBundle\HttplugBundle(),
    ];
}
```

Usage

```
httplug:
    plugins:
        logger: ~
    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            plugins: ['httplug.plugin.logger']
            config:
                timeout: 2
```

```
$request = $this->container->get('httplug.psr17_request_factory')->createRequest('GET',
↪'http://example.com');
$response = $this->container->get('httplug.client.acme')->sendRequest($request);
```

Autowiring

The first configured client is considered the "default" client. The default clients are available for autowiring: The PSR-18 `Psr\Http\Client\ClientInterface` and the `Http\Client\HttpAsyncClient`. Autowiring can be convenient to build your application.

However, if you configured several different clients and need to be sure that the correct client is used in each service, it can also hide mistakes. Therefore you can disable autowiring with a configuration option:

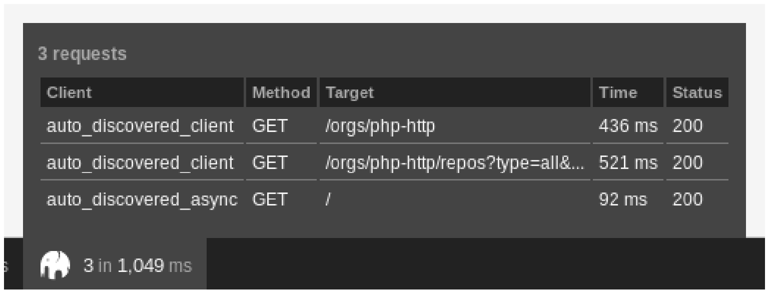
```
httplug:
    default_client_autowiring: false
```

When using this bundle with Symfony 5.3 or newer, you can use the Symfony *#[Target]* attribute to select a client by name. For a client configured as `httplug.clients.acme`, this would be:

```php
use Psr\Http\Client\ClientInterface;
use Symfony\Component\DependencyInjection\Attribute as DI;

final class MyService
{
    public function __construct(
        #[DI\Target('acme')] ClientInterface $client
    ) {}
}
```

Web Debug Toolbar



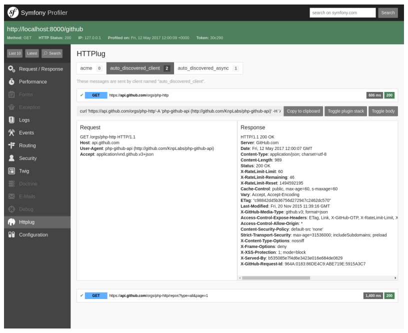
3 requests

Client	Method	Target	Time	Status
auto_discovered_client	GET	/orgs/php-http	436 ms	200
auto_discovered_client	GET	/orgs/php-http/repos?type=all&...	521 ms	200
auto_discovered_async	GET	/	92 ms	200

3 in 1,049 ms

When using a client configured with `HttplugBundle`, you will get debug information in the web debug toolbar. It will tell you how many request were made and how many of those that were successful or not. It will also show you detailed information about each request.

The web profiler page will show you lots of information about the request and also how different plugins changes the message. See example screen shots below.



http://localhost:8000/github

Method: GET **HTTP Status:** 200 **IP:** 127.0.0.1 **Profiled on:** Fri, 12 May 2017 12:00:09 +0000 **Token:** 30c290

HTTPlug

acme 0 | auto_discovered_client 2 | auto_discovered_async 1

These messages are sent by client named "auto_discovered_client".

✓ GET https://api.github.com/orgs/php-http 606 ms 200

```
curl 'https://api.github.com/orgs/php-http'-A 'php-github-api (http://github.com/KnpLabs/php-github-api)' -H '/
```

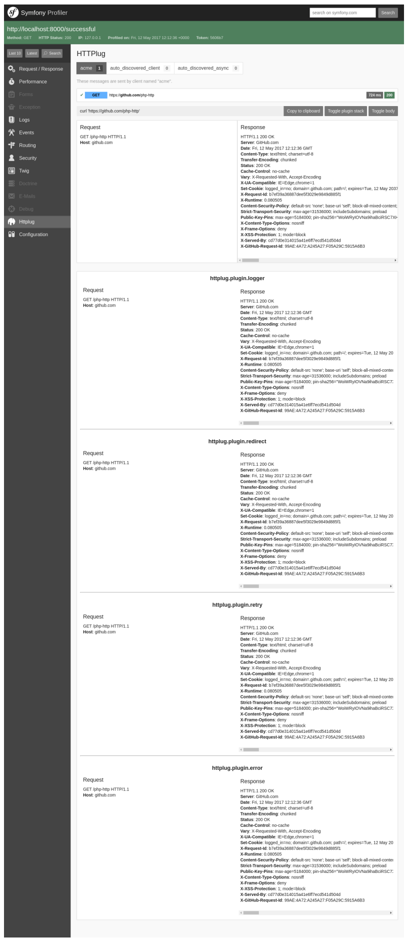
Request

```
GET /orgs/php-http HTTP/1.1
Host: api.github.com
User-Agent: php-github-api (http://github.com/KnpLabs/php-github-api)
Accept: application/vnd.github.v3+json
```

Response

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Fri, 12 May 2017 12:00:07 GMT
Content-Type: application/json; charset=utf-8
Content-Length: 989
Status: 200 OK
X-RateLimit-Limit: 60
X-RateLimit-Remaining: 46
X-RateLimit-Reset: 1494592195
Cache-Control: public, max-age=60, s-maxage=60
Vary: Accept, Accept-Encoding
ETag: "c98842d45b36756d272947c2462dc570"
Last-Modified: Fri, 20 Nov 2015 11:39:16 GMT
X-GitHub-Media-Type: github.v3; format=json
Access-Control-Expose-Headers: ETag, Link, X-GitHub-OTP, X-RateLimit-Limit, X
Access-Control-Allow-Origin: *
Content-Security-Policy: default-src 'none'
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: b535085e7f4d6e3423e016e684de0829
X-GitHub-Request-Id: 964A:0183:86DE4C9:ABE719E:5915A3C7
```

✓ GET https://api.github.com/orgs/php-http/repos?type=all&page=1 1,400 ms 200



http://localhost:8000/successful

Method: GET **HTTP Status:** 200 **IP:** 127.0.0.1 **Profiled on:** Fri, 12 May 2017 12:12:36 +0000 **Token:** 5606b7

HTTPlug

acme 1 | auto_discovered_client 0 | auto_discovered_async 0

These messages are sent by client named "acme".

✓ GET https://github.com/php-http 724 ms 200

```
curl 'https://github.com/php-http'
```

Request

```
GET /php-http HTTP/1.1
Host: github.com
```

Response

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Fri, 12 May 2017 12:12:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Status: 200 OK
Cache-Control: no-cache
Vary: X-Requested-With, Accept-Encoding
X-UA-Compatible: IE=Edge,chrome=1
Set-Cookie: logged_in=no; domain=.github.com; path=/; expires=Tue, 12 May 2037
X-Request-Id: b7ef39a36887dee5f3029e9849d885f1
X-Runtime: 0.080505
Content-Security-Policy: default-src 'none'; base-uri 'self'; block-all-mixed-content;
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Public-Key-Pins: max-age=5184000; pin-sha256="WoiWRyIOVNa9ihaBciRSC7XH
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: cd77d0e314015a41e6ff7ecd541d504d
X-GitHub-Request-Id: 99AE:4A72:A245A27:F05A29C:5915A6B3
```

httplug.plugin.logger

Request

```
GET /php-http HTTP/1.1
Host: github.com
```

Response

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Fri, 12 May 2017 12:12:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Status: 200 OK
Cache-Control: no-cache
Vary: X-Requested-With, Accept-Encoding
X-UA-Compatible: IE=Edge,chrome=1
Set-Cookie: logged_in=no; domain=.github.com; path=/; expires=Tue, 12 May 20
X-Request-Id: b7ef39a36887dee5f3029e9849d885f1
X-Runtime: 0.080505
Content-Security-Policy: default-src 'none'; base-uri 'self'; block-all-mixed-conte
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Public-Key-Pins: max-age=5184000; pin-sha256="WoiWRyIOVNa9ihaBciRSC7:
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: cd77d0e314015a41e6ff7ecd541d504d
X-GitHub-Request-Id: 99AE:4A72:A245A27:F05A29C:5915A6B3
```

httplug.plugin.redirect

Request

```
GET /php-http HTTP/1.1
Host: github.com
```

Response

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Fri, 12 May 2017 12:12:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Status: 200 OK
Cache-Control: no-cache
Vary: X-Requested-With, Accept-Encoding
X-UA-Compatible: IE=Edge,chrome=1
Set-Cookie: logged_in=no; domain=.github.com; path=/; expires=Tue, 12 May 20
X-Request-Id: b7ef39a36887dee5f3029e9849d885f1
X-Runtime: 0.080505
Content-Security-Policy: default-src 'none'; base-uri 'self'; block-all-mixed-conte
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Public-Key-Pins: max-age=5184000; pin-sha256="WoiWRyIOVNa9ihaBciRSC7:
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: cd77d0e314015a41e6ff7ecd541d504d
X-GitHub-Request-Id: 99AE:4A72:A245A27:F05A29C:5915A6B3
```

httplug.plugin.retry

Request

```
GET /php-http HTTP/1.1
Host: github.com
```

Response

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Fri, 12 May 2017 12:12:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Status: 200 OK
Cache-Control: no-cache
Vary: X-Requested-With, Accept-Encoding
X-UA-Compatible: IE=Edge,chrome=1
Set-Cookie: logged_in=no; domain=.github.com; path=/; expires=Tue, 12 May 20
X-Request-Id: b7ef39a36887dee5f3029e9849d885f1
X-Runtime: 0.080505
Content-Security-Policy: default-src 'none'; base-uri 'self'; block-all-mixed-conte
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Public-Key-Pins: max-age=5184000; pin-sha256="WoiWRyIOVNa9ihaBciRSC7:
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: cd77d0e314015a41e6ff7ecd541d504d
X-GitHub-Request-Id: 99AE:4A72:A245A27:F05A29C:5915A6B3
```

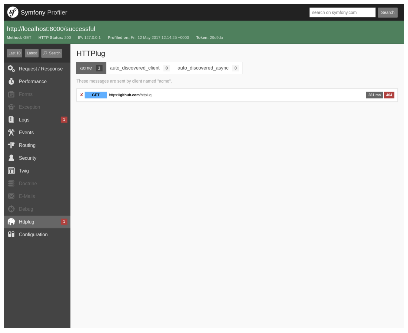
httplug.plugin.error

Request

```
GET /php-http HTTP/1.1
Host: github.com
```

Response

```
HTTP/1.1 200 OK
Server: GitHub.com
Date: Fri, 12 May 2017 12:12:36 GMT
Content-Type: text/html; charset=utf-8
Transfer-Encoding: chunked
Status: 200 OK
Cache-Control: no-cache
Vary: X-Requested-With, Accept-Encoding
X-UA-Compatible: IE=Edge,chrome=1
Set-Cookie: logged_in=no; domain=.github.com; path=/; expires=Tue, 12 May 20
X-Request-Id: b7ef39a36887dee5f3029e9849d885f1
X-Runtime: 0.080505
Content-Security-Policy: default-src 'none'; base-uri 'self'; block-all-mixed-conte
Strict-Transport-Security: max-age=31536000; includeSubdomains; preload
Public-Key-Pins: max-age=5184000; pin-sha256="WoiWRyIOVNa9ihaBciRSC7:
X-Content-Type-Options: nosniff
X-Frame-Options: deny
X-XSS-Protection: 1; mode=block
X-Served-By: cd77d0e314015a41e6ff7ecd541d504d
X-GitHub-Request-Id: 99AE:4A72:A245A27:F05A29C:5915A6B3
```



http://localhost:8000/successful

Method: GET **HTTP Status:** 200 **IP:** 127.0.0.1 **Profiled on:** Fri, 12 May 2017 12:14:25 +0000 **Token:** 29d9da

HTTPlug

acme 1 | auto_discovered_client 0 | auto_discovered_async 0

These messages are sent by client named "acme".

✗ GET https://github.com/httplug 381 ms 404

The body of the HTTP messages is not captured by default because of performance reasons. Turn this on by changing the `captured_body_length` configuration.

```yaml
httplug:
    profiling:
        captured_body_length: 1000 # Capture the first 1000 chars of the HTTP body
```

You can set `captured_body_length` to `null` to avoid body limitation size.

```yaml
httplug:
    profiling:
        captured_body_length: ~ # Avoid truncation of body content
```

The profiling is automatically turned off when `kernel.debug = false`. You can also disable the profiling by configuration.

```yaml
httplug:
    profiling: false
```

You can configure the bundle to show debug information for clients found with discovery. You may also force a specific client to be found when a third party library is using discovery. The configuration below makes sure the client with service id `httplug.client.my_guzzle7` is returned when calling `Psr18ClientDiscovery::find()` . It does also make sure to show debug info for asynchronous clients.

Note: Ideally, you would always use dependency injection and never rely on auto discovery to find a client.

```yaml
httplug:
    clients:
        my_guzzle7:
            factory: 'httplug.factory.guzzle7'
    discovery:
        client: 'httplug.client.my_guzzle7'
        async_client: 'auto'
```

For normal clients, the auto discovery debug info is enabled by default. For async clients, debug is not enabled by default to avoid errors when using the bundle with a client that can not do async. To get debug information for async clients, set `discovery.async_client` to `'auto'` or an explicit client.

You can turn off all interaction of the bundle with auto discovery by setting the value of `discovery.client` to `false`.

Discovery of Factory Classes

You need to specify all the factory classes for you client. The following example shows how you configure factory classes using Guzzle:

```yaml
httplug:
    classes:
        client: Http\Adapter\Guzzle7\Client
        psr17_request_factory: GuzzleHttp\Psr7\HttpFactory
        psr17_response_factory: GuzzleHttp\Psr7\HttpFactory
        psr17_uri_factory: GuzzleHttp\Psr7\HttpFactory
        psr17_stream_factory: GuzzleHttp\Psr7\HttpFactory
```

Configure Clients

You can configure your clients with default options. These default values will be specific to you client you are using. The clients are later registered as services.

```yaml
httplug:
    clients:
        my_guzzle7:
            factory: 'httplug.factory.guzzle7'
            config:
                # These options are given to Guzzle without validation.
```

(continues on next page)

(continued from previous page)

```yaml
                defaults:
                    # timeout if connection is not established after 4 seconds
                    timeout: 4
        acme:
            factory: 'httplug.factory.curl'
            config:
                # timeout if connection is not established after 4 seconds
                CURLOPT_CONNECTTIMEOUT: 4
                # throttle sending data if more than ~ 1MB / second
                CURLOPT_MAX_SEND_SPEED_LARGE: 1000000
```

```php
$httpClient = $this->container->get('httplug.client.my_guzzle7');
$httpClient = $this->container->get('httplug.client.acme');

// will be the same as ``httplug.client.my_guzzle7``
$httpClient = $this->container->get('httplug.client');
```

The bundle has client factory services that you can use to build your client. If you need a very custom made client you could create your own factory service implementing `Http\HttplugBundle\ClientFactory\ClientFactory`. The built-in services are:

- `httplug.factory.curl`
- `httplug.factory.buzz`
- `httplug.factory.guzzle6`
- `httplug.factory.guzzle7`
- `httplug.factory.react`
- `httplug.factory.socket`
- `httplug.factory.symfony`
- `httplug.factory.mock` (Install `php-http/mock-client` first)

Note: New in version 1.10: If you already have a client service registered you can skip using the `factory` and use the `service` key instead.

```yaml
httplug:
    clients:
        my_client:
            service: 'my_custom_client_service'
```

New in version 1.17: All configured clients are tagged with `'httplug.client'` (the value of the constant `Http\HttplugBundle\DependencyInjection\HttplugExtension::HTTPLUG_CLIENT_TAG`), so they can be easily retrieved. This is useful for functional tests, where one might want to replace every configured client with a mock client, so they can be retrieved and configured later

```php
use Http\HttplugBundle\DependencyInjection\HttplugExtension;
use Http\Mock\Client;
use Symfony\Component\DependencyInjection\ContainerBuilder;

/** @var ContainerBuilder $container */
$serviceIds = array_keys($container->findTaggedServiceIds(HttplugExtension::HTTPLUG_
```

(continues on next page)

(continued from previous page)

```
↪CLIENT_TAG));

foreach ($serviceIds as $serviceId) {
    $decoratingServiceId = \sprintf(
        '%s.mock',
        $serviceId
    );

    $container->register($decoratingServiceId, Client::class)
        ->setDecoratedService($serviceId)
        ->setPublic(true);
}
```

Plugins

Clients can have plugins that act on the request before it is sent out and/or on the response before it is returned to the caller. Generic plugins from `php-http/client-common` (e.g. retry or redirect) can be configured globally. You can tell the client which of those plugins to use, as well as specify the service names of custom plugins that you want to use.

Additionally you can configure any of the `php-http/plugins` specifically on a client. For some plugins this is the only place where they can be configured. The order in which you specify the plugins **does** matter.

See *the plugin documentation* for more information on the plugins.

See *full configuration* for the full list of plugins you can configure through this bundle. If a plugin is not available in the configuration, you can configure it as a service and reference the plugin by service id as you would do for a *custom plugin*.

You can configure many of the plugins directly on the client:

```
// config.yml
httplug:
    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            plugins:
                - error:
                    only_server_exception: true
                - add_host:
                    host: "http://localhost:8000"
                - header_defaults:
                    headers:
                        "X-FOO": bar
                - authentication:
                    acme_basic:
                        type: 'basic'
                        username: 'my_username'
                        password: 'p4ssw0rd'
```

Alternatively, the same configuration also works on a global level. With this, you can configure plugins once and then use them in several clients. The plugin service names follow the pattern `httplug.plugin.<name>`:

```
// config.yml
httplug:
    plugins:
        cache:
            cache_pool: 'my_cache_pool'
    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            plugins:
                - 'httplug.plugin.cache'
        app:
            plugins:
                - 'httplug.plugin.cache'
```

Note: To configure HTTP caching, you need to require `php-http/cache-plugin` in your project. It is available as a separate composer package.

Configure a Custom Plugin

To use a custom plugin or when you need specific configuration that is not covered by the bundle configuration, you can configure the plugin as a normal Symfony service and then reference that service name in the plugin list of your client:

```
// services.yml
acme_plugin:
    class: Acme\Plugin\MyCustomPlugin
    arguments: ["%some_parameter%"]
```

```
// config.yml
httplug:
    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            plugins:
                - 'acme_plugin'
```

Authentication

You can configure a client with authentication. Valid authentication types are `basic`, `bearer`, `service`, `wsse`, `query_param` and `header`. See more examples at the *full configuration*.

```
// config.yml
httplug:
    plugins:
        authentication:
            my_wsse:
                type: 'wsse'
                username: 'my_username'
```

(continues on next page)

(continued from previous page)

```
                password: 'p4ssw0rd'

    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            plugins: ['httplug.plugin.authentication.my_wsse']
```

Warning: Using query parameters for authentication is *not safe*. The auth params will appear on the URL and we recommend to NOT log your request, especially on production side.

VCR Plugin

The *VCR Plugin* allows to record and/or replay HTTP requests. You can configure the mode you want, how to find recorded responses and how to match requests with responses. The mandatory options are:

```
// config.yml
httplug:
    clients:
        acme:
            plugins:
            - vcr:
                mode: replay # record | replay | replay_or_record
                fixtures_directory: '%kernel.project_dir%/fixtures/http' # mandatory for
↪"filesystem" recorder
                # recorder: filesystem
```

See *Full configuration* for the full list of configuration options.

Warning: You have to explicitly require this plugin with composer (`composer require --dev php-http/vcr-plugin`) before using it, as it isn't included by default.

Special HTTP Clients

If you want to use the `FlexibleHttpClient` or `HttpMethodsClient` from the `php-http/client-common` package, you may specify that on the client configuration.

```
// config.yml
httplug:
    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            flexible_client: true

        foobar:
            factory: 'httplug.factory.guzzle6'
            http_methods_client: true
```

List of Services

Service id	Description
`httplug.psr17_request_factory`	Service* that provides the *PsrHttpMessageRequestFactoryInterface*
`httplug.psr17_response_factory`	Service* that provides the *PsrHttpMessageResponseFactoryInterface*
`httplug.psr17_uri_factory`	Service* that provides the *PsrHttpMessageUriFactoryInterface*
`httplug.psr17_stream_factory`	Service* that provides the *PsrHttpMessageStreamFactoryInterface*
`httplug.client.[name]`	There is one service per named client.
`httplug.client`	If there is a client named "default", this service is an alias to that client, otherwise it is an alias to the first client configured.

`httplug.plugin.content_length`
`httplug.plugin.decoder`
`httplug.plugin.logger`
`httplug.plugin.redirect`
`httplug.plugin.retry`
`httplug.plugin.stopwatch`
	These are plugins that are enabled by default.
	These services are private and should only be used to configure
	clients or other services.

`httplug.plugin.cache`
`httplug.plugin.cookie`
`httplug.plugin.history`
`httplug.plugin.error`
	These are plugins that are disabled by default and only get
	activated when configured.
	These services are private and should only be used to configure
	clients or other services.

* *These services are always an alias to another service. You can specify your own service or leave the default, which is the same name with `.default` appended.*

Usage for Reusable Bundles

Rather than code against specific HTTP clients, you want to use the HTTPlug `Client` interface. To avoid building your own infrastructure to define services for the client, simply `require: php-http/httplug-bundle` in your bundles `composer.json`. You SHOULD provide a configuration option to specify which HTTP client service to use for each of your services. This option should default to `httplug.client`. This way, the default case needs no additional configuration for your users, but they have the option of using specific clients with each of your services.

The only steps they need is `require` one of the adapter implementations in their projects `composer.json` and instantiating the `HttplugBundle` in their kernel.

Mock Responses In Functional Tests

First thing to do is add the *php-http/mock-client* to your `require-dev` section. Then, use the mock client factory in your test environment configuration:

```
# config_test.yml
httplug:
    clients:
        my_awesome_backend:
            factory: 'httplug.factory.mock' # replace factory
```

The client is always wrapped into a plugin client. Therefore you need to access the inner client to get the mock client. It is available in the container with the suffix `.inner`. For the example above, the full name is `httplug.clients.my_awesome_backend.inner`.

If you enable a decorator like `http_methods_client:  true`, the actual mock client will be at `httplug.client.my_awesome_backend.http_methods.inner`. Use the `container:debug` command to make sure you grab the correct service.

To mock a response in your tests, do:

```
// SomeWebTestCase.php
$client = static::createClient();

// If your test has the client (BrowserKit) make multiple requests, you need to disable␣
↪reboot as the kernel is rebooted on each request.
// $client->disableReboot();

$response = $this->createMock('Psr\Http\Message\ResponseInterface');
$response->method('getBody')->willReturn(/* Psr\Http\Message\Interface instance␣
↪containing expected response content. */);
$client->getContainer()->get('httplug.clients.my_awesome_backend.client')->addResponse(
↪$response);
```

If you do not specify the factory in your configuration, you can also directly overwrite the HTTPlug services:

```
# config/services_test.yaml
services:
    # overwrite the http clients for mocking
    httplug.client.my_awesome_backend:
        class: Http\Mock\Client
        public: true
```

With this method, the plugin client is not applied. However, if you configure a decorator, your mock client will still be decorated and the mock available as service `...<decorator>.inner`.

Read more on how the mock client works in the *mock client documentation*.

4.7.2 Full configuration

This page shows an example of all configuration values provided by the bundle.

Hint: See *the plugin documentation* for more information on the plugins.

If a plugin is not listed in the configuration reference below, you can configure it as a service and reference the plugin by service id as you would do for a *custom plugin*.

```
// config.yml
httplug:
    # allows to disable autowiring of the clients
    default_client_autowiring: true
    # define which service to use as httplug.<type>
    # this does NOT change autowiring, which will always go to the "default" client
    main_alias:
        client: httplug.client.default
        psr17_request_factory: httplug.psr17_request_factory.default
        psr17_response_factory: httplug.psr17_response_factory.default
        psr17_uri_factory: httplug.psr17_uri_factory.default
        psr17_stream_factory: httplug.psr17_stream_factory.default
    classes:
        # uses discovery if not specified
        client: ~
        psr17_request_factory: ~
        psr17_response_factory: ~
        psr17_uri_factory: ~
        psr17_stream_factory: ~

    plugins: # Global plugin configuration. When configured here, plugins need to be␣
↪explicitly added to clients by service name.
        authentication:
            # The names can be freely chosen, the authentication type is specified in␣
↪the "type" option
            my_basic:
                type: 'basic'
                username: 'my_username'
                password: 'p4ssw0rd'
            my_wsse:
                type: 'wsse'
                username: 'my_username'
                password: 'p4ssw0rd'
            my_bearer:
                type: 'bearer'
                token: 'authentication_token_hash'
            my_query_param:
                type: 'query_param'
                params:
                    access_token: '9zh987g86fg87gh978hg9g79'
            my_header:
                type: 'header'
                header_name: 'ApiKey'
```

(continues on next page)

(continued from previous page)

```
                header_value: '9zh987g86fg87gh978hg9g79'
            my_service:
                type: 'service'
                service: 'my_authentication_service'
        cache: # requires the php-http/cache-plugin package to be installed in your␣
↪package
            cache_pool: 'my_cache_pool'
            stream_factory: 'httplug.stream_factory'
            config:
                default_ttl: 3600
                respect_cache_headers: true
                cache_key_generator: null # This must be a service id to a service␣
↪implementing 'Http\Client\Common\Plugin\Cache\Generator\CacheKeyGenerator'. If 'null'
↪'Http\Client\Common\Plugin\Cache\Generator\SimpleGenerator' will be used.
        cookie:
            cookie_jar: my_cookie_jar
        decoder:
            use_content_encoding: true
        history:
            journal: my_journal
        logger:
            logger: 'logger'
            formatter: null
        redirect:
            preserve_header: true
            use_default_for_multiple: true
        retry:
            retry: 1
        stopwatch:
            stopwatch: 'debug.stopwatch'
        error:
            enabled: false
            only_server_exception: false

    profiling:
        enabled: true # Defaults to kernel.debug
        formatter: null # Defaults to \Http\Message\Formatter\FullHttpMessageFormatter
        captured_body_length: 0

    discovery:
        client: 'auto'
        async_client: false

    clients:
        acme:
            factory: 'httplug.factory.guzzle6'
            service: 'my_service'       # Can not be used with "factory" or "config"
            flexible_client: false      # Can only be true if http_methods_client is␣
↪false
            http_methods_client: false  # Can only be true if flexible_client is false
            public: null                # Set to true if you really cannot use␣
↪dependency injection and need to make the client service public
```

(continues on next page)

(continued from previous page)

```
            config:
                # Options to the Guzzle 6 constructor
                timeout: 2
            plugins:
                # Can reference a globally configured plugin service
                - 'httplug.plugin.authentication.my_wsse'
                # Can configure a plugin customized for this client
                - cache:
                    cache_pool: 'my_other_pool'
                    config:
                        default_ttl: 120
                # Can configure plugins that can not be configured globally
                - add_host:
                    # Host name including protocol and optionally the port number, e.g.␣
↪https://api.local:8000
                    host: http://localhost:80 # Required
                    # Whether to replace the host if request already specifies it
                    replace: false
                - add_path:
                    # Path to be added, e.g. /api/v1
                    path: /api/v1 # Required
                - base_uri:
                    # Base Uri including protocol, optionally the port number and␣
↪prepend path, e.g. https://api.local:8000/api
                    uri: http://localhost:80 # Required
                    # Whether to replace the host if request already specifies one
                    replace: false
                # Set content-type header based on request body, if the header is not␣
↪already set
                - content_type:
                    # skip content-type detection if body is larger than size_limit
                    skip_detection: true
                    # size_limit in bytes for when skip_detection is enabled
                    size_limit: 200000
                # Append headers to the request. If the header already exists the value␣
↪will be appended to the current value.
                - header_append:
                    # Keys are the header names, values the header values
                    headers:
                        'X-FOO': bar # contrary to default symfony behavior, hyphens "-"␣
↪are NOT translated to underscores "_" for the headers.
                # Set header to default value if it does not exist.
                - header_defaults:
                    # Keys are the header names, values the header values
                    headers:
                        'X-FOO': bar
                # Set headers to requests. If the header does not exist it wil be set,␣
↪if the header already exists it will be replaced.
                - header_set:
                    # Keys are the header names, values the header values
                    headers:
                        'X-FOO': bar
```

(continues on next page)

(continued from previous page)

```
                # Remove headers from requests.
                - header_remove:
                    # List of header names to remove
                    headers: ["X-FOO"]
                # Sets query parameters to default value if they are not present in the␣
↪request.
                - query_defaults:
                    parameters:
                        locale: en
                # Plugins to ensure the request resp response body is seekable
                - request_seekable_body:
                    use_file_buffer: true
                    memory_buffer_size: 2097152
                - response_seekable_body:
                    use_file_buffer: true
                    memory_buffer_size: 2097152
                # Enable VCR plugin integration (Must be installed first).
                - vcr:
                    mode: replay # record | replay | replay_or_record
                    fixtures_directory: '%kernel.project_dir%/fixtures/http' # mandatory␣
↪for "filesystem" recorder
                    # recorder: filesystem  ## Can be filesystem, in_memory or the id of␣
↪your custom recorder
                    # naming_strategy: service_id.of.naming_strategy # or "default"
                    # naming_strategy_options: # options for the default naming strategy,
↪ see VCR plugin documentation
                    #     hash_headers: []
                    #     hash_body_methods: []
```

- Nette Framework Integration (external Documentation)

4.8 Backwards compatibility

Backwards compatibility is an important topic for us, as it should be in every open source project. We follow Semver which allows us to only break backwards compatibility between major versions. We use deprecation notices to inform you about the changes made before they are removed.

Our backwards compatibility promise does not include classes or functions with the `@internal` annotation.

4.8.1 Symfony Bundle

The HttplugBundle is just a Symfony integration for HTTPlug and it does not have any classes which falls under the BC promise. The backwards compatibility of the bundle is only the configuration and its values (and of course the behavior of those values).

4.8.2 Discovery

The order of the strategies is not part of our BC promise. The strategies themselves are marked as `@internal` so they are also not part of our BC promise. However, we do promise that we will not remove a strategy neither will we remove classes from the `CommonClassesStrategy`.

The consequences of the BC promise for the discovery library is that you can not rely on the *same* client to be returned in the future. However, if discovery does find a client now, you can be sure that after future updates it will still discover a client.

4.9 Message

This package contains various PSR-7 tools which might be useful in an HTTP workflow:

4.9.1 Authentication

The Authentication component allows you to to implement authentication methods which can simply update the request with authentication detail (for example by adding an `Authorization` header). This is useful when you have to send multiple requests to the same endpoint. Using an authentication implementation, these details can be separated from the actual requests.

Installation

```
$ composer require php-http/message
```

Authentication Methods

Method	Parameters	Behavior
Basic Auth	Username and password	`Authorization` header of the HTTP specification
Bearer	Token	`Authorization` header of the HTTP specification
WSSE	Username and password	`Authorization` header of the HTTP specification
Query Params	Array of param-value pairs	URI parameters
Chain	Array of authentication instances	Behaviors of the underlying authentication methods
Matching	An authentication instance and a matcher callback	Behavior of the underlying authentication method if the matcher callback passes
Header	Header name and value	Add an arbitrary authentication header

Integration with HTTPlug

Normally requests must be authenticated "by hand" which is not really convenient.

If you use HTTPlug, you can integrate this component into the client using the *authentication plugin*.

Examples

General usage looks like the following:

```
$authentication = new AuthenticationMethod();

/** @var Psr\Http\Message\RequestInterface */
$authentication->authenticate($request);
```

Basic Auth

```
use Http\Message\Authentication\BasicAuth;

$authentication = new BasicAuth('username', 'password');
```

Bearer

```
use Http\Message\Authentication\Bearer;

$authentication = new Bearer('token');
```

WSSE

```
use Http\Message\Authentication\Wsse;

$authentication = new Wsse('username', 'password');
```

For better security, also pass the 3rd optional parameter to use a better hashing algorithm than `sha1`, e.g.

```
use Http\Message\Authentication\Wsse;

$authentication = new Wsse('username', 'password', 'sha512');
```

Query Params

http://api.example.com/endpoint?access_token=9zh987g86fg87gh978hg9g79:

```
use Http\Message\Authentication\QueryParam;

$authentication = new QueryParam([
    'access_token' => '9zh987g86fg87gh978hg9g79',
]);
```

Warning: Using query parameters for authentication is not safe. Only use it when this is the only authentication method offered by a third party application.

Chain

The idea behind this authentication method is that in some cases you might need to authenticate the request with multiple methods.

For example it's a common practice to protect development APIs with Basic Auth and the regular token authentication as well to protect the API from unnecessary processing:

```
use Http\Message\Authentication\Chain;

$authenticationChain = [
    new AuthenticationMethod1(),
    new AuthenticationMethod2(),
];

$authentication = new Chain($authenticationChain);
```

Matching

With this authentication method you can conditionally add authentication details to your request by passing a callable to it. When a request is passed, the callable is called and used as a boolean value in order to decide whether the request should be authenticated or not. It also accepts an authentication method instance which does the actual authentication when the condition is fulfilled.

For example a common use case is to authenticate requests sent to certain paths:

```
use Http\Message\Authentication\Matching;
use Psr\Http\Message\RequestInterface;

$authentication = new Matching(
    new AuthenticationMethod1(),
    function (RequestInterface $request) {
        $path = $request->getUri()->getPath();

        return 0 === strpos($path, '/api');
    }
);
```

In order to ease creating matchers for URLs/paths, there is a static factory method for this purpose: `createUrlMatcher` The first argument is an authentication method, the second is a regular expression to match against the URL:

```
use Http\Message\Authentication\Matching;

$authentication = Matching::createUrlMatcher(new AuthenticationMethod(), '\/api');
```

Header

With this authentication method you can add arbitrary headers.

In the following example, we are setting a `X-AUTH-TOKEN` header with it's value:

```
use Http\Message\Authentication\Header;

$authentication = new Header('X-AUTH-TOKEN', '9zh987g86fg87gh978hg9g79');
```

Implement Your Own

Implementing an authentication method is easy: only one method needs to be implemented:

```
use Http\Message\Authentication;
use Psr\Http\Message\RequestInterface;

class MyAuth implements Authentication
{
    public function authenticate(RequestInterface $request)
    {
        // do something with the request

        // keep in mind that the request is immutable - return the updated
        // version of the request with the authentication information added
        // to it.
        return $request;
    }
}
```

4.9.2 HTTP Factories (deprecated)

Factory interfaces for PSR-7 HTTP objects.

This package has been superseded by PSR-17. Our HTTP-PHP factories have been retired and the repository archived. The PHP-HTTP libraries switched to use the PSR-17 factories. Please migrate your code to the PSR-17 factories too.

Rationale

At the time of building this, PSR-17 did not yet exist. Read the documentation of PSR-17 to learn why a standard for factories is useful.

Factories

The *php-http/message-factory* package defines interfaces for PSR-7 factories including:

- `RequestFactory`
- `ResponseFactory`
- `MessageFactory` (combination of request and response factories)
- `StreamFactory`
- `UriFactory`

Implementations of the interfaces above for Laminas Diactoros (and its abandoned predecessor Zend Diactoros), Guzzle PSR-7 and the Slim PSR-7 can be found in `php-http/message`.

Usage

Instantiate the factories in your bootstrap code or use discovery for them. Inject the factories into the rest of your code to limit the implementation choice to the bootstrapping code:

```php
// ApiClient.php

use Http\Message\RequestFactory;
use Http\Message\StreamFactory;
use Http\Message\UriFactory;

class ApiClient
{
    /**
     * @var RequestFactory
     */
    private $requestFactory;

    /**
     * @var StreamFactory
     */
    private $streamFactory;

    /**
     * @var UriFactory
     */
    private $uriFactory;

    public function __construct(
        RequestFactory $requestFactory,
        StreamFactory $streamFactory,
        UriFactory $uriFactory
    ) {
```

(continues on next page)

(continued from previous page)

```
        $this->requestFactory = $requestFactory;
        $this->streamFactory = $streamFactory;
        $this->uriFactory = $uriFactory;
    }

    public function doStuff()
    {
        $request = $this->requestFactory->createRequest('GET', 'http://httplug.io');
        $stream = $this->streamFactory->createStream('stream content');
        $uri = $this->uriFactory->createUri('http://httplug.io');
        ...
    }
}
```

The bootstrapping code could look like this:

```
// bootstrap.php
use Http\Message\MessageFactory\DiactorosMessageFactory;
use Http\Message\StreamFactory\DiactorosStreamFactory;
use Http\Message\UriFactory\DiactorosUriFactory;

$apiClient = new ApiClient(
    new DiactorosMessageFactory(),
    new DiactorosStreamFactory(),
    new DiactorosUriFactory()
);
```

You could also use *Discovery* to make the factory arguments optional and automatically find an available factory in the client:

```
// ApiClient.php

use Http\Discovery\MessageFactoryDiscovery;
use Http\Discovery\StreamFactoryDiscovery;
use Http\Discovery\UriFactoryDiscovery;
use Http\Message\RequestFactory;
use Http\Message\StreamFactory;
use Http\Message\UriFactory;

class ApiClient
{
    public function __construct(
        RequestFactory $requestFactory = null,
        StreamFactory $streamFactory = null,
        UriFactory $uriFactory = null
    ) {
        $this->requestFactory = $requestFactory ?: MessageFactoryDiscovery::find(),
        $this->streamFactory = $streamFactory ?: StreamFactoryDiscovery::find();
        $this->uriFactory = $uriFactory ?: UriFactoryDiscovery::find();;
    }

    ...
}
```

Hint: If you create requests only and no responses, use `RequestFactory` in the type hint, instead of the `MessageFactory`. And vice versa if you create responses only.

- Authentication method implementations
- Various Stream encoding tools
- Message decorators
- Cookie implementation

4.10 Client Common

The client-common package provides some useful tools for working with HTTPlug. Include them in your project with composer:

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

4.10.1 HttpMethodsClient

This client wraps the HttpClient and provides convenience methods for common HTTP requests like `GET` and `POST`. To be able to do that, it also wraps a message factory:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Discovery\MessageFactoryDiscovery;

$client = new HttpMethodsClient(
    HttpClientDiscovery::find(),
    MessageFactoryDiscovery::find()
);

$foo = $client->get('http://example.com/foo');
$bar = $client->get('http://example.com/bar', ['accept-encoding' => 'application/json']);
$post = $client->post('http://example.com/update', [], 'My post body');
```

..versionadded:: 2.0
`HttpMethodsClient` is final since version 2.0. You can typehint the `HttpMethodsClientInterface` to allow mocking the client in unit tests.

4.10.2 BatchClient

This client wraps a HttpClient and extends it with the possibility to send an array of requests and to retrieve their responses as a `BatchResult`:

```
use Http\Discovery\HttpClientDiscovery;
use Http\Discovery\MessageFactoryDiscovery;

$messageFactory = MessageFactoryDiscovery::find();

$requests = [
```

(continues on next page)

(continued from previous page)

```
    $messageFactory->createRequest('GET', 'http://example.com/foo'),
    $messageFactory->createRequest('POST', 'http://example.com/update', [], 'My post body
↪'),
];

$client = new BatchClient(
    HttpClientDiscovery::find()
);

$batchResult = $client->sendRequests($requests);
```

..versionadded:: 2.0

> `BatchClient` is final since version 2.0. You can typehint the `BatchClientInterface` to allow mocking the client in unit tests.

The `BatchResult` itself is an object that contains responses for all requests sent. It provides methods that give appropriate information based on a given request:

```
$requests = [
    $messageFactory->createRequest('GET', 'http://example.com/foo'),
    $messageFactory->createRequest('POST', 'http://example.com/update', [], 'My post body
↪'),
];

$batchResult = $client->sendRequests($requests);

if ($batchResult->hasResponses()) {
    $fooSuccessful = $batchResult->isSuccessful($requests[0]);
    $updateResponse = $batchResult->getResponseFor($request[1]);
}
```

If one or more of the requests throw exceptions, they are added to the `BatchResult` and the `BatchClient` will ultimately throw a `BatchException` containing the `BatchResult` and therefore its exceptions:

```
$requests = [
    $messageFactory->createRequest('GET', 'http://example.com/update'),
];

try {
    $batchResult = $client->sendRequests($requests);
} catch (BatchException $e) {
    var_dump($e->getResult()->getExceptions());
}
```

4.10.3 PluginClient

See *the documentation about plugins*

4.10.4 HttpClientPool

The `HttpClientPool` allows to balance requests between a pool of `HttpClient` and/or `HttpAsyncClient`.

The use cases are:

- Using a cluster (like an Elasticsearch service with multiple master nodes)
- Using fallback servers with the combination of the `RetryPlugin` (see *Retry Plugin*)

You can attach HTTP clients to this kind of client by using the `addHttpClient` method:

```
use Http\Client\Common\HttpClientPool\LeastUsedClientPool;
use Http\Discovery\HttpAsyncClientDiscovery;
use Http\Discovery\HttpClientDiscovery;
use Http\Discovery\MessageFactoryDiscovery;

$messageFactory = MessageFactoryDiscovery::find();

$httpClient = HttpClientDiscovery::find();
$httpAsyncClient = HttpAsyncClientDiscovery::find();

$httpClientPool = new LeastUsedClientPool();
$httpClientPool->addHttpClient($httpClient);
$httpClientPool->addHttpClient($httpAsyncClient);

$httpClientPool->sendRequest($messageFactory->createRequest('GET', 'http://example.com/
↪update'));
```

Clients added to the pool are decorated with the `HttpClientPoolItem` class unless they already are an instance of this class. The pool item class lets the pool be aware of the number of requests currently being processed by that client. It is also used to deactivate clients when they receive errors. Deactivated clients can be reactivated after a certain amount of time, however, by default, they stay deactivated forever. To enable the behavior, wrap the clients with the `HttpClientPoolItem` class yourself and specify the re-enable timeout:

```
// Reactivate after 30 seconds
$httpClientPool->addHttpClient(new HttpClientPoolItem($httpClient, 30));
// Reactivate after each call
$httpClientPool->addHttpClient(new HttpClientPoolItem($httpClient, 0));
// Never reactivate the client (default)
$httpClientPool->addHttpClient(new HttpClientPoolItem($httpClient, null));
```

`HttpClientPool` is an interface. There are three concrete implementations with specific strategies on how to choose clients:

LeastUsedClientPool

`LeastUsedClientPool` choose the client with the fewest requests in progress. As it sounds the best strategy for sending a request on a pool of clients, this strategy has some limitations: :

- The counter is not shared between PHP process, so this strategy is not so useful in a web context, however it will make better sense in a PHP command line context.
- This pool only makes sense with asynchronous clients. If you use `sendRequest`, the call is blocking, and the pool will only ever use the first client as its request count will be 0 once `sendRequest` finished.

A deactivated client will be removed for the pool until it is reactivated, if none are available it will throw a `NotFoundHttpClientException`

RoundRobinClientPool

`RoundRobinClientPool` keeps an internal pointer on the pool. At each call the pointer is moved to the next client, if the current client is disabled it will move to the next client, and if none are available it will throw a `NotFoundHttpClientException`

The pointer is not shared across PHP processes, so for each new one it will always start on the first client.

RandomClientPool

`RandomClientPool` randomly choose an available client, throw a `NotFoundHttpClientException` if none are available.

4.10.5 HTTP Client Router

This client accepts pairs of clients and request matchers. Every request is "routed" through the `HttpClientRouter`, checked against the request matchers and sent using the first matched client. If there is no matching client, an exception is thrown.

This allows a single client to be used for different requests.

In the following example we use the client router to access an API protected by basic auth and also to download an image from a static host:

```php
use Http\Client\Common\HttpClientRouter;
use Http\Client\Common\PluginClient;
use Http\Client\Common\Plugin\AuthenticationPlugin;
use Http\Client\Common\Plugin\CachePlugin;
use Http\Discovery\HttpClientDiscovery;
use Http\Discovery\MessageFactoryDiscovery;
use Http\Message\Authentication\BasicAuth;
use Http\Message\RequestMatcher\RequestMatcher;

$client = new HttpClientRouter();

$requestMatcher = new RequestMatcher(null, 'api.example.com');
$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [new AuthenticationPlugin(new BasicAuth('user', 'password'))]
);
```

(continues on next page)

(continued from previous page)

```
$client->addClient($pluginClient, $requestMatcher);


$requestMatcher = new RequestMatcher(null, 'images.example.com');

/** @var \Psr\Cache\CacheItemPoolInterface $pool */
$pool = ...
/** @var \Http\Message\StreamFactory $streamFactory */
$streamFactory = ...

$pluginClient = new PluginClient(
    HttpClientDiscovery::find(),
    [new CachePlugin($pool, $streamFactory)]
);

$client->addClient($pluginClient, $requestMatcher);


$messageFactory = MessageFactoryDiscovery::find();

// Get the user data
$request = $messageFactory->createRequest('GET', 'https://api.example.com/user/1');

$response = $client->send($request);
$imagePath = json_decode((string) $response->getBody(), true)['image_path'];

// Download the image and store it in cache
$request = $messageFactory->createRequest('GET', 'https://images.example.com/user/'.
↪$imagePath);

$response = $client->send($request);

file_put_contents('path/to/images/'.$imagePath, (string) $response->getBody());

$request = $messageFactory->createRequest('GET', 'https://api2.example.com/user/1');

// Throws an Http\Client\Exception\RequestException
$client->send($request);
```

Note: When you have small difference between the underlying clients (for example different credentials based on host) it's easier to use the `RequestConditionalPlugin` and the `PluginClient`, but in that case the routing logic is integrated into the linear request flow which might make debugging harder.

..versionadded:: 2.0
`HttpClientRouter` is final since version 2.0. You can typehint the `HttpClientRouterInterface` to allow mocking the client in unit tests.

4.11 Adapter Integration Tests

TODO

4.12 Promise

A promise represents a single result of an asynchronous operation. It is not necessarily available at a specific time, but should become in the future.

The PHP-HTTP promise follows the Promises/A+ standard.

Note: Work is underway for a Promise PSR. When that PSR has been released, we will use it in HTTPlug and deprecate our `Http\Promise\Promise` interface.

4.12.1 Asynchronous requests

Asynchronous requests enable non-blocking HTTP operations. When sending asynchronous HTTP requests, a promise is returned. The promise acts as a proxy for the response or error result, which is not yet known.

To execute such a request with HTTPlug:

```php
$request = $messageFactory->createRequest('GET', 'http://php-http.org');

// Where $client implements HttpAsyncClient
$promise = $client->sendAsyncRequest($request);

// This code will be executed right after the request is sent, but before
// the response is returned.
echo 'Wow, non-blocking!';
```

See *HTTP Factories (deprecated)* on how to use message factories.

Although the promise itself is not restricted to resolve a specific result type, in HTTP context it resolves a PSR-7 `Psr\Http\Message\ResponseInterface` or fails with an `Http\Client\Exception`.

Note: An asynchronous request will never throw an exception directly but always return a promise. All exceptions SHOULD implement `Http\Client\Exception`. See *Exceptions* for more information on the exceptions you might encounter.

4.12.2 Wait

The `$promise` that is returned implements `Http\Promise\Promise`. At this point in time, the response is not known yet. You can be polite and wait for that response to arrive:

```php
try {
    $response = $promise->wait();
} catch (\Exception $exception) {
```

(continues on next page)

(continued from previous page)

```php
    echo $exception->getMessage();
}
```

4.12.3 Then

Instead of waiting, however, you can handle things asynchronously. Call the `then` method with two arguments: one callback that will be executed if the request turns out to be successful and/or a second callback that will be executed if the request results in an error:

```php
$promise->then(
    // The success callback
    function (ResponseInterface $response) {
        echo 'Yay, we have a shiny new response!';

        // Write status code to some log file
        file_put_contents('responses.log', $response->getStatusCode() . "\n", FILE_
↪APPEND);

        return $response;
    },

    // The failure callback
    function (\Exception $exception) {
        echo 'Oh darn, we have a problem';

        throw $exception;
    }
);
```

The failure callback can also return a `Promise`. This can be useful to implement a retry mechanism, as follows:

```php
use Http\Discovery\HttpAsyncClientDiscovery;
use Http\Discovery\Psr17FactoryDiscovery;

$client = HttpAsyncClientDiscovery::find();
$requestFactory = Psr17FactoryDiscovery::findRequestFactory();
$retries = 2; // number of HTTP retries
$request = $requestFactory->createRequest("GET", "http://localhost:8080/test");

// success callback
$success = function (ResponseInterface $response) {
    return $response;
};
// failure callback
$failure = function (Exception $e) use ($client, $request) {
    // $request can be changed, e.g. using a Round-Robin algorithm

    // try another execution
    return $client->sendAsyncRequest($request);
};

```

(continues on next page)

(continued from previous page)

```php
$promise = $client->sendAsyncRequest($request);
for ($i=0; $i < $retries; $i++) {
    $promise = $promise->then($success, $failure);
}
// Add the last callable to manage the exceeded maximum number of retries
$promise->then($success, function(\Exception $e) {
    throw new \Exception(sprintf(
        "Exceeded maximum number of retries (%d): %s",
        $retries,
        $e->getMessage()
    ));
});
```

4.13 Discovery

The discovery service allows to find installed resources and auto-install missing ones.

Currently available discovery services:

- PSR-17 Factory Discovery
- PSR-18 HTTP Client Discovery
- PSR-7 Message Factory Discovery (deprecated in favor of PSR-17)
- PSR-7 URI Factory Discovery (deprecated in favor of PSR-17)
- PSR-7 Stream Factory Discovery (deprecated in favor of PSR-17)
- HTTP Async Client Discovery
- HTTP Client Discovery (deprecated in favor of PSR-18)
- Mock Client Discovery (not enabled by default)

The principle is always the same: you call the static `find` method on the discovery service. The discovery service will try to locate a suitable implementation. If no implementation is found, an `Http\Discovery\Exception\NotFoundException` is thrown.

Discovery is simply a convenience wrapper to statically access clients and factories for when Dependency Injection is not an option. Discovery is particularly useful in libraries that want to offer zero-configuration services relying on the virtual packages.

4.13.1 Using discovery in a shared library

The goal of the PSR standards is that libraries do not depend on specific implementations but only on the standard. The library should only require the PSR standards.

To run tests, you might still need an implementation. We recommend to explicitly require that, but only for development. To build a library that needs to send HTTP requests, you could do:

```
$ composer require --dev symfony/http-client
$ composer require --dev nyholm/psr7
```

Then, you can disable the Composer plugin provided by `php-http/discovery` because you just installed the `dev` dependencies you need for testing:

```
$ composer config allow-plugins.php-http/discovery false
```

Finally, you need to require `php-http/discovery` and the generic implementations that your library is going to need:

```
$ composer require php-http/discovery:^1.17
$ composer require psr/http-client-implementation:*
$ composer require psr/http-factory-implementation:*
```

Now, you're ready to make an HTTP request:

```
use Http\Discovery\Psr18Client;

$client = new Psr18Client();

$request = $client->createRequest('GET', 'https://example.com');
$response = $client->sendRequest($request);
```

New in version 1.17: The `Psr18Client` is available since v1.17.

Internally, this code will use whatever PSR-7, PSR-17 and PSR-18 implementations your users have installed.

It is best practice to allow the users of your library to optionally specify the `ClientInterface` instance and only fallback to discovery when no explicit client has been specified.

4.13.2 Auto-installation

New in version 1.15: Auto-installation of missing dependencies is available since v1.15.

Discovery embeds a composer plugin that can auto-install missing implementations when an application does not specify any specific implementation.

If a library requires both `php-http/discovery` and one of the supported virtual packages (see *HTTPlug for Library Developers*), but no implementation for the virtual package is already installed, the plugin will auto-install the best matching known implementation.

For example, if the project requires `react/event-loop`, the plugin will select `php-http/react-adapter` to meet a missing dependency on `php-http/client-implementation`.

The following abstractions are currently supported:

- `php-http/async-client-implementation`
- `php-http/client-implementation`
- `psr/http-client-implementation`
- `psr/http-factory-implementation`
- `psr/http-message-implementation`

Note: Auto-installation is only done for libraries that directly require `php-http/discovery` to avoid unexpected dependency installation.

If you do not want auto-installation to happen, you can chose to not enable the composer plugin of the discovery component:

```
composer config allow-plugins.php-http/discovery false
```

4.13.3 Strategies

The package uses strategies to select an implementation.

The default strategy contains a list of preferences that looks for well-known implementations: Symfony, Guzzle, Diactoros and Slim Framework.

Once a strategy provided a candidate, the result is cached in memory and reused for further discovery calls in the same process.

To register a custom strategy, implement the `Http\Discovery\Strategy\DiscoveryStrategy` interface and register your strategy with the `prependStrategy`, `appendStrategy` or `setStrategies` method of the corresponding discovery type.

4.13.4 Implementation Pinning

New in version 1.17: Pinning the preferred implementation is available since v1.17.

In case there are several implementations available, the application can pin which implementation to prefer. You can specify the implementation for one of the standards:

```
$ composer config extra.discovery.psr/http-factory-implementation GuzzleHttp\Psr7\
↪HttpFactory
```

This will update your `composer.json` file to add the following configuration:

```
"extra": {
    "discovery": {
        "psr/http-factory-implementation": "GuzzleHttp\\Psr7\\HttpFactory"
    }
}
```

You can also pin single interfaces, e.g. for the PSR-17 factories:

```
"extra": {
    "discovery": {
        "Psr\\Http\\Message\\RequestFactoryInterface": "Slim\\Psr7\\Factory\\
↪RequestFactory"
    }
}
```

Don't forget to run composer install to apply the changes, and ensure that the composer plugin is enabled:

```
$ composer config allow-plugins.php-http/discovery true
$ composer install
```

Note: Implementation pinning only works if the composer plugin of discovery is allowed. If you disabled the plugin, you need to configure your own discovery if you need a specific implementation selection.

4.13.5 Installation

```
$ composer require php-http/discovery
```

4.13.6 Common Errors

Could not find resource using any discovery strategy

If you get an error saying "*Could not find resource using any discovery strategy.*" it means that all the discovery *strategies* have failed. Most likely, your project is missing the message factories and/or a PRS-7 implementation. See the *user documentation*.

To resolve this you may run

```
$ composer require php-http/curl-client guzzlehttp/psr7 php-http/message
```

No factories found

The error "*No message factories found. To use Guzzle, Diactoros or Slim Framework factories install php-http/message and the chosen message implementation.*" tells you that no discovery strategy could find an installed implementation of PSR-7 and/or factories for that implementation. You need to install those libraries. If you want to use Guzzle you may run:

```
$ composer require php-http/message guzzlehttp/psr7
```

No HTTPlug clients found

The error "*No HTTPlug clients found. Make sure to install a package providing 'php-http/client-implementation'*" says that we cannot find a client. See our *list of clients* and install one of them.

```
$ composer require php-http/curl-client
```

4.13.7 HTTP Client Discovery

This type of discovery finds an HTTP Client implementation:

```
use Http\Client\HttpClient;
use Http\Discovery\HttpClientDiscovery;

class MyClass
{
    /**
     * @var HttpClient
     */
    private $httpClient;

    /**
     * @param HttpClient|null $httpClient Client to do HTTP requests, if not set, auto␣
↪discovery will be used to find a HTTP client.
```

(continues on next page)

(continued from previous page)

```
     */
    public function __construct(HttpClient $httpClient = null)
    {
        $this->httpClient = $httpClient ?: HttpClientDiscovery::find();
    }
}
```

4.13.8 HTTP Asynchronous Client Discovery

This type of discovery finds a HTTP asynchronous Client implementation:

```
use Http\Client\HttpAsyncClient;
use Http\Discovery\HttpAsyncClientDiscovery;

class MyClass
{
    /**
     * @var HttpAsyncClient
     */
    private $httpAsyncClient;

    /**
     * @param HttpAsyncClient|null $httpAsyncClient Client to do HTTP requests, if not␣
↪set, auto discovery will be used to find an asynchronous client.
     */
    public function __construct(HttpAsyncClient $httpAsyncClient = null)
    {
        $this->httpAsyncClient = $httpAsyncClient ?: HttpAsyncClientDiscovery::find();
    }
}
```

4.13.9 PSR-17 Factory Discovery

This type of discovery finds a factory for a PSR-17 implementation:

```
use Psr\Http\Message\RequestFactoryInterface;
use Psr\Http\Message\ResponseFactoryInterface;
use Http\Discovery\Psr17FactoryDiscovery;

class MyClass
{
    /**
     * @var RequestFactoryInterface
     */
    private $requestFactory;

    /**
     * @var ResponseFactoryInterface
     */
    private $responseFactory;
```

(continues on next page)

(continued from previous page)

```
    /**
     * @var ServerRequestFactoryInterface
     */
    private $serverRequestFactory;

    /**
     * @var StreamFactoryInterface
     */
    private $streamFactory;

    /**
     * @var UploadedFileFactoryInterface
     */
    private $uploadedFileFactory;

    /**
     * @var UriFactoryInterface
     */
    private $uriFactory;

    public function __construct(
        RequestFactoryInterface $requestFactory = null,
        ResponseFactoryInterface $responseFactory = null,
        ServerRequestFactoryInterface $serverRequestFactory = null,
        StreamFactoryInterface $streamFactory = null,
        UploadedFileFactoryInterface $uploadedFileFactory = null,
        UriFactoryInterface = $uriFactoryInterface = null
    ) {
        $this->requestFactory = $requestFactory ?:␣
↪Psr17FactoryDiscovery::findRequestFactory();
        $this->responseFactory = $responseFactory ?:␣
↪Psr17FactoryDiscovery::findResponseFactory();
        $this->serverRequestFactory = $serverRequestFactory ?:␣
↪Psr17FactoryDiscovery::findServerRequestFactory();
        $this->streamFactory = $streamFactory ?:␣
↪Psr17FactoryDiscovery::findStreamFactory();
        $this->uploadedFileFactory = $uploadedFileFactory ?:␣
↪Psr17FactoryDiscovery::findUploadedFileFactory();
        $this->uriFactory = $uriFactory ?: Psr17FactoryDiscovery::findUriFactory();
    }
}
```

4.13.10 PSR-17 Factory

The package also provides an `Http\Discovery\Psr17Factory` class that can be instantiated to get a generic PSR-17 factory:

```php
use Http\Discovery\Psr17Factory;

$factory = new Psr17Factory();

// use any PSR-17 methods, e.g.
$request = $factory->createRequest();
```

Internally, this class relies on the concrete PSR-17 factories that are installed in your project and can use discovery to find implementations if you do not specify them in the constructor.

`Psr17Factory` provides two additional methods that allow creating server requests or URI objects from the PHP superglobals:

```php
$serverRequest = $factory->createServerRequestFromGlobals();
$uri = $factory->createUriFromGlobals();
```

New in version 1.15: The `Psr17Factory` class is available since version 1.15.

4.13.11 PSR-18 Client Discovery

This type of discovery finds a PSR-18 HTTP Client implementation:

```php
use Psr\Http\Client\ClientInterface;
use Http\Discovery\Psr18ClientDiscovery;

class MyClass
{
    /**
     * @var ClientInterface
     */
    private $httpClient;

    public function __construct(ClientInterface $httpClient = null)
    {
        $this->httpClient = $httpClient ?: Psr18ClientDiscovery::find();
    }
}
```

4.13.12 PSR-7 Message Factory Discovery

New in version 1.6: This is deprecated and will be removed in 2.0. Consider using PSR-17 Factory Discovery.

This type of discovery finds a *HTTP Factories (deprecated)* for a PSR-7 Message implementation:

```php
use Http\Message\MessageFactory;
use Http\Discovery\MessageFactoryDiscovery;

class MyClass
```

(continues on next page)

(continued from previous page)

```
{
    /**
     * @var MessageFactory
     */
    private $messageFactory;

    /**
     * @param MessageFactory|null $messageFactory to create PSR-7 requests.
     */
    public function __construct(MessageFactory $messageFactory = null)
    {
        $this->messageFactory = $messageFactory ?: MessageFactoryDiscovery::find();
    }
}
```

4.13.13 PSR-7 URI Factory Discovery

New in version 1.6: This is deprecated and will be removed in 2.0. Consider using PSR-17 Factory Discovery.

This type of discovery finds a URI factory for a PSR-7 URI implementation:

```
use Http\Message\UriFactory;
use Http\Discovery\UriFactoryDiscovery;

class MyClass
{
    /**
     * @var UriFactory
     */
    private $uriFactory;

    /**
     * @param UriFactory|null $uriFactory to create UriInterface instances from strings.
     */
    public function __construct(UriFactory $uriFactory = null)
    {
        $this->uriFactory = $uriFactory ?: UriFactoryDiscovery::find();
    }
}
```

4.13.14 Mock Client Discovery

You may find yourself testing parts of your application that are dependent on an HTTP Client using the Discovery Service, but do not necessarily need to perform the request nor contain any special configuration. In this case, the `Http\Mock\Client` from the `php-http/mock-client` package is typically used to fake requests and keep your tests nicely decoupled. However, for the best stability in a production environment, the mock client is not set to be found via the Discovery Service. Attempting to run a test which relies on discovery and uses a mock client will result in an `Http\Discovery\Exception\NotFoundException`. Thankfully, Discovery gives us a Mock Client strategy that can be added straight to the Discovery. Let's take a look:

```
use MyCustomService;
use Http\Mock\Client as MockClient;
use Http\Discovery\Psr18ClientDiscovery;
use Http\Discovery\Strategy\MockClientStrategy;

class MyCustomServiceTest extends TestCase
{
    public function setUp()
    {
        Psr18ClientDiscovery::prependStrategy(MockClientStrategy::class);

        $this->service = new MyCustomService;
    }

    public function testMyCustomServiceDoesSomething()
    {
        // Test...
    }
}
```

In the example of a test class above, we have our `MyCustomService` which relies on an HTTP Client implementation. We do not need to test that the actual request our custom service makes is successful in this test class, so it makes sense to use the Mock Client. However, we do want to make sure that our dependency injection using the Discovery service properly works, as this is a major feature of our service. By calling the `HttpClientDiscovery`'s `prependStrategy` method and passing in the `MockClientStrategy` namespace, we have now added the ability to discover the mock client and our tests will work as desired.

It is important to note that you must explicitly enable the `MockClientStrategy` and that it is not used by the Discovery Service by default. It is simply provided as a convenient option when writing tests.

4.14 Multipart Stream Builder

A multipart stream is a special kind of stream that is used to transfer files over HTTP. There is currently no PSR-7 support for multipart streams as they are considered to be normal streams with a special content. A multipart stream HTTP request may look like this:

```
POST / HTTP/1.1
Host: example.com
Content-Type: multipart/form-data; boundary="578de3b0e3c46.2334ba3"

--578de3b0e3c46.2334ba3
Content-Disposition: form-data; name="foo"
Content-Length: 15

A normal stream
--578de3b0e3c46.2334ba3
Content-Disposition: form-data; name="bar"; filename="bar.png"
Content-Length: 71
Content-Type: image/png

?PNG

```

(continues on next page)

(continued from previous page)

```
???
IHDR??? ??? ?????? ???? IDATx{c???51?)?:??????IEND?B`?
--578de3b0e3c46.2334ba3
Content-Type: text/plain
Content-Disposition: form-data; name="baz"
Content-Length: 6

string
--578de3b0e3c46.2334ba3--
```

In the request above you see a set of HTTP headers and a body with two streams. The body starts and ends with a "boundary" and it is also this boundary that separates the streams. That boundary also needs to be specified in the `Content-Type` header.

4.14.1 Building a Multipart Stream

To build a multipart stream you may use the `MultipartStreamBuilder`. It is not coupled to any stream implementation so it needs a `StreamFactory` to create the streams.

```
$streamFactory = StreamFactoryDiscovery::find();
$builder = new MultipartStreamBuilder($streamFactory);
$builder
  ->addResource('foo', $stream)
  ->addResource('bar', fopen($filePath, 'r'), ['filename' => 'bar.png'])
  ->addData('baz', ['headers' => ['Content-Type' => 'text/plain']]);

$multipartStream = $builder->build();
$boundary = $builder->getBoundary();

$request = MessageFactoryDiscovery::find()->createRequest(
  'POST',
  'http://example.com',
  ['Content-Type' => 'multipart/form-data; boundary="'.$boundary.'"'],
  $multipartStream
);

$response = HttpClientDiscovery::find()->sendRequest($request);
```

The second parameter of `MultipartStreamBuilder::addResource()` is the content of the stream. The supported input is the same as `StreamFactory::createStream()`.

4.15 Development

4.15.1 Contributor Code of Conduct

As contributors and maintainers of this project, and in the interest of fostering an open and welcoming community, we pledge to respect all people who contribute through reporting issues, posting feature requests, updating documentation, submitting pull requests or patches, and other activities.

We are committed to making participation in this project a harassment-free experience for everyone, regardless of level of experience, gender, gender identity and expression, sexual orientation, disability, personal appearance, body size, race, ethnicity, age, religion, or nationality.

Examples of unacceptable behavior by participants include:

- The use of sexualized language or imagery
- Personal attacks
- Trolling or insulting/derogatory comments
- Public or private harassment
- Publishing other's private information, such as physical or electronic addresses, without explicit permission
- Other unethical or unprofessional conduct

Project maintainers have the right and responsibility to remove, edit, or reject comments, commits, code, wiki edits, issues, and other contributions that are not aligned to this Code of Conduct, or to ban temporarily or permanently any contributor for other behaviors that they deem inappropriate, threatening, offensive, or harmful.

By adopting this Code of Conduct, project maintainers commit themselves to fairly and consistently applying these principles to every aspect of managing this project. Project maintainers who do not follow or enforce the Code of Conduct may be permanently removed from the project team.

This Code of Conduct applies both within project spaces and in public spaces when an individual is representing the project or its community.

Instances of abusive, harassing, or otherwise unacceptable behavior may be reported by contacting a project maintainer at team@php-http.org. All complaints will be reviewed and investigated and will result in a response that is deemed necessary and appropriate to the circumstances. Maintainers are obligated to maintain confidentiality with regard to the reporter of an incident.

This Code of Conduct is adapted from the Contributor Covenant, version 1.3.0, available at http://contributor-covenant.org/version/1/3/0/

4.15.2 Contributing

If you're here, you would like to contribute to this project and you're really welcome!

Bug Reports

If you find a bug or a documentation issue, please report it or even better: fix it :). If you report it, please be as precise as possible. Here is a little list of required information:

- Precise description of the bug
- Details of your environment (for example: OS, PHP version, installed extensions)
- Backtrace which might help identifying the bug

Security Issues

If you discover any security related issues, please contact us at security@php-http.org instead of submitting an issue on GitHub. This allows us to fix the issue and release a security hotfix without publicly disclosing the vulnerability.

Feature Requests

If you think a feature is missing, please report it or even better: implement it :). If you report it, describe the more precisely what you would like to see implemented and we will discuss what is the best approach for it. If you can do some research before submitting it and link the resources to your description, you're awesome! It will allow us to more easily understood/implement it.

Sending a Pull Request

If you're here, you are going to fix a bug or implement a feature and you're the best! To do it, first fork the repository, clone it and create a new branch with the following commands:

```
$ git clone git@github.com:your-name/repo-name.git
$ git checkout -b feature-or-bug-fix-description
```

Then install the dependencies through Composer:

```
$ composer install
```

Write code and tests. When you are ready, run the tests. (This is usually PHPUnit or PHPSpec)

```
$ composer test
```

When you are ready with the code, tested it and documented it, you can commit and push it with the following commands:

```
$ git commit -m "Feature or bug fix description"
$ git push origin feature-or-bug-fix-description
```

Note: Please write your commit messages in the imperative and follow the guidelines for clear and concise messages.

Then create a pull request on GitHub.

Please make sure that each individual commit in your pull request is meaningful. If you had to make multiple intermediate commits while developing, please squash them before submitting with the following commands (here, we assume you would like to squash 3 commits in a single one):

```
$ git rebase -i HEAD~3
```

If your branch conflicts with the master branch, you will need to rebase and re-push it with the following commands:

```
$ git remote add upstream git@github.com:orga/repo-name.git
$ git pull --rebase upstream master
$ git push -f origin feature-or-bug-fix-description
```

Coding Standard

This repository follows the PSR-2 standard and so, if you want to contribute, you must follow these rules.

Semver

We are trying to follow semver. When you are making BC breaking changes, please let us know why you think it is important. In this case, your patch can only be included in the next major version.

Contributor Code of Conduct

This project is released with a *Contributor Code of Conduct*. By participating in this project you agree to abide by its terms.

License

All of our packages are licensed under the *MIT license*.

4.15.3 Building the Documentation

We build the documentation with Sphinx. You could install it on your system or use Docker.

Install Sphinx

Install on Local Machine

The installation for Sphinx differs between system. See Sphinx installation page for details. When Sphinx is installed you need to install enchant (e.g. `sudo apt-get install enchant`).

Using Docker

If you are using docker. Run the following commands from the repository root.

```
$ docker run --rm -t -v "$PWD":/doc webplates/readthedocs build
$ docker run --rm -t -v "$PWD":/doc webplates/readthedocs check
```

Alternatively you can run the make commands as well:

```
$ docker run --rm -t -v "$PWD":/doc webplates/readthedocs make html
$ docker run --rm -t -v "$PWD":/doc webplates/readthedocs make spelling
```

To automatically rebuild the documentation upon change run:

```
$ docker run --rm -t -v "$PWD":/doc webplates/readthedocs watch
```

For more details see the readthedocs image documentation.

Build Documentation

Before building the documentation make sure to install all requirements.

```
$ pip install -r requirements.txt
```

To build the docs:

```
$ make html
$ make spelling
```

4.15.4 License

INDEX