Web Scraping with PHP

Second Edition

by Matthew Turland

a php[architect] guide

Web Scraping with PHP

2nd Edition

by Matthew Turland

php[architect] edition

Web Scraping with PHP

php[architect] edition published: August 2019

Print ISBN:	978-1-940111-67-4
PDF ISBN:	978-1-940111-68-1
ePub ISBN:	978-1-940111-60-8
Mobi ISBN	978-1-940111-70-4

Produced & Printed in the United States

Disclaimer

Although every effort has been made in the preparation of this book to ensure the accuracy of the information contained therein, this book is provided "as-is" and the publisher, the author(s), their distributors and retailers, as well as all affiliated, related or subsidiary parties take no responsibility for any inaccuracy and any and all damages caused, either directly or indirectly, by the use of such information. We have endeavored to properly provide trademark information on all companies and products mentioned in the book by the appropriate use of capitals. However, we cannot guarantee the accuracy of such information.

Written by
Matthew Turland

Managing Editor
Oscar Merida

Editor
Kara Ferguson

Published by
musketeers.me, LLC.
4627 University Dr
Fairfax, VA 22030 USA

240-348-5PHP (240-348-5747)
info@phparch.com
www.phparch.com

Table of Contents

Chapter 3

HTTP Streams Wrapper

At this point, you should be reasonably well-acquainted with some of the general concepts involved in using an HTTP client. The next chapters review some of the more popular mainstream client libraries, including common use cases and the advantages and disadvantages of each. The client covered in this chapter is the HTTP streams wrapper[1].

PHP 4.3 saw the addition of the Streams extension to the core. According to the related section of the PHP manual, the intention was to provide "a way of generalizing file, network, data compression, and other operations which share a common set of functions and uses." One of the concepts that streams introduced was the **wrapper**. The job of a wrapper is to define how a stream handles communications in a specific protocol or using a specific encoding. One such protocol for which a wrapper is available is HTTP.

The primary advantages of the HTTP streams wrapper are its ease of use and availability. Its API is minimal; it's easy and quick to get something simple working. The HTTP streams wrapper is part

[1] HTTP streams wrapper: http://php.net/wrappers.http

of the PHP core; thus, it's available in all PHP installations, as opposed to an optional extension that may not be, and has no other installation requirements.

The disadvantage of the HTTP streams wrapper is its minimal feature set. It gives you the ability to send HTTP requests without having to construct them entirely on your own (by specifying the body and optionally any headers you want to add) and access data in the response. That's about it. The ability to debug requests is one example of a feature it does not include at the time of this writing.

The fact that the wrapper is C code is a bit of a double-edged sword. On the positive side, there is a significant performance difference between C code and PHP code (though it's more noticeable in a high load environment). On the negative side, you have to either know C or depend on the community to deliver patches for any issues which may arise. This also applies to extensions written in C covered in later sections.

Simple Request and Response Handling

Here's a simple example of the HTTP streams wrapper in action.

```
$response = file_get_contents('http://example.com');
print_r($http_response_header);
```

Some notes:

- You must enable the `allow_url_fopen` PHP configuration setting for this to work, it's enabled in most environments.
- In this example, `file_get_contents()`[2] makes a `GET` request for the specified URL `http://example.com`.
- `$response` will contain the response body after the call to the `file_get_contents()` function completes.
- `$http_response_header` is implicitly populated with the HTTP response status line and headers after the `file_get_contents()` call because it uses the HTTP streams wrapper *within the current scope*.

While this example does work, it violates a core principle of good coding practices: no unexpected side effects. The origin of `$http_response_header` is not entirely obvious because PHP populates it implicitly. It's also more restrictive because the variable is unavailable outside the scope containing the call to `file_get_contents()`. Here's a better way to get access to the same data from the response headers.

```
$handle = fopen('http://example.com', 'r');
$response = stream_get_contents($handle);
$meta = stream_get_meta_data($handle);
print_r($meta['wrapper_data']);
```

[2] `file_get_contents()`: http://php.net/file_get_contents

Let's step through this.

1. `fopen()` opens a connection to the URL `http://example.com`; the resource `$handle` references a stream for that connection.
2. `stream_get_contents()` reads the remaining data on the stream pointed to by the `$handle` resource into `$response`.
3. `stream_get_meta_data()` reads metadata for the stream pointed to by the `$handle` resource into `$meta`.
4. At this point, `$meta['wrapper_data']` contains the same array as `$http_response_header` would within the current scope. You can call `stream_get_metadata()` with `$handle` in any scope in which the latter is available. This makes it more flexible than `$http_response_header`.

Stream Contexts and POST Requests

Another concept introduced by streams is the **context**[3], which is a set of configuration options used in a streams operation. `stream_context_create()` receives an associative array of context options and their corresponding values and returns a context. When using the HTTP streams wrapper, one use of contexts is to make `POST` requests, as the wrapper uses the `GET` method by default.

Listing 3.1

```
<?php
$context = stream_context_create([
    'http' => [
        'method' => 'POST',
        'header' => implode(
            "\r\n", [
                'Content-Type: application/x-www-form-urlencoded',
                'Referer: http://example.com'
            ]
        ),
        'content' => http_build_query([
            'param1' => 'value1',
            'param2' => 'value2'
        ]),
    ]
]);

$response = file_get_contents(
    'http://example.com/process', false, $context
);
```

[3] **context**: *http://php.net/context.http*

Here is a walk-through of the example in Listing 3.1.

- `'http'` is the streams wrapper used.
- `'POST'` is the HTTP method of the request.
- The `'header'` stream context setting references a string containing HTTP header key-value pairs, in this case for the `Content-Type` and `Referer` HTTP headers. The `Content-Type` header indicates the request body data is URL-encoded. If you need to set more than one custom header, you must separate them with a carriage return-line feed sequence (`"\r\n"` also known as a CRLF). `implode()`[4] is useful for this if you store key-value pairs for headers.
- `http_build_query()`[5] constructs the body of the request. It can also construct query strings of URLs for `GET` requests. One useful aspect is that it automatically handles encoding key-value pairs and delimiting them with an ampersand.
- `http://example.com/process` is the URL of the requested resource.
- `file_get_contents()`[6] executes the request using options from the context `$context` created using `stream_context_create()`[7].
- `$response` receives the body of the response returned by `file_get_contents()`.

Error Handling

Before PHP 5.3.0, an HTTP streams wrapper operation resulting in an HTTP error response (i.e., a 4xx or 5xx status code) emits a PHP-level warning. This warning contains the HTTP version, the status code, and the status code description. The function calls for such operations generally return `false` as a result, and leave you without a stream resource to check for more information. Listing 3.2 is an example of how to get what data you can.

Listing 3.2

```
<?php
function error_handler($errno, $errstr, $errfile, $errline, array $errcontext) {
    // $errstr will contain something like this:
    // fopen(http:_example.com/404): failed to open stream:
    // HTTP request failed! HTTP/1.0 404 Not Found
    if ($httperr = strstr($errstr, 'HTTP/')) {
        // $httperr will contain HTTP/1.0 404 Not Found in the case
        // of the above example, do something useful with that here
    }
}

set_error_handler('error_handler', E_WARNING);

```

[4] `implode()`: http://php.net/implode
[5] `http_build_query()`: http://php.net/http_build_query
[6] `file_get_contents()`: http://php.net/file_get_contents
[7] `stream_context_create()`: http://php.net/stream_context_create

```
// If the following statement fails, $stream will be assigned
// false and error_handler will be called automatically
$stream = fopen('http://example.com/404', 'r');

// If error_handler() does not terminate the script, control
// will be returned here once it completes its execution
restore_error_handler();
```

This situation improved somewhat in PHP 5.3 with the addition of the `ignore_errors` context setting. When you set this setting to `true`, PHP treats operations resulting in errors the same way as successful operations and emits no warnings. Listing 3.3 is an example of what it might look like.

Listing 3.3

```
<?php
$context = stream_context_create([
                                    'http' => [
                                       'ignore_errors' => true
                                    ]
                                ]);

$stream = fopen('http://example.com/404', 'r', false, $context);

// $stream will be a stream resource at this point regardless of
// the outcome of the operation
$body = stream_get_contents($stream);
$meta = stream_get_meta_data($stream);

// $meta['wrapper_data'][0] will equal something like 'HTTP/1.0 404 Not Found'
// at this point, with subsequent array elements being other headers
$response = explode(' ', $meta['wrapper_data'][0], 3);
list($version, $status, $description) = $response;

switch (substr($status, 0, 1)) {
    case '4':
    case '5':
        $result = false;
        break;

    default:
        $result = true;
}
```

HTTP Authentication

The HTTP stream wrapper has no context options for HTTP authentication credentials, but you can include credentials as part of the requested URL. See the example below.

```
$response = file_get_contents('http://username:password@example.com');
```

Note that credentials are not pre-encoded; the stream wrapper handles encoding transparently when making the request.

Also, this feature supports Basic HTTP authentication, but you must handle Digest authentication manually. As such, if support for Digest authentication is a desirable feature for your project, consider using a different client library, such as one of the others discussed in later chapters of this book.

More Options

Below are other stream context options for the HTTP streams wrapper that may prove useful.

- `'user_agent'` allows you to set the user agent string to use in the operation. You can also set it manually by specifying a value for the `User-Agent` header in the `'header'` context option value.
- `'max_redirects'` sets the highest number of redirects that the operation processes before assuming the application is misbehaving and terminating the request. This option is unavailable in PHP versions before 5.1.0 and uses a default value of `20`.
- `'follow_location'` became available in PHP 5.3.4. If you set `'max_redirects'` to `1`, the operation will not process redirects, but will emit an error. Setting `'follow_location'` to `0` suppresses this error.
- `'timeout'` sets a limit on the amount of time in seconds a read operation executes before it terminates. It defaults to the value of the `default_socket_timeout` PHP configuration setting.

All other features utilizing headers are accessible by specifying request headers in the `'header'` context option and checking either `$http_response_header` or the `'wrapper_data'` index of the array returned by `stream_get_meta_data()`[8] for response headers.

[8] `stream_get_meta_data()`: *http://php.net/stream_get_meta_data*

Index

E

G

H

S

T

U

W

X

Z

php[architect] Books

The php[architect] series of books cover topics relevant to modern PHP programming. We offer our books in both print and digital formats. Print copy price includes free shipping to the US. Books sold digitally are available to you DRM-free in PDF, ePub, or Mobi formats for viewing on any device that supports these.

To view the complete selection of books and order a copy of your own, please visit: *http://phparch.com/books/*.

- **Security Principles for PHP Applications**
  By Eric Mann
  ISBN: 978-1940111612

- **Docker for Developers, 2nd Edition**
  By Chris Tankersley
  ISBN: 978-1940111568 (Print edition)

- **What's Next? Professional Development Advice**
  Edited by Oscar Merida
  ISBN: 978-1940111513

- **Functional Programing in PHP, 2nd Edition**
  By: Simon Holywell
  ISBN: 978-1940111469

- **Web Security 2016**
  Edited by Oscar Merida
  ISBN: 978-1940111414

- **Building Exceptional Sites with WordPress & Thesis**
  By Peter MacIntyre
  ISBN: 978-1940111315

- **Integrating Web Services with OAuth and PHP**
  By Matthew Frost
  ISBN: 978-1940111261

- **Zend Framework 1 to 2 Migration Guide**
  By Bart McLeod
  ISBN: 978-1940111216

- **XML Parsing with PHP**
  By John M. Stokes
  ISBN: 978-1940111162

- **Zend PHP 5 Certification Study Guide, Third Edition**
  By Davey Shafik with Ben Ramsey
  ISBN: 978-1940111100

- **Mastering the SPL Library**
  By Joshua Thijssen
  ISBN: 978-1940111001