BY MATTHEW SETTER • 2021-06-29

5 Ways to Make HTTP Requests in PHP

HTTP requests are a hallmark of modern, web-based applications—especially in PHP. We have to interact with all manner of APIs and external services, such as Google Cloud, Facebook, and AWS, on almost a daily basis. I'd go so far as to say that it might well be one of the first things that you learn after you've mastered PHP's foundations. It was for me.

That said, like many modern software development languages, there's more than one way to make HTTP requests. So in this post, I'm going to introduce you to five of the most common options for making them in PHP. I'll show how to use them and cover some of their respective strengths and weaknesses. I won't explore them in intimate detail, rather give a broad introduction to each one.

The approaches that I'll cover are:

- PHP's HTTP/s stream wrapper
- PHP's cURL extension
- GuzzleHttp
- Httpful
- Symfony's HTTP client

With each one, we're going to step through a code example that will download a selection of 10 images of Kakadu National Park (in Australia) using Flickr's API, such as the one below.

"Sunset over Yellow Waters Billabong, Kakadu National Park, NT, Australia" by Geoff Whalan, which is licensed under CC BY-NC-ND 2.0.

Prerequisites

To complete the tutorial, you will need the following things:

- PHP 7.4 or newer (ideally version 8) with the cURL and OpenSSL extensions installed and enabled, and the allow_url_fopen runtime setting enabled
- Composer installed globally
- Git (required for Composer to work fully)
- Grep
- A Flickr account and a Flickr API key

Before we begin

Throughout this tutorial, we're going to create five PHP scripts, however, before we can get started we need to do a few things:

- Create a project directory
- Install PHP dotenv
- Add our Flickr API key in a *.env* file so that we don't accidentally store it in the example code

To complete the first two, run the three commands below.

```
mkdir -p php-http/photos
cd php-http
composer require vlucas/phpdotenv
```

Next, in the project directory create a new file named *.env*, and in that file insert the code below. Replace the placeholder, `<FLICKR API KEY>`, with your Flickr API key.

```
API_KEY=<FLICKR API KEY>
```

With that, we're ready to begin***!***

Core PHP Functionality & Extensions

Let's start with the options available as part of PHP's core functionality and extensions.

HTTP/S stream wrapper

Quoting the PHP manual:

> *Allows read-only access to files/resources via HTTP 1.0, using the HTTP GET method. A* `Host:` *header is sent with the request to handle name-based virtual hosts. If you have configured a user_agent string using your php.ini file or the stream context, it will also be included in the request.*

As streams are part of PHP's core, you don't have to do much to make use of their functionality. What's more, they integrate with many of PHP's core functions, such as fopen and file_get_contents. Consequently, you don't need to install a third-party library or custom extension to start using it.

On the flip side, they don't have as intuitive an interface, nor the helper utility methods that third-party libraries such as GuzzleHttp and Symfony's HTTP client, do. In addition, it:

- Only supports read (GET) **not** write (POST, PUT, and DELETE), requests
- Only supports a limited set of context options, such as the *user agent*, *redirect*, *headers*, *timeout*, and *proxy.* That said, you can do quite a bit with these.

Note: *Depending on your use case, being limited to HTTP/1.0 might be a problem. However, in the five examples in this tutorial, it isn't. If you'd like to learn more about the differences between HTTP 1.0, 1.1. and 2, check out this excellent post from Digital Ocean.*

That said, you can get up and running with it fairly quickly. Let's step through an example of how to do so. In the *php-http* directory, create an empty PHP file named *http-stream.php*, and in there, paste the code below.

```
<?php

declare(strict_types=1);

require_once('vendor/autoload.php');
```

```
$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();

const BASE_URL = 'https://api.flickr.com/services';
const PHOTO_URL = 'https://live.staticflickr.com/%s/%s_%s_b.jpg';
const PHOTOS_DIR = 'photos';

$apiKey=$_ENV['API_KEY'];

$opts = [
    'https' => [
        'max_redirects' => 3,
    ],
];
$queryString = http_build_query([
    'api_key' => $apiKey,
    'content_type' => 1,
    'format' => 'php_serial',
    'media' => 'photos',
    'method' => 'flickr.photos.search',
    'per_page' => 10,
    'safe_search' => 1,
    'text' => 'Kakadu National Park'
]);
$requestUri = sprintf(
    '%s/rest/?%s',
    BASE_URL,
    $queryString
);
$fp = fopen($requestUri, 'r');

$photoData = unserialize(stream_get_contents($fp));

foreach ($photoData['photos']['photo'] as $photoDatum) {
    printf("Downloading %s.jpg\n", $photoDatum['title']);
    $photoFile = file_get_contents(
        sprintf(
            PHOTO_URL,
            $photoDatum['server'],
            $photoDatum['id'],
            $photoDatum['secret']
        )
    );
    file_put_contents(PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg', $photoFile);
    printf("  Downloaded %s.jpg\n", $photoDatum['title']);
}

fclose($fp);
```

The code above first defines a set of constants that the script will make use of for interacting with Flickr's API and for saving the downloaded images.

After that, as stream wrappers provide no native way to build an HTTP query string, one is built using PHP's http_build_query function. Alternatively, we could use native string concatenation or sprintf, but `http_build_query` is cleaner and more intuitive.

After that, fopen is used to create a read-only HTTP connection to Flickr's API. Following that, stream_get_contents retrieves the contents of the request, which is then unserialized into a plain old PHP object.

In short, these two function calls make a transparent photo search request against Flickr's API and download a maximum of 10 images. If there are results for the request, they're then downloaded by calling file_get_contents, and written to the local filesystem using file_put_contents.

You can say many things about PHP, but it sure is extremely flexible and accommodating. The ability to use

existing PHP functions is a definite positive, even if, in this case, the function names aren't the most intuitive for what is being done. However, by using stream wrappers no additional dependencies are required, so you can get up and running fairly quickly.

Test that the code works

Now that the code's written, let's test it to make sure that it works. To do that, run the commands below.

```
php http-stream.php
```

You should see output similar to the example below.

```
Downloading Happy Snappy.jpg
  Downloaded Happy Snappy.jpg
Downloading Miyumba (Riversleigh World Heritage Area, North West Queensland).jpg
  Downloaded Miyumba (Riversleigh World Heritage Area, North West Queensland).jpg
Downloading Riversleigh Fossil Exploration: Site D (Riversleigh World Heritage Area, North West Queensland)
  Downloaded Riversleigh Fossil Exploration: Site D (Riversleigh World Heritage Area, North West Queenslan
Downloading Miyumba (Riversleigh World Heritage Area, North West Queensland).jpg
  Downloaded Miyumba (Riversleigh World Heritage Area, North West Queensland).jpg
Downloading Black-tailed Treecreeper (Climacteris melanurus melanurus).jpg
  Downloaded Black-tailed Treecreeper (Climacteris melanurus melanurus).jpg
Downloading Wilkins' Rock-Wallaby (Petrogale wilkinsi).jpg
  Downloaded Wilkins' Rock-Wallaby (Petrogale wilkinsi).jpg
Downloading Red-tailed Black Cockatoo.jpg
  Downloaded Red-tailed Black Cockatoo.jpg
Downloading 790515Tu-AyersRock-OU34a-bus-RSmith-ss.jpg
  Downloaded 790515Tu-AyersRock-OU34a-bus-RSmith-ss.jpg
Downloading Spotted Nightjar (Eurostopodus argus).jpg
  Downloaded Spotted Nightjar (Eurostopodus argus).jpg
Downloading Chestnut-quilled Rock Pigeon (Petrophassa rufipennis).jpg
  Downloaded Chestnut-quilled Rock Pigeon (Petrophassa rufipennis).jpg
```

After the command completes, if you look in the *photos* directory, you should see ten JPEG files.

PHP's cURL extension

Next, let's see how to do the same thing using PHP's cURL extension. Before we can begin, check that the cURL extension is available in your PHP runtime by running the following command:

```
php --ri curl | grep "cURL support => enabled"
```

If the extension is installed and enabled, you will see "*cURL support => enabled*" printed to the terminal. Create a new file in the project directory named *curl.php* and copy the code below into it.

```
<?php

declare(strict_types=1);

require_once('vendor/autoload.php');

$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();

// ... constants, the call to http_build_query, and definition of $requestUri, and $queryString

$ch = curl_init();
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
```

```
curl_setopt($ch, CURLOPT_URL, $requestUri);
curl_setopt($ch, CURLOPT_SSH_COMPRESSION, true);
curl_setopt_array($ch, [
        CURLOPT_RETURNTRANSFER => true,
        CURLOPT_URL => $requestUri
]);
$result = curl_exec($ch);
$photoData = unserialize($result);

curl_close($ch);

foreach ($photoData['photos']['photo'] as $photoDatum) {
    printf("Downloading %s.jpg\n", $photoDatum['title']);
    $file_url = sprintf(
        PHOTO_URL,
        $photoDatum['server'],
        $photoDatum['id'],
        $photoDatum['secret']
    );
    $destination_path = PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg';

    $fp = fopen($destination_path, "w");
    $ch = curl_init($file_url);
    curl_setopt($ch, CURLOPT_FILE, $fp);
    curl_exec($ch);
    $st_code = curl_getinfo($ch, CURLINFO_HTTP_CODE);
    curl_close($ch);
    fclose($fp);
    printf("  Downloaded %s.jpg\n", $photoDatum['title']);
}
```

Note: *In the example code above, I've deliberately not included the constants, the call to* `http_build_query` *, and the definition of* `$requestUri` *, as they're the same as in the stream example. Make sure you copy those from the stream example if you're following along locally.*

To begin, we have to call curl_init which initializes a cURL session. This provides a handle which the remaining code requires to make HTTP requests. After that, two options are set:

- `CURLOPT_RETURNTRANSFER` : This instructs cURL to return the result of a request instead of echoing it out immediately
- `CURLOPT_URL` : This sets the URL to request

Both of these options were set individually. However, we could have set them in one call using curl_setopt_array, as in the example below.

```
curl_setopt_array($ch, [
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_URL => $requestUri,
    CURLOPT_SSH_COMPRESSION => true,
]);
```

With cURL configured, curl_exec is called to make the request to the Flickr API, storing the result in `$result` . After that, the response is unserialized to a plain old PHP object, as in the previous example, and curl_close is called to close the cURL session and free all associated resources.

With the response returned, the code then iterates through it and downloads each of the 10 images. The process is reasonably similar to the streams example and could have used the download code verbatim. However, I wanted to show a more cURL-specific implementation.

In this example, the code opens a file handle for writing, downloads the file content and writes it to the new file, and then closes both the curl session and file handle after doing so.

Writing the image data to the file is simplified by calling `curl_setopt($ch, CURLOPT_FILE, $fp);`, which writes the image data to the file instead of STDOUT.

This example highlights how the cURL extension is more attuned to making HTTP requests than stream wrappers are. It also shows how the cURL extension is much more configurable and powerful, potentially. In addition, the predefined constants are quite logical and intuitive.

Another thing worth pointing out is that many third-party packages, such as GuzzleHttp and Symfony's HTTP client use the cURL extension internally. Consequently, it's a very battle-tested extension that you may already be using—even if you're not directly aware of it!

Note: *Make sure that you test that the code works.*

Third-party PHP HTTP packages

Now that we've looked at two native ways to make HTTP/S requests in PHP, let's see how to do so using three of the most well-known third-party packages.

GuzzleHttp

Created by Michael Dowling, GuzzleHttp is likely *the* most well-known and used of the three third-party libraries that I'm covering in this tutorial.

Here's a bit of trivia. Did you know that if you're using Laravel's HTTP client, then you're using GuzzleHttp under the hood? Check out the Laravel documentation.

To install the library, run the command below in the root directory of the project.

```
composer require guzzlehttp/guzzle:^7.0
```

Then, create a new file in the project directory named *guzzlehttp.php* and copy the code below into it.

```
<?php

declare(strict_types=1);

use GuzzleHttp\Client;

require_once('vendor/autoload.php');

$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();

//...pre-defined constants

$requestData = [
    'debug' => true,
    'query' => [
        'api_key' => $_ENV['API_KEY'],
        'content_type' => 1,
        'format' => 'php_serial',
        'media' => 'photos',
        'method' => 'flickr.photos.search',
        'per_page' => 10,
```

```
        'safe_search' => 1,
        'text' => 'Kakadu National Park',
    ],
];

$client = new Client([
    'base_uri' => BASE_URL,
]);

$response = $client->request('GET', '/services/rest', $requestData);
$photoData = unserialize($response->getBody()->getContents());

foreach ($photoData['photos']['photo'] as $photoDatum) {
    printf("Downloading %s.jpg\n", $photoDatum['title']);
    $fileUrl = sprintf(
        PHOTO_URL,
        $photoDatum['server'],
        $photoDatum['id'],
        $photoDatum['secret']
    );
    $filePath = PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg';
    $response = $client->get($fileUrl, ['sink' => $filePath]);
    printf("  Downloaded %s.jpg\n", $photoDatum['title']);
}
```

Note: *I've omitted the constants to shorten the example.*

This example is a little longer than the previous two, however, that's not necessarily a bad thing. It starts off by initializing an array that contains a set of request options:

- `debug`: This enables debug output; helpful for when things go wrong
- `query`: This associative array is used to build a query string for the request avoiding the need to call http_build_query

After that, a new `Client` object is initialized, with an associative array passed to its constructor, containing one key/value pair: `base_uri`. The value of this key is a base URL that is prepended to all requests using the object.

With the `Client` object initialized, a request is made by calling the object's `request` method, with three parameters passed to the call:

- The HTTP method to use (GET)
- The request's path (`/services/rest`)
- An associative array of request data (`$requestData`)

Following that, the results for the request are retrieved and unserialized into a PHP object representation by passing the results of `$response->getBody()->getContents()` to PHP's unserialize function; this is quite similar to the two previous examples.

After that, the images are downloaded to the photos directory. It's quite similar to the stream wrapper example, however, there is a key difference. Instead of using a combination of `file_get_contents` and `file_put_contents`, Guzzle's able to download the file with one call to `$client->get` because of the array provided as the second parameter to the call.

The array's `sink` option instructs Guzzle to write the response content in the file specified in `sink`'s value. If `sink` *wasn't* provided, the response would be stored in a temporary file using PHP's `php://temp` stream wrapper.

This example shows where a custom library can shine over PHP's native functionality. It has classes, methods, and method parameters that are logical, intuitive, and specifically designed for making HTTP/S requests; ones that simplify the effort required.

While longer than the previous examples, I'm generally more comfortable using Guzzle than using either PHP streams or the cURL extension.

Finally, here are two final, bonus points about Guzzle:

- It implements both PSR-7 (HTTP message interfaces) and PSR-18 (HTTP Client), making it interoperable with code and frameworks that make use of those standards
- The library is nicely documented. Good docs often make a library a pleasure to use.

Note: *Make sure that you test that the code works.*

Httpful

Next, let's take a look at Httpful. To quote the documentation:

> *Httpful is a simple, chainable, readable PHP library intended to make speaking HTTP sane. It lets the developer focus on interacting with APIs instead of sifting through curl set_opt pages and is an ideal PHP REST client.*

Here's a nice excerpt from its `Request` class that reinforces the quote:

> *There is an emphasis on readability without losing concise syntax.*

To install the library, run the command below.

```
composer require nategood/httpful
```

Then, create a new file in the project directory named *httpful.php* and copy the code below into it.

```
<?php

declare(strict_types=1);

require_once('vendor/autoload.php');

$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();

use Httpful\Request;

//...pre-defined constants

$queryString = http_build_query([
    'api_key' => $_ENV['API_KEY'],
    'content_type' => 1,
    'format' => 'php_serial',
```

```
    'media' => 'photos',
    'method' => 'flickr.photos.search',
    'per_page' => 10,
    'safe_search' => 1,
    'text' => 'Kakadu National Park'
]);
$response = Request::get(BASE_URL . '/rest?' . $queryString)
    ->followRedirects()
    ->withoutStrictSSL()
    ->send();
$photoData = unserialize($response->body);

foreach ($photoData['photos']['photo'] as $photoDatum) {
    printf("Downloading %s.jpg\n", $photoDatum['title']);
   $fileUrl = sprintf(PHOTO_URL, $photoDatum['server'], $photoDatum['id'], $photoDatum['secret']);
    $filePath = PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg';
    $file = Request::get($fileUrl)
        ->followRedirects()
        ->withoutStrictSSL()
        ->send();
    file_put_contents(PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg', $file->body);
    printf("  Downloaded %s.jpg\n", $photoDatum['title']);
}
```

Note: *As with the other examples, I've omitted code common to the earlier examples.*

The code starts by building a query string using `http_build_query`. This is then concatenated with the `BASE_URL` constant to create the request URL to send to Flickr. The URL is then passed to a static call to `Request`'s `get` method to make a GET request to Flickr's API.

`Request` is the class through which requests are made with the library. It supports several methods to make them, including `post`, `patch`, `delete`, `options`, and `head`, in addition to `get`.

It also includes many other supporting methods, including:

- `attach` for attaching files
- `addHeader/s` for adding headers to the request
- `basicAuth` / `digestAuth` for authentication
- `mime` for setting the Content-Type and Expected headers
- `followRedirects` for following redirect requests

As with the previous examples, the body of the response (`$response->body`) was then unserialized into a plain old PHP object, if the body was available.

The code for downloading the images is quite similar to the previous examples, making use of `file_put_contents` to write the files to the local filesystem, as the library does not provide a method to do that.

Httpful is, perhaps, the simplest to use of the three libraries. It has a fair bit in common with GuzzleHttp, but it also does things in its own way as well. It doesn't convert an associative array into a query string, which I would have liked, but the methods that it supports are both intuitive and logical, and the fluent interface makes it quick and easy to build and send a request.

One final point is worth noting: *the library is based around PHP's cURL extension*. This further reinforces just how trusted the cURL extension is **and** how you can build upon it to create more powerful and

expressive tooling.

Note: *Make sure that you test that the code works.*

Symfony's HTTP client

Finally, let's round out the article by looking at Symfony's HTTP client. To quote the documentation:

> *The HttpClient component is a low-level HTTP client with support for both PHP stream wrappers and cURL. It provides utilities to consume APIs and supports synchronous and asynchronous operations.*

To install it, run the command below.

```
composer require symfony/http-client
```

Then, create a new file in the project directory named *symfony.php* and copy the code below into it.

```
<?php

declare(strict_types=1);

require_once('vendor/autoload.php');

$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->load();

use Symfony\Component\HttpClient\HttpClient;

// Request constants

$requestData = [
    'debug' => true,
    'query' => [
        'api_key' => $_ENV['API_KEY'],
        'content_type' => 1,
        'format' => 'php_serial',
        'media' => 'photos',
        'method' => 'flickr.photos.search',
        'per_page' => 10,
        'safe_search' => 1,
        'text' => 'Kakadu National Park',
    ],
];

$client = HttpClient::create([
    'max_redirects' => 3,
]);
$response = $client->request(
    'GET',
    BASE_URL . '/rest',
    [
        'query' => $requestData['query']
    ]
);

echo $statusCode = $response->getStatusCode();
echo $contentType = $response->getHeaders()['content-type'][0];
$photoData = unserialize($response->getContent());

foreach ($photoData['photos']['photo'] as $photoDatum) {
    printf("Downloading %s.jpg\n", $photoDatum['title']);
    $fileUrl = sprintf(PHOTO_URL, $photoDatum['server'], $photoDatum['id'], $photoDatum['secret']);
    $filePath = PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg';
    $response = $client->request('GET', $fileUrl);
```

```
    file_put_contents(PHOTOS_DIR . '/' . $photoDatum['title'] . '.jpg', $response->getContent());
    printf("  Downloaded %s.jpg\n", $photoDatum['title']);
}
```

The code starts, similarly to the GuzzleHttp example, by defining an associative array of request data. As before, it defines `debug` to enable debug mode and `query` to build the query string from.

After that, it calls `HttpClient`'s static `create` method to instantiate an HTTP client for making requests, passing to it some default request options, ones that will be used on all requests.

After that, `$client`'s `request` method is called to make the request, where the HTTP method, `GET`, the base request URL, `BASE_URL`, and query data are supplied, with the response being stored in `$response`.

Here, we departed a little from the previous examples by first retrieving the status code of the response (`getStatusCode`) and the content-type header (`$response->getHeaders()['content-type'][0]`), before retrieving and unserializing the response body into a plain old PHP object (`$response->getContent()`).

Following that, how the images are downloaded is, again, largely similar to the previous examples. As with the Httpful example, it makes use of `file_put_contents` to write the files to the local filesystem, as there is no method in the library that does it for us.

After a bit of usage, I found Symfony's HTTP client comparable to GuzzleHttp in terms of functionality, expressiveness, and intuitiveness. The class, method, and parameter names make sense, and the library is easy to use, e.g., it simplifies creating a query string from an associative array. Consequently, it greatly simplifies the effort required in making HTTP/S requests.

Note: *Make sure that you test that the code works.*

That's 5 ways to make HTTP requests in PHP

We looked at how to make them using PHP's HTTP/S stream wrapper, PHP's cURL extension, GuzzleHttp, Httpful, and Symfony's HTTP client. In addition to a fairly simplistic usage example for each one, we briefly considered some of the merits of each approach.

While each of the 5 options covered here can be used in any project, from experience, the larger and more complex your project the better you *may* well be served by using one of the third-party packages. If so, my recommendation is GuzzleHttp.

In these instances, the third-party packages will likely improve both efficiency and enjoyment, resulting in considerable developer time saved. That said, while not wanting to talk them down, PHP's cURL extension or HTTP/S stream wrapper may be all that you need. Sometimes less is more!

One last thing: thank you to all those who provided feedback to my Tweet when I was researching this article! It helped me immeasurably in both writing and refining it.

Happy hacking! I can't wait to see what you build with them.

Matthew Setter is a PHP Editor in the Twilio Voices team and (naturally) a PHP developer. He's also the author of Mezzio Essentials and Docker Essentials. When he's not writing PHP code, he's editing great PHP articles here at Twilio. You can find him at msetter@twilio.com, Twitter, and GitHub.

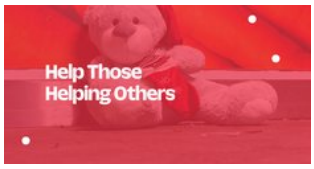
REVIEWERS

Miguel Grinberg

Marcus Battle

Matthew Gilliard

Related Posts



Help Those Helping Others

Dec 06, 2022

In this tutorial, you'll learn how to build an application to help anyone quickly donate to charities and nonprofits using Slim Framework, Vue.js, and Tailwin...

Learn the Maasai Language the Fun Way

Oct 20, 2022

In this tutorial, you will build an app that helps you learn the Maasai language in an easy and fun way with WhatsApp.

Pub/Sub in Laravel - An In-depth Understanding

Oct 20, 2022

In this tutorial, you will learn about Pub/Sub, a message-driven software design pattern and how to implement it in Laravel.

HTTP PHP

Search

POSTS BY STACK

JAVA PHP RUBY PYTHON .NET SWIFT ARDUINO GO JAVASCRIPT

POSTS BY PRODUCT

EMAIL SMS VOICE TWILIO CLIENT MMS VIDEO CONVERSATIONS TASK ROUTER VERIFY FLEX SIP IOT STUDIO

CATEGORIES

Code, Tutorials and Hacks
Customer Highlights
Developers Drawing The Owl
Enterprise
Life Inside: We Build At Twilio
News
Stories From The Road

SIGN UP AND START BUILDING

Not ready yet? Talk to an expert.