# Guzzle

***Release***

January 31, 2019

Contents

Guzzle is a PHP HTTP client that makes it easy to send HTTP requests and trivial to integrate with web services.

- Simple interface for building query strings, POST requests, streaming large uploads, streaming large downloads, using HTTP cookies, uploading JSON data, etc...
- Can send both synchronous and asynchronous requests using the same interface.
- Uses PSR-7 interfaces for requests, responses, and streams. This allows you to utilize other PSR-7 compatible libraries with Guzzle.
- Abstracts away the underlying HTTP transport, allowing you to write environment and transport agnostic code; i.e., no hard dependency on cURL, PHP streams, sockets, or non-blocking event loops.
- Middleware system allows you to augment and compose client behavior.

```php
$client = new GuzzleHttp\Client();
$res = $client->request('GET', 'https://api.github.com/user', [
    'auth' => ['user', 'pass']
]);
echo $res->getStatusCode();
// "200"
echo $res->getHeader('content-type')[0];
// 'application/json; charset=utf8'
echo $res->getBody();
// {"type":"User"...'

// Send an asynchronous request.
$request = new \GuzzleHttp\Psr7\Request('GET', 'http://httpbin.org');
$promise = $client->sendAsync($request)->then(function ($response) {
    echo 'I completed! ' . $response->getBody();
});
$promise->wait();
```

CHAPTER 1

# User Guide

## Overview

### Requirements

1. PHP 5.5.0
2. To use the PHP stream handler, `allow_url_fopen` must be enabled in your system's php.ini.
3. To use the cURL handler, you must have a recent version of cURL >= 7.19.4 compiled with OpenSSL and zlib.

**Note:** Guzzle no longer requires cURL in order to send HTTP requests. Guzzle will use the PHP stream wrapper to send HTTP requests if cURL is not installed. Alternatively, you can provide your own HTTP handler used to send requests.

### Installation

The recommended way to install Guzzle is with Composer. Composer is a dependency management tool for PHP that allows you to declare the dependencies your project needs and installs them into your project.

```
# Install Composer
curl -sS https://getcomposer.org/installer | php
```

You can add Guzzle as a dependency using the composer.phar CLI:

```
php composer.phar require guzzlehttp/guzzle:~6.0
```

Alternatively, you can specify Guzzle as a dependency in your project's existing composer.json file:

```
 {
   "require": {
      "guzzlehttp/guzzle": "~6.0"
   }
}
```

After installing, you need to require Composer's autoloader:

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

You can find out more on how to install Composer, configure autoloading, and other best-practices for defining dependencies at getcomposer.org.

### Bleeding edge

During your development, you can keep up with the latest changes on the master branch by setting the version requirement for Guzzle to `~6.0@dev`.

```
{
   "require": {
      "guzzlehttp/guzzle": "~6.0@dev"
   }
}
```

## License

Licensed using the MIT license.

> Copyright (c) 2015 Michael Dowling <https://github.com/mtdowling>
>
> Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:
>
> The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.
>
> THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

## Contributing

### Guidelines

1. Guzzle utilizes PSR-1, PSR-2, PSR-4, and PSR-7.
2. Guzzle is meant to be lean and fast with very few dependencies. This means that not every feature request will be accepted.
3. Guzzle has a minimum PHP version requirement of PHP 5.5. Pull requests must not require a PHP version greater than PHP 5.5 unless the feature is only utilized conditionally.
4. All pull requests must include unit tests to ensure the change works as expected and to prevent regressions.

### Running the tests

In order to contribute, you'll need to checkout the source from GitHub and install Guzzle's dependencies using Composer:

```
git clone https://github.com/guzzle/guzzle.git
cd guzzle && curl -s http://getcomposer.org/installer | php && ./composer.phar install --dev
```

Guzzle is unit tested with PHPUnit. Run the tests using the Makefile:

```
make test
```

**Note:** You'll need to install node.js v0.5.0 or newer in order to perform integration tests on Guzzle's HTTP handlers.

### Reporting a security vulnerability

We want to ensure that Guzzle is a secure HTTP client library for everyone. If you've discovered a security vulnerability in Guzzle, we appreciate your help in disclosing it to us in a responsible manner.

Publicly disclosing a vulnerability can put the entire community at risk. If you've discovered a security concern, please email us at security@guzzlephp.org. We'll work with you to make sure that we understand the scope of the issue, and that we fully address your concern. We consider correspondence sent to security@guzzlephp.org our highest priority, and work to address any issues that arise as quickly as possible.

After a security vulnerability has been corrected, a security hotfix release will be deployed as soon as possible.

## Quickstart

This page provides a quick introduction to Guzzle and introductory examples. If you have not already installed, Guzzle, head over to the *Installation* page.

### Making a Request

You can send requests with Guzzle using a `GuzzleHttp\ClientInterface` object.

#### Creating a Client

```
use GuzzleHttp\Client;

$client = new Client([
    // Base URI is used with relative requests
    'base_uri' => 'http://httpbin.org',
    // You can set any number of default request options.
    'timeout'  => 2.0,
]);
```

Clients are immutable in Guzzle 6, which means that you cannot change the defaults used by a client after it's created.

The client constructor accepts an associative array of options:

**`base_uri`** (string|UriInterface) Base URI of the client that is merged into relative URIs. Can be a string or instance of UriInterface. When a relative URI is provided to a client, the client will combine the base URI with the relative URI using the rules described in RFC 3986, section 2.

```
    // Create a client with a base URI
    $client = new GuzzleHttp\Client(['base_uri' => 'https://foo.com/api/']);
    // Send a request to https://foo.com/api/test
    $response = $client->request('GET', 'test');
    // Send a request to https://foo.com/root
    $response = $client->request('GET', '/root');
```

Don't feel like reading RFC 3986? Here are some quick examples on how a `base_uri` is resolved with another URI.

| base_uri | URI | Result |
|---|---|---|
| `http://foo.com` | `/bar` | `http://foo.com/bar` |
| `http://foo.com/foo` | `/bar` | `http://foo.com/bar` |
| `http://foo.com/foo` | `bar` | `http://foo.com/bar` |
| `http://foo.com/foo/` | `bar` | `http://foo.com/foo/bar` |
| `http://foo.com` | `http://baz.com` | `http://baz.com` |
| `http://foo.com/?bar` | `bar` | `http://foo.com/bar` |

**handler** (callable) Function that transfers HTTP requests over the wire. The function is called with a `Psr7\Http\Message\RequestInterface` and array of transfer options, and must return a `GuzzleHttp\Promise\PromiseInterface` that is fulfilled with a `Psr7\Http\Message\ResponseInterface` on success. `handler` is a constructor only option that cannot be overridden in per/request options.

**...** (mixed) All other options passed to the constructor are used as default request options with every request created by the client.

### Sending Requests

Magic methods on the client make it easy to send synchronous requests:

```
$response = $client->get('http://httpbin.org/get');
$response = $client->delete('http://httpbin.org/delete');
$response = $client->head('http://httpbin.org/get');
$response = $client->options('http://httpbin.org/get');
$response = $client->patch('http://httpbin.org/patch');
$response = $client->post('http://httpbin.org/post');
$response = $client->put('http://httpbin.org/put');
```

You can create a request and then send the request with the client when you're ready:

```
use GuzzleHttp\Psr7\Request;

$request = new Request('PUT', 'http://httpbin.org/put');
$response = $client->send($request, ['timeout' => 2]);
```

Client objects provide a great deal of flexibility in how request are transferred including default request options, default handler stack middleware that are used by each request, and a base URI that allows you to send requests with relative URIs.

You can find out more about client middleware in the Handlers and Middleware page of the documentation.

### Async Requests

You can send asynchronous requests using the magic methods provided by a client:

```
$promise = $client->getAsync('http://httpbin.org/get');
$promise = $client->deleteAsync('http://httpbin.org/delete');
$promise = $client->headAsync('http://httpbin.org/get');
$promise = $client->optionsAsync('http://httpbin.org/get');
$promise = $client->patchAsync('http://httpbin.org/patch');
$promise = $client->postAsync('http://httpbin.org/post');
$promise = $client->putAsync('http://httpbin.org/put');
```

You can also use the *sendAsync()* and *requestAsync()* methods of a client:

```
use GuzzleHttp\Psr7\Request;

// Create a PSR-7 request object to send
$headers = ['X-Foo' => 'Bar'];
$body = 'Hello!';
$request = new Request('HEAD', 'http://httpbin.org/head', $headers, $body);
$promise = $client->sendAsync($request);

// Or, if you don't need to pass in a request instance:
$promise = $client->requestAsync('GET', 'http://httpbin.org/get');
```

The promise returned by these methods implements the Promises/A+ spec, provided by the Guzzle promises library. This means that you can chain `then()` calls off of the promise. These then calls are either fulfilled with a successful `Psr\Http\Message\ResponseInterface` or rejected with an exception.

```
use Psr\Http\Message\ResponseInterface;
use GuzzleHttp\Exception\RequestException;

$promise = $client->requestAsync('GET', 'http://httpbin.org/get');
$promise->then(
    function (ResponseInterface $res) {
        echo $res->getStatusCode() . "\n";
    },
    function (RequestException $e) {
        echo $e->getMessage() . "\n";
        echo $e->getRequest()->getMethod();
    }
);
```

### Concurrent requests

You can send multiple requests concurrently using promises and asynchronous requests.

```
use GuzzleHttp\Client;
use GuzzleHttp\Promise;

$client = new Client(['base_uri' => 'http://httpbin.org/']);

// Initiate each request but do not block
$promises = [
    'image' => $client->getAsync('/image'),
    'png'   => $client->getAsync('/image/png'),
    'jpeg'  => $client->getAsync('/image/jpeg'),
    'webp'  => $client->getAsync('/image/webp')
];

// Wait for the requests to complete; throws a ConnectException
// if any of the requests fail
$responses = Promise\unwrap($promises);

// Wait for the requests to complete, even if some of them fail
$responses = Promise\settle($promises)->wait();

// You can access each response using the key of the promise
echo $responses['image']->getHeader('Content-Length')[0]
echo $responses['png']->getHeader('Content-Length')[0]
```

You can use the `GuzzleHttp\Pool` object when you have an indeterminate amount of requests you wish to send.

```
use GuzzleHttp\Client;
use GuzzleHttp\Exception\RequestException;
use GuzzleHttp\Pool;
use GuzzleHttp\Psr7\Request;
use GuzzleHttp\Psr7\Response;

$client = new Client();

$requests = function ($total) {
    $uri = 'http://127.0.0.1:8126/guzzle-server/perf';
    for ($i = 0; $i < $total; $i++) {
        yield new Request('GET', $uri);
    }
};

$pool = new Pool($client, $requests(100), [
    'concurrency' => 5,
    'fulfilled' => function (Response $response, $index) {
        // this is delivered each successful response
    },
    'rejected' => function (RequestException $reason, $index) {
        // this is delivered each failed request
    },
]);

// Initiate the transfers and create a promise
$promise = $pool->promise();

// Force the pool of requests to complete.
$promise->wait();
```

Or using a closure that will return a promise once the pool calls the closure.

```
$client = new Client();

$requests = function ($total) use ($client) {
    $uri = 'http://127.0.0.1:8126/guzzle-server/perf';
    for ($i = 0; $i < $total; $i++) {
        yield function() use ($client, $uri) {
            return $client->getAsync($uri);
        };
    }
};

$pool = new Pool($client, $requests(100));
```

## Using Responses

In the previous examples, we retrieved a `$response` variable or we were delivered a response from a promise. The response object implements a PSR-7 response, `Psr\Http\Message\ResponseInterface`, and contains lots of helpful information.

You can get the status code and reason phrase of the response:

```
$code = $response->getStatusCode(); // 200
$reason = $response->getReasonPhrase(); // OK
```

You can retrieve headers from the response:

```
// Check if a header exists.
if ($response->hasHeader('Content-Length')) {
    echo "It exists";
}

// Get a header from the response.
echo $response->getHeader('Content-Length')[0];

// Get all of the response headers.
foreach ($response->getHeaders() as $name => $values) {
    echo $name . ': ' . implode(', ', $values) . "\r\n";
}
```

The body of a response can be retrieved using the `getBody` method. The body can be used as a string, cast to a string, or used as a stream like object.

```
$body = $response->getBody();
// Implicitly cast the body to a string and echo it
echo $body;
// Explicitly cast the body to a string
$stringBody = (string) $body;
// Read 10 bytes from the body
$tenBytes = $body->read(10);
// Read the remaining contents of the body as a string
$remainingBytes = $body->getContents();
```

## Query String Parameters

You can provide query string parameters with a request in several ways.

You can set query string parameters in the request's URI:

```
$response = $client->request('GET', 'http://httpbin.org?foo=bar');
```

You can specify the query string parameters using the `query` request option as an array.

```
$client->request('GET', 'http://httpbin.org', [
    'query' => ['foo' => 'bar']
]);
```

Providing the option as an array will use PHP's `http_build_query` function to format the query string.

And finally, you can provide the `query` request option as a string.

```
$client->request('GET', 'http://httpbin.org', ['query' => 'foo=bar']);
```

## Uploading Data

Guzzle provides several methods for uploading data.

You can send requests that contain a stream of data by passing a string, resource returned from `fopen`, or an instance of a `Psr\Http\Message\StreamInterface` to the `body` request option.

```
// Provide the body as a string.
$r = $client->request('POST', 'http://httpbin.org/post', [
    'body' => 'raw data'
```

```
]);

// Provide an fopen resource.
$body = fopen('/path/to/file', 'r');
$r = $client->request('POST', 'http://httpbin.org/post', ['body' => $body]);

// Use the stream_for() function to create a PSR-7 stream.
$body = \GuzzleHttp\Psr7\stream_for('hello!');
$r = $client->request('POST', 'http://httpbin.org/post', ['body' => $body]);
```

An easy way to upload JSON data and set the appropriate header is using the `json` request option:

```
$r = $client->request('PUT', 'http://httpbin.org/put', [
    'json' => ['foo' => 'bar']
]);
```

### POST/Form Requests

In addition to specifying the raw data of a request using the `body` request option, Guzzle provides helpful abstractions over sending POST data.

#### Sending form fields

Sending `application/x-www-form-urlencoded` POST requests requires that you specify the POST fields as an array in the `form_params` request options.

```
$response = $client->request('POST', 'http://httpbin.org/post', [
    'form_params' => [
        'field_name' => 'abc',
        'other_field' => '123',
        'nested_field' => [
            'nested' => 'hello'
        ]
    ]
]);
```

#### Sending form files

You can send files along with a form (`multipart/form-data` POST requests), using the `multipart` request option. `multipart` accepts an array of associative arrays, where each associative array contains the following keys:

- name: (required, string) key mapping to the form field name.
- contents: (required, mixed) Provide a string to send the contents of the file as a string, provide an fopen resource to stream the contents from a PHP stream, or provide a `Psr\Http\Message\StreamInterface` to stream the contents from a PSR-7 stream.

```
$response = $client->request('POST', 'http://httpbin.org/post', [
    'multipart' => [
        [
            'name'     => 'field_name',
            'contents' => 'abc'
        ],
        [
            'name'     => 'file_name',
```

```
            'contents' => fopen('/path/to/file', 'r')
        ],
        [
            'name'     => 'other_file',
            'contents' => 'hello',
            'filename' => 'filename.txt',
            'headers'  => [
                'X-Foo' => 'this is an extra header to include'
            ]
        ]
    ]
]);
```

## Cookies

Guzzle can maintain a cookie session for you if instructed using the `cookies` request option. When sending a request, the `cookies` option must be set to an instance of `GuzzleHttp\Cookie\CookieJarInterface`.

```
// Use a specific cookie jar
$jar = new \GuzzleHttp\Cookie\CookieJar;
$r = $client->request('GET', 'http://httpbin.org/cookies', [
    'cookies' => $jar
]);
```

You can set `cookies` to `true` in a client constructor if you would like to use a shared cookie jar for all requests.

```
// Use a shared client cookie jar
$client = new \GuzzleHttp\Client(['cookies' => true]);
$r = $client->request('GET', 'http://httpbin.org/cookies');
```

Different implementations exist for the `GuzzleHttp\Cookie\CookieJarInterface`:

- The `GuzzleHttp\Cookie\CookieJar` class stores cookies as an array.
- The `GuzzleHttp\Cookie\FileCookieJar` class persists non-session cookies using a JSON formatted file.
- The `GuzzleHttp\Cookie\SessionCookieJar` class persists cookies in the client session.

You can manually set cookies into a cookie jar with the named constructor `fromArray(array $cookies, $domain)`.

```
$jar = \GuzzleHttp\Cookie\CookieJar::fromArray(
    [
        'some_cookie' => 'foo',
        'other_cookie' => 'barbaz1234'
    ],
    'example.org'
);
```

You can get a cookie by its name with the `getCookieByName($name)` method which returns a `GuzzleHttp\Cookie\SetCookie` instance.

```
$cookie = $jar->getCookieByName('some_cookie');

$cookie->getValue(); // 'foo'
$cookie->getDomain(); // 'example.org'
$cookie->getExpires(); // expiration date as a Unix timestamp
```

The cookies can be also fetched into an array thanks to the *toArray()* method. The `GuzzleHttp\Cookie\CookieJarInterface` interface extends `Traversable` so it can be iterated in a foreach loop.

## Redirects

Guzzle will automatically follow redirects unless you tell it not to. You can customize the redirect behavior using the `allow_redirects` request option.

- Set to `true` to enable normal redirects with a maximum number of 5 redirects. This is the default setting.
- Set to `false` to disable redirects.
- Pass an associative array containing the 'max' key to specify the maximum number of redirects and optionally provide a 'strict' key value to specify whether or not to use strict RFC compliant redirects (meaning redirect POST requests with POST requests vs. doing what most browsers do which is redirect POST requests with GET requests).

```
$response = $client->request('GET', 'http://github.com');
echo $response->getStatusCode();
// 200
```

The following example shows that redirects can be disabled.

```
$response = $client->request('GET', 'http://github.com', [
    'allow_redirects' => false
]);
echo $response->getStatusCode();
// 301
```

## Exceptions

### Tree View

The following tree view describes how the Guzzle Exceptions depend on each other.

```
. \RuntimeException
-- SeekException (implements GuzzleException)
-- TransferException (implements GuzzleException)
    -- RequestException
        -- BadResponseException
        |   -- ServerException
        |   -- ClientException
        -- ConnectException
        -- TooManyRedirectsException
```

Guzzle throws exceptions for errors that occur during a transfer.

- In the event of a networking error (connection timeout, DNS errors, etc.), a `GuzzleHttp\Exception\RequestException` is thrown. This exception extends from `GuzzleHttp\Exception\TransferException`. Catching this exception will catch any exception that can be thrown while transferring requests.

```
    use GuzzleHttp\Psr7;
    use GuzzleHttp\Exception\RequestException;

    try {
        $client->request('GET', 'https://github.com/_abc_123_404');
```

```
} catch (RequestException $e) {
    echo Psr7\str($e->getRequest());
    if ($e->hasResponse()) {
        echo Psr7\str($e->getResponse());
    }
}
```

- A `GuzzleHttp\Exception\ConnectException` exception is thrown in the event of a networking error. This exception extends from `GuzzleHttp\Exception\RequestException`.
- A `GuzzleHttp\Exception\ClientException` is thrown for 400 level errors if the `http_errors` request option is set to true. This exception extends from `GuzzleHttp\Exception\BadResponseException` and `GuzzleHttp\Exception\BadResponseException` extends from `GuzzleHttp\Exception\RequestException`.

```
use GuzzleHttp\Psr7;
use GuzzleHttp\Exception\ClientException;

try {
    $client->request('GET', 'https://github.com/_abc_123_404');
} catch (ClientException $e) {
    echo Psr7\str($e->getRequest());
    echo Psr7\str($e->getResponse());
}
```

- A `GuzzleHttp\Exception\ServerException` is thrown for 500 level errors if the `http_errors` request option is set to true. This exception extends from `GuzzleHttp\Exception\BadResponseException`.
- A `GuzzleHttp\Exception\TooManyRedirectsException` is thrown when too many redirects are followed. This exception extends from `GuzzleHttp\Exception\RequestException`.

All of the above exceptions extend from `GuzzleHttp\Exception\TransferException`.

## Environment Variables

Guzzle exposes a few environment variables that can be used to customize the behavior of the library.

**`GUZZLE_CURL_SELECT_TIMEOUT`** Controls the duration in seconds that a curl_multi_* handler will use when selecting on curl handles using `curl_multi_select()`. Some systems have issues with PHP's implementation of `curl_multi_select()` where calling this function always results in waiting for the maximum duration of the timeout.

**`HTTP_PROXY`** Defines the proxy to use when sending requests using the "http" protocol.

Note: because the HTTP_PROXY variable may contain arbitrary user input on some (CGI) environments, the variable is only used on the CLI SAPI. See https://httpoxy.org for more information.

**`HTTPS_PROXY`** Defines the proxy to use when sending requests using the "https" protocol.

**`NO_PROXY`** Defines URLs for which a proxy should not be used. See *proxy* for usage.

### Relevant ini Settings

Guzzle can utilize PHP ini settings when configuring clients.

**`openssl.cafile`** Specifies the path on disk to a CA file in PEM format to use when sending requests over "https". See: https://wiki.php.net/rfc/tls-peer-verification#phpini_defaults

# Request Options

You can customize requests created and transferred by a client using **request options**. Request options control various aspects of a request including, headers, query string parameters, timeout settings, the body of a request, and much more.

All of the following examples use the following client:

```
$client = new GuzzleHttp\Client(['base_uri' => 'http://httpbin.org']);
```

## allow_redirects

**Summary** Describes the redirect behavior of a request

**Types**

- bool
- array

**Default**

```
[
    'max'             => 5,
    'strict'          => false,
    'referer'         => false,
    'protocols'       => ['http', 'https'],
    'track_redirects' => false
]
```

**Constant** `GuzzleHttp\RequestOptions::ALLOW_REDIRECTS`

Set to `false` to disable redirects.

```
$res = $client->request('GET', '/redirect/3', ['allow_redirects' => false]);
echo $res->getStatusCode();
// 302
```

Set to `true` (the default setting) to enable normal redirects with a maximum number of 5 redirects.

```
$res = $client->request('GET', '/redirect/3');
echo $res->getStatusCode();
// 200
```

You can also pass an associative array containing the following key value pairs:

- max: (int, default=5) maximum number of allowed redirects.
- strict: (bool, default=false) Set to true to use strict redirects. Strict RFC compliant redirects mean that POST redirect requests are sent as POST requests vs. doing what most browsers do which is redirect POST requests with GET requests.
- referer: (bool, default=false) Set to true to enable adding the Referer header when redirecting.
- protocols: (array, default=['http', 'https']) Specified which protocols are allowed for redirect requests.
- on_redirect: (callable) PHP callable that is invoked when a redirect is encountered. The callable is invoked with the original request and the redirect response that was received. Any return value from the on_redirect function is ignored.

- track_redirects: (bool) When set to `true`, each redirected URI and status code encountered will be tracked in the `X-Guzzle-Redirect-History` and `X-Guzzle-Redirect-Status-History` headers respectively. All URIs and status codes will be stored in the order which the redirects were encountered.

  Note: When tracking redirects the `X-Guzzle-Redirect-History` header will exclude the initial request's URI and the `X-Guzzle-Redirect-Status-History` header will exclude the final status code.

```
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use Psr\Http\Message\UriInterface;

$onRedirect = function(
    RequestInterface $request,
    ResponseInterface $response,
    UriInterface $uri
) {
    echo 'Redirecting! ' . $request->getUri() . ' to ' . $uri . "\n";
};

$res = $client->request('GET', '/redirect/3', [
    'allow_redirects' => [
        'max'             => 10,        // allow at most 10 redirects.
        'strict'          => true,      // use "strict" RFC compliant redirects.
        'referer'         => true,      // add a Referer header
        'protocols'       => ['https'], // only allow https URLs
        'on_redirect'     => $onRedirect,
        'track_redirects' => true
    ]
]);

echo $res->getStatusCode();
// 200

echo $res->getHeaderLine('X-Guzzle-Redirect-History');
// http://first-redirect, http://second-redirect, etc...

echo $res->getHeaderLine('X-Guzzle-Redirect-Status-History');
// 301, 302, etc...
```

**Warning:** This option only has an effect if your handler has the `GuzzleHttp\Middleware::redirect` middleware. This middleware is added by default when a client is created with no handler, and is added by default when creating a handler with `GuzzleHttp\HandlerStack::create`.

## auth

**Summary** Pass an array of HTTP authentication parameters to use with the request. The array must contain the username in index [0], the password in index [1], and you can optionally provide a built-in authentication type in index [2]. Pass `null` to disable authentication for a request.

**Types**

- array
- string
- null

**Default** None

**Constant** `GuzzleHttp\RequestOptions::AUTH`

The built-in authentication types are as follows:

**basic** Use basic HTTP authentication in the `Authorization` header (the default setting used if none is specified).

```
$client->request('GET', '/get', ['auth' => ['username', 'password']]);
```

**digest** Use digest authentication (must be supported by the HTTP handler).

```
$client->request('GET', '/get', [
    'auth' => ['username', 'password', 'digest']
]);
```

---

**Note:** This is currently only supported when using the cURL handler, but creating a replacement that can be used with any HTTP handler is planned.

---

**ntlm** Use Microsoft NTLM authentication (must be supported by the HTTP handler).

```
$client->request('GET', '/get', [
    'auth' => ['username', 'password', 'ntlm']
]);
```

---

**Note:** This is currently only supported when using the cURL handler.

---

## body

**Summary** The `body` option is used to control the body of an entity enclosing request (e.g., PUT, POST, PATCH).

**Types**

- string
- `fopen()` resource
- `Psr\Http\Message\StreamInterface`

**Default** None

**Constant** `GuzzleHttp\RequestOptions::BODY`

This setting can be set to any of the following types:

- string

```
// You can send requests that use a string as the message body.
$client->request('PUT', '/put', ['body' => 'foo']);
```

- resource returned from `fopen()`

```
// You can send requests that use a stream resource as the body.
$resource = fopen('http://httpbin.org', 'r');
$client->request('PUT', '/put', ['body' => $resource]);
```

- `Psr\Http\Message\StreamInterface`

```
    // You can send requests that use a Guzzle stream object as the body
    $stream = GuzzleHttp\Psr7\stream_for('contents...');
    $client->request('POST', '/post', ['body' => $stream]);
```

**Note:** This option cannot be used with `form_params`, `multipart`, or `json`

## cert

**Summary** Set to a string to specify the path to a file containing a PEM formatted client side certificate. If a password is required, then set to an array containing the path to the PEM file in the first array element followed by the password required for the certificate in the second array element.

**Types**

- string
- array

**Default** None

**Constant** `GuzzleHttp\RequestOptions::CERT`

```
$client->request('GET', '/', ['cert' => ['/path/server.pem', 'password']]);
```

## cookies

**Summary** Specifies whether or not cookies are used in a request or what cookie jar to use or what cookies to send.

**Types** `GuzzleHttp\Cookie\CookieJarInterface`

**Default** None

**Constant** `GuzzleHttp\RequestOptions::COOKIES`

You must specify the cookies option as a `GuzzleHttp\Cookie\CookieJarInterface` or `false`.

```
$jar = new \GuzzleHttp\Cookie\CookieJar();
$client->request('GET', '/get', ['cookies' => $jar]);
```

**Warning:** This option only has an effect if your handler has the `GuzzleHttp\Middleware::cookies` middleware. This middleware is added by default when a client is created with no handler, and is added by default when creating a handler with `GuzzleHttp\default_handler`.

**Tip:** When creating a client, you can set the default cookie option to `true` to use a shared cookie session associated with the client.

## connect_timeout

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

**Types** float

**Default** 0

**Constant** GuzzleHttp\RequestOptions::CONNECT_TIMEOUT

```
// Timeout if the client fails to connect to the server in 3.14 seconds.
$client->request('GET', '/delay/5', ['connect_timeout' => 3.14]);
```

---

**Note:** This setting must be supported by the HTTP handler used to send a request. connect_timeout is currently only supported by the built-in cURL handler.

---

## debug

**Summary** Set to true or set to a PHP stream returned by fopen() to enable debug output with the handler used to send a request. For example, when using cURL to transfer requests, cURL's verbose of CURLOPT_VERBOSE will be emitted. When using the PHP stream wrapper, stream wrapper notifications will be emitted. If set to true, the output is written to PHP's STDOUT. If a PHP stream is provided, output is written to the stream.

**Types**

- bool
- fopen() resource

**Default** None

**Constant** GuzzleHttp\RequestOptions::DEBUG

```
$client->request('GET', '/get', ['debug' => true]);
```

Running the above example would output something like the following:

```
* About to connect() to httpbin.org port 80 (#0)
*   Trying 107.21.213.98... * Connected to httpbin.org (107.21.213.98) port 80 (#0)
> GET /get HTTP/1.1
Host: httpbin.org
User-Agent: Guzzle/4.0 curl/7.21.4 PHP/5.5.7

< HTTP/1.1 200 OK
< Access-Control-Allow-Origin: *
< Content-Type: application/json
< Date: Sun, 16 Feb 2014 06:50:09 GMT
< Server: gunicorn/0.17.4
< Content-Length: 335
< Connection: keep-alive
<
* Connection #0 to host httpbin.org left intact
```

## decode_content

**Summary** Specify whether or not Content-Encoding responses (gzip, deflate, etc.) are automatically decoded.

**Types**

- string

- bool

**Default** `true`

**Constant** `GuzzleHttp\RequestOptions::DECODE_CONTENT`

This option can be used to control how content-encoded response bodies are handled. By default, `decode_content` is set to true, meaning any gzipped or deflated response will be decoded by Guzzle.

When set to `false`, the body of a response is never decoded, meaning the bytes pass through the handler unchanged.

```
// Request gzipped data, but do not decode it while downloading
$client->request('GET', '/foo.js', [
    'headers'        => ['Accept-Encoding' => 'gzip'],
    'decode_content' => false
]);
```

When set to a string, the bytes of a response are decoded and the string value provided to the `decode_content` option is passed as the `Accept-Encoding` header of the request.

```
// Pass "gzip" as the Accept-Encoding header.
$client->request('GET', '/foo.js', ['decode_content' => 'gzip']);
```

## delay

**Summary** The number of milliseconds to delay before sending the request.

**Types**

- integer
- float

**Default** null

**Constant** `GuzzleHttp\RequestOptions::DELAY`

## expect

**Summary** Controls the behavior of the "Expect: 100-Continue" header.

**Types**

- bool
- integer

**Default** `1048576`

**Constant** `GuzzleHttp\RequestOptions::EXPECT`

Set to `true` to enable the "Expect: 100-Continue" header for all requests that sends a body. Set to `false` to disable the "Expect: 100-Continue" header for all requests. Set to a number so that the size of the payload must be greater than the number in order to send the Expect header. Setting to a number will send the Expect header for all requests in which the size of the payload cannot be determined or where the body is not rewindable.

By default, Guzzle will add the "Expect: 100-Continue" header when the size of the body of a request is greater than 1 MB and a request is using HTTP/1.1.

**Note:** This option only takes effect when using HTTP/1.1. The HTTP/1.0 and HTTP/2.0 protocols do not support the "Expect: 100-Continue" header. Support for handling the "Expect: 100-Continue" workflow must be implemented by Guzzle HTTP handlers used by a client.

## force_ip_resolve

**Summary** Set to "v4" if you want the HTTP handlers to use only ipv4 protocol or "v6" for ipv6 protocol.

**Types** string

**Default** null

**Constant** `GuzzleHttp\RequestOptions::FORCE_IP_RESOLVE`

```
// Force ipv4 protocol
$client->request('GET', '/foo', ['force_ip_resolve' => 'v4']);

// Force ipv6 protocol
$client->request('GET', '/foo', ['force_ip_resolve' => 'v6']);
```

**Note:** This setting must be supported by the HTTP handler used to send a request. `force_ip_resolve` is currently only supported by the built-in cURL and stream handlers.

## form_params

**Summary** Used to send an *application/x-www-form-urlencoded* POST request.

**Types** array

**Constant** `GuzzleHttp\RequestOptions::FORM_PARAMS`

Associative array of form field names to values where each value is a string or array of strings. Sets the Content-Type header to application/x-www-form-urlencoded when no Content-Type header is already present.

```
$client->request('POST', '/post', [
    'form_params' => [
        'foo' => 'bar',
        'baz' => ['hi', 'there!']
    ]
]);
```

**Note:** `form_params` cannot be used with the `multipart` option. You will need to use one or the other. Use `form_params` for `application/x-www-form-urlencoded` requests, and `multipart` for `multipart/form-data` requests.

This option cannot be used with `body`, `multipart`, or `json`

## headers

**Summary** Associative array of headers to add to the request. Each key is the name of a header, and each value is a string or array of strings representing the header field values.

**Types** array

**Defaults** None

**Constant** `GuzzleHttp\RequestOptions::HEADERS`

```
// Set various headers on a request
$client->request('GET', '/get', [
    'headers' => [
        'User-Agent' => 'testing/1.0',
        'Accept'     => 'application/json',
        'X-Foo'      => ['Bar', 'Baz']
    ]
]);
```

Headers may be added as default options when creating a client. When headers are used as default options, they are only applied if the request being created does not already contain the specific header. This includes both requests passed to the client in the `send()` and `sendAsync()` methods, and requests created by the client (e.g., `request()` and `requestAsync()`).

```
$client = new GuzzleHttp\Client(['headers' => ['X-Foo' => 'Bar']]);

// Will send a request with the X-Foo header.
$client->request('GET', '/get');

// Sets the X-Foo header to "test", which prevents the default header
// from being applied.
$client->request('GET', '/get', ['headers' => ['X-Foo' => 'test']]);

// Will disable adding in default headers.
$client->request('GET', '/get', ['headers' => null]);

// Will not overwrite the X-Foo header because it is in the message.
use GuzzleHttp\Psr7\Request;
$request = new Request('GET', 'http://foo.com', ['X-Foo' => 'test']);
$client->send($request);

// Will overwrite the X-Foo header with the request option provided in the
// send method.
use GuzzleHttp\Psr7\Request;
$request = new Request('GET', 'http://foo.com', ['X-Foo' => 'test']);
$client->send($request, ['headers' => ['X-Foo' => 'overwrite']]);
```

## http_errors

**Summary** Set to `false` to disable throwing exceptions on an HTTP protocol errors (i.e., 4xx and 5xx responses). Exceptions are thrown by default when HTTP protocol errors are encountered.

**Types** bool

**Default** `true`

**Constant** `GuzzleHttp\RequestOptions::HTTP_ERRORS`

```
$client->request('GET', '/status/500');
// Throws a GuzzleHttp\Exception\ServerException

$res = $client->request('GET', '/status/500', ['http_errors' => false]);
```

```
echo $res->getStatusCode();
// 500
```

**Warning:** This option only has an effect if your handler has the `GuzzleHttp\Middleware::httpErrors` middleware. This middleware is added by default when a client is created with no handler, and is added by default when creating a handler with `GuzzleHttp\default_handler`.

## json

**Summary** The `json` option is used to easily upload JSON encoded data as the body of a request. A Content-Type header of `application/json` will be added if no Content-Type header is already present on the message.

**Types** Any PHP type that can be operated on by PHP's `json_encode()` function.

**Default** None

**Constant** `GuzzleHttp\RequestOptions::JSON`

```
$response = $client->request('PUT', '/put', ['json' => ['foo' => 'bar']]);
```

Here's an example of using the `tap` middleware to see what request is sent over the wire.

```
use GuzzleHttp\Middleware;

// Grab the client's handler instance.
$clientHandler = $client->getConfig('handler');
// Create a middleware that echoes parts of the request.
$tapMiddleware = Middleware::tap(function ($request) {
    echo $request->getHeaderLine('Content-Type');
    // application/json
    echo $request->getBody();
    // {"foo":"bar"}
});

$response = $client->request('PUT', '/put', [
    'json'    => ['foo' => 'bar'],
    'handler' => $tapMiddleware($clientHandler)
]);
```

**Note:** This request option does not support customizing the Content-Type header or any of the options from PHP's json_encode() function. If you need to customize these settings, then you must pass the JSON encoded data into the request yourself using the `body` request option and you must specify the correct Content-Type header using the `headers` request option.

This option cannot be used with `body`, `form_params`, or `multipart`

## multipart

**Summary** Sets the body of the request to a *multipart/form-data* form.

**Types** array

**Constant** `GuzzleHttp\RequestOptions::MULTIPART`

The value of `multipart` is an array of associative arrays, each containing the following key value pairs:

- `name`: (string, required) the form field name
- `contents`: (StreamInterface/resource/string, required) The data to use in the form element.
- `headers`: (array) Optional associative array of custom headers to use with the form element.
- `filename`: (string) Optional string to send as the filename in the part.

```
$client->request('POST', '/post', [
    'multipart' => [
        [
            'name'     => 'foo',
            'contents' => 'data',
            'headers'  => ['X-Baz' => 'bar']
        ],
        [
            'name'     => 'baz',
            'contents' => fopen('/path/to/file', 'r')
        ],
        [
            'name'     => 'qux',
            'contents' => fopen('/path/to/file', 'r'),
            'filename' => 'custom_filename.txt'
        ],
    ]
]);
```

**Note:** `multipart` cannot be used with the `form_params` option. You will need to use one or the other. Use `form_params` for `application/x-www-form-urlencoded` requests, and `multipart` for `multipart/form-data` requests.

This option cannot be used with `body`, `form_params`, or `json`

## on_headers

**Summary** A callable that is invoked when the HTTP headers of the response have been received but the body has not yet begun to download.

**Types**

- callable

**Constant** `GuzzleHttp\RequestOptions::ON_HEADERS`

The callable accepts a `Psr\Http\ResponseInterface` object. If an exception is thrown by the callable, then the promise associated with the response will be rejected with a `GuzzleHttp\Exception\RequestException` that wraps the exception that was thrown.

You may need to know what headers and status codes were received before data can be written to the sink.

```
// Reject responses that are greater than 1024 bytes.
$client->request('GET', 'http://httpbin.org/stream/1024', [
    'on_headers' => function (ResponseInterface $response) {
        if ($response->getHeaderLine('Content-Length') > 1024) {
            throw new \Exception('The file is too big!');
        }
```

```
    }
]);
```

**Note:** When writing HTTP handlers, the `on_headers` function must be invoked before writing data to the body of the response.

## on_stats

**Summary** `on_stats` allows you to get access to transfer statistics of a request and access the lower level transfer details of the handler associated with your client. `on_stats` is a callable that is invoked when a handler has finished sending a request. The callback is invoked with transfer statistics about the request, the response received, or the error encountered. Included in the data is the total amount of time taken to send the request.

**Types**

- callable

**Constant** `GuzzleHttp\RequestOptions::ON_STATS`

The callable accepts a `GuzzleHttp\TransferStats` object.

```
use GuzzleHttp\TransferStats;

$client = new GuzzleHttp\Client();

$client->request('GET', 'http://httpbin.org/stream/1024', [
    'on_stats' => function (TransferStats $stats) {
        echo $stats->getEffectiveUri() . "\n";
        echo $stats->getTransferTime() . "\n";
        var_dump($stats->getHandlerStats());

        // You must check if a response was received before using the
        // response object.
        if ($stats->hasResponse()) {
            echo $stats->getResponse()->getStatusCode();
        } else {
            // Error data is handler specific. You will need to know what
            // type of error data your handler uses before using this
            // value.
            var_dump($stats->getHandlerErrorData());
        }
    }
]);
```

## progress

**Summary** Defines a function to invoke when transfer progress is made.

**Types**

- callable

**Default** None

**Constant** `GuzzleHttp\RequestOptions::PROGRESS`

The function accepts the following positional arguments:

- the total number of bytes expected to be downloaded
- the number of bytes downloaded so far
- the total number of bytes expected to be uploaded
- the number of bytes uploaded so far

```
// Send a GET request to /get?foo=bar
$result = $client->request(
    'GET',
    '/',
    [
        'progress' => function(
            $downloadTotal,
            $downloadedBytes,
            $uploadTotal,
            $uploadedBytes
        ) {
            //do something
        },
    ]
);
```

## proxy

**Summary** Pass a string to specify an HTTP proxy, or an array to specify different proxies for different protocols.

**Types**

- string
- array

**Default** None

**Constant** `GuzzleHttp\RequestOptions::PROXY`

Pass a string to specify a proxy for all protocols.

```
$client->request('GET', '/', ['proxy' => 'tcp://localhost:8125']);
```

Pass an associative array to specify HTTP proxies for specific URI schemes (i.e., "http", "https"). Provide a `no` key value pair to provide a list of host names that should not be proxied to.

**Note:** Guzzle will automatically populate this value with your environment's `NO_PROXY` environment variable. However, when providing a `proxy` request option, it is up to your to provide the `no` value parsed from the `NO_PROXY` environment variable (e.g., `explode(',', getenv('NO_PROXY'))`).

```
$client->request('GET', '/', [
    'proxy' => [
        'http'  => 'tcp://localhost:8125', // Use this proxy with "http"
        'https' => 'tcp://localhost:9124', // Use this proxy with "https",
        'no' => ['.mit.edu', 'foo.com']    // Don't use a proxy with these
    ]
]);
```

**Note:** You can provide proxy URLs that contain a scheme, username, and password. For example, `"http://username:password@192.168.16.1:10"`.

## query

**Summary** Associative array of query string values or query string to add to the request.

**Types**

- array
- string

**Default** None

**Constant** `GuzzleHttp\RequestOptions::QUERY`

```
// Send a GET request to /get?foo=bar
$client->request('GET', '/get', ['query' => ['foo' => 'bar']]);
```

Query strings specified in the `query` option will overwrite all query string values supplied in the URI of a request.

```
// Send a GET request to /get?foo=bar
$client->request('GET', '/get?abc=123', ['query' => ['foo' => 'bar']]);
```

## read_timeout

**Summary** Float describing the timeout to use when reading a streamed body

**Types** float

**Default** Defaults to the value of the `default_socket_timeout` PHP ini setting

**Constant** `GuzzleHttp\RequestOptions::READ_TIMEOUT`

The timeout applies to individual read operations on a streamed body (when the `stream` option is enabled).

```
$response = $client->request('GET', '/stream', [
    'stream' => true,
    'read_timeout' => 10,
]);

$body = $response->getBody();

// Returns false on timeout
$data = $body->read(1024);

// Returns false on timeout
$line = fgets($body->detach());
```

## sink

**Summary** Specify where the body of a response will be saved.

**Types**

- string (path to file on disk)

- `fopen()` resource
- `Psr\Http\Message\StreamInterface`

**Default** PHP temp stream

**Constant** `GuzzleHttp\RequestOptions::SINK`

Pass a string to specify the path to a file that will store the contents of the response body:

```
$client->request('GET', '/stream/20', ['sink' => '/path/to/file']);
```

Pass a resource returned from `fopen()` to write the response to a PHP stream:

```
$resource = fopen('/path/to/file', 'w');
$client->request('GET', '/stream/20', ['sink' => $resource]);
```

Pass a `Psr\Http\Message\StreamInterface` object to stream the response body to an open PSR-7 stream.

```
$resource = fopen('/path/to/file', 'w');
$stream = GuzzleHttp\Psr7\stream_for($resource);
$client->request('GET', '/stream/20', ['save_to' => $stream]);
```

---

**Note:** The `save_to` request option has been deprecated in favor of the `sink` request option. Providing the `save_to` option is now an alias of `sink`.

---

## ssl_key

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

**Types**

- string
- array

**Default** None

**Constant** `GuzzleHttp\RequestOptions::SSL_KEY`

---

**Note:** `ssl_key` is implemented by HTTP handlers. This is currently only supported by the cURL handler, but might be supported by other third-part handlers.

---

## stream

**Summary** Set to `true` to stream a response rather than download it all up-front.

**Types** bool

**Default** `false`

**Constant** `GuzzleHttp\RequestOptions::STREAM`

```php
$response = $client->request('GET', '/stream/20', ['stream' => true]);
// Read bytes off of the stream until the end of the stream is reached
$body = $response->getBody();
while (!$body->eof()) {
    echo $body->read(1024);
}
```

**Note:** Streaming response support must be implemented by the HTTP handler used by a client. This option might not be supported by every HTTP handler, but the interface of the response object remains the same regardless of whether or not it is supported by the handler.

## synchronous

**Summary** Set to true to inform HTTP handlers that you intend on waiting on the response. This can be useful for optimizations.

**Types** bool

**Default** none

**Constant** `GuzzleHttp\RequestOptions::SYNCHRONOUS`

## verify

**Summary** Describes the SSL certificate verification behavior of a request.

- Set to `true` to enable SSL certificate verification and use the default CA bundle provided by operating system.
- Set to `false` to disable certificate verification (this is insecure!).
- Set to a string to provide the path to a CA bundle to enable verification using a custom certificate.

**Types**

- bool
- string

**Default** `true`

**Constant** `GuzzleHttp\RequestOptions::VERIFY`

```php
// Use the system's CA bundle (this is the default setting)
$client->request('GET', '/', ['verify' => true]);

// Use a custom SSL certificate on disk.
$client->request('GET', '/', ['verify' => '/path/to/cert.pem']);

// Disable validation entirely (don't do this!).
$client->request('GET', '/', ['verify' => false]);
```

Not all system's have a known CA bundle on disk. For example, Windows and OS X do not have a single common location for CA bundles. When setting "verify" to `true`, Guzzle will do its best to find the most appropriate CA bundle on your system. When using cURL or the PHP stream wrapper on PHP versions >= 5.6, this happens by default. When using the PHP stream wrapper on versions < 5.6, Guzzle tries to find your CA bundle in the following order:

1. Check if `openssl.cafile` is set in your php.ini file.
2. Check if `curl.cainfo` is set in your php.ini file.
3. Check if `/etc/pki/tls/certs/ca-bundle.crt` exists (Red Hat, CentOS, Fedora; provided by the ca-certificates package)
4. Check if `/etc/ssl/certs/ca-certificates.crt` exists (Ubuntu, Debian; provided by the ca-certificates package)
5. Check if `/usr/local/share/certs/ca-root-nss.crt` exists (FreeBSD; provided by the ca_root_nss package)
6. Check if `/usr/local/etc/openssl/cert.pem` (OS X; provided by homebrew)
7. Check if `C:\windows\system32\curl-ca-bundle.crt` exists (Windows)
8. Check if `C:\windows\curl-ca-bundle.crt` exists (Windows)

The result of this lookup is cached in memory so that subsequent calls in the same process will return very quickly. However, when sending only a single request per-process in something like Apache, you should consider setting the `openssl.cafile` environment variable to the path on disk to the file so that this entire process is skipped.

If you do not need a specific certificate bundle, then Mozilla provides a commonly used CA bundle which can be downloaded here (provided by the maintainer of cURL). Once you have a CA bundle available on disk, you can set the "openssl.cafile" PHP ini setting to point to the path to the file, allowing you to omit the "verify" request option. Much more detail on SSL certificates can be found on the cURL website.

## timeout

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

**Types** float

**Default** `0`

**Constant** `GuzzleHttp\RequestOptions::TIMEOUT`

```
// Timeout if a server does not return a response in 3.14 seconds.
$client->request('GET', '/delay/5', ['timeout' => 3.14]);
// PHP Fatal error:  Uncaught exception 'GuzzleHttp\Exception\RequestException'
```

## version

**Summary** Protocol version to use with the request.

**Types** string, float

**Default** `1.1`

**Constant** `GuzzleHttp\RequestOptions::VERSION`

```
// Force HTTP/1.0
$request = $client->request('GET', '/get', ['version' => 1.0]);
```

# Guzzle and PSR-7

Guzzle utilizes PSR-7 as the HTTP message interface. This allows Guzzle to work with any other library that utilizes PSR-7 message interfaces.

Guzzle is an HTTP client that sends HTTP requests to a server and receives HTTP responses. Both requests and responses are referred to as messages.

Guzzle relies on the `guzzlehttp/psr7` Composer package for its message implementation of PSR-7.

You can create a request using the `GuzzleHttp\Psr7\Request` class:

```
use GuzzleHttp\Psr7\Request;

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

// You can provide other optional constructor arguments.
$headers = ['X-Foo' => 'Bar'];
$body = 'hello!';
$request = new Request('PUT', 'http://httpbin.org/put', $headers, $body);
```

You can create a response using the `GuzzleHttp\Psr7\Response` class:

```
use GuzzleHttp\Psr7\Response;

// The constructor requires no arguments.
$response = new Response();
echo $response->getStatusCode(); // 200
echo $response->getProtocolVersion(); // 1.1

// You can supply any number of optional arguments.
$status = 200;
$headers = ['X-Foo' => 'Bar'];
$body = 'hello!';
$protocol = '1.1';
$response = new Response($status, $headers, $body, $protocol);
```

## Headers

Both request and response messages contain HTTP headers.

### Accessing Headers

You can check if a request or response has a specific header using the `hasHeader()` method.

```
use GuzzleHttp\Psr7;

$request = new Psr7\Request('GET', '/', ['X-Foo' => 'bar']);

if ($request->hasHeader('X-Foo')) {
    echo 'It is there';
}
```

You can retrieve all the header values as an array of strings using `getHeader()`.

```
$request->getHeader('X-Foo'); // ['bar']

// Retrieving a missing header returns an empty array.
$request->getHeader('X-Bar'); // []
```

You can iterate over the headers of a message using the `getHeaders()` method.

```
foreach ($request->getHeaders() as $name => $values) {
    echo $name . ': ' . implode(', ', $values) . "\r\n";
}
```

### Complex Headers

Some headers contain additional key value pair information. For example, Link headers contain a link and several key value pairs:

```
<http://foo.com>; rel="thing"; type="image/jpeg"
```

Guzzle provides a convenience feature that can be used to parse these types of headers:

```
use GuzzleHttp\Psr7;

$request = new Psr7\Request('GET', '/', [
    'Link' => '<http:/.../front.jpeg>; rel="front"; type="image/jpeg"'
]);

$parsed = Psr7\parse_header($request->getHeader('Link'));
var_export($parsed);
```

Will output:

```
array (
  0 =>
  array (
    0 => '<http:/.../front.jpeg>',
    'rel' => 'front',
    'type' => 'image/jpeg',
  ),
)
```

The result contains a hash of key value pairs. Header values that have no key (i.e., the link) are indexed numerically while headers parts that form a key value pair are added as a key value pair.

## Body

Both request and response messages can contain a body.

You can retrieve the body of a message using the `getBody()` method:

```
$response = GuzzleHttp\get('http://httpbin.org/get');
echo $response->getBody();
// JSON string: { ... }
```

The body used in request and response objects is a `Psr\Http\Message\StreamInterface`. This stream is used for both uploading data and downloading data. Guzzle will, by default, store the body of a message in a stream that uses PHP temp streams. When the size of the body exceeds 2 MB, the stream will automatically switch to storing data on disk rather than in memory (protecting your application from memory exhaustion).

The easiest way to create a body for a message is using the `stream_for` function from the `GuzzleHttp\Psr7` namespace – `GuzzleHttp\Psr7\stream_for`. This function accepts strings, resources, callables, iterators, other streamables, and returns an instance of `Psr\Http\Message\StreamInterface`.

The body of a request or response can be cast to a string or you can read and write bytes off of the stream as needed.

```
use GuzzleHttp\Stream\Stream;
$response = $client->request('GET', 'http://httpbin.org/get');

echo $response->getBody()->read(4);
echo $response->getBody()->read(4);
echo $response->getBody()->read(1024);
var_export($response->eof());
```

## Requests

Requests are sent from a client to a server. Requests include the method to be applied to a resource, the identifier of the resource, and the protocol version to use.

### Request Methods

When creating a request, you are expected to provide the HTTP method you wish to perform. You can specify any method you'd like, including a custom method that might not be part of RFC 7231 (like "MOVE").

```
// Create a request using a completely custom HTTP method
$request = new \GuzzleHttp\Psr7\Request('MOVE', 'http://httpbin.org/move');

echo $request->getMethod();
// MOVE
```

You can create and send a request using methods on a client that map to the HTTP method you wish to use.

**GET** `$client->get('http://httpbin.org/get', [/** options **/])`

**POST** `$client->post('http://httpbin.org/post', [/** options **/])`

**HEAD** `$client->head('http://httpbin.org/get', [/** options **/])`

**PUT** `$client->put('http://httpbin.org/put', [/** options **/])`

**DELETE** `$client->delete('http://httpbin.org/delete', [/** options **/])`

**OPTIONS** `$client->options('http://httpbin.org/get', [/** options **/])`

**PATCH** `$client->patch('http://httpbin.org/put', [/** options **/])`

For example:

```
$response = $client->patch('http://httpbin.org/patch', ['body' => 'content']);
```

### Request URI

The request URI is represented by a `Psr\Http\Message\UriInterface` object. Guzzle provides an implementation of this interface using the `GuzzleHttp\Psr7\Uri` class.

When creating a request, you can provide the URI as a string or an instance of `Psr\Http\Message\UriInterface`.

```
$response = $client->request('GET', 'http://httpbin.org/get?q=foo');
```

### Scheme

The scheme of a request specifies the protocol to use when sending the request. When using Guzzle, the scheme can be set to "http" or "https".

```
$request = new Request('GET', 'http://httpbin.org');
echo $request->getUri()->getScheme(); // http
echo $request->getUri(); // http://httpbin.org
```

### Host

The host is accessible using the URI owned by the request or by accessing the Host header.

```
$request = new Request('GET', 'http://httpbin.org');
echo $request->getUri()->getHost(); // httpbin.org
echo $request->getHeader('Host'); // httpbin.org
```

### Port

No port is necessary when using the "http" or "https" schemes.

```
$request = new Request('GET', 'http://httpbin.org:8080');
echo $request->getUri()->getPort(); // 8080
echo $request->getUri(); // http://httpbin.org:8080
```

### Path

The path of a request is accessible via the URI object.

```
$request = new Request('GET', 'http://httpbin.org/get');
echo $request->getUri()->getPath(); // /get
```

The contents of the path will be automatically filtered to ensure that only allowed characters are present in the path. Any characters that are not allowed in the path will be percent-encoded according to RFC 3986 section 3.3

### Query string

The query string of a request can be accessed using the `getQuery()` of the URI object owned by the request.

```
$request = new Request('GET', 'http://httpbin.org/?foo=bar');
echo $request->getUri()->getQuery(); // foo=bar
```

The contents of the query string will be automatically filtered to ensure that only allowed characters are present in the query string. Any characters that are not allowed in the query string will be percent-encoded according to RFC 3986 section 3.4

## Responses

Responses are the HTTP messages a client receives from a server after sending an HTTP request message.

### Start-Line

The start-line of a response contains the protocol and protocol version, status code, and reason phrase.

```
$client = new \GuzzleHttp\Client();
$response = $client->request('GET', 'http://httpbin.org/get');

echo $response->getStatusCode(); // 200
echo $response->getReasonPhrase(); // OK
echo $response->getProtocolVersion(); // 1.1
```

### Body

As described earlier, you can get the body of a response using the `getBody()` method.

```
$body = $response->getBody();
echo $body;
// Cast to a string: { ... }
$body->seek(0);
// Rewind the body
$body->read(1024);
// Read bytes of the body
```

## Streams

Guzzle uses PSR-7 stream objects to represent request and response message bodies. These stream objects allow you to work with various types of data all using a common interface.

HTTP messages consist of a start-line, headers, and a body. The body of an HTTP message can be very small or extremely large. Attempting to represent the body of a message as a string can easily consume more memory than intended because the body must be stored completely in memory. Attempting to store the body of a request or response in memory would preclude the use of that implementation from being able to work with large message bodies. The StreamInterface is used in order to hide the implementation details of where a stream of data is read from or written to.

The PSR-7 `Psr\Http\Message\StreamInterface` exposes several methods that enable streams to be read from, written to, and traversed effectively.

Streams expose their capabilities using three methods: `isReadable()`, `isWritable()`, and `isSeekable()`. These methods can be used by stream collaborators to determine if a stream is capable of their requirements.

Each stream instance has various capabilities: they can be read-only, write-only, read-write, allow arbitrary random access (seeking forwards or backwards to any location), or only allow sequential access (for example in the case of a socket or pipe).

Guzzle uses the `guzzlehttp/psr7` package to provide stream support. More information on using streams, creating streams, converting streams to PHP stream resource, and stream decorators can be found in the Guzzle PSR-7 documentation.

### Creating Streams

The best way to create a stream is using the `GuzzleHttp\Psr7\stream_for` function. This function accepts strings, resources returned from `fopen()`, an object that implements `__toString()`, iterators, callables, and instances of `Psr\Http\Message\StreamInterface`.

```php
use GuzzleHttp\Psr7;

$stream = Psr7\stream_for('string data');
echo $stream;
// string data
echo $stream->read(3);
// str
echo $stream->getContents();
// ing data
var_export($stream->eof());
// true
var_export($stream->tell());
// 11
```

You can create streams from iterators. The iterator can yield any number of bytes per iteration. Any excess bytes returned by the iterator that were not requested by a stream consumer will be buffered until a subsequent read.

```php
use GuzzleHttp\Psr7;

$generator = function ($bytes) {
    for ($i = 0; $i < $bytes; $i++) {
        yield '.';
    }
};

$iter = $generator(1024);
$stream = Psr7\stream_for($iter);
echo $stream->read(3); // ...
```

### Metadata

Streams expose stream metadata through the `getMetadata()` method. This method provides the data you would retrieve when calling PHP's stream_get_meta_data() function, and can optionally expose other custom data.

```php
use GuzzleHttp\Psr7;

$resource = fopen('/path/to/file', 'r');
$stream = Psr7\stream_for($resource);
echo $stream->getMetadata('uri');
// /path/to/file
var_export($stream->isReadable());
// true
var_export($stream->isWritable());
// false
var_export($stream->isSeekable());
// true
```

### Stream Decorators

Adding custom functionality to streams is very simple with stream decorators. Guzzle provides several built-in decorators that provide additional stream functionality.

- AppendStream
- BufferStream
- CachingStream

- DroppingStream
- FnStream
- InflateStream
- LazyOpenStream
- LimitStream
- NoSeekStream
- PumpStream

# Handlers and Middleware

Guzzle clients use a handler and middleware system to send HTTP requests.

## Handlers

A handler function accepts a `Psr\Http\Message\RequestInterface` and array of request options and returns a `GuzzleHttp\Promise\PromiseInterface` that is fulfilled with a `Psr\Http\Message\ResponseInterface` or rejected with an exception.

You can provide a custom handler to a client using the `handler` option of a client constructor. It is important to understand that several request options used by Guzzle require that specific middlewares wrap the handler used by the client. You can ensure that the handler you provide to a client uses the default middlewares by wrapping the handler in the `GuzzleHttp\HandlerStack::create(callable $handler = null)` static method.

```
use GuzzleHttp\Client;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Handler\CurlHandler;

$handler = new CurlHandler();
$stack = HandlerStack::create($handler); // Wrap w/ middleware
$client = new Client(['handler' => $stack]);
```

The `create` method adds default handlers to the `HandlerStack`. When the `HandlerStack` is resolved, the handlers will execute in the following order:

1. Sending request:

1. `http_errors` - No op when sending a request. The response status code is checked in the response processing when returning a response promise up the stack.
2. `allow_redirects` - No op when sending a request. Following redirects occurs when a response promise is being returned up the stack.
3. `cookies` - Adds cookies to requests.
4. `prepare_body` - The body of an HTTP request will be prepared (e.g., add default headers like Content-Length, Content-Type, etc.).
5. <send request with handler>

2. Processing response:

1. `prepare_body` - no op on response processing.
2. `cookies` - extracts response cookies into the cookie jar.

3. `allow_redirects` - Follows redirects.
4. `http_errors` - throws exceptions when the response status code >= 300.

When provided no `$handler` argument, `GuzzleHttp\HandlerStack::create()` will choose the most appropriate handler based on the extensions available on your system.

**Important:** The handler provided to a client determines how request options are applied and utilized for each request sent by a client. For example, if you do not have a cookie middleware associated with a client, then setting the `cookies` request option will have no effect on the request.

## Middleware

Middleware augments the functionality of handlers by invoking them in the process of generating responses. Middleware is implemented as a higher order function that takes the following form.

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

function my_middleware()
{
    return function (callable $handler) {
        return function (RequestInterface $request, array $options) use ($handler) {
            return $handler($request, $options);
        };
    };
}
```

Middleware functions return a function that accepts the next handler to invoke. This returned function then returns another function that acts as a composed handler– it accepts a request and options, and returns a promise that is fulfilled with a response. Your composed middleware can modify the request, add custom request options, and modify the promise returned by the downstream handler.

Here's an example of adding a header to each request.

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

function add_header($header, $value)
{
    return function (callable $handler) use ($header, $value) {
        return function (
            RequestInterface $request,
            array $options
        ) use ($handler, $header, $value) {
            $request = $request->withHeader($header, $value);
            return $handler($request, $options);
        };
    };
}
```

Once a middleware has been created, you can add it to a client by either wrapping the handler used by the client or by decorating a handler stack.

```
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Handler\CurlHandler;
use GuzzleHttp\Client;

$stack = new HandlerStack();
```

```
$stack->setHandler(new CurlHandler());
$stack->push(add_header('X-Foo', 'bar'));
$client = new Client(['handler' => $stack]);
```

Now when you send a request, the client will use a handler composed with your added middleware, adding a header to each request.

Here's an example of creating a middleware that modifies the response of the downstream handler. This example adds a header to the response.

```
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Handler\CurlHandler;
use GuzzleHttp\Client;

function add_response_header($header, $value)
{
    return function (callable $handler) use ($header, $value) {
        return function (
            RequestInterface $request,
            array $options
        ) use ($handler, $header, $value) {
            $promise = $handler($request, $options);
            return $promise->then(
                function (ResponseInterface $response) use ($header, $value) {
                    return $response->withHeader($header, $value);
                }
            );
        };
    };
}

$stack = new HandlerStack();
$stack->setHandler(new CurlHandler());
$stack->push(add_response_header('X-Foo', 'bar'));
$client = new Client(['handler' => $stack]);
```

Creating a middleware that modifies a request is made much simpler using the `GuzzleHttp\Middleware::mapRequest()` middleware. This middleware accepts a function that takes the request argument and returns the request to send.

```
use Psr\Http\Message\RequestInterface;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Handler\CurlHandler;
use GuzzleHttp\Client;
use GuzzleHttp\Middleware;

$stack = new HandlerStack();
$stack->setHandler(new CurlHandler());

$stack->push(Middleware::mapRequest(function (RequestInterface $request) {
    return $request->withHeader('X-Foo', 'bar');
}));

$client = new Client(['handler' => $stack]);
```

Modifying a response is also much simpler using the `GuzzleHttp\Middleware::mapResponse()` middleware.

```php
use Psr\Http\Message\ResponseInterface;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Handler\CurlHandler;
use GuzzleHttp\Client;
use GuzzleHttp\Middleware;

$stack = new HandlerStack();
$stack->setHandler(new CurlHandler());

$stack->push(Middleware::mapResponse(function (ResponseInterface $response) {
    return $response->withHeader('X-Foo', 'bar');
}));

$client = new Client(['handler' => $stack]);
```

## HandlerStack

A handler stack represents a stack of middleware to apply to a base handler function. You can push middleware to the stack to add to the top of the stack, and unshift middleware onto the stack to add to the bottom of the stack. When the stack is resolved, the handler is pushed onto the stack. Each value is then popped off of the stack, wrapping the previous value popped off of the stack.

```php
use Psr\Http\Message\RequestInterface;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Middleware;
use GuzzleHttp\Client;

$stack = new HandlerStack();
$stack->setHandler(\GuzzleHttp\choose_handler());

$stack->push(Middleware::mapRequest(function (RequestInterface $r) {
    echo 'A';
    return $r;
});

$stack->push(Middleware::mapRequest(function (RequestInterface $r) {
    echo 'B';
    return $r;
});

$stack->push(Middleware::mapRequest(function (RequestInterface $r) {
    echo 'C';
    return $r;
});

$client->request('GET', 'http://httpbin.org/');
// echoes 'ABC';

$stack->unshift(Middleware::mapRequest(function (RequestInterface $r) {
    echo '0';
    return $r;
});

$client = new Client(['handler' => $stack]);
$client->request('GET', 'http://httpbin.org/');
// echoes '0ABC';
```

You can give middleware a name, which allows you to add middleware before other named middleware, after other named middleware, or remove middleware by name.

```
use Psr\Http\Message\RequestInterface;
use GuzzleHttp\Middleware;

// Add a middleware with a name
$stack->push(Middleware::mapRequest(function (RequestInterface $r) {
    return $r->withHeader('X-Foo', 'Bar');
}, 'add_foo');

// Add a middleware before a named middleware (unshift before).
$stack->before('add_foo', Middleware::mapRequest(function (RequestInterface $r) {
    return $r->withHeader('X-Baz', 'Qux');
}, 'add_baz');

// Add a middleware after a named middleware (pushed after).
$stack->after('add_baz', Middleware::mapRequest(function (RequestInterface $r) {
    return $r->withHeader('X-Lorem', 'Ipsum');
});

// Remove a middleware by name
$stack->remove('add_foo');
```

## Creating a Handler

As stated earlier, a handler is a function accepts a `Psr\Http\Message\RequestInterface` and array of request options and returns a `GuzzleHttp\Promise\PromiseInterface` that is fulfilled with a `Psr\Http\Message\ResponseInterface` or rejected with an exception.

A handler is responsible for applying the following Request Options. These request options are a subset of request options called "transfer options".

- *cert*
- *connect_timeout*
- *debug*
- *delay*
- *decode_content*
- *expect*
- *proxy*
- *sink*
- *timeout*
- *ssl_key*
- *stream*
- *verify*

# Testing Guzzle Clients

Guzzle provides several tools that will enable you to easily mock the HTTP layer without needing to send requests over the internet.

- Mock handler
- History middleware
- Node.js web server for integration testing

## Mock Handler

When testing HTTP clients, you often need to simulate specific scenarios like returning a successful response, returning an error, or returning specific responses in a certain order. Because unit tests need to be predictable, easy to bootstrap, and fast, hitting an actual remote API is a test smell.

Guzzle provides a mock handler that can be used to fulfill HTTP requests with a response or exception by shifting return values off of a queue.

```
use GuzzleHttp\Client;
use GuzzleHttp\Handler\MockHandler;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Psr7\Response;
use GuzzleHttp\Psr7\Request;
use GuzzleHttp\Exception\RequestException;

// Create a mock and queue two responses.
$mock = new MockHandler([
    new Response(200, ['X-Foo' => 'Bar']),
    new Response(202, ['Content-Length' => 0]),
    new RequestException('Error Communicating with Server', new Request('GET', 'test'))
]);

$handlerStack = HandlerStack::create($mock);
$client = new Client(['handler' => $handlerStack]);

// The first request is intercepted with the first response.
echo $client->request('GET', '/')->getStatusCode();
//> 200
// The second request is intercepted with the second response.
echo $client->request('GET', '/')->getStatusCode();
//> 202
```

When no more responses are in the queue and a request is sent, an `OutOfBoundsException` is thrown.

## History Middleware

When using things like the `Mock` handler, you often need to know if the requests you expected to send were sent exactly as you intended. While the mock handler responds with mocked responses, the history middleware maintains a history of the requests that were sent by a client.

```
use GuzzleHttp\Client;
use GuzzleHttp\HandlerStack;
use GuzzleHttp\Middleware;

$container = [];
```

```php
$history = Middleware::history($container);

$handlerStack = HandlerStack::create();
// or $handlerStack = HandlerStack::create($mock); if using the Mock handler.

// Add the history middleware to the handler stack.
$handlerStack->push($history);

$client = new Client(['handler' => $handlerStack]);

$client->request('GET', 'http://httpbin.org/get');
$client->request('HEAD', 'http://httpbin.org/get');

// Count the number of transactions
echo count($container);
//> 2

// Iterate over the requests and responses
foreach ($container as $transaction) {
    echo $transaction['request']->getMethod();
    //> GET, HEAD
    if ($transaction['response']) {
        echo $transaction['response']->getStatusCode();
        //> 200, 200
    } elseif ($transaction['error']) {
        echo $transaction['error'];
        //> exception
    }
    var_dump($transaction['options']);
    //> dumps the request options of the sent request.
}
```

## Test Web Server

Using mock responses is almost always enough when testing a web service client. When implementing custom HTTP handlers, you'll need to send actual HTTP requests in order to sufficiently test the handler. However, a best practice is to contact a local web server rather than a server over the internet.

- Tests are more reliable
- Tests do not require a network connection
- Tests have no external dependencies

### Using the test server

**Warning:** The following functionality is provided to help developers of Guzzle develop HTTP handlers. There is no promise of backwards compatibility when it comes to the node.js test server or the `GuzzleHttp\Tests\Server` class. If you are using the test server or `Server` class outside of guzzle-http/guzzle, then you will need to configure autoloading and ensure the web server is started manually.

**Hint:** You almost never need to use this test web server. You should only ever consider using it when developing HTTP handlers. The test web server is not necessary for mocking requests. For that, please use the Mock handler and

history middleware.

Guzzle ships with a node.js test server that receives requests and returns responses from a queue. The test server exposes a simple API that is used to enqueue responses and inspect the requests that it has received.

Any operation on the `Server` object will ensure that the server is running and wait until it is able to receive requests before returning.

`GuzzleHttp\Tests\Server` provides a static interface to the test server. You can queue an HTTP response or an array of responses by calling `Server::enqueue()`. This method accepts an array of `Psr\Http\Message\ResponseInterface` and `Exception` objects.

```
use GuzzleHttp\Client;
use GuzzleHttp\Psr7\Response;
use GuzzleHttp\Tests\Server;

// Start the server and queue a response
Server::enqueue([
    new Response(200, ['Content-Length' => 0])
]);

$client = new Client(['base_uri' => Server::$url]);
echo $client->request('GET', '/foo')->getStatusCode();
// 200
```

When a response is queued on the test server, the test server will remove any previously queued responses. As the server receives requests, queued responses are dequeued and returned to the request. When the queue is empty, the server will return a 500 response.

You can inspect the requests that the server has retrieved by calling `Server::received()`.

```
foreach (Server::received() as $response) {
    echo $response->getStatusCode();
}
```

You can clear the list of received requests from the web server using the `Server::flush()` method.

```
Server::flush();
echo count(Server::received());
// 0
```

# FAQ

## Does Guzzle require cURL?

No. Guzzle can use any HTTP handler to send requests. This means that Guzzle can be used with cURL, PHP's stream wrapper, sockets, and non-blocking libraries like React. You just need to configure an HTTP handler to use a different method of sending requests.

**Note:** Guzzle has historically only utilized cURL to send HTTP requests. cURL is an amazing HTTP client (arguably the best), and Guzzle will continue to use it by default when it is available. It is rare, but some developers don't have cURL installed on their systems or run into version specific issues. By allowing swappable HTTP handlers, Guzzle is now much more customizable and able to adapt to fit the needs of more developers.

## Can Guzzle send asynchronous requests?

Yes. You can use the `requestAsync`, `sendAsync`, `getAsync`, `headAsync`, `putAsync`, `postAsync`, `deleteAsync`, and `patchAsync` methods of a client to send an asynchronous request. The client will return a `GuzzleHttp\Promise\PromiseInterface` object. You can chain `then` functions off of the promise.

```
$promise = $client->requestAsync('GET', 'http://httpbin.org/get');
$promise->then(function ($response) {
    echo 'Got a response! ' . $response->getStatusCode();
});
```

You can force an asynchronous response to complete using the `wait()` method of the returned promise.

```
$promise = $client->requestAsync('GET', 'http://httpbin.org/get');
$response = $promise->wait();
```

## How can I add custom cURL options?

cURL offers a huge number of customizable options. While Guzzle normalizes many of these options across different handlers, there are times when you need to set custom cURL options. This can be accomplished by passing an associative array of cURL settings in the **curl** key of a request.

For example, let's say you need to customize the outgoing network interface used with a client.

```
$client->request('GET', '/', [
    'curl' => [
        CURLOPT_INTERFACE => 'xxx.xxx.xxx.xxx'
    ]
]);
```

## How can I add custom stream context options?

You can pass custom stream context options using the **stream_context** key of the request option. The **stream_context** array is an associative array where each key is a PHP transport, and each value is an associative array of transport options.

For example, let's say you need to customize the outgoing network interface used with a client and allow self-signed certificates.

```
$client->request('GET', '/', [
    'stream' => true,
    'stream_context' => [
        'ssl' => [
            'allow_self_signed' => true
        ],
        'socket' => [
            'bindto' => 'xxx.xxx.xxx.xxx'
        ]
    ]
]);
```

## Why am I getting an SSL verification error?

You need to specify the path on disk to the CA bundle used by Guzzle for verifying the peer certificate. See *verify*.

## What is this Maximum function nesting error?

> Maximum function nesting level of '100' reached, aborting

You could run into this error if you have the XDebug extension installed and you execute a lot of requests in callbacks. This error message comes specifically from the XDebug extension. PHP itself does not have a function nesting limit. Change this setting in your php.ini to increase the limit:

```
xdebug.max_nesting_level = 1000
```

## Why am I getting a 417 error response?

This can occur for a number of reasons, but if you are sending PUT, POST, or PATCH requests with an `Expect: 100-Continue` header, a server that does not support this header will return a 417 response. You can work around this by setting the `expect` request option to `false`:

```
$client = new GuzzleHttp\Client();

// Disable the expect header on a single request
$response = $client->request('PUT', '/', ['expect' => false]);

// Disable the expect header on all client requests
$client = new GuzzleHttp\Client(['expect' => false]);
```

## How can I track redirected requests?

You can enable tracking of redirected URIs and status codes via the *track_redirects* option. Each redirected URI and status code will be stored in the `X-Guzzle-Redirect-History` and the `X-Guzzle-Redirect-Status-History` header respectively.

The initial request's URI and the final status code will be excluded from the results. With this in mind you should be able to easily track a request's full redirect path.

For example, let's say you need to track redirects and provide both results together in a single report:

```
// First you configure Guzzle with redirect tracking and make a request
$client = new Client([
    RequestOptions::ALLOW_REDIRECTS => [
        'max'             => 10,        // allow at most 10 redirects.
        'strict'          => true,      // use "strict" RFC compliant redirects.
        'referer'         => true,      // add a Referer header
        'track_redirects' => true,
    ],
]);
$initialRequest = '/redirect/3'; // Store the request URI for later use
$response = $client->request('GET', $initialRequest); // Make your request

// Retrieve both Redirect History headers
$redirectUriHistory = $response->getHeader('X-Guzzle-Redirect-History')[0]; // retrieve Redirect URI
$redirectCodeHistory = $response->getHeader('X-Guzzle-Redirect-Status-History')[0]; // retrieve Redi

// Add the initial URI requested to the (beginning of) URI history
array_unshift($redirectUriHistory, $initialRequest);

// Add the final HTTP status code to the end of HTTP response history
array_push($redirectCodeHistory, $response->getStatusCode());
```

```php
// (Optional) Combine the items of each array into a single result set
$fullRedirectReport = [];
foreach ($redirectUriHistory as $key => $value) {
    $fullRedirectReport[$key] = ['location' => $value, 'code' => $redirectCodeHistory[$key]];
}
echo json_encode($fullRedirectReport);
```