Guzzle

Release

Jun 02, 2017

Contents

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

- Manages things like persistent connections, represents query strings as collections, simplifies sending streaming POST requests with fields and files, and abstracts away the underlying HTTP transport layer.
- Can send both synchronous and asynchronous requests using the same interface without requiring a dependency on a specific event loop.
- Pluggable HTTP handlers allows Guzzle to integrate with any method you choose for sending HTTP requests over the wire (e.g., cURL, sockets, PHP's stream wrapper, non-blocking event loops like React, etc.).
- Guzzle makes it so that you no longer need to fool around with cURL options, stream contexts, or sockets.

```php
$client = new GuzzleHttp\Client();
$response = $client->get('http://guzzlephp.org');
$res = $client->get('https://api.github.com/user', ['auth' =>  ['user', 'pass']]);
echo $res->getStatusCode();
// "200"
echo $res->getHeader('content-type');
// 'application/json; charset=utf8'
echo $res->getBody();
// {"type":"User"...'
var_export($res->json());
// Outputs the JSON decoded data

// Send an asynchronous request.
$req = $client->createRequest('GET', 'http://httpbin.org', ['future' => true]);
$client->send($req)->then(function ($response) {
    echo 'I completed! ' . $response;
});
```

CHAPTER 1

User guide

Overview

Requirements

1. PHP 5.4.0
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 >= 7.16.2 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:~5.0
```

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

```
{
  "require": {
```

```
        "guzzlehttp/guzzle": "~5.0"
    }
}
```

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 `~5.0@dev`.

```
{
    "require": {
        "guzzlehttp/guzzle": "~5.0@dev"
    }
}
```

License

Licensed using the MIT license.

> Copyright (c) 2014 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 follows PSR-0, PSR-1, and PSR-2.
2. Guzzle is meant to be lean and fast with very few dependencies.
3. Guzzle has a minimum PHP version requirement of PHP 5.4. Pull requests must not require a PHP version greater than PHP 5.4.

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 vendored PHPUnit binary:

```
vendor/bin/phpunit
```

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.

Make a Request

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

Creating a Client

The procedural API is simple but not very testable; it's best left for quick prototyping. If you want to use Guzzle in a more flexible and testable way, then you'll need to use a `GuzzleHttp\ClientInterface` object.

```
use GuzzleHttp\Client;

$client = new Client();
$response = $client->get('http://httpbin.org/get');

// You can use the same methods you saw in the procedural API
```

```
$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 with a client and then send the request with the client when you're ready.

```
$request = $client->createRequest('GET', 'http://www.foo.com');
$response = $client->send($request);
```

Client objects provide a great deal of flexibility in how request are transferred including default request options, subscribers that are attached to each request, and a base URL that allows you to send requests with relative URLs. You can find out all about clients in the *Clients* page of the documentation.

Using Responses

In the previous examples, we retrieved a `$response` variable. This value is actually a `GuzzleHttp\Message\ResponseInterface` object 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
```

By providing the `future` request option to a request, you can send requests asynchronously using the promise interface of a future response.

```
$client->get('http://httpbin.org', ['future' => true])
    ->then(function ($response) {
        echo $response->getStatusCode();
    });
```

Response Body

The body of a response can be retrieved and cast to a string.

```
$body = $response->getBody();
echo $body;
// { "some_json_data" ...}
```

You can also read read bytes from body of a response like a stream.

```
$body = $response->getBody();

while (!$body->eof()) {
    echo $body->read(1024);
}
```

JSON Responses

You can more easily work with JSON responses using the `json()` method of a response.

```
$response = $client->get('http://httpbin.org/get');
$json = $response->json();
var_dump($json[0]['origin']);
```

Guzzle internally uses PHP's `json_decode()` function to parse responses. If Guzzle is unable to parse the JSON response body, then a `GuzzleHttp\Exception\ParseException` is thrown.

XML Responses

You can use a response's `xml()` method to more easily work with responses that contain XML data.

```
$response = $client->get('https://github.com/mtdowling.atom');
$xml = $response->xml();
echo $xml->id;
// tag:github.com,2008:/mtdowling
```

Guzzle internally uses a `SimpleXMLElement` object to parse responses. If Guzzle is unable to parse the XML response body, then a `GuzzleHttp\Exception\ParseException` is thrown.

Query String Parameters

Sending query string parameters with a request is easy. You can set query string parameters in the request's URL.

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

You can also specify the query string parameters using the `query` request option.

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

And finally, you can build up the query string of a request as needed by calling the `getQuery()` method of a request and modifying the request's `GuzzleHttp\Query` object as needed.

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

// You can use the query string object like an array
$query['baz'] = 'bam';

// The query object can be cast to a string
echo $query;
// foo=bar&baz=bam

// Setting a value to false or null will cause the "=" sign to be omitted
$query['empty'] = null;
echo $query;
// foo=bar&baz=bam&empty

// Use an empty string to include the "=" sign with an empty value
```

```
$query['empty'] = '';
echo $query;
// foo=bar&baz=bam&empty=
```

Request and Response Headers

You can specify request headers when sending or creating requests with a client. In the following example, we send the `X-Foo-Header` with a value of `value` by setting the `headers` request option.

```
$response = $client->get('http://httpbin.org/get', [
    'headers' => ['X-Foo-Header' => 'value']
]);
```

You can view the headers of a response using header specific methods of a response class. Headers work exactly the same way for request and response object.

You can retrieve a header from a request or response using the `getHeader()` method of the object. This method is case-insensitive and by default will return a string containing the header field value.

```
$response = $client->get('http://www.yahoo.com');
$length = $response->getHeader('Content-Length');
```

Header fields that contain multiple values can be retrieved as a string or as an array. Retrieving the field values as a string will naively concatenate all of the header values together with a comma. Because not all header fields should be represented this way (e.g., `Set-Cookie`), you can pass an optional flag to the `getHeader()` method to retrieve the header values as an array.

```
$values = $response->getHeader('Set-Cookie', true);
foreach ($values as $value) {
    echo $value;
}
```

You can test if a request or response has a specific header using the `hasHeader()` method. This method accepts a case-insensitive string and returns true if the header is present or false if it is not.

You can retrieve all of the headers of a message using the `getHeaders()` method of a request or response. The return value is an associative array where the keys represent the header name as it will be sent over the wire, and each value is an array of strings associated with the header.

```
$headers = $response->getHeaders();
foreach ($message->getHeaders() as $name => $values) {
    echo $name . ": " . implode(", ", $values);
}
```

Modifying headers

The headers of a message can be modified using the `setHeader()`, `addHeader()`, `setHeaders()`, and `removeHeader()` methods of a request or response object.

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

// Set a single value for a header
$request->setHeader('User-Agent', 'Testing!');

// Set multiple values for a header in one call
```

```php
$request->setHeader('X-Foo', ['Baz', 'Bar']);

// Add a header to the message
$request->addHeader('X-Foo', 'Bam');

echo $request->getHeader('X-Foo');
// Baz, Bar, Bam

// Remove a specific header using a case-insensitive name
$request->removeHeader('x-foo');
echo $request->getHeader('X-Foo');
// Echoes an empty string: ''
```

Uploading Data

Guzzle provides several methods of uploading data.

You can send requests that contain a stream of data by passing a string, resource returned from `fopen`, or a `GuzzleHttp\Stream\StreamInterface` object to the `body` request option.

```php
$r = $client->post('http://httpbin.org/post', ['body' => 'raw data']);
```

You can easily upload JSON data using the `json` request option.

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

POST 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 POST Fields

Sending `application/x-www-form-urlencoded` POST requests requires that you specify the body of a POST request as an array.

```php
$response = $client->post('http://httpbin.org/post', [
    'body' => [
        'field_name' => 'abc',
        'other_field' => '123'
    ]
]);
```

You can also build up POST requests before sending them.

```php
$request = $client->createRequest('POST', 'http://httpbin.org/post');
$postBody = $request->getBody();

// $postBody is an instance of GuzzleHttp\Post\PostBodyInterface
$postBody->setField('foo', 'bar');
echo $postBody->getField('foo');
// 'bar'
```

```
echo json_encode($postBody->getFields());
// {"foo": "bar"}

// Send the POST request
$response = $client->send($request);
```

Sending POST Files

Sending `multipart/form-data` POST requests (POST requests that contain files) is the same as sending `application/x-www-form-urlencoded`, except some of the array values of the POST fields map to PHP `fopen` resources, or `GuzzleHttp\Stream\StreamInterface`, or `GuzzleHttp\Post\PostFileInterface` objects.

```
use GuzzleHttp\Post\PostFile;

$response = $client->post('http://httpbin.org/post', [
    'body' => [
        'field_name' => 'abc',
        'file_filed' => fopen('/path/to/file', 'r'),
        'other_file' => new PostFile('other_file', 'this is the content')
    ]
]);
```

Just like when sending POST fields, you can also build up POST requests with files before sending them.

```
use GuzzleHttp\Post\PostFile;

$request = $client->createRequest('POST', 'http://httpbin.org/post');
$postBody = $request->getBody();
$postBody->setField('foo', 'bar');
$postBody->addFile(new PostFile('test', fopen('/path/to/file', 'r')));
$response = $client->send($request);
```

Cookies

Guzzle can maintain a cookie session for you if instructed using the `cookies` request option.

- Set to `true` to use a shared cookie session associated with the client.
- Pass an associative array containing cookies to send in the request and start a new cookie session.
- Set to a `GuzzleHttp\Subscriber\CookieJar\CookieJarInterface` object to use an existing cookie jar.

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->get('http://github.com');
echo $response->getStatusCode();
// 200
echo $response->getEffectiveUrl();
// 'https://github.com/'
```

The following example shows that redirects can be disabled.

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

Exceptions

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 (non-parallel) requests.

```
use GuzzleHttp\Exception\RequestException;

try {
    $client->get('https://github.com/_abc_123_404');
} catch (RequestException $e) {
    echo $e->getRequest();
    if ($e->hasResponse()) {
        echo $e->getResponse();
    }
}
```

- A `GuzzleHttp\Exception\ClientException` is thrown for 400 level errors if the `exceptions` request option is set to true. This exception extends from `GuzzleHttp\Exception\BadResponseException` and `GuzzleHttp\Exception\BadResponseException` extends from `GuzzleHttp\Exception\RequestException`.

```
use GuzzleHttp\Exception\ClientException;

try {
    $client->get('https://github.com/_abc_123_404');
} catch (ClientException $e) {
    echo $e->getRequest();
    echo $e->getResponse();
}
```

- A `GuzzleHttp\Exception\ServerException` is thrown for 500 level errors if the `exceptions` 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.

Clients

Clients are used to create requests, create transactions, send requests through an HTTP handler, and return a response. You can add default request options to a client that are applied to every request (e.g., default headers, default query string parameters, etc.), and you can add event listeners and subscribers to every request created by a client.

Creating a client

The constructor of a client accepts an associative array of configuration options.

base_url Configures a base URL for the client so that requests created using a relative URL are combined with the base_url of the client according to section 5.2 of RFC 3986.

```
// Create a client with a base URL
$client = new GuzzleHttp\Client(['base_url' => 'https://github.com']);
// Send a request to https://github.com/notifications
$response = $client->get('/notifications');
```

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

base_url	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 Configures the RingPHP handler used to transfer the HTTP requests of a client. Guzzle will, by default, utilize a stacked handlers that chooses the best handler to use based on the provided request options and based on the extensions available in the environment.

message_factory Specifies the factory used to create HTTP requests and responses (GuzzleHttp\Message\MessageFactoryInterface).

defaults Associative array of *Request Options* that are applied to every request created by the client. This allows you to specify things like default headers (e.g., User-Agent), default query string parameters, SSL configurations, and any other supported request options.

emitter Specifies an event emitter (GuzzleHttp\Event\EmitterInterface) instance to be used by the client to emit request events. This option is useful if you need to inject an emitter with listeners/subscribers already attached.

Here's an example of creating a client with various options.

```
use GuzzleHttp\Client;

$client = new Client([
    'base_url' => ['https://api.twitter.com/{version}/', ['version' => 'v1.1']],
```

```
    'defaults' => [
        'headers' => ['Foo' => 'Bar'],
        'query'   => ['testing' => '123'],
        'auth'    => ['username', 'password'],
        'proxy'   => 'tcp://localhost:80'
    ]
]);
```

Sending Requests

Requests can be created using various methods of a client. You can create **and** send requests using one of the following methods:

- `GuzzleHttp\Client::get`: Sends a GET request.
- `GuzzleHttp\Client::head`: Sends a HEAD request
- `GuzzleHttp\Client::post`: Sends a POST request
- `GuzzleHttp\Client::put`: Sends a PUT request
- `GuzzleHttp\Client::delete`: Sends a DELETE request
- `GuzzleHttp\Client::options`: Sends an OPTIONS request

Each of the above methods accepts a URL as the first argument and an optional associative array of *Request Options* as the second argument.

Synchronous Requests

Guzzle sends synchronous (blocking) requests when the `future` request option is not specified. This means that the request will complete immediately, and if an error is encountered, a `GuzzleHttp\Exception\RequestException` will be thrown.

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

$client->put('http://httpbin.org', [
    'headers'         => ['X-Foo' => 'Bar'],
    'body'            => 'this is the body!',
    'save_to'         => '/path/to/local/file',
    'allow_redirects' => false,
    'timeout'         => 5
]);
```

Synchronous Error Handling

When a recoverable error is encountered while calling the `send()` method of a client, a `GuzzleHttp\Exception\RequestException` is thrown.

```
use GuzzleHttp\Client;
use GuzzleHttp\Exception\RequestException;

$client = new Client();

try {
```

```
    $client->get('http://httpbin.org');
} catch (RequestException $e) {
    echo $e->getRequest() . "\n";
    if ($e->hasResponse()) {
        echo $e->getResponse() . "\n";
    }
}
```

`GuzzleHttp\Exception\RequestException` always contains a `GuzzleHttp\Message\RequestInterface` object that can be accessed using the exception's `getRequest()` method.

A response might be present in the exception. In the event of a networking error, no response will be received. You can check if a `RequestException` has a response using the `hasResponse()` method. If the exception has a response, then you can access the associated `GuzzleHttp\Message\ResponseInterface` using the `getResponse()` method of the exception.

Asynchronous Requests

You can send asynchronous requests by setting the `future` request option to `true` (or a string that your handler understands). This creates a `GuzzleHttp\Message\FutureResponse` object that has not yet completed. Once you have a future response, you can use a promise object obtained by calling the `then` method of the response to take an action when the response has completed or encounters an error.

```
$response = $client->put('http://httpbin.org/get', ['future' => true]);

// Call the function when the response completes
$response->then(function ($response) {
    echo $response->getStatusCode();
});
```

You can call the `wait()` method of a future response to block until it has completed. You also use a future response object just like a normal response object by accessing the methods of the response. Using a future response like a normal response object, also known as *dereferencing*, will block until the response has completed.

```
$response = $client->put('http://httpbin.org/get', ['future' => true]);

// Block until the response has completed
echo $response->getStatusCode();
```

Important: If an exception occurred while transferring the future response, then the exception encountered will be thrown when dereferencing.

Note: It depends on the RingPHP handler used by a client, but you typically need to use the same RingPHP handler in order to utilize asynchronous requests across multiple clients.

Asynchronous Error Handling

Handling errors with future response object promises is a bit different. When using a promise, exceptions are forwarded to the `$onError` function provided to the second argument of the `then()` function.

```php
$response = $client->put('http://httpbin.org/get', ['future' => true]);

$response
    ->then(
        function ($response) {
            // This is called when the request succeeded
            echo 'Success: ' . $response->getStatusCode();
            // Returning a value will forward the value to the next promise
            // in the chain.
            return $response;
        },
        function ($error) {
            // This is called when the exception failed.
            echo 'Exception: ' . $error->getMessage();
            // Throwing will "forward" the exception to the next promise
            // in the chain.
            throw $error;
        }
    )
    ->then(
        function($response) {
            // This is called after the first promise in the chain. It
            // receives the value returned from the first promise.
            echo $response->getReasonPhrase();
        },
        function ($error) {
            // This is called if the first promise error handler in the
            // chain rethrows the exception.
            echo 'Error: ' . $error->getMessage();
        }
    );
```

Please see the React/Promises project documentation for more information on how promise resolution and rejection forwarding works.

HTTP Errors

If the `exceptions` request option is not set to `false`, then exceptions are thrown for HTTP protocol errors as well: `GuzzleHttp\Exception\ClientErrorResponseException` for 4xx level HTTP responses and `GuzzleHttp\Exception\ServerException` for 5xx level responses, both of which extend from `GuzzleHttp\Exception\BadResponseException`.

Creating Requests

You can create a request without sending it. This is useful for building up requests over time or sending requests in concurrently.

```php
$request = $client->createRequest('GET', 'http://httpbin.org', [
    'headers' => ['X-Foo' => 'Bar']
]);

// Modify the request as needed
$request->setHeader('Baz', 'bar');
```

After creating a request, you can send it with the client's `send()` method.

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

Sending Requests With a Pool

You can send requests concurrently using a fixed size pool via the `GuzzleHttp\Pool` class. The Pool class is an implementation of `GuzzleHttp\Ring\Future\FutureInterface`, meaning it can be dereferenced at a later time or cancelled before sending. The Pool constructor accepts a client object, iterator or array that yields `GuzzleHttp\Message\RequestInterface` objects, and an optional associative array of options that can be used to affect the transfer.

```
use GuzzleHttp\Pool;

$requests = [
    $client->createRequest('GET', 'http://httpbin.org'),
    $client->createRequest('DELETE', 'http://httpbin.org/delete'),
    $client->createRequest('PUT', 'http://httpbin.org/put', ['body' => 'test'])
];

$options = [];

// Create a pool. Note: the options array is optional.
$pool = new Pool($client, $requests, $options);

// Send the requests
$pool->wait();
```

The Pool constructor accepts the following associative array of options:

- **pool_size**: Integer representing the maximum number of requests that are allowed to be sent concurrently.
- **before**: Callable or array representing the event listeners to add to each request's *before* event.
- **complete**: Callable or array representing the event listeners to add to each request's *complete* event.
- **error**: Callable or array representing the event listeners to add to each request's *error* event.
- **end**: Callable or array representing the event listeners to add to each request's *end* event.

The "before", "complete", "error", and "end" event options accept a callable or an array of associative arrays where each associative array contains a "fn" key with a callable value, an optional "priority" key representing the event priority (with a default value of 0), and an optional "once" key that can be set to true so that the event listener will be removed from the request after it is first triggered.

```
use GuzzleHttp\Pool;
use GuzzleHttp\Event\CompleteEvent;

// Add a single event listener using a callable.
Pool::send($client, $requests, [
    'complete' => function (CompleteEvent $event) {
        echo 'Completed request to ' . $event->getRequest()->getUrl() . "\n";
        echo 'Response: ' . $event->getResponse()->getBody() . "\n\n";
    }
]);

// The above is equivalent to the following, but the following structure
// allows you to add multiple event listeners to the same event name.
Pool::send($client, $requests, [
    'complete' => [
```

```
        [
            'fn'       => function (CompleteEvent $event) { /* ... */ },
            'priority' => 0,    // Optional
            'once'     => false // Optional
        ]
    ]
]);
```

Asynchronous Response Handling

When sending requests concurrently using a pool, the request/response/error lifecycle must be handled asynchronously. This means that you give the Pool multiple requests and handle the response or errors that is associated with the request using event callbacks.

```
use GuzzleHttp\Pool;
use GuzzleHttp\Event\ErrorEvent;

Pool::send($client, $requests, [
    'complete' => function (CompleteEvent $event) {
        echo 'Completed request to ' . $event->getRequest()->getUrl() . "\n";
        echo 'Response: ' . $event->getResponse()->getBody() . "\n\n";
        // Do something with the completion of the request...
    },
    'error' => function (ErrorEvent $event) {
        echo 'Request failed: ' . $event->getRequest()->getUrl() . "\n";
        echo $event->getException();
        // Do something to handle the error...
    }
]);
```

The `GuzzleHttp\Event\ErrorEvent` event object is emitted when an error occurs during a transfer. With this event, you have access to the request that was sent, the response that was received (if one was received), access to transfer statistics, and the ability to intercept the exception with a different `GuzzleHttp\Message\ResponseInterface` object. See *Event System* for more information.

Handling Errors After Transferring

It sometimes might be easier to handle all of the errors that occurred during a transfer after all of the requests have been sent. Here we are adding each failed request to an array that we can use to process errors later.

```
use GuzzleHttp\Pool;
use GuzzleHttp\Event\ErrorEvent;

$errors = [];
Pool::send($client, $requests, [
    'error' => function (ErrorEvent $event) use (&$errors) {
        $errors[] = $event;
    }
]);

foreach ($errors as $error) {
    // Handle the error...
}
```

Batching Requests

Sometimes you just want to send a few requests concurrently and then process the results all at once after they've been sent. Guzzle provides a convenience function `GuzzleHttp\Pool::batch()` that makes this very simple:

```
use GuzzleHttp\Pool;
use GuzzleHttp\Client;

$client = new Client();

$requests = [
    $client->createRequest('GET', 'http://httpbin.org/get'),
    $client->createRequest('HEAD', 'http://httpbin.org/get'),
    $client->createRequest('PUT', 'http://httpbin.org/put'),
];

// Results is a GuzzleHttp\BatchResults object.
$results = Pool::batch($client, $requests);

// Can be accessed by index.
echo $results[0]->getStatusCode();

// Can be accessed by request.
echo $results->getResult($requests[0])->getStatusCode();

// Retrieve all successful responses
foreach ($results->getSuccessful() as $response) {
    echo $response->getStatusCode() . "\n";
}

// Retrieve all failures.
foreach ($results->getFailures() as $requestException) {
    echo $requestException->getMessage() . "\n";
}
```

`GuzzleHttp\Pool::batch()` accepts an optional associative array of options in the third argument that allows you to specify the 'before', 'complete', 'error', and 'end' events as well as specify the maximum number of requests to send concurrently using the 'pool_size' option key.

Request Options

You can customize requests created 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_url' => 'http://httpbin.org']);
```

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

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

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
- `GuzzleHttp\Stream\StreamInterface`
- `GuzzleHttp\Post\PostBodyInterface`

Default None

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->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->put('/put', ['body' => $resource]);
```

- Array

  Use an array to send POST style requests that use a `GuzzleHttp\Post\PostBodyInterface` object as the body.

```
// You can send requests that use a POST body containing fields & files.
$client->post('/post', [
    'body' => [
        'field' => 'abc',
        'other_field' => '123',
        'file_name' => fopen('/path/to/file', 'r')
    ]
]);
```

- `GuzzleHttp\Stream\StreamInterface`

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

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

```
$request = $client->createRequest('PUT', '/put', ['json' => ['foo' => 'bar']]);
echo $request->getHeader('Content-Type');
// application/json
echo $request->getBody();
// {"foo":"bar"}
```

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.

---

query

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

Types

- array
- `GuzzleHttp\Query`

Default None

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

Query strings specified in the `query` option are combined with any query string values that are parsed from the URL.

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

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

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

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

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

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

Important: The authentication type (whether it's provided as a string or as the third option in an array) is always converted to a lowercase string. Take this into account when implementing custom authentication types and when implementing custom message factories.

Custom Authentication Schemes

You can also provide a string representing a custom authentication type name. When using a custom authentication type string, you will need to implement the authentication method in an event listener that checks the `auth` request option of a request before it is sent. Authentication listeners that require a request is not modified after they are signed should have a very low priority to ensure that they are fired last or near last in the event chain.

```
use GuzzleHttp\Event\BeforeEvent;
use GuzzleHttp\Event\RequestEvents;

/**
 * Custom authentication listener that handles the "foo" auth type.
 *
 * Listens to the "before" event of a request and only modifies the request
 * when the "auth" config setting of the request is "foo".
 */
class FooAuth implements GuzzleHttp\Event\SubscriberInterface
{
    private $password;

    public function __construct($password)
    {
        $this->password = $password;
    }

    public function getEvents()
    {
        return ['before' => ['sign', RequestEvents::SIGN_REQUEST]];
    }

    public function sign(BeforeEvent $e)
    {
        if ($e->getRequest()->getConfig()['auth'] == 'foo') {
            $e->getRequest()->setHeader('X-Foo', 'Foo ' . $this->password);
        }
    }
}
```

```
$client->getEmitter()->attach(new FooAuth('password'));
$client->get('/', ['auth' => 'foo']);
```

Adapter Specific Authentication Schemes

If you need to use authentication methods provided by cURL (e.g., NTLM, GSS, etc.), then you need to specify a curl handler option in the `options` request option array. See *config* for more information.

cookies

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

Types

- bool
- array
- `GuzzleHttp\Cookie\CookieJarInterface`

Default None

Set to `true` to use a shared cookie session associated with the client.

```
// Enable cookies using the shared cookie jar of the client.
$client->get('/get', ['cookies' => true]);
```

Pass an associative array containing cookies to send in the request and start a new cookie session.

```
// Enable cookies and send specific cookies
$client->get('/get', ['cookies' => ['foo' => 'bar']]);
```

Set to a `GuzzleHttp\Cookie\CookieJarInterface` object to use an existing cookie jar.

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

allow_redirects

Summary Describes the redirect behavior of a request

Types

- bool
- array

Default

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

Set to `false` to disable redirects.

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

Pass an associative array containing the 'max' key to specify the maximum number of redirects, 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), provide a 'referer' key to specify whether or not the "Referer" header should be added when redirecting, and provide a 'protocols' array that specifies which protocols are supported for redirects (defaults to `['http', 'https']`).

```
$res = $client->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
    ]
]);
echo $res->getStatusCode();
// 200
```

decode_content

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

Types

- string
- bool

Default `true`

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->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->get('/foo.js', ['decode_content' => 'gzip']);
```

save_to

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

Types

- string
- `fopen()` resource
- `GuzzleHttp\Stream\StreamInterface`

Default PHP temp stream

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

```
$client->get('/stream/20', ['save_to' => '/path/to/file']);
```

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

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

Pass a `GuzzleHttp\Stream\StreamInterface` object to stream the response body to an open Guzzle stream:

```
$resource = fopen('/path/to/file', 'w');
$stream = GuzzleHttp\Stream\Stream::factory($resource);
$client->get('/stream/20', ['save_to' => $stream]);
```

events

Summary An associative array mapping event names to a callable. Or an associative array containing the 'fn' key that maps to a callable, an optional 'priority' key used to specify the event priority, and an optional 'once' key used to specify if the event should remove itself the first time it is triggered.

Types array

Default None

```
use GuzzleHttp\Event\BeforeEvent;
use GuzzleHttp\Event\HeadersEvent;
use GuzzleHttp\Event\CompleteEvent;
use GuzzleHttp\Event\ErrorEvent;

$client->get('/', [
    'events' => [
        'before' => function (BeforeEvent $e) { echo 'Before'; },
        'complete' => function (CompleteEvent $e) { echo 'Complete'; },
        'error' => function (ErrorEvent $e) { echo 'Error'; },
    ]
]);
```

Here's an example of using the associative array format for control over the priority and whether or not an event should be triggered more than once.

```
$client->get('/', [
    'events' => [
        'before' => [
            'fn'       => function (BeforeEvent $e) { echo 'Before'; },
```

```
            'priority' => 100,
            'once'     => true
        ]
    ]
]);
```

subscribers

Summary Array of event subscribers to add to the request. Each value in the array must be an instance of `GuzzleHttp\Event\SubscriberInterface`.

Types array

Default None

```
use GuzzleHttp\Subscriber\History;
use GuzzleHttp\Subscriber\Mock;
use GuzzleHttp\Message\Response;

$history = new History();
$mock = new Mock([new Response(200)]);
$client->get('/', ['subscribers' => [$history, $mock]]);

echo $history;
// Outputs the request and response history
```

exceptions

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`

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

$res = $client->get('/status/500', ['exceptions' => false]);
echo $res->getStatusCode();
// 500
```

timeout

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

Types float

Default `0`

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

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`

```
// Timeout if the client fails to connect to the server in 3.14 seconds.
$client->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.

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`

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

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

// Disable validation entirely (don't do this!).
$client->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.

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

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

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

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.

---

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

Pass a string to specify a proxy for all protocols.

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

Pass an associative array to specify HTTP proxies for specific URI schemes (i.e., "http", "https").

```
$client->get('/', [
    'proxy' => [
        'http'  => 'tcp://localhost:8125', // Use this proxy with "http"
        'https' => 'tcp://localhost:9124'  // Use this proxy with "https"
    ]
]);
```

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

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

```
$client->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
```

stream

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

Types bool

Default `false`

```
$response = $client->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.

expect

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

Types

- bool
- integer

Default `1048576`

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.

version

Summary Protocol version to use with the request.

Types string, float

Default `1.1`

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

config

Summary Associative array of config options that are forwarded to a request's configuration collection. These values are used as configuration options that can be consumed by plugins and handlers.

Types array

Default None

```
$request = $client->createRequest('GET', '/get', ['config' => ['foo' => 'bar']]);
echo $request->getConfig('foo');
// 'bar'
```

Some HTTP handlers allow you to specify custom handler-specific settings. For example, you can pass custom cURL options to requests by passing an associative array in the `config` request option under the `curl` key.

```
// Use custom cURL options with the request. This example uses NTLM auth
// to authenticate with a server.
$client->get('/', [
    'config' => [
        'curl' => [
            CURLOPT_HTTPAUTH => CURLAUTH_NTLM,
            CURLOPT_USERPWD  => 'username:password'
        ]
    ]
]);
```

future

Summary Specifies whether or not a response SHOULD be an instance of a `GuzzleHttp\Message\FutureResponse` object.

Types

- bool
- string

Default `false`

By default, Guzzle requests should be synchronous. You can create asynchronous future responses by passing the `future` request option as `true`. The response will only be executed when it is used like a normal response, the `wait()` method of the response is called, or the corresponding handler that created the response is destructing and there are futures that have not been resolved.

Important: This option only has an effect if your handler can create and return future responses. However, even if a response is completed synchronously, Guzzle will ensure that a FutureResponse object is returned for API consistency.

```
$response = $client->get('/foo', ['future' => true])
    ->then(function ($response) {
        echo 'I got a response! ' . $response;
    });
```

Event Subscribers

Requests emit lifecycle events when they are transferred. A client object has a `GuzzleHttp\Common\EventEmitter` object that can be used to add event *listeners* and event *subscribers* to all requests created by the client.

Important: **Every** event listener or subscriber added to a client will be added to every request created by the client.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\BeforeEvent;

$client = new Client();

// Add a listener that will echo out requests before they are sent
$client->getEmitter()->on('before', function (BeforeEvent $e) {
    echo 'About to send request: ' . $e->getRequest();
});

$client->get('http://httpbin.org/get');
// Outputs the request as a string because of the event
```

See *Event System* for more information on the event system used in Guzzle.

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.

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

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 and Response Messages

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.

Headers

Both request and response messages contain HTTP headers.

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\Message\Request;

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

$parsed = Request::parseHeader($request, '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.

See *Request and Response Headers* for information on how the headers of a request and response can be accessed and modified.

Body

Both request and response messages can contain a body.

You can check to see if a request or response has a body using the `getBody()` method:

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

The body used in request and response objects is a `GuzzleHttp\Stream\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).

You can change the body used in a request or response using the `setBody()` method:

```
use GuzzleHttp\Stream\Stream;
$request = $client->createRequest('PUT', 'http://httpbin.org/put');
$request->setBody(Stream::factory('foo'));
```

The easiest way to create a body for a request is using the static `GuzzleHttp\Stream\Stream::factory()` method. This method accepts various inputs like strings, resources returned from `fopen()`, and other `GuzzleHttp\Stream\StreamInterface` objects.

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;
$request = $client->createRequest('PUT', 'http://httpbin.org/put', ['body' =>
↪'testing...']);

echo $request->getBody()->read(4);
// test
echo $request->getBody()->read(4);
// ing.
echo $request->getBody()->read(1024);
// ..
var_export($request->eof());
// true
```

You can find out more about Guzzle stream objects in *Streams*.

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.

Clients are used to create request messages. More precisely, clients use a `GuzzleHttp\Message\MessageFactoryInterface` to create request messages. You create requests with a client using the `createRequest()` method.

```
// Create a request but don't send it immediately
$request = $client->createRequest('GET', 'http://httpbin.org/get');
```

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 = $client->createRequest('MOVE', 'http://httpbin.org/move', ['exceptions' =>␣
↪false]);

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

$response = $client->send($request);
echo $response->getStatusCode();
// 405
```

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 **/])`

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

Request URI

The resource you are requesting with an HTTP request is identified by the path of the request, the query string, and the "Host" header of the request.

When creating a request, you can provide the entire resource URI as a URL.

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

Using the above code, you will send a request that uses `httpbin.org` as the Host header, sends the request over port 80, uses `/get` as the path, and sends `?q=foo` as the query string. All of this is parsed automatically from the provided URI.

Sometimes you don't know what the entire request will be when it is created. In these cases, you can modify the request as needed before sending it using the `createRequest()` method of the client and methods on the request that allow you to change it.

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

You can change the path of the request using `setPath()`:

```
$request->setPath('/get');
echo $request->getPath();
// /get
echo $request->getUrl();
// http://httpbin.com/get
```

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".

You can change the scheme of the request using the `setScheme()` method:

```
$request = $client->createRequest('GET', 'http://httbin.org');
$request->setScheme('https');
echo $request->getScheme();
// https
echo $request->getUrl();
// https://httpbin.com/get
```

Port

No port is necessary when using the "http" or "https" schemes, but you can override the port using `setPort()`. If you need to modify the port used with the specified scheme from the default setting, then you must use the `setPort()` method.

```php
$request = $client->createRequest('GET', 'http://httbin.org');
$request->setPort(8080);
echo $request->getPort();
// 8080
echo $request->getUrl();
// https://httpbin.com:8080/get

// Set the port back to the default value for the scheme
$request->setPort(443);
echo $request->getUrl();
// https://httpbin.com/get
```

Query string

You can get the query string of the request using the `getQuery()` method. This method returns a `GuzzleHttp\Query` object. A Query object can be accessed like a PHP array, iterated in a foreach statement like a PHP array, and cast to a string.

```php
$request = $client->createRequest('GET', 'http://httbin.org');
$query = $request->getQuery();
$query['foo'] = 'bar';
$query['baz'] = 'bam';
$query['bam'] = ['test' => 'abc'];

echo $request->getQuery();
// foo=bar&baz=bam&bam%5Btest%5D=abc

echo $request->getQuery()['foo'];
// bar
echo $request->getQuery()->get('foo');
// bar
echo $request->getQuery()->get('foo');
// bar

var_export($request->getQuery()['bam']);
// array('test' => 'abc')

foreach ($query as $key => $value) {
    var_export($value);
}

echo $request->getUrl();
// https://httpbin.com/get?foo=bar&baz=bam&bam%5Btest%5D=abc
```

Query Aggregators

Query objects can store scalar values or arrays of values. When an array of values is added to a query object, the query object uses a query aggregator to convert the complex structure into a string. Query objects will use PHP style query strings when complex query string parameters are converted to a string. You can customize how complex query string parameters are aggregated using the `setAggregator()` method of a query string object.

```php
$query->setAggregator($query::duplicateAggregator());
```

In the above example, we've changed the query object to use the "duplicateAggregator". This aggregator will allow duplicate entries to appear in a query string rather than appending "[n]" to each value. So if you had a query string with `['a' => ['b', 'c']]`, the duplicate aggregator would convert this to "a=b&a=c" while the default aggregator would convert this to "a[0]=b&a[1]=c" (with urlencoded brackets).

The `setAggregator()` method accepts a `callable` which is used to convert a deeply nested array of query string variables into a flattened array of key value pairs. The callable accepts an array of query data and returns a flattened array of key value pairs where each value is an array of strings. You can use the `GuzzleHttp\Query::walkQuery()` static function to easily create custom query aggregators.

Host

You can change the host header of the request in a predictable way using the `setHost()` method of a request:

```
$request->setHost('www.google.com');
echo $request->getHost();
// www.google.com
echo $request->getUrl();
// https://www.google.com/get?foo=bar&baz=bam
```

Note: The Host header can also be changed by modifying the Host header of a request directly, but modifying the Host header directly could result in sending a request to a different Host than what is specified in the Host header (sometimes this is actually the desired behavior).

Resource

You can use the `getResource()` method of a request to return the path and query string of a request in a single string.

```
$request = $client->createRequest('GET', 'http://httpbin.org/get?baz=bar');
echo $request->getResource();
// /get?baz=bar
```

Request Config

Request messages contain a configuration collection that can be used by event listeners and HTTP handlers to modify how a request behaves or is transferred over the wire. For example, many of the request options that are specified when creating a request are actually set as config options that are only acted upon by handlers and listeners when the request is sent.

You can get access to the request's config object using the `getConfig()` method of a request.

```
$request = $client->createRequest('GET', '/');
$config = $request->getConfig();
```

The config object is a `GuzzleHttp\Collection` object that acts like an associative array. You can grab values from the collection using array like access. You can also modify and remove values using array like access.

```
$config['foo'] = 'bar';
echo $config['foo'];
// bar
```

```
var_export(isset($config['foo']));
// true

unset($config['foo']);
var_export(isset($config['foo']));
// false

var_export($config['foo']);
// NULL
```

HTTP handlers and event listeners can expose additional customization options through request config settings. For example, in order to specify custom cURL options to the cURL handler, you need to specify an associative array in the `curl config` request option.

```
$client->get('/', [
    'config' => [
        'curl' => [
            CURLOPT_HTTPAUTH => CURLAUTH_NTLM,
            CURLOPT_USERPWD  => 'username:password'
        ]
    ]
]);
```

Consult the HTTP handlers and event listeners you are using to see if they allow customization through request configuration options.

Event Emitter

Request objects implement `GuzzleHttp\Event\HasEmitterInterface`, so they have a method called `getEmitter()` that can be used to get an event emitter used by the request. Any listener or subscriber attached to a request will only be triggered for the lifecycle events of a specific request. Conversely, adding an event listener or subscriber to a client will listen to all lifecycle events of all requests created by the client.

See *Event System* for more information.

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.

```
$response = GuzzleHttp\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.

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

When working with JSON responses, you can use the `json()` method of a response:

```
$json = $response->json();
```

Note: Guzzle uses the `json_decode()` method of PHP and uses arrays rather than `stdClass` objects for objects.

You can use the `xml()` method when working with XML data.

```
$xml = $response->xml();
```

Note: Guzzle uses the `SimpleXMLElement` objects when converting response bodies to XML.

Effective URL

The URL that was ultimately accessed that returned a response can be accessed using the `getEffectiveUrl()` method of a response. This method will return the URL of a request or the URL of the last redirected URL if any redirects occurred while transferring a request.

```
$response = GuzzleHttp\get('http://httpbin.org/get');
echo $response->getEffectiveUrl();
// http://httpbin.org/get

$response = GuzzleHttp\get('http://httpbin.org/redirect-to?url=http://www.google.com
↪');
echo $response->getEffectiveUrl();
// http://www.google.com
```

Event System

Guzzle uses an event emitter to allow you to easily extend the behavior of a request, change the response associated with a request, and implement custom error handling. All events in Guzzle are managed and emitted by an **event emitter**.

Event Emitters

Clients, requests, and any other class that implements the `GuzzleHttp\Event\HasEmitterInterface` interface have a `GuzzleHttp\Event\Emitter` object. You can add event *listeners* and event *subscribers* to an event *emitter*.

emitter An object that implements `GuzzleHttp\Event\EmitterInterface`. This object emits named events to event listeners. You may register event listeners on subscribers on an emitter.

event listeners Callable functions that are registered on an event emitter for specific events. Event listeners are registered on an emitter with a *priority* setting. If no priority is provided, `0` is used by default.

event subscribers Classes that tell an event emitter what methods to listen to and what functions on the class to invoke when the event is triggered. Event subscribers subscribe event listeners to an event emitter. They should be used when creating more complex event based logic in applications (i.e., cookie handling is implemented using an event subscriber because it's easier to share a subscriber than an anonymous function and because handling cookies is a complex process).

priority Describes the order in which event listeners are invoked when an event is emitted. The higher a priority value, the earlier the event listener will be invoked (a higher priority means the listener is more important). If no priority is provided, the priority is assumed to be `0`.

When specifying an event priority, you can pass `"first"` or `"last"` to dynamically specify the priority based on the current event priorities associated with the given event name in the emitter. Use `"first"` to set the priority to the current highest priority plus one. Use `"last"` to set the priority to the current lowest event priority minus one. It is important to remember that these dynamic priorities are calculated only at the point of insertion into the emitter and they are not rearranged after subsequent listeners are added to an emitter.

propagation Describes whether or not other event listeners are triggered. Event emitters will trigger every event listener registered to a specific event in priority order until all of the listeners have been triggered **or** until the propagation of an event is stopped.

Getting an EventEmitter

You can get the event emitter of `GuzzleHttp\Event\HasEmitterInterface` object using the `getEmitter()` method. Here's an example of getting a client object's event emitter.

```
$client = new GuzzleHttp\Client();
$emitter = $client->getEmitter();
```

Note: You'll notice that the event emitter used in Guzzle is very similar to the Symfony2 EventDispatcher component. This is because the Guzzle event system is based on the Symfony2 event system with several changes. Guzzle uses its own event emitter to improve performance, isolate Guzzle from changes to the Symfony, and provide a few improvements that make it easier to use for an HTTP client (e.g., the addition of the `once()` method).

Adding Event Listeners

After you have the emitter, you can register event listeners that listen to specific events using the `on()` method. When registering an event listener, you must tell the emitter what event to listen to (e.g., "before", "after", "progress", "complete", "error", etc.), what callable to invoke when the event is triggered, and optionally provide a priority.

```
use GuzzleHttp\Event\BeforeEvent;

$emitter->on('before', function (BeforeEvent $event) {
```

```
    echo $event->getRequest();
});
```

When a listener is triggered, it is passed an event that implements the `GuzzleHttp\Event\EventInterface` interface, the name of the event, and the event emitter itself. The above example could more verbosely be written as follows:

```
use GuzzleHttp\Event\BeforeEvent;

$emitter->on('before', function (BeforeEvent $event, $name) {
    echo $event->getRequest();
});
```

You can add an event listener that automatically removes itself after it is triggered using the `once()` method of an event emitter.

```
$client = new GuzzleHttp\Client();
$client->getEmitter()->once('before', function () {
    echo 'This will only happen once... per request!';
});
```

Event Propagation

Event listeners can prevent other event listeners from being triggered by stopping an event's propagation.

Stopping event propagation can be useful, for example, if an event listener has changed the state of the subject to such an extent that allowing subsequent event listeners to be triggered could place the subject in an inconsistent state. This technique is used in Guzzle extensively when intercepting error events with responses.

You can stop the propagation of an event using the `stopPropagation()` method of a `GuzzleHttp\Event\EventInterface` object:

```
use GuzzleHttp\Event\ErrorEvent;

$emitter->on('error', function (ErrorEvent $event) {
    $event->stopPropagation();
});
```

After stopping the propagation of an event, any subsequent event listeners that have not yet been triggered will not be triggered. You can check to see if the propagation of an event was stopped using the `isPropagationStopped()` method of the event.

```
$client = new GuzzleHttp\Client();
$emitter = $client->getEmitter();
// Note: assume that the $errorEvent was created
if ($emitter->emit('error', $errorEvent)->isPropagationStopped()) {
    echo 'It was stopped!';
}
```

Hint: When emitting events, the event that was emitted is returned from the emitter. This allows you to easily chain calls as shown in the above example.

Event Subscribers

Event subscribers are classes that implement the `GuzzleHttp\Event\SubscriberInterface` object. They are used to register one or more event listeners to methods of the class. Event subscribers tell event emitters exactly which events to listen to and what method to invoke on the class when the event is triggered by called the `getEvents()` method of a subscriber.

The following example registers event listeners to the `before` and `complete` event of a request. When the `before` event is emitted, the `onBefore` instance method of the subscriber is invoked. When the `complete` event is emitted, the `onComplete` event of the subscriber is invoked. Each array value in the `getEvents()` return value MUST contain the name of the method to invoke and can optionally contain the priority of the listener (as shown in the `before` listener in the example).

```
use GuzzleHttp\Event\EmitterInterface;
use GuzzleHttp\Event\SubscriberInterface;
use GuzzleHttp\Event\BeforeEvent;
use GuzzleHttp\Event\CompleteEvent;

class SimpleSubscriber implements SubscriberInterface
{
    public function getEvents()
    {
        return [
            // Provide name and optional priority
            'before'   => ['onBefore', 100],
            'complete' => ['onComplete'],
            // You can pass a list of listeners with different priorities
            'error'    => [['beforeError', 'first'], ['afterError', 'last']]
        ];
    }

    public function onBefore(BeforeEvent $event, $name)
    {
        echo 'Before!';
    }

    public function onComplete(CompleteEvent $event, $name)
    {
        echo 'Complete!';
    }
}
```

To register the listeners the subscriber needs to be attached to the emitter:

```
$client = new GuzzleHttp\Client();
$emitter = $client->getEmitter();
$subscriber = new SimpleSubscriber();
$emitter->attach($subscriber);

//to remove the listeners
$emitter->detach($subscriber);
```

Note: You can specify event priorities using integers or "`first`" and "`last`" to dynamically determine the priority.

Event Priorities

When adding event listeners or subscribers, you can provide an optional event priority. This priority is used to determine how early or late a listener is triggered. Specifying the correct priority is an important aspect of ensuring a listener behaves as expected. For example, if you wanted to ensure that cookies associated with a redirect were added to a cookie jar, you'd need to make sure that the listener that collects the cookies is triggered before the listener that performs the redirect.

In order to help make the process of determining the correct event priority of a listener easier, Guzzle provides several pre-determined named event priorities. These priorities are exposed as constants on the `GuzzleHttp\Event\RequestEvents` object.

last Use `"last"` as an event priority to set the priority to the current lowest event priority minus one.

first Use `"first"` as an event priority to set the priority to the current highest priority plus one.

`GuzzleHttp\Event\RequestEvents::EARLY` Used when you want a listener to be triggered as early as possible in the event chain.

`GuzzleHttp\Event\RequestEvents::LATE` Used when you want a listener to be to be triggered as late as possible in the event chain.

`GuzzleHttp\Event\RequestEvents::PREPARE_REQUEST` Used when you want a listener to be trigger while a request is being prepared during the `before` event. This event priority is used by the `GuzzleHttp\Subscriber\Prepare` event subscriber which is responsible for guessing a Content-Type, Content-Length, and Expect header of a request. You should subscribe after this event is triggered if you want to ensure that this subscriber has already been triggered.

`GuzzleHttp\Event\RequestEvents::SIGN_REQUEST` Used when you want a listener to be triggered when a request is about to be signed. Any listener triggered at this point should expect that the request object will no longer be mutated. If you are implementing a custom signature subscriber, then you should use this event priority to sign requests.

`GuzzleHttp\Event\RequestEvents::VERIFY_RESPONSE` Used when you want a listener to be triggered when a response is being validated during the `complete` event. The `GuzzleHttp\Subscriber\HttpError` event subscriber uses this event priority to check if an exception should be thrown due to a 4xx or 5xx level response status code. If you are doing any kind of verification of a response during the complete event, it should happen at this priority.

`GuzzleHttp\Event\RequestEvents::REDIRECT_RESPONSE` Used when you want a listener to be triggered when a response is being redirected during the `complete` event. The `GuzzleHttp\Subscriber\Redirect` event subscriber uses this event priority when performing redirects.

You can use the above event priorities as a guideline for determining the priority of you event listeners. You can use these constants and add to or subtract from them to ensure that a listener happens before or after the named priority.

Note: "first" and "last" priorities are not adjusted after they added to an emitter. For example, if you add a listener with a priority of "first", you can still add subsequent listeners with a higher priority which would be triggered before the listener added with a priority of "first".

Working With Request Events

Requests emit lifecycle events when they are transferred.

Important: Excluding the `end` event, request lifecycle events may be triggered multiple times due to redirects, retries, or reusing a request multiple times. Use the `once()` method want the event to be triggered once. You can also remove an event listener from an emitter by using the emitter which is provided to the listener.

before

The `before` event is emitted before a request is sent. The event emitted is a `GuzzleHttp\Event\BeforeEvent`.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\EmitterInterface;
use GuzzleHttp\Event\BeforeEvent;

$client = new Client(['base_url' => 'http://httpbin.org']);
$request = $client->createRequest('GET', '/');
$request->getEmitter()->on(
    'before',
    function (BeforeEvent $e, $name) {
        echo $name . "\n";
        // "before"
        echo $e->getRequest()->getMethod() . "\n";
        // "GET" / "POST" / "PUT" / etc.
        echo get_class($e->getClient());
        // "GuzzleHttp\Client"
    }
);
```

You can intercept a request with a response before the request is sent over the wire. The `intercept()` method of the `BeforeEvent` accepts a `GuzzleHttp\Message\ResponseInterface`. Intercepting the event will prevent the request from being sent over the wire and stops the propagation of the `before` event, preventing subsequent event listeners from being invoked.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\BeforeEvent;
use GuzzleHttp\Message\Response;

$client = new Client(['base_url' => 'http://httpbin.org']);
$request = $client->createRequest('GET', '/status/500');
$request->getEmitter()->on('before', function (BeforeEvent $e) {
    $response = new Response(200);
    $e->intercept($response);
});

$response = $client->send($request);
echo $response->getStatusCode();
// 200
```

Attention: Any exception encountered while executing the `before` event will trigger the `error` event of a request.

complete

The `complete` event is emitted after a transaction completes and an entire response has been received. The event is a `GuzzleHttp\Event\CompleteEvent`.

You can intercept the `complete` event with a different response if needed using the `intercept()` method of the event. This can be useful, for example, for changing the response for caching.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\CompleteEvent;
use GuzzleHttp\Message\Response;

$client = new Client(['base_url' => 'http://httpbin.org']);
$request = $client->createRequest('GET', '/status/302');
$cachedResponse = new Response(200);

$request->getEmitter()->on(
    'complete',
    function (CompleteEvent $e) use ($cachedResponse) {
        if ($e->getResponse()->getStatusCode() == 302) {
            // Intercept the original transaction with the new response
            $e->intercept($cachedResponse);
        }
    }
);

$response = $client->send($request);
echo $response->getStatusCode();
// 200
```

Attention: Any `GuzzleHttp\Exception\RequestException` encountered while executing the `complete` event will trigger the `error` event of a request.

error

The `error` event is emitted when a request fails (whether it's from a networking error or an HTTP protocol error). The event emitted is a `GuzzleHttp\Event\ErrorEvent`.

This event is useful for retrying failed requests. Here's an example of retrying failed basic auth requests by re-sending the original request with a username and password.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\ErrorEvent;

$client = new Client(['base_url' => 'http://httpbin.org']);
$request = $client->createRequest('GET', '/basic-auth/foo/bar');
$request->getEmitter()->on('error', function (ErrorEvent $e) {
    if ($e->getResponse()->getStatusCode() == 401) {
        // Add authentication stuff as needed and retry the request
        $e->getRequest()->setHeader('Authorization', 'Basic ' . base64_encode('foo:bar
↪'));
        // Get the client of the event and retry the request
        $newResponse = $e->getClient()->send($e->getRequest());
        // Intercept the original transaction with the new response
        $e->intercept($newResponse);
```

```
    }
});
```

Attention: If an `error` event is intercepted with a response, then the `complete` event of a request is triggered. If the `complete` event fails, then the `error` event is triggered once again.

progress

The `progress` event is emitted when data is uploaded or downloaded. The event emitted is a `GuzzleHttp\Event\ProgressEvent`.

You can access the emitted progress values using the corresponding public properties of the event object:

- `$downloadSize`: The number of bytes that will be downloaded (if known)
- `$downloaded`: The number of bytes that have been downloaded
- `$uploadSize`: The number of bytes that will be uploaded (if known)
- `$uploaded`: The number of bytes that have been uploaded

This event cannot be intercepted.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\ProgressEvent;

$client = new Client(['base_url' => 'http://httpbin.org']);
$request = $client->createRequest('PUT', '/put', [
    'body' => str_repeat('.', 100000)
]);

$request->getEmitter()->on('progress', function (ProgressEvent $e) {
    echo 'Downloaded ' . $e->downloaded . ' of ' . $e->downloadSize . ' '
        . 'Uploaded ' . $e->uploaded . ' of ' . $e->uploadSize . "\r";
});

$client->send($request);
echo "\n";
```

end

The `end` event is a terminal event, emitted once per request, that provides access to the response that was received or the exception that was encountered. The event emitted is a `GuzzleHttp\Event\EndEvent`.

This event can be intercepted, but keep in mind that the `complete` event will not fire after intercepting this event.

```
use GuzzleHttp\Client;
use GuzzleHttp\Event\EndEvent;

$client = new Client(['base_url' => 'http://httpbin.org']);
$request = $client->createRequest('PUT', '/put', [
    'body' => str_repeat('.', 100000)
]);

$request->getEmitter()->on('end', function (EndEvent $e) {
```

```
    if ($e->getException()) {
        echo 'Error: ' . $e->getException()->getMessage();
    } else {
        echo 'Response: ' . $e->getResponse();
    }
});

$client->send($request);
echo "\n";
```

Streams

Guzzle uses 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.

Guzzle's 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).

Creating Streams

The best way to create a stream is using the static factory method, `GuzzleHttp\Stream\Stream::factory()`. This factory accepts strings, resources returned from `fopen()`, an object that implements `__toString()`, and an object that implements `GuzzleHttp\Stream\StreamInterface`.

```
use GuzzleHttp\Stream\Stream;

$stream = Stream::factory('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
```

Metadata

Guzzle 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.

```
use GuzzleHttp\Stream\Stream;

$resource = fopen('/path/to/file', 'r');
$stream = Stream::factory($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

With the small and focused interface, add custom functionality to streams is very simple with stream decorators. Guzzle provides several built-in decorators that provide additional stream functionality.

CachingStream

The CachingStream is used to allow seeking over previously read bytes on non-seekable streams. This can be useful when transferring a non-seekable entity body fails due to needing to rewind the stream (for example, resulting from a redirect). Data that is read from the remote stream will be buffered in a PHP temp stream so that previously read bytes are cached first in memory, then on disk.

```
use GuzzleHttp\Stream\Stream;
use GuzzleHttp\Stream\CachingStream;

$original = Stream::factory(fopen('http://www.google.com', 'r'));
$stream = new CachingStream($original);

$stream->read(1024);
echo $stream->tell();
// 1024

$stream->seek(0);
echo $stream->tell();
// 0
```

LimitStream

LimitStream can be used to read a subset or slice of an existing stream object. This can be useful for breaking a large file into smaller pieces to be sent in chunks (e.g. Amazon S3's multipart upload API).

```
use GuzzleHttp\Stream\Stream;
use GuzzleHttp\Stream\LimitStream;

$original = Stream::factory(fopen('/tmp/test.txt', 'r+'));
```

```php
echo $original->getSize();
// >>> 1048576

// Limit the size of the body to 1024 bytes and start reading from byte 2048
$stream = new LimitStream($original, 1024, 2048);
echo $stream->getSize();
// >>> 1024
echo $stream->tell();
// >>> 0
```

NoSeekStream

NoSeekStream wraps a stream and does not allow seeking.

```php
use GuzzleHttp\Stream\Stream;
use GuzzleHttp\Stream\LimitStream;

$original = Stream::factory('foo');
$noSeek = new NoSeekStream($original);

echo $noSeek->read(3);
// foo
var_export($noSeek->isSeekable());
// false
$noSeek->seek(0);
var_export($noSeek->read(3));
// NULL
```

Creating Custom Decorators

Creating a stream decorator is very easy thanks to the `GuzzleHttp\Stream\StreamDecoratorTrait`. This trait provides methods that implement `GuzzleHttp\Stream\StreamInterface` by proxying to an underlying stream. Just `use` the `StreamDecoratorTrait` and implement your custom methods.

For example, let's say we wanted to call a specific function each time the last byte is read from a stream. This could be implemented by overriding the `read()` method.

```php
use GuzzleHttp\Stream\StreamDecoratorTrait;

class EofCallbackStream implements StreamInterface
{
    use StreamDecoratorTrait;

    private $callback;

    public function __construct(StreamInterface $stream, callable $callback)
    {
        $this->stream = $stream;
        $this->callback = $callback;
    }

    public function read($length)
    {
        $result = $this->stream->read($length);
```

```
        // Invoke the callback when EOF is hit.
        if ($this->eof()) {
            call_user_func($this->callback);
        }

        return $result;
    }
}
```

This decorator could be added to any existing stream and used like so:

```
use GuzzleHttp\Stream\Stream;

$original = Stream::factory('foo');
$eofStream = new EofCallbackStream($original, function () {
    echo 'EOF!';
});

$eofStream->read(2);
$eofStream->read(1);
// echoes "EOF!"
$eofStream->seek(0);
$eofStream->read(3);
// echoes "EOF!"
```

RingPHP Handlers

Guzzle uses RingPHP handlers to send HTTP requests over the wire. RingPHP provides a low-level library that can be used to "glue" Guzzle with any transport method you choose. By default, Guzzle utilizes cURL and PHP's stream wrappers to send HTTP requests.

RingPHP handlers makes it extremely simple to integrate Guzzle with any HTTP transport. For example, you could quite easily bridge Guzzle and React to use Guzzle in React's event loop.

Using a handler

You can change the handler used by a client using the `handler` option in the `GuzzleHttp\Client` constructor.

```
use GuzzleHttp\Client;
use GuzzleHttp\Ring\Client\MockHandler;

// Create a mock handler that always returns a 200 response.
$handler = new MockHandler(['status' => 200]);

// Configure to client to use the mock handler.
$client = new Client(['handler' => $handler]);
```

At its core, handlers are simply PHP callables that accept a request array and return a `GuzzleHttp\Ring\Future\FutureArrayInterface`. This future array can be used just like a normal PHP array, causing it to block, or you can use the promise interface using the `then()` method of the future. Guzzle hooks up to the RingPHP project using a very simple bridge class (`GuzzleHttp\RingBridge`).

Creating a handler

See the RingPHP project documentation for more information on creating custom handlers that can be used with Guzzle clients.

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 subscriber
- Mock handler
- Node.js web server for integration testing

Mock Subscriber

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 subscriber that can be attached to clients or requests that allows you to queue up a list of responses to use rather than hitting a remote API.

```
use GuzzleHttp\Client;
use GuzzleHttp\Subscriber\Mock;
use GuzzleHttp\Message\Response;

$client = new Client();

// Create a mock subscriber and queue two responses.
$mock = new Mock([
    new Response(200, ['X-Foo' => 'Bar']),          // Use response object
    "HTTP/1.1 202 OK\r\nContent-Length: 0\r\n\r\n"  // Use a response string
]);

// Add the mock subscriber to the client.
$client->getEmitter()->attach($mock);
// The first request is intercepted with the first response.
echo $client->get('/')->getStatusCode();
//> 200
// The second request is intercepted with the second response.
echo $client->get('/')->getStatusCode();
//> 202
```

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

History Subscriber

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

```php
use GuzzleHttp\Client;
use GuzzleHttp\Subscriber\History;

$client = new Client();
$history = new History();

// Add the history subscriber to the client.
$client->getEmitter()->attach($history);

$client->get('http://httpbin.org/get');
$client->head('http://httpbin.org/get');

// Count the number of transactions
echo count($history);
//> 2
// Get the last request
$lastRequest = $history->getLastRequest();
// Get the last response
$lastResponse = $history->getLastResponse();

// Iterate over the transactions that were sent
foreach ($history as $transaction) {
    echo $transaction['request']->getMethod();
    //> GET, HEAD
    echo $transaction['response']->getStatusCode();
    //> 200, 200
}
```

The history subscriber can also be printed, revealing the requests and responses that were sent as a string, in order.

```php
echo $history;
```

```
> GET /get HTTP/1.1
Host: httpbin.org
User-Agent: Guzzle/4.0-dev curl/7.21.4 PHP/5.5.8

< HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-Type: application/json
Date: Tue, 25 Mar 2014 03:53:27 GMT
Server: gunicorn/0.17.4
Content-Length: 270
Connection: keep-alive

{
  "headers": {
    "Connection": "close",
    "X-Request-Id": "3d0f7d5c-c937-4394-8248-2b8e03fcccdb",
    "User-Agent": "Guzzle/4.0-dev curl/7.21.4 PHP/5.5.8",
    "Host": "httpbin.org"
  },
  "origin": "76.104.247.1",
  "args": {},
  "url": "http://httpbin.org/get"
}

> HEAD /get HTTP/1.1
Host: httpbin.org
```

```
User-Agent: Guzzle/4.0-dev curl/7.21.4 PHP/5.5.8

< HTTP/1.1 200 OK
Access-Control-Allow-Origin: *
Content-length: 270
Content-Type: application/json
Date: Tue, 25 Mar 2014 03:53:27 GMT
Server: gunicorn/0.17.4
Connection: keep-alive
```

Mock Adapter

In addition to using the Mock subscriber, you can use the `GuzzleHttp\Ring\Client\MockHandler` as the handler of a client to return the same response over and over or return the result of a callable function.

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 subcribers and History subscriber.

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.

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

// Start the server and queue a response
Server::enqueue("HTTP/1.1 200 OK\r\n\Content-Length: 0r\n\r\n");
```

```
$client = new Client(['base_url' => Server::$url]);
echo $client->get('/foo')->getStatusCode();
// 200
```

`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 a string representing an HTTP response message, a `GuzzleHttp\Message\ResponseInterface`, or an array of HTTP message strings / `GuzzleHttp\Message\ResponseInterface` objects.

```
// Queue single response
Server::enqueue("HTTP/1.1 200 OK\r\n\Content-Length: 0r\n\r\n");

// Clear the queue and queue an array of responses
Server::enqueue([
    "HTTP/1.1 200 OK\r\n\Content-Length: 0r\n\r\n",
    "HTTP/1.1 404 Not Found\r\n\Content-Length: 0r\n\r\n"
]);
```

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()`. This method accepts an optional `$hydrate` parameter that specifies if you are retrieving an array of HTTP requests as strings or an array of `GuzzleHttp\Message\RequestInterface` objects.

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

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

Why should I use Guzzle?

Guzzle makes it easy to send HTTP requests and super simple to integrate with web services. Guzzle manages things like persistent connections, represents query strings as collections, makes it simple to send streaming POST requests with fields and files, and abstracts away the underlying HTTP transport layer. By providing an object oriented interface for HTTP clients, requests, responses, headers, and message bodies, Guzzle makes it so that you no longer need to fool around with cURL options, stream contexts, or sockets.

Asynchronous and Synchronous Requests

Guzzle allows you to send both asynchronous and synchronous requests using the same interface and no direct dependency on an event loop. This flexibility allows Guzzle to send an HTTP request using the most appropriate HTTP handler based on the request being sent. For example, when sending synchronous requests, Guzzle will by default send requests using cURL easy handles to ensure you're using the fastest possible method for serially transferring HTTP requests. When sending asynchronous requests, Guzzle might use cURL's multi interface or any other asynchronous handler you configure. When you request streaming data, Guzzle will by default use PHP's stream wrapper.

Streams

Request and response message bodies use *Guzzle Streams*, allowing you to stream data without needing to load it all into memory. Guzzle's stream layer provides a large suite of functionality:

- You can modify streams at runtime using custom or a number of pre-made decorators.
- You can emit progress events as data is read from a stream.
- You can validate the integrity of a stream using a rolling hash as data is read from a stream.

Event System and Plugins

Guzzle's event system allows you to completely modify the behavior of a client or request at runtime to cater them for any API. You can send a request with a client, and the client can do things like automatically retry your request if it fails, automatically redirect, log HTTP messages that are sent over the wire, emit progress events as data is uploaded and downloaded, sign requests using OAuth 1.0, verify the integrity of messages before and after they are sent over the wire, and anything else you might need.

Testable

Another important aspect of Guzzle is that it's really *easy to test clients*. You can mock HTTP responses and when testing an handler implementation, Guzzle provides a mock node.js web server.

Ecosystem

Guzzle has a large ecosystem of plugins, including service descriptions which allows you to abstract web services using service descriptions. These service descriptions define how to serialize an HTTP request and how to parse an HTTP response into a more meaningful model object.

- Guzzle Command: Provides the building blocks for service description abstraction.
- Guzzle Services: Provides an implementation of "Guzzle Command" that utilizes Guzzle's service description format.

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 a RingPHP 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. Pass the `future` true request option to a request to send it asynchronously. Guzzle will then return a `GuzzleHttp\Message\FutureResponse` object that can be used synchronously by accessing the response object like a normal response, and it can be used asynchronously using a promise that is notified when the response is resolved with a real response or rejected with an exception.

```
$request = $client->createRequest('GET', ['future' => true]);
$client->send($request)->then(function ($response) {
    echo 'Got a response! ' . $response;
});
```

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

```
$request = $client->createRequest('GET', ['future' => true]);
$futureResponse = $client->send($request);
$futureResponse->wait();
```

How can I add custom cURL options?

cURL offer 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 the **config** request option.

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

```
$client->get('/', [
    'config' => [
        '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 **config** 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->get('/', [
    'stream' => true,
    'config' => [
        '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->put('/', [], 'the body', [
    'expect' => false
]);

// Disable the expect header on all client requests
$client->setDefaultOption('expect', false)
```

CHAPTER 2

HTTP Components

There are a number of optional libraries you can use along with Guzzle's HTTP layer to add capabilities to the client.

Log Subscriber Logs HTTP requests and responses sent over the wire using customizable log message templates.

OAuth Subscriber Signs requests using OAuth 1.0.

Cache Subscriber Implements a private transparent proxy cache that caches HTTP responses.

Retry Subscriber Retries failed requests using customizable retry strategies (e.g., retry based on response status code, cURL error codes, etc.)

Message Integrity Subscriber Verifies the message integrity of HTTP responses using customizable validators. This plugin can be used, for example, to verify the Content-MD5 headers of responses.

CHAPTER 3

Service Description Commands

You can use the **Guzzle Command** library to encapsulate interaction with a web service using command objects. Building on top of Guzzle's command abstraction allows you to easily implement things like service description that can be used to serialize requests and parse responses using a meta-description of a web service.

Guzzle Command Provides the foundational elements used to build high-level, command based, web service clients with Guzzle.

Guzzle Services Provides an implementation of the *Guzzle Command* library that uses Guzzle service descriptions to describe web services, serialize requests, and parse responses into easy to use model structures.