Ya Retorno Boleto Documentation

Versão 1.0.0

Italo Lelis de Vietro

26/02/2016

CAPÍTULO 1

Iniciando

1.1 Bem vindo ao YA Retorno Boleto

1.1.1 O que é?

Yet Another Retorno Boleto é uma biblioteca em PHP para leitura de arquivos de retorno de títulos de cobrança de bancos brasileiros.

Principais funcionalidades

- Parser de arquivos de retorno da FEBRABAN em uma unica interface.
- Fácil extensão para funcionar com qualquer arquivo de retorno não suportado.

```php
use Umbrella\Ya\RetornoBoleto\ProcessFactory;
use Umbrella\Ya\RetornoBoleto\ProcessHandler;

// Utilizamos a factory para construir o objeto correto para um determinado arquivo de retorno
$cnab = ProcessFactory::getRetorno('arquivo-retorno.ret');

// Passamos o objeto contruido para o handler
$processor = new ProcessHandler($cnab);

// Processamos o arquivo. Isso retornará um objeto parseado com todas as propriedades do arquvio.
$retorno = $processor->processar();
```

License

Licensed using the MIT license.

> The MIT License (MIT)
>
> Copyright (c) 2014 Umbrella Tech
>
> 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:

1.2 Instalação

1.2.1 Requisitos

1. PHP 5.3.3+ compilado com a extensão cURL
2. A versão atual do cURL 7.16.2+ compilado com OpenSSL e zlib

1.2.2 Instalando YA Retorno Boleto

Composer

A maneira recomendada de instalar YA Retorno Boleto é com o 'Composer <http://getcomposer.org> '_. Composer é uma ferramenta de gerenciamento de dependência para PHP que lhe permite declarar as dependências que o seu projeto precisa e instala-los em seu projeto.

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

# Adicionando YA Retorno Boleto como dependencia
php composer.phar require umbrella/retorno-boleto:~1.2
```

Após a instalação, é necessário carregar o autoloader do composer:

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

Você pode encontrar mais informações sobre como instalar o Composer, configurar o carregamento automático, e outras boas práticas para a definição dependências em *getcomposer.org <http://getcomposer.org>* _.

Mantendo-se atualizado

Durante o desenvolvimento, você pode manter-se com as últimas alterações do branch master, definindo a versão do YA Retorno Boleto para '' dev-master''.

```
{
    "require": {
        "umbrella/retorno-boleto": "dev-master"
    }
}
```

CAPÍTULO 2

O componente

2.1 The Guzzle HTTP client

Guzzle gives PHP developers complete control over HTTP requests while utilizing HTTP/1.1 best practices. Guzzle's HTTP functionality is a robust framework built on top of the PHP libcurl bindings.

The three main parts of the Guzzle HTTP client are:

Clients	`Guzzle\Http\Client` (creates and sends requests, associates a response with a request)
Requests	`Guzzle\Http\Message\Request` (requests with no body), `Guzzle\Http\Message\EntityEnclosingRequest` (requests with a body)
Responses	`Guzzle\Http\Message\Response`

2.1.1 Creating a Client

Clients create requests, send requests, and set responses on a request object. When instantiating a client object, you can pass an optional "base URL" and optional array of configuration options. A base URL is a `URI template` that contains the URL of a remote server. When creating requests with a relative URL, the base URL of a client will be merged into the request's URL.

```
use Guzzle\Http\Client;

// Create a client and provide a base URL
$client = new Client('https://api.github.com');

$request = $client->get('/user');
$request->setAuth('user', 'pass');
echo $request->getUrl();
// >>> https://api.github.com/user

// You must send a request in order for the transfer to occur
$response = $request->send();

echo $response->getBody();
// >>> {"type":"User", ...

echo $response->getHeader('Content-Length');
// >>> 792

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

```
echo $data['type'];
// >>> User
```

Base URLs

Notice that the URL provided to the client's `get()` method is relative. Relative URLs will always merge into the base URL of the client. There are a few rules that control how the URLs are merged.

Dica: Guzzle follows RFC 3986 when merging base URLs and relative URLs.

In the above example, we passed `/user` to the `get()` method of the client. This is a relative URL, so it will merge into the base URL of the client– resulting in the derived URL of `https://api.github.com/users`.

`/user` is a relative URL but uses an absolute path because it contains the leading slash. Absolute paths will overwrite any existing path of the base URL. If an absolute path is provided (e.g. `/path/to/something`), then the path specified in the base URL of the client will be replaced with the absolute path, and the query string provided by the relative URL will replace the query string of the base URL.

Omitting the leading slash and using relative paths will add to the path of the base URL of the client. So using a client base URL of `https://api.twitter.com/v1.1` and creating a GET request with `statuses/user_timeline.json` will result in a URL of `https://api.twitter.com/v1.1/statuses/user_timeline.json`. If a relative path and a query string are provided, then the relative path will be appended to the base URL path, and the query string provided will be merged into the query string of the base URL.

If an absolute URL is provided (e.g. `http://httpbin.org/ip`), then the request will completely use the absolute URL as-is without merging in any of the URL parts specified in the base URL.

Configuration options

The second argument of the client's constructor is an array of configuration data. This can include URI template data or special options that alter the client's behavior:

`request.options`	Associative array of *Request options* to apply to every request created by the client.
`redirect.disable`	Disable HTTP redirects for every request created by the client.
`curl.options`	Associative array of cURL options to apply to every request created by the client. if either the key or value of an entry in the array is a string, Guzzle will attempt to find a matching defined cURL constant automatically (e.g. "CURLOPT_PROXY" will be converted to the constant `CURLOPT_PROXY`).
`ssl.certificate_authority`	Set to true to use the Guzzle bundled SSL certificate bundle (this is used by default, 'system' to use the bundle on your system, a string pointing to a file to use a specific certificate file, a string pointing to a directory to use multiple certificates, or `false` to disable SSL validation (not recommended). When using Guzzle inside of a phar file, the bundled SSL certificate will be extracted to your system's temp folder, and each time a client is created an MD5 check will be performed to ensure the integrity of the certificate.
`command.params`	When using a `Guzzle\Service\Client` object, this is an associative array of default options to set on each command created by the client.

Here's an example showing how to set various configuration options, including default headers to send with each request, default query string parameters to add to each request, a default auth scheme for each request, and a proxy to use for each request. Values can be injected into the client's base URL using variables from the configuration array.

```
use Guzzle\Http\Client;

$client = new Client('https://api.twitter.com/{version}', array(
    'version'        => 'v1.1',
    'request.options' => array(
        'headers' => array('Foo' => 'Bar'),
        'query'   => array('testing' => '123'),
        'auth'    => array('username', 'password', 'Basic|Digest|NTLM|Any'),
        'proxy'   => 'tcp://localhost:80'
    )
));
```

Setting a custom User-Agent

The default Guzzle User-Agent header is `Guzzle/<Guzzle_Version> curl/<curl_version> PHP/<PHP_VERSION>`. You can customize the User-Agent header of a client by calling the `setUserAgent()` method of a Client object.

```
// Completely override the default User-Agent
$client->setUserAgent('Test/123');

// Prepend a string to the default User-Agent
$client->setUserAgent('Test/123', true);
```

2.1.2 Creating requests with a client

A Client object exposes several methods used to create Request objects:

- Create a custom HTTP request: `$client->createRequest($method, $uri, array $headers, $body, $options)`
- Create a GET request: `$client->get($uri, array $headers, $options)`
- Create a HEAD request: `$client->head($uri, array $headers, $options)`
- Create a DELETE request: `$client->delete($uri, array $headers, $body, $options)`
- Create a POST request: `$client->post($uri, array $headers, $postBody, $options)`
- Create a PUT request: `$client->put($uri, array $headers, $body, $options)`
- Create a PATCH request: `$client->patch($uri, array $headers, $body, $options)`

```php
use Guzzle\Http\Client;

$client = new Client('http://baseurl.com/api/v1');

// Create a GET request using Relative to base URL
// URL of the request: http://baseurl.com/api/v1/path?query=123&value=abc)
$request = $client->get('path?query=123&value=abc');
$response = $request->send();

// Create HEAD request using a relative URL with an absolute path
// URL of the request: http://baseurl.com/path?query=123&value=abc
$request = $client->head('/path?query=123&value=abc');
$response = $request->send();

// Create a DELETE request using an absolute URL
$request = $client->delete('http://www.example.com/path?query=123&value=abc');
$response = $request->send();

// Create a PUT request using the contents of a PHP stream as the body
// Specify custom HTTP headers
$request = $client->put('http://www.example.com/upload', array(
    'X-Header' => 'My Header'
), fopen('http://www.test.com/', 'r'));
$response = $request->send();

// Create a POST request and add the POST files manually
$request = $client->post('http://localhost:8983/solr/update')
    ->addPostFiles(array('file' => '/path/to/documents.xml'));
$response = $request->send();

// Check if a resource supports the DELETE method
$supportsDelete = $client->options('/path')->send()->isMethodAllowed('DELETE');
$response = $request->send();
```

Client objects create Request objects using a request factory (`Guzzle\Http\Message\RequestFactoryInterface`). You can inject a custom request factory into the Client using `$client->setRequestFactory()`, but you can typically rely on a Client's default request factory.

2.1.3 Static clients

You can use Guzzle's static client facade to more easily send simple HTTP requests.

```
// Mount the client so that you can access it at \Guzzle
Guzzle\Http\StaticClient::mount();
$response = Guzzle::get('http://guzzlephp.org');
```

Each request method of the static client (e.g. `get()`, `post()` `, ``put()`, etc) accepts an associative array of request options to apply to the request.

```
$response = Guzzle::post('http://test.com', array(
    'headers' => array('X-Foo' => 'Bar'),
    'body'    => array('Test' => '123'),
    'timeout' => 10
));
```

2.1.4 Request options

Request options can be specified when creating a request or in the `request.options` parameter of a client. These options can control various aspects of a request including: headers to send, query string data, where the response should be downloaded, proxies, auth, etc.

headers

Associative array of headers to apply to the request. When specified in the `$options` argument of a client creational method (e.g. `get()`, `post()`, etc), the headers in the `$options` array will overwrite headers specified in the `$headers` array.

```
$request = $client->get($url, array(), array(
    'headers' => array('X-Foo' => 'Bar')
));
```

Headers can be specified on a client to add default headers to every request sent by a client.

```
$client = new Guzzle\Http\Client();

// Set a single header using path syntax
$client->setDefaultOption('headers/X-Foo', 'Bar');

// Set all headers
$client->setDefaultOption('headers', array('X-Foo' => 'Bar'));
```

Nota: In addition to setting request options when creating requests or using the `setDefaultOption()` method, any default client request option can be set using a client's config object:

```
$client->getConfig()->setPath('request.options/headers/X-Foo', 'Bar');
```

query

Associative array of query string parameters to the request. When specified in the `$options` argument of a client creational method, the query string parameters in the `$options` array will overwrite query string parameters specified in the *$url*.

```
$request = $client->get($url, array(), array(
    'query' => array('abc' => '123')
));
```

Query string parameters can be specified on a client to add default query string parameters to every request sent by a client.

```
$client = new Guzzle\Http\Client();

// Set a single query string parameter using path syntax
$client->setDefaultOption('query/abc', '123');

// Set an array of default query string parameters
$client->setDefaultOption('query', array('abc' => '123'));
```

body

Sets the body of a request. The value supplied to the body option can be a `Guzzle\Http\EntityBodyInterface`, string, fopen resource, or array when sending POST requests. When a `body` request option is supplied, the option value will overwrite the `$body` argument of a client creational method.

auth

Specifies and array of HTTP authorization parameters parameters to use with the request. The array must contain the username in index [0], the password in index [1], and can optionally contain the authentication type in index [2]. The available authentication types are: "Basic" (default), "Digest", "NTLM", or "Any".

```
$request = $client->get($url, array(), array(
    'auth' => array('username', 'password', 'Digest')
));

// You can add auth headers to every request of a client
$client->setDefaultOption('auth', array('username', 'password', 'Digest'));
```

cookies

Specifies an associative array of cookies to add to the request.

allow_redirects

Specifies whether or not the request should follow redirects. Requests will follow redirects by default. Set `allow_redirects` to `false` to disable redirects.

save_to

The `save_to` option specifies where the body of a response is downloaded. You can pass the path to a file, an fopen resource, or a `Guzzle\Http\EntityBodyInterface` object.

See *Changing where a response is downloaded* for more information on setting the *save_to* option.

events

The *events* option makes it easy to attach listeners to the various events emitted by a request object. The *events* options must be an associative array mapping an event name to a Closure or array the contains a Closure and the priority of the event.

```
$request = $client->get($url, array(), array(
    'events' => array(
        'request.before_send' => function (\Guzzle\Common\Event $e) {
            echo 'About to send ' . $e['request'];
        }
    )
));

// Using the static client:
Guzzle::get($url, array(
    'events' => array(
        'request.before_send' => function (\Guzzle\Common\Event $e) {
            echo 'About to send ' . $e['request'];
        }
    )
));
```

plugins

The *plugins* options makes it easy to attach an array of plugins to a request.

```
// Using the static client:
Guzzle::get($url, array(
    'plugins' => array(
        new Guzzle\Plugin\Cache\CachePlugin(),
        new Guzzle\Plugin\Cookie\CookiePlugin()
    )
));
```

exceptions

The *exceptions* option can be used to disable throwing exceptions for unsuccessful HTTP response codes (e.g. 404, 500, etc). Set *exceptions* to false to not throw exceptions.

params

The *params* options can be used to specify an associative array of data parameters to add to a request. Note that these are not query string parameters.

timeout / connect_timeout

You can specify the maximum number of seconds to allow for an entire transfer to take place before timing out using the *timeout* request option. You can specify the maximum number of seconds to wait while trying to connect using the *connect_timeout* request option. Set either of these options to 0 to wait indefinitely.

```
$request = $client->get('http://www.example.com', array(), array(
    'timeout'         => 20,
    'connect_timeout' => 1.5
));
```

verify

Set to true to enable SSL certificate validation (the default), false to disable SSL certificate validation, or supply the path to a CA bundle to enable verification using a custom certificate.

cert

The *cert* option lets you specify a PEM formatted SSL client certificate to use with servers that require one. If the certificate requires a password, provide an array with the password as the second item.

This would typically be used in conjunction with the *ssl_key* option.

```
$request = $client->get('https://www.example.com', array(), array(
    'cert' => '/etc/pki/client_certificate.pem'
)

$request = $client->get('https://www.example.com', array(), array(
    'cert' => array('/etc/pki/client_certificate.pem', 's3cr3tp455w0rd')
)
```

ssl_key

The *ssl_key* option lets you specify a file containing your PEM formatted private key, optionally protected by a password. Note: your password is sensitive, keep the PHP script containing it safe.

This would typically be used in conjunction with the *cert* option.

```
$request = $client->get('https://www.example.com', array(), array(
    'ssl_key' => '/etc/pki/private_key.pem'
)

$request = $client->get('https://www.example.com', array(), array(
    'ssl_key' => array('/etc/pki/private_key.pem', 's3cr3tp455w0rd')
)
```

proxy

The *proxy* option is used to specify an HTTP proxy (e.g. *http://username:password@192.168.16.1:10*).

debug

The *debug* option is used to show verbose cURL output for a transfer.

stream

When using a static client, you can set the *stream* option to true to return a *GuzzleStreamStream* object that can be used to pull data from a stream as needed (rather than have cURL download the entire contents of a response to a stream all at once).

```
$stream = Guzzle::get('http://guzzlephp.org', array('stream' => true));
while (!$stream->feof()) {
    echo $stream->readLine();
}
```

2.1.5 Sending requests

Requests can be sent by calling the `send()` method of a Request object, but you can also send requests using the `send()` method of a Client.

```
$request = $client->get('http://www.amazon.com');
$response = $client->send($request);
```

Sending requests in parallel

The Client's `send()` method accept a single `Guzzle\Http\Message\RequestInterface` object or an array of RequestInterface objects. When an array is specified, the requests will be sent in parallel.

Sending many HTTP requests serially (one at a time) can cause an unnecessary delay in a script's execution. Each request must complete before a subsequent request can be sent. By sending requests in parallel, a pool of HTTP requests can complete at the speed of the slowest request in the pool, significantly reducing the amount of time needed to execute multiple HTTP requests. Guzzle provides a wrapper for the curl_multi functions in PHP.

Here's an example of sending three requests in parallel using a client object:

```
use Guzzle\Common\Exception\MultiTransferException;

try {
    $responses = $client->send(array(
        $client->get('http://www.google.com/'),
        $client->head('http://www.google.com/'),
        $client->get('https://www.github.com/')
    ));
} catch (MultiTransferException $e) {

    echo "The following exceptions were encountered:\n";
    foreach ($e as $exception) {
        echo $exception->getMessage() . "\n";
    }

    echo "The following requests failed:\n";
    foreach ($e->getFailedRequests() as $request) {
        echo $request . "\n\n";
    }

    echo "The following requests succeeded:\n";
    foreach ($e->getSuccessfulRequests() as $request) {
        echo $request . "\n\n";
    }
}
```

If the requests succeed, an array of `Guzzle\Http\Message\Response` objects are returned. A single request failure will not cause the entire pool of requests to fail. Any exceptions thrown while transferring a pool of requests will be aggregated into a `Guzzle\Common\Exception\MultiTransferException` exception.

2.1.6 Plugins and events

Guzzle provides easy to use request plugins that add behavior to requests based on signal slot event notifications powered by the Symfony2 Event Dispatcher component. Any event listener or subscriber attached to a Client object will automatically be attached to each request created by the client.

Using the same cookie session for each request

Attach a `Guzzle\Plugin\Cookie\CookiePlugin` to a client which will in turn add support for cookies to every request created by a client, and each request will use the same cookie session:

```
use Guzzle\Plugin\Cookie\CookiePlugin;
use Guzzle\Plugin\Cookie\CookieJar\ArrayCookieJar;

// Create a new cookie plugin
$cookiePlugin = new CookiePlugin(new ArrayCookieJar());

// Add the cookie plugin to the client
$client->addSubscriber($cookiePlugin);
```

Events emitted from a client

A `Guzzle\Http\Client` object emits the following events:

Event name	Description	Event data
client.create_request	Called when a client creates a request	• client: The client • request: The created request

```
use Guzzle\Common\Event;
use Guzzle\Http\Client;

$client = new Client();

// Add a listener that will echo out requests as they are created
$client->getEventDispatcher()->addListener('client.create_request', function (Event $e) {
    echo 'Client object: ' . spl_object_hash($e['client']) . "\n";
    echo "Request object: {$e['request']}\n";
});
```

2.2 Using Request objects

2.2.1 HTTP request messages

Request objects are all about building an HTTP message. Each part of an HTTP request message can be set individually using methods on the request object or set in bulk using the `setUrl()` method. Here's the format of an HTTP request with each part of the request referencing the method used to change it:

```
PUT(a) /path(b)?query=123(c) HTTP/1.1(d)
X-Header(e): header
Content-Length(e): 4

data(f)
```

1. **Method**	The request method can only be set when instantiating a request
2. **Path**	`$request->setPath('/path');`
3. **Query**	`$request->getQuery()->set('query', '123');`
4. **Protocol version**	`$request->setProtocolVersion('1.1');`
5. **Header**	`$request->setHeader('X-Header', 'header');`
6. **Entity Body**	`$request->setBody('data'); // Only available with PUT, POST, PATCH, DELETE`

2.2.2 Creating requests with a client

Client objects are responsible for creating HTTP request objects.

GET requests

GET requests are the most common form of HTTP requests. When you visit a website in your browser, the HTML of the website is downloaded using a GET request. GET requests are idempotent requests that are typically used to download content (an entity) identified by a request URL.

```
use Guzzle\Http\Client;

$client = new Client();

// Create a request that has a query string and an X-Foo header
$request = $client->get('http://www.amazon.com?a=1', array('X-Foo' => 'Bar'));

// Send the request and get the response
$response = $request->send();
```

You can change where the body of a response is downloaded on any request using the `$request->setResponseBody(string|EntityBodyInterface|resource)` method of a request. You can also set the `save_to` option of a request:

```
// Send the response body to a file
$request = $client->get('http://test.com', array(), array('save_to' => '/path/to/file'));

// Send the response body to an fopen resource
$request = $client->get('http://test.com', array(), array('save_to' => fopen('/path/to/file', 'w')));
```

HEAD requests

HEAD requests work exactly like GET requests except that they do not actually download the response body (entity) of the response message. HEAD requests are useful for retrieving meta information about an entity identified by a Request-URI.

```
$client = new Guzzle\Http\Client();
$request = $client->head('http://www.amazon.com');
$response = $request->send();
echo $response->getContentLength();
// >>> Will output the Content-Length header value
```

DELETE requests

A DELETE method requests that the origin server delete the resource identified by the Request-URI.

```
$client = new Guzzle\Http\Client();
$request = $client->delete('http://example.com');
$response = $request->send();
```

POST requests

While POST requests can be used for a number of reasons, POST requests are often used when submitting HTML form data to a website. POST requests can include an entity body in the HTTP request.

POST requests in Guzzle are sent with an `application/x-www-form-urlencoded` Content-Type header if POST fields are present but no files are being sent in the POST. If files are specified in the POST request, then the Content-Type header will become `multipart/form-data`.

The `post()` method of a client object accepts four arguments: the URL, optional headers, post fields, and an array of request options. To send files in the POST request, prepend the `@` symbol to the array value (just like you would if you were using the PHP `curl_setopt` function).

Here's how to create a multipart/form-data POST request containing files and fields:

```
$request = $client->post('http://httpbin.org/post', array(), array(
    'custom_field' => 'my custom value',
    'file_field'   => '@/path/to/file.xml'
));

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

Nota: Remember to **always** sanitize user input when sending POST requests:

```
// Prevent users from accessing sensitive files by sanitizing input
$_POST = array('firstname' => '@/etc/passwd');
$request = $client->post('http://www.example.com', array(), array (
    'firstname' => str_replace('@', '', $_POST['firstname'])
));
```

You can alternatively build up the contents of a POST request.

```
$request = $client->post('http://httpbin.org/post')
    ->setPostField('custom_field', 'my custom value')
    ->addPostFile('file', '/path/to/file.xml');

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

Raw POST data

POST requests can also contain raw POST data that is not related to HTML forms.

```
$request = $client->post('http://httpbin.org/post', array(), 'this is the body');
$response = $request->send();
```

You can set the body of POST request using the `setBody()` method of the `Guzzle\Http\Message\EntityEnclosingRequest` object. This method accepts a string, a resource returned from `fopen`, or a `Guzzle\Http\EntityBodyInterface` object.

```
$request = $client->post('http://httpbin.org/post');
// Set the body of the POST to stream the contents of /path/to/large_body.txt
$request->setBody(fopen('/path/to/large_body.txt', 'r'));
$response = $request->send();
```

PUT requests

The PUT method requests that the enclosed entity be stored under the supplied Request-URI. PUT requests are similar to POST requests in that they both can send an entity body in the request message.

The body of a PUT request (any any `Guzzle\Http\Message\EntityEnclosingRequestInterface` object) is always stored as a `Guzzle\Http\Message\EntityBodyInterface` object. This allows a great deal of flexibility when sending data to a remote server. For example, you can stream the contents of a stream returned by fopen, stream the contents of a callback function, or simply send a string of data.

```
$request = $client->put('http://httpbin.org/put', array(), 'this is the body');
$response = $request->send();
```

Just like with POST, PATH, and DELETE requests, you can set the body of a PUT request using the `setBody()` method.

```
$request = $client->put('http://httpbin.org/put');
$request->setBody(fopen('/path/to/large_body.txt', 'r'));
$response = $request->send();
```

PATCH requests

PATCH requests are used to modify a resource.

```
$request = $client->patch('http://httpbin.org', array(), 'this is the body');
$response = $request->send();
```

OPTIONS requests

The OPTIONS method represents a request for information about the communication options available on the request/response chain identified by the Request-URI.

```
$request = $client->options('http://httpbin.org');
$response = $request->send();

// Check if the PUT method is supported by this resource
var_export($response->isMethodAllows('PUT'));
```

Custom requests

You can create custom HTTP requests that use non-standard HTTP methods using the `createRequest()` method of a client object.

```
$request = $client->createRequest('COPY', 'http://example.com/foo', array(
    'Destination' => 'http://example.com/bar',
    'Overwrite'   => 'T'
));
$response = $request->send();
```

2.2.3 Query string parameters

Query string parameters of a request are owned by a request's `Guzzle\Http\Query` object that is accessible by calling `$request->getQuery()`. The Query class extends from `Guzzle\Common\Collection` and allows you to set one or more query string parameters as key value pairs. You can set a parameter on a Query object using the `set($key, $value)` method or access the query string object like an associative array. Any previously specified value for a key will be overwritten when using `set()`. Use `add($key, $value)` to add a value to query string object, and in the event of a collision with an existing value at a specific key, the value will be converted to an array that contains all of the previously set values.

```
$request = new Guzzle\Http\Message\Request('GET', 'http://www.example.com?foo=bar&abc=123');

$query = $request->getQuery();
echo "{$query}\n";
// >>> foo=bar&abc=123

$query->remove('abc');
echo "{$query}\n";
// >>> foo=bar

$query->set('foo', 'baz');
echo "{$query}\n";
// >>> foo=baz

$query->add('foo', 'bar');
echo "{$query}\n";
// >>> foo%5B0%5D=baz&foo%5B1%5D=bar
```

Whoah! What happened there? When `foo=bar` was added to the existing `foo=baz` query string parameter, the aggregator associated with the Query object was used to help convert multi-value query string parameters into a string. Let's disable URL-encoding to better see what's happening.

```
$query->useUrlEncoding(false);
echo "{$query}\n";
// >>> foo[0]=baz&foo[1]=bar
```

Nota: URL encoding can be disabled by passing false, enabled by passing true, set to use RFC 1738 by passing `Query::FORM_URLENCODED` (internally uses PHP's `urlencode` function), or set to RFC 3986 by passing `Query::RFC_3986` (this is the default and internally uses PHP's `rawurlencode` function).

As you can see, the multiple values were converted into query string parameters following the default PHP convention of adding numerically indexed square bracket suffixes to each key (`foo[0]=baz&foo[1]=bar`). The strategy used to convert multi-value parameters into a string can be customized using the `setAggregator()` method of the Query class. Guzzle ships with the following query string aggregators by default:

1. `Guzzle\Http\QueryAggregator\PhpAggregator`: Aggregates using PHP style brackets (e.g. `foo[0]=baz&foo[1]=bar`)
2. `Guzzle\Http\QueryAggregator\DuplicateAggregator`: Performs no aggregation and allows for key value pairs to be repeated in a URL (e.g. `foo=baz&foo=bar`)
3. `Guzzle\Http\QueryAggregator\CommaAggregator`: Aggregates using commas (e.g. `foo=baz,bar`)

2.2.4 HTTP Message Headers

HTTP message headers are case insensitive, multiple occurrences of any header can be present in an HTTP message (whether it's valid or not), and some servers require specific casing of particular headers. Because of this, request and response headers are stored in `Guzzle\Http\Message\Header` objects. The Header object can be cast as a string, counted, or iterated to retrieve each value from the header. Casting a Header object to a string will return all of the header values concatenated together using a glue string (typically ", ").

A request (and response) object have several methods that allow you to retrieve and modify headers.

- `getHeaders()`: Get all of the headers of a message as a `Guzzle\Http\Message\Header\HeaderCollection` object.
- `getHeader($header)`: Get a specific header from a message. If the header exists, you'll get a `Guzzle\Http\Message\Header` object. If the header does not exist, this methods returns `null`.
- `hasHeader($header)`: Returns true or false based on if the message has a particular header.
- `setHeader($header, $value)`: Set a header value and overwrite any previously set value for this header.
- `addHeader($header, $value)`: Add a header with a particular name. If a previous value was already set by the same, then the header will contain multiple values.
- `removeHeader($header)`: Remove a header by name from the message.

```php
$request = new Request('GET', 'http://httpbin.com/cookies');
// addHeader will set and append to any existing header values
$request->addHeader('Foo', 'bar');
$request->addHeader('foo', 'baz');
// setHeader overwrites any existing values
$request->setHeader('Test', '123');

// Request headers can be cast as a string
echo $request->getHeader('Foo');
// >>> bar, baz
echo $request->getHeader('Test');
// >>> 123

// You can count the number of headers of a particular case insensitive name
echo count($request->getHeader('foO'));
// >>> 2

// You can iterate over Header objects
foreach ($request->getHeader('foo') as $header) {
    echo $header . "\n";
}

// You can get all of the request headers as a Guzzle\Http\Message\Header\HeaderCollection object
$headers = $request->getHeaders();
```

```
// Missing headers return NULL
var_export($request->getHeader('Missing'));
// >>> null

// You can see all of the different variations of a header by calling raw() on the Header
var_export($request->getHeader('foo')->raw());
```

2.2.5 Setting the body of a request

Requests that can send a body (e.g. PUT, POST, DELETE, PATCH) are instances of `Guzzle\Http\Message\EntityEnclosingRequestInterface`. Entity enclosing requests contain several methods that allow you to specify the body to send with a request.

Use the `setBody()` method of a request to set the body that will be sent with a request. This method accepts a string, a resource returned by `fopen()`, an array, or an instance of `Guzzle\Http\EntityBodyInterface`. The body will then be streamed from the underlying `EntityBodyInterface` object owned by the request. When setting the body of the request, you can optionally specify a Content-Type header and whether or not to force the request to use chunked Transfer-Encoding.

```
$request = $client->put('/user.json');
$request->setBody('{"foo":"baz"}', 'application/json');
```

Content-Type header

Guzzle will automatically add a Content-Type header to a request if the Content-Type can be guessed based on the file extension of the payload being sent or the file extension present in the path of a request.

```
$request = $client->put('/user.json', array(), '{"foo":"bar"}');
// The Content-Type was guessed based on the path of the request
echo $request->getHeader('Content-Type');
// >>> application/json

$request = $client->put('/user.json');
$request->setBody(fopen('/tmp/user_data.json', 'r'));
// The Content-Type was guessed based on the path of the entity body
echo $request->getHeader('Content-Type');
// >>> application/json
```

Transfer-Encoding: chunked header

When sending HTTP requests that contain a payload, you must let the remote server know how to determine when the entire message has been sent. This usually is done by supplying a `Content-Length` header that tells the origin server the size of the body that is to be sent. In some cases, the size of the payload being sent in a request cannot be known before initiating the transfer. In these cases (when using HTTP/1.1), you can use the `Transfer-Encoding: chunked` header.

If the Content-Length cannot be determined (i.e. using a PHP `http://` stream), then Guzzle will automatically add the `Transfer-Encoding: chunked` header to the request.

```
$request = $client->put('/user.json');
$request->setBody(fopen('http://httpbin.org/get', 'r'));

// The Content-Length could not be determined
```

```
echo $request->getHeader('Transfer-Encoding');
// >>> chunked
```

See `/http-client/entity-bodies` for more information on entity bodies.

Expect: 100-Continue header

The `Expect: 100-Continue` header is used to help a client prevent sending a large payload to a server that will reject the request. This allows clients to fail fast rather than waste bandwidth sending an erroneous payload. Guzzle will automatically add the `Expect: 100-Continue` header to a request when the size of the payload exceeds 1MB or if the body of the request is not seekable (this helps to prevent errors when a non-seekable body request is redirected).

Nota: If you find that your larger requests are taking too long to complete, you should first check if the `Expect: 100-Continue` header is being sent with the request. Some servers do not respond well to this header, which causes cURL to sleep for 1 second.

POST fields and files

Any entity enclosing request can send POST style fields and files. This includes POST, PUT, PATCH, and DELETE requests. Any request that has set POST fields or files will use cURL's POST message functionality.

```
$request = $client->post('/post');
// Set an overwrite any previously specified value
$request->setPostField('foo', 'bar');
// Append a value to any existing values
$request->getPostFields()->add('foo', 'baz');
// Remove a POST field by name
$request->removePostField('fizz');

// Add a file to upload (forces multipart/form-data)
$request->addPostFile('my_file', '/path/to/file', 'plain/text');
// Remove a POST file by POST key name
$request->removePostFile('my_other_file');
```

Dica: Adding a large number of POST fields to a POST request is faster if you use the `addPostFields()` method so that you can add and process multiple fields with a single call. Adding multiple POST files is also faster using `addPostFiles()`.

2.2.6 Working with cookies

Cookies can be modified and retrieved from a request using the following methods:

```
$request->addCookie($name, $value);
$request->removeCookie($name);
$value = $request->getCookie($name);
$valueArray = $request->getCookies();
```

Use the `cookie plugin` if you need to reuse cookies between requests.

2.2.7 Changing where a response is downloaded

When a request is sent, the body of the response will be stored in a PHP temp stream by default. You can change the location in which the response will be downloaded using `$request->setResponseBody($body)` or the `save_to` request option. This can be useful for downloading the contents of a URL to a specific file.

Here's an example of using request options:

```
$request = $this->client->get('http://example.com/large.mov', array(), array(
    'save_to' => '/tmp/large_file.mov'
));
$request->send();
var_export(file_exists('/tmp/large_file.mov'));
// >>> true
```

Here's an example of using `setResponseBody()`:

```
$body = fopen('/tmp/large_file.mov', 'w');
$request = $this->client->get('http://example.com/large.mov');
$request->setResponseBody($body);

// You can more easily specify the name of a file to save the contents
// of the response to by passing a string to ``setResponseBody()``.

$request = $this->client->get('http://example.com/large.mov');
$request->setResponseBody('/tmp/large_file.mov');
```

2.2.8 Custom cURL options

Most of the functionality implemented in the libcurl bindings has been simplified and abstracted by Guzzle. Developers who need access to cURL specific functionality can still add cURL handle specific behavior to Guzzle HTTP requests by modifying the cURL options collection of a request:

```
$request->getCurlOptions()->set(CURLOPT_LOW_SPEED_LIMIT, 200);
```

Other special options that can be set in the `curl.options` array include:

debug	Adds verbose cURL output to a temp stream owned by the cURL handle object
progress	Instructs cURL to emit events when IO events occur. This allows you to be notified when bytes are transferred over the wire by subscribing to a request's `curl.callback.read`, `curl.callback.write`, and `curl.callback.progress` events.

2.2.9 Request options

Requests options can be specified when creating a request or in the `request.options` parameter of a client. These options can control various aspects of a request including: headers to send, query string data, where the response should be downloaded, proxies, auth, etc.

```
$request = $client->get($url, $headers, array('proxy' => 'http://proxy.com'));
```

See *Request options* for more information.

2.2.10 Working with errors

HTTP errors

Requests that receive a 4xx or 5xx response will throw a `Guzzle\Http\Exception\BadResponseException`. More specifically, 4xx errors throw a `Guzzle\Http\Exception\ClientErrorResponseException`, and 5xx errors throw a `Guzzle\Http\Exception\ServerErrorResponseException`. You can catch the specific exceptions or just catch the BadResponseException to deal with either type of error. Here's an example of catching a generic BadResponseException:

```
try {
    $response = $client->get('/not_found.xml')->send();
} catch (Guzzle\Http\Exception\BadResponseException $e) {
    echo 'Uh oh! ' . $e->getMessage();
    echo 'HTTP request URL: ' . $e->getRequest()->getUrl() . "\n";
    echo 'HTTP request: ' . $e->getRequest() . "\n";
    echo 'HTTP response status: ' . $e->getResponse()->getStatusCode() . "\n";
    echo 'HTTP response: ' . $e->getResponse() . "\n";
}
```

Throwing an exception when a 4xx or 5xx response is encountered is the default behavior of Guzzle requests. This behavior can be overridden by adding an event listener with a higher priority than -255 that stops event propagation. You can subscribe to `request.error` to receive notifications any time an unsuccessful response is received.

You can change the response that will be associated with the request by calling `setResponse()` on the `$event['request']` object passed into your listener, or by changing the `$event['response']` value of the `Guzzle\Common\Event` object that is passed to your listener. Transparently changing the response associated with a request by modifying the event allows you to retry failed requests without complicating the code that uses the client. This might be useful for sending requests to a web service that has expiring auth tokens. When a response shows that your token has expired, you can get a new token, retry the request with the new token, and return the successful response to the user.

Here's an example of retrying a request using updated authorization credentials when a 401 response is received, overriding the response of the original request with the new response, and still allowing the default exception behavior to be called when other non-200 response status codes are encountered:

```
// Add custom error handling to any request created by this client
$client->getEventDispatcher()->addListener('request.error', function(Event $event) {

    if ($event['response']->getStatusCode() == 401) {

        $newRequest = $event['request']->clone();
        $newRequest->setHeader('X-Auth-Header', MyApplication::getNewAuthToken());
        $newResponse = $newRequest->send();

        // Set the response object of the request without firing more events
        $event['response'] = $newResponse;

        // You can also change the response and fire the normal chain of
        // events by calling $event['request']->setResponse($newResponse);

        // Stop other events from firing when you override 401 responses
        $event->stopPropagation();
    }

});
```

cURL errors

Connection problems and cURL specific errors can also occur when transferring requests using Guzzle. When Guzzle encounters cURL specific errors while transferring a single request, a `Guzzle\Http\Exception\CurlException` is thrown with an informative error message and access to the cURL error message.

A `Guzzle\Http\Exception\MultiTransferException` exception is thrown when a cURL specific error occurs while transferring multiple requests in parallel. You can then iterate over all of the exceptions encountered during the transfer.

2.2.11 Plugins and events

Guzzle request objects expose various events that allow you to hook in custom logic. A request object owns a `Symfony\Component\EventDispatcher\EventDispatcher` object that can be accessed by calling `$request->getEventDispatcher()`. You can use the event dispatcher to add listeners (a simple callback function) or event subscribers (classes that listen to specific events of a dispatcher). You can add event subscribers to a request directly by just calling `$request->addSubscriber($mySubscriber);`.

Events emitted from a request

A `Guzzle\Http\Message\Request` and `Guzzle\Http\Message\EntityEnclosingRequest` object emit the following events:

Event name	Description	Event data
request.before_send	About to send request	• request: Request to be sent
request.sent	Sent the request	• request: Request that was sent • response: Received response
request.complete	Completed a full HTTP transaction	• request: Request that was sent • response: Received response
request.success	Completed a successful request	• request: Request that was sent • response: Received response
request.error	Completed an unsuccessful request	• request: Request that was sent • response: Received response
request.exception	An unsuccessful response was received.	• request: Request • response: Received response • exception: BadResponseException
request.receive.status_line	Received the start of a response	• line: Full response start line • status_code: Status code • reason_phrase: Reason phrase • previous_response: (e.g. redirect)
curl.callback.progress	cURL progress event (only dispatched when `emit_io` is set on a request's curl options)	• handle: CurlHandle • download_size: Total download size • downloaded: Bytes downloaded • upload_size: Total upload bytes • uploaded: Bytes uploaded
curl.callback.write	cURL event called when data is written to an outgoing stream	• request: Request • write: Data being written
curl.callback.read	cURL event called when data is written to an incoming stream	• request: Request • read: Data being read

Creating a request event listener

Here's an example that listens to the `request.complete` event of a request and prints the request and response.

```php
use Guzzle\Common\Event;

$request = $client->get('http://www.google.com');

// Echo out the response that was received
$request->getEventDispatcher()->addListener('request.complete', function (Event $e) {
    echo $e['request'] . "\n\n";
    echo $e['response'];
});
```

CAPÍTULO 3

Eventos

3.1 Plugin system overview

The workflow of sending a request and parsing a response is driven by Guzzle's event system, which is powered by the Symfony2 Event Dispatcher component.

Any object in Guzzle that emits events will implement the `Guzzle\Common\HasEventDispatcher` interface. You can add event subscribers directly to these objects using the `addSubscriber()` method, or you can grab the `Symfony\Component\EventDispatcher\EventDispatcher` object owned by the object using `getEventDispatcher()` and add a listener or event subscriber.

3.1.1 Adding event subscribers to clients

Any event subscriber or event listener attached to the EventDispatcher of a `Guzzle\Http\Client` or `Guzzle\Service\Client` object will automatically be attached to all request objects created by the client. This allows you to attach, for example, a HistoryPlugin to a client object, and from that point on, every request sent through that client will utilize the HistoryPlugin.

```
use Guzzle\Plugin\History\HistoryPlugin;
use Guzzle\Service\Client;

$client = new Client();

// Create a history plugin and attach it to the client
$history = new HistoryPlugin();
$client->addSubscriber($history);

// Create and send a request. This request will also utilize the HistoryPlugin
$client->get('http://httpbin.org')->send();

// Echo out the last sent request by the client
echo $history->getLastRequest();
```

Dica: Create event subscribers, or *plugins*, to implement reusable logic that can be shared across clients. Event subscribers are also easier to test than anonymous functions.

3.1.2 Pre-Built plugins

Guzzle provides easy to use request plugins that add behavior to requests based on signal slot event notifications powered by the Symfony2 Event Dispatcher component.

- `async-plugin`
- `backoff-plugin`
- `cache-plugin`
- `cookie-plugin`
- `curl-auth-plugin`
- `history-plugin`
- `log-plugin`
- `md5-validator-plugin`
- `mock-plugin`
- `oauth-plugin`

3.2 Creating plugins

Guzzle is extremely extensible because of the behavioral modifications that can be added to requests, clients, and commands using an event system. Before and after the majority of actions are taken in the library, an event is emitted with the name of the event and context surrounding the event. Observers can subscribe to a subject and modify the subject based on the events received. Guzzle's event system utilizes the Symfony2 EventDispatcher and is the backbone of its plugin architecture.

3.2.1 Overview

Plugins must implement the `Symfony\Component\EventDispatcher\EventSubscriberInterface` interface. The `EventSubscriberInterface` requires that your class implements a static method, `getSubscribedEvents()`, that returns an associative array mapping events to methods on the object. See the Symfony2 documentation for more information.

Plugins can be attached to any subject, or object in Guzzle that implements that `Guzzle\Common\HasDispatcherInterface`.

Subscribing to a subject

You can subscribe an instantiated observer to an event by calling `addSubscriber` on a subject.

```
$testPlugin = new TestPlugin();
$client->addSubscriber($testPlugin);
```

You can also subscribe to only specific events using a closure:

```
$client->getEventDispatcher()->addListener('request.create', function(Event $event) {
    echo $event->getName();
    echo $event['request'];
});
```

`Guzzle\Common\Event` objects are passed to notified functions. The Event object has a `getName()` method which return the name of the emitted event and may contain contextual information that can be accessed like an array.

Knowing what events to listen to

Any class that implements the `Guzzle\Common\HasDispatcherInterface` must implement a static method, `getAllEvents()`, that returns an array of the events that are emitted from the object. You can browse the source to see each event, or you can call the static method directly in your code to get a list of available events.

3.2.2 Event hooks

- *Events emitted from a client*
- service-client-events
- *Events emitted from a request*
- `Guzzle\Http\Curl\CurlMulti`:
- service-builder-events

3.2.3 Examples of the event system

Simple Echo plugin

This simple plugin prints a string containing the request that is about to be sent by listening to the `request.before_send` event:

```
use Symfony\Component\EventDispatcher\EventSubscriberInterface;

class EchoPlugin implements EventSubscriberInterface
{
    public static function getSubscribedEvents()
    {
        return array('request.before_send' => 'onBeforeSend');
    }

    public function onBeforeSend(Guzzle\Common\Event $event)
    {
        echo 'About to send a request: ' . $event['request'] . "\n";
    }
}

$client = new Guzzle\Service\Client('http://www.test.com/');

// Create the plugin and add it as an event subscriber
$plugin = new EchoPlugin();
$client->addSubscriber($plugin);

// Send a request and notice that the request is printed to the screen
$client->get('/')->send();
```

Running the above code will print a string containing the HTTP request that is about to be sent.