



Symfony  
4.3

Consume APIs in a snap!

# HttpClient

Provides utilities to consume APIs

## Install

HttpClient is a standalone package

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

Create the low-level HTTP client that makes requests

## Using the HttpClient

```
use Symfony\Component\HttpClient\HttpClient;
```

**Create Options:**  
options defined here are added to all requests made by this client

```
$httpClient = HttpClient::create([], 6, 50);
```

default values

max host connections (optional)  
max pending pushes (optional) (only cURL)

**Request Options:**  
here you can define options that apply only to this request (overrides any global option defined by the HttpClient::create)

The request() method perform all kinds of HTTP requests

```
$response = $httpClient->request('GET', 'https://symfony.com/versions.json', []);
```

code execution continues immediately, it doesn't wait to receive the response

HTTP method

URL

```
$statusCode = $response->getStatusCode();
```

getting the response headers waits until they arrive

returns the status code  
E.g.: 200

```
$contentType = $response->getHeaders()['content-type'][0];
```

returns: 'application/json'

```
$content = $response->getContent();
```

getting the response contents will block the execution until the full response contents are received (use streaming responses for full async apps)

returns:  
{ "lts": "3.4.28", "latest": "4.2.9", "dev": "4.3.0-RC1", ... }

```
$content = $response->toArray();
```

returns:  
["lts" => "3.4.28", "latest" => "4.2.9", "dev" => "4.3.0-RC1", "2.0" => "2.0.25", ...]

Only supported when using cURL

## HTTP/2 request

HTTP/2 will be used by default if:  
\* cURL-based transport used  
\* libcurl version is >= 7.36  
\* request using HTTP's protocol

To enable for HTTP requests:

```
$httpClient = HttpClient::create(['http_version' => '2.0']);
```

## HTTP/2 PUSH support

Available when:  
\* libcurl >= 7.61 is used  
\* PHP >= 7.2.17 / 7.3.4

Pushed responses are put into a temporary cache and are used when a subsequent request is triggered for the corresponding URLs.

## HttpClient supports native PHP streams and cURL

HttpClient::create() selects cURL transport if cURL PHP extension is enabled and falls back to PHP streams otherwise.

### Explicitly selecting the transport

```
use Symfony\Component\HttpClient\CurlHttpClient;
use Symfony\Component\HttpClient\NativeHttpClient;

// native PHP streams
$httpClient = new NativeHttpClient();

// cURL PHP extension
$httpClient = new CurlHttpClient();
```



# HttpClient

Symfony  
4.3

## Options for Create and Request

option	default value	definition and examples
authentication	auth_basic	<p> <code>null</code>            An array containing the username as first value, and optionally the password as the second one; or string like <code>username:password</code> - enabling HTTP Basic authentication (RFC 7617).         </p>
	auth_bearer	<p> <code>null</code>            A token enabling HTTP Bearer authorization (RFC 6750).         </p> <pre> \$httpClient = HttpClient::create([     'auth_basic' =&gt; ['the-username'],     'auth_basic' =&gt; ['the-username', 'the-password'],     'auth_bearer' =&gt; 'the-bearer-token', ]); \$response = \$httpClient-&gt;request('GET', 'https://...', [     'auth_basic' =&gt; ['the-username', 'the-password'], ]); </pre> <p> <i>Use the same authentication for all requests</i>  <i>HTTP Basic authentication with only the username</i>  <i>HTTP Basic authentication with username and password</i>  <i>HTTP Bearer authentication (also called token authentication)</i>  <i>use a different HTTP Basic authentication only for this request</i> </p>
query string params	query	<p> <code>[]</code>            Associative array of query string values to merge with the request's URL.         </p> <pre> \$response = \$httpClient-&gt;request('GET', 'https://httpbin.org/get', [     'query' =&gt; [         'token' =&gt; '...',         'name' =&gt; '...',     ], ]); </pre> <p> <i>these values are automatically encoded before including them in the URL</i> </p>
setting HTTP headers	headers	<p> <code>[]</code>            Headers names provided as keys or as part of values.         </p> <pre> \$httpClient = HttpClient::create(['headers' =&gt; [     'User-Agent' =&gt; 'My Fancy App', ]]); \$response = \$httpClient-&gt;request('POST', 'https://...', [     'headers' =&gt; [         'Content-Type' =&gt; 'text/plain',     ], ]); </pre> <p> <i>header added to all requests made by this client</i>  <i>header only included in this request and overrides the value of the same header if defined globally by create()</i> </p>
uploading data	body	<p> <code>''</code>            You can use regular strings, closures, iterables and resources to upload data. They'll be processed automatically when making the requests.         </p> <pre> \$response = \$httpClient-&gt;request('POST', 'https://...', [     'body' =&gt; 'raw data',     'body' =&gt; ['parameter1' =&gt; 'value1', '...'],     'body' =&gt; function () {         // ...     },     'body' =&gt; fopen('/path/to/file', 'r'), ]); </pre> <p> <i>using a regular string</i>  <i>using an array of parameters</i>  <i>using a closure to generate the uploaded data</i>  <i>using a resource to get the data from it</i> </p>
json payload	json	<p> <code>null</code>            When uploading JSON payloads, use the <code>json</code> option instead of <code>body</code>. The given content will be JSON-encoded automatically and the request will add the <code>Content-Type: application/json</code> automatically too.         </p> <pre> \$response = \$httpClient-&gt;request('POST', 'https://...', [     'json' =&gt; ['param1' =&gt; 'value1', '...'], ]); </pre>
	user_data	<p> <code>null</code>            Any extra data to attach to the request (scalar, callable, object...) that must be available via <code>\$response-&gt;getInfo('user_data')</code> - not used internally.         </p>
	max_redirects	<p> <code>20</code>            The maximum number of redirects to follow; a value lower or equal to zero means redirects should not be followed; "Authorization" and "Cookie" headers must not follow except for the initial host name. If the number of redirects is higher than the configured value, you'll get a <code>RedirectionException</code>.         </p>
	http_version	<p> <code>null</code>            Defaults to the best supported version, typically 1.1 or 2.0.         </p>
	base_uri	<p> <code>null</code>            The URI to resolve relative URLs, following rules in RFC 3986, section 2.         </p>
	buffer	<p> <code>true</code>            Whether the content of the response should be buffered or not.         </p>



# HttpClient PSR-18 compatible

Symfony  
4.3

option	default value	definition and examples
<code>on_progress</code>	<code>null</code>	Details about the response progress (e.g. display a progress bar) / abort a request throwing any exceptions. <pre> \$url = 'https://releases.ubuntu.com/18.04.1/ubuntu-18.04.1-desktop-amd64.iso'; \$response = \$httpClient-&gt;request('GET', \$url, [     'buffer' =&gt; false,     'on_progress' =&gt; function (int \$dlNow, int \$dlSize, array \$info): void {         // ...     }, ]); </pre> <p><i>optional: if you don't want to buffer the response in memory</i></p> <p><i>optional: to display details about the response progress</i></p>
<code>resolve</code>	<code>[]</code>	A map of host to IP address that should replace DNS resolution. Protect webhooks against calls to internal endpoints.
<code>proxy</code>	<code>null</code>	Get through an HTTP proxy. By default, the proxy-related env vars handled by cURL should be honored.
<code>no_proxy</code>	<code>null</code>	A comma separated list of hosts that do not require a proxy to be reached.
<code>timeout</code>	<code>null</code>	The inactivity timeout - defaults to <code>ini_get('default_socket_timeout')</code> .
<code>bindto</code>	<code>0</code>	The interface or the local socket to bind to.
<code>verify_peer</code>	<code>true</code>	Require verification of SSL certificate used.
<code>verify_host</code>	<code>true</code>	
<code>cafile</code>	<code>null</code>	Location of Certificate Authority file on local filesystem which should be used with the <code>verify_peer</code> context option to authenticate the identity of the remote peer.
<code>capath</code>	<code>null</code>	If <code>cafile</code> is not specified or if the certificate is not found there, the directory pointed to by <code>capath</code> is searched for a suitable certificate. <code>capath</code> must be a correctly hashed certificate directory.
<code>local_cert</code>	<code>null</code>	Path to local certificate file on filesystem.
<code>local_pk</code>	<code>null</code>	Path to local private key file on filesystem in case of separate files for certificate ( <code>local_cert</code> ) and private key.
<code>passphrase</code>	<code>null</code>	Passphrase with which your <code>local_cert</code> file was encoded.
<code>ciphers</code>	<code>null</code>	Sets the list of available ciphers.
<code>peer_fingerprint</code>	<code>null</code>	Pin public keys of remote certificates. Aborts when the remote certificate digest doesn't match the specified hash.
<code>capture_peer_cert_chain</code>	<code>false</code>	If set to <code>TRUE</code> a <code>peer_certificate_chain</code> context option will be created containing the certificate chain.
<code>extra</code>	<code>[]</code>	Additional options that can be ignored if unsupported, unlike regular options

SSL / certificates (<https://php.net/context.ssl>)

## Cookies

HttpClient is stateless so it doesn't handle cookies automatically. You can:

- handle cookies yourself using the Cookie HTTP header
- use the BrowserKit component which provides this feature and integrates seamlessly with the HttpClient component

## Caching Requests and Responses

The `CachingHttpClient` decorator allows caching responses and serving them from the local storage for next requests.

The implementation leverages the `HttpCache` class under the hood so that the `HttpKernel` component needs to be installed in your app.

```

use Symfony\Component\HttpClient\HttpClient;
use Symfony\Component\HttpClient\CachingHttpClient;
use Symfony\Component\HttpKernel\HttpCache\Store;

```

```

$store = new Store('/path/to/cache/storage/');
$client = HttpClient::create();
$client = new CachingHttpClient($client, $store);

```

*accepts a third argument to set the options for HttpCache*

```

$response = $client->request('GET', 'https://example.com/cacheable-resource');

```

*won't hit the network if the resource is already in the cache*



# HttpClient

Supports synchronous and asynchronous operations

Symfony 4.3

The response is an object of type `ResponseInterface` → **Response**

Responses are always asynchronous: the call to the method returns immediately instead of waiting to receive the response

## Response Methods

```

$response = $httpClient->request('GET', 'https://...');

$statusCode = $response->getStatusCode(); // returns the HTTP status code of the response

$headers = $response->getHeaders(); // gets the HTTP headers as string[][] with the header names lower-cased

$content = $response->getContent(); // gets the response body as a string

$httpInfo = $response->getInfo(); // gets info coming from the transport layer

$startTime = $response->getInfo('start_time'); // gets individual info

```

Info coming from the transport layer

## `$response->getInfo()` Options

- user\_data
- response\_headers
- debug
- url
- error
- http\_method
- http\_code
- redirect\_count
- start\_time
- connect\_time
- redirect\_time
- starttransfer\_time
- total\_time
- namelookup\_time
- size\_upload
- size\_download
- primary\_ip
- primary\_port
- redirect\_url

is non-blocking: it returns live info about the response

## Streaming Responses ← for full async apps

```

$url = 'https://releases.ubuntu.com/18.04.1/ubuntu-18.04.1-desktop-amd64.iso';
$response = $httpClient->request('GET', $url, [
    'buffer' => false,
    'on_progress' => function (int $dlNow, int $dlSize, array $info): void {
        // ...
    },
]);

```

responses are lazy: this code is executed as soon as headers are received

```

if (200 !== $response->getStatusCode()) {
    throw new \Exception('...');
}

```

(optional) max number of seconds to wait before yielding a timeout chunk

```

$fileHandler = fopen('/ubuntu.iso', 'w');

foreach ($httpClient->stream($response, 0.0) as $chunk) {
    fwrite($fileHandler, $chunk->getContent());
}

```

get the response contents in chunk

stream: get chunks of the response sequentially instead of waiting for the entire response

gets detailed logs about the HTTP transaction  
E.g.: `$response->getInfo('debug')`

response chunks implement `Symfony\Contracts\HttpClient\ChunkInterface`

autoconfigure the HTTP client based on the requested URL

## Scoping Client

HTTP client options that depend on the URL of the request

```

use Symfony\Component\HttpClient\HttpClient;
use Symfony\Component\HttpClient\ScopingHttpClient;

$client = HttpClient::create();
$httpClient = new ScopingHttpClient($client, [
    'https://api.github.com/' => [
        'headers' => [
            'Accept' => 'application/vnd.github.v3+json',
            'Authorization' => 'token '.$githubToken,
        ],
        'base_uri' => 'https://api.github.com/',
    ],
    'https://api.github.com/'
]);

```

the key is a regexp which must match the beginning of the request URL

the (optional) 3rd argument is the regexp applied to all relative URLs (when using base\_uri)

the options defined as values apply only to the URLs matching the regular expressions defined as key



Symfony  
4.3

# HttpClient

you can configure multiple clients with different configurations and inject them into your services

## Symfony Framework Integration

```
# config/packages/framework.yaml
framework:
  # ...
  http_client:
    max_host_connections: 10
    default_options:
      max_redirects: 7
```

Use the `http_client` key to configure the default HTTP client used in the app

```
# config/packages/framework.yaml
framework:
  # ...
  http_client:
    scoped_clients:
      crawler.client:
        headers: { 'X-Powered-By': 'ACME App' }
        http_version: '1.0'
      some_api.client:
        max_redirects: 5
```

Defining multiple HTTP clients

you can choose the service using any available method in Symfony

### Multiple HTTP clients

Each `scoped client` also defines a corresponding named autowiring `alias`

E.g.: when using as type and name of an argument:

```
Symfony\Contracts\HttpClient\HttpClientInterface $someApiClient
```

autowiring will inject the `some_api.client` service

## Injecting the HTTP Client into Services

### One HTTP client

```
use Symfony\Contracts\HttpClient\HttpClientInterface;

class SomeService
{
    private $httpClient;

    public function __construct(HttpClientInterface $httpClient)
    {
        $this->httpClient = $httpClient;
    }
}
```

inject the HTTP client into any service by type-hinting a constructor argument with the `HttpClientInterface`

### Multiple HTTP clients

Each `scoped client` also defines a corresponding named autowiring `alias`

E.g.: when using as type and name of an argument:

```
Symfony\Contracts\HttpClient\HttpClientInterface $someApiClient
```

autowiring will inject the `some_api.client` service

## Dealing with Network Errors

Network errors (broken pipe, failed DNS resolution, etc.) are thrown as instances of `TransportExceptionInterface`

### To catch errors

Wrap calls to `$client->request()` but also calls to any methods of the returned responses

because responses are lazy so errors can happen in any method, (except `$response->getInfo()` that is non-blocking)

```
try {
    // both lines can potentially throw
    $response = $client->request(...);
    $headers = $response->getHeaders();
    // ...
} catch (TransportExceptionInterface $e) {
    // ...
}
```

### Multiplexing responses

Deal with errors for individual streams by catching `TransportExceptionInterface` in the foreach loop

```
foreach ($client->stream($responses) as $response => $chunk) {
    try {
        if ($chunk->isLast()) {
            // ... do something with $response
        }
    } catch (TransportExceptionInterface $e) {
        // ...
    }
}
```

## Handling Exceptions

When the HTTP status code of the response is in the 300-599 range (i.e. 3xx, 4xx or 5xx) your code is expected to handle it. If you don't do that, the `getHeaders()` and `getContent()` methods throw an appropriate exception:

```
$response = $httpClient->request('GET', 'https://httpbin.org/status/403');
```

the response of this request will be a 403 HTTP error

```
// this code results in a Symfony\Component\HttpClient\Exception\ClientException
// because it doesn't check the status code of the response
$content = $response->getContent();
```

```
$content = $response->getContent(false);
```

pass `FALSE` as the optional argument to not throw an exception and return instead the original response content (even if it's an error message)