



## HTTPX

# Environment Variables

The HTTPX library can be configured via environment variables. Environment variables are used by default. To ignore environment variables, `trust_env` has to be set `False`. There are two ways to set `trust_env` to disable environment variables:

- On the client via `httpx.Client(trust_env=False)`.
- Using the top-level API, such as `httpx.get("<url>", trust_env=False)`.

Here is a list of environment variables that HTTPX recognizes and what function they serve:

### SSLKEYLOGFILE

Valid values: a filename

If this environment variable is set, TLS keys will be appended to the specified file, creating it if it doesn't exist, whenever key material is generated or received. The keylog file is designed for debugging purposes only.

Support for `SSLKEYLOGFILE` requires Python 3.8 and OpenSSL 1.1.1 or newer.

Example:

```
# test_script.py
import httpx
```

```
with httpx.AsyncClient() as client:
    r = client.get("https://google.com")
```

```
SSLKEYLOGFILE=test.log python test_script.py
cat test.log
# TLS secrets log file, generated by OpenSSL / Python
SERVER_HANDSHAKE_TRAFFIC_SECRET XXXX
EXPORTER_SECRET XXXX
SERVER_TRAFFIC_SECRET_0 XXXX
CLIENT_HANDSHAKE_TRAFFIC_SECRET XXXX
CLIENT_TRAFFIC_SECRET_0 XXXX
SERVER_HANDSHAKE_TRAFFIC_SECRET XXXX
EXPORTER_SECRET XXXX
SERVER_TRAFFIC_SECRET_0 XXXX
CLIENT_HANDSHAKE_TRAFFIC_SECRET XXXX
CLIENT_TRAFFIC_SECRET_0 XXXX
```

### SSL\_CERT\_FILE

Valid values: a filename

If this environment variable is set then HTTPX will load CA certificate from the specified file instead of the default location.

Example:

```
SSL_CERT_FILE=/path/to/ca-certs/ca-bundle.crt python -c "import httpx; httpx.get('https://example.com')"
```

## SSL\_CERT\_DIR

Valid values: a directory following an OpenSSL specific layout.

If this environment variable is set and the directory follows an OpenSSL specific layout (ie. you ran `c_rehash`) then HTTPX will load CA certificates from this directory instead of the default location.

Example:

```
SSL_CERT_DIR=/path/to/ca-certs/ python -c "import httpx; httpx.get('https://example.com')"
```

## Proxies

The environment variables documented below are used as a convention by various HTTP tooling, including:

- cURL
- requests

For more information on using proxies in HTTPX, see [HTTP Proxying](#).

HTTP\_PROXY , HTTPS\_PROXY , ALL\_PROXY

Valid values: A URL to a proxy

HTTP\_PROXY , HTTPS\_PROXY , ALL\_PROXY set the proxy to be used for `http` , `https` , or all requests respectively.

```
export HTTP_PROXY=http://my-external-proxy.com:1234
```

```
# This request will be sent through the proxy
python -c "import httpx; httpx.get('http://example.com')"
```

```
# This request will be sent directly, as we set `trust_env=False`
python -c "import httpx; httpx.get('http://example.com', trust_env=False)"
```

NO\_PROXY

Valid values: a comma-separated list of hostnames/urls

NO\_PROXY disables the proxy for specific urls

```
export HTTP_PROXY=http://my-external-proxy.com:1234
export NO_PROXY=http://127.0.0.1,python-httpx.org
```

```
# As in the previous example, this request will be sent through the proxy
python -c "import httpx; httpx.get('http://example.com')"
```

```
# These requests will be sent directly, bypassing the proxy
python -c "import httpx; httpx.get('http://127.0.0.1:5000/my-api')"
```

```
python -c "import httpx; httpx.get('https://www.python-httpx.org')"
```

Made with Material for MkDocs



## HTTPX

# QuickStart

First, start by importing HTTPX:

```
>>> import httpx
```

Now, let's try to get a webpage.

```
>>> r = httpx.get('https://httpbin.org/get')
>>> r
<Response [200 OK]>
```

Similarly, to make an HTTP POST request:

```
>>> r = httpx.post('https://httpbin.org/post', data={'key': 'value'})
```

The PUT, DELETE, HEAD, and OPTIONS requests all follow the same style:

```
>>> r = httpx.put('https://httpbin.org/put', data={'key': 'value'})
>>> r = httpx.delete('https://httpbin.org/delete')
>>> r = httpx.head('https://httpbin.org/get')
>>> r = httpx.options('https://httpbin.org/get')
```

## Passing Parameters in URLs

To include URL query parameters in the request, use the `params` keyword:

```
>>> params = {'key1': 'value1', 'key2': 'value2'}
>>> r = httpx.get('https://httpbin.org/get', params=params)
```

To see how the values get encoding into the URL string, we can inspect the resulting URL that was used to make the request:

```
>>> r.url
URL('https://httpbin.org/get?key2=value2&key1=value1')
```

You can also pass a list of items as a value:

```
>>> params = {'key1': 'value1', 'key2': ['value2', 'value3']}
>>> r = httpx.get('https://httpbin.org/get', params=params)
>>> r.url
URL('https://httpbin.org/get?key1=value1&key2=value2&key2=value3')
```

## Response Content

HTTPX will automatically handle decoding the response content into Unicode text.

```
>>> r = httpx.get('https://www.example.org/')
>>> r.text
'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>...'
```

You can inspect what encoding will be used to decode the response.

```
>>> r.encoding
'UTF-8'
```

In some cases the response may not contain an explicit encoding, in which case HTTPX will attempt to automatically determine an encoding to use.

```
>>> r.encoding
None
>>> r.text
'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>...'
```

If you need to override the standard behaviour and explicitly set the encoding to use, then you can do that too.

```
>>> r.encoding = 'ISO-8859-1'
```

## Binary Response Content

The response content can also be accessed as bytes, for non-text responses:

```
>>> r.content
b'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>...'
```

Any `gzip` and `deflate` HTTP response encodings will automatically be decoded for you. If `brotlipy` is installed, then the `brotli` response encoding will also be supported.

For example, to create an image from binary data returned by a request, you can use the following code:

```
>>> from PIL import Image
>>> from io import BytesIO
>>> i = Image.open(BytesIO(r.content))
```

## JSON Response Content

Often Web API responses will be encoded as JSON.

```
>>> r = httpx.get('https://api.github.com/events')
>>> r.json()
[{'u'repository': {'u'open_issues': 0, u'url': 'https://github.com/...' ... } }]
```

## Custom Headers

To include additional headers in the outgoing request, use the `headers` keyword argument:

```
>>> url = 'https://httpbin.org/headers'
>>> headers = {'user-agent': 'my-app/0.0.1'}
>>> r = httpx.get(url, headers=headers)
```

## Sending Form Encoded Data

Some types of HTTP requests, such as `POST` and `PUT` requests, can include data in the request body. One common way of including that is as form-encoded data, which is used for HTML forms.

```
>>> data = {'key1': 'value1', 'key2': 'value2'}
>>> r = httpx.post("https://httpbin.org/post", data=data)
>>> print(r.text)
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

Form encoded data can also include multiple values from a given key.

```
>>> data = {'key1': ['value1', 'value2']}
>>> r = httpx.post("https://httpbin.org/post", data=data)
>>> print(r.text)
{
  ...
  "form": {
    "key1": [
      "value1",
      "value2"
    ]
  },
  ...
}
```

## Sending Multipart File Uploads

You can also upload files, using HTTP multipart encoding:

```
>>> files = {'upload-file': open('report.xls', 'rb')}
>>> r = httpx.post("https://httpbin.org/post", files=files)
>>> print(r.text)
{
  ...
  "files": {
    "upload-file": "<... binary content ...>"
  },
  ...
}
```

You can also explicitly set the filename and content type, by using a tuple of items for the file value:

```
>>> files = {'upload-file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel')}
>>> r = httpx.post("https://httpbin.org/post", files=files)
>>> print(r.text)
{
  ...
  "files": {
    "upload-file": "<... binary content ...>"
  },
  ...
}
```

If you need to include non-file data fields in the multipart form, use the `data=...` parameter:

```
>>> data = {'message': 'Hello, world!'}
>>> files = {'file': open('report.xls', 'rb')}
>>> r = httpx.post("https://httpbin.org/post", data=data, files=files)
>>> print(r.text)
{
  ...
  "files": {
    "file": "<... binary content ...>"
  },
  "form": {
    "message": "Hello, world!",
  },
  ...
}
```

## Sending JSON Encoded Data

Form encoded data is okay if all you need is a simple key-value data structure. For more complicated data structures you'll often want to use JSON encoding instead.

```
>>> data = {'integer': 123, 'boolean': True, 'list': ['a', 'b', 'c']}
>>> r = httpx.post("https://httpbin.org/post", json=data)
>>> print(r.text)
{
  ...
  "json": {
    "boolean": true,
    "integer": 123,
    "list": [
      "a",
      "b",
      "c"
    ]
  },
  ...
}
```

## Sending Binary Request Data

For other encodings, you should use the `content=...` parameter, passing either a `bytes` type or a generator that yields `bytes`.

```
>>> content = b'Hello, world'
>>> r = httpx.post("https://httpbin.org/post", content=content)
```

You may also want to set a custom `Content-Type` header when uploading binary data.

## Response Status Codes

We can inspect the HTTP status code of the response:

```
>>> r = httpx.get('https://httpbin.org/get')
>>> r.status_code
200
```

HTTPX also includes an easy shortcut for accessing status codes by their text phrase.

```
>>> r.status_code == httpx.codes.OK
True
```

We can raise an exception for any responses which are not a 2xx success code:

```
>>> not_found = httpx.get('https://httpbin.org/status/404')
>>> not_found.status_code
404
>>> not_found.raise_for_status()
Traceback (most recent call last):
  File "/Users/tomchristie/GitHub/encode/httpcore/httpx/models.py", line 837, in raise_for_status
    raise HTTPStatusError(message, response=self)
httpx_exceptions.HTTPStatusError: 404 Client Error: Not Found for url: https://httpbin.org/status/404
For more information check: https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/404
```

Any successful response codes will return the `Response` instance rather than raising an exception.

```
>>> r.raise_for_status()
```

The method returns the response instance, allowing you to use it inline. For example:

```
>>> r = httpx.get('...').raise_for_status()
>>> data = httpx.get('...').raise_for_status().json()
```

# Response Headers

The response headers are available as a dictionary-like interface.

```
>>> r.headers
Headers({
  'content-encoding': 'gzip',
  'transfer-encoding': 'chunked',
  'connection': 'close',
  'server': 'nginx/1.0.4',
  'x-runtime': '148ms',
  'etag': '"e1ca502697e5c9317743dc078f67693f"',
  'content-type': 'application/json'
})
```

The `Headers` data type is case-insensitive, so you can use any capitalization.

```
>>> r.headers['Content-Type']
'application/json'

>>> r.headers.get('content-type')
'application/json'
```

Multiple values for a single response header are represented as a single comma-separated value, as per RFC 7230:

A recipient MAY combine multiple header fields with the same field name into one “field-name: field-value” pair, without changing the semantics of the message, by appending each subsequent field-value to the combined field value in order, separated by a comma.

## Streaming Responses

For large downloads you may want to use streaming responses that do not load the entire response body into memory at once.

You can stream the binary content of the response...

```
>>> with httpx.stream("GET", "https://www.example.com") as r:
...     for data in r.iter_bytes():
...         print(data)
```

Or the text of the response...

```
>>> with httpx.stream("GET", "https://www.example.com") as r:
...     for text in r.iter_text():
...         print(text)
```

Or stream the text, on a line-by-line basis...

```
>>> with httpx.stream("GET", "https://www.example.com") as r:
...     for line in r.iter_lines():
...         print(line)
```

HTTPX will use universal line endings, normalising all cases to `\n`.

In some cases you might want to access the raw bytes on the response without applying any HTTP content decoding. In this case any content encoding that the web server has applied such as `gzip`, `deflate`, or `brrotli` will not be automatically decoded.

```
>>> with httpx.stream("GET", "https://www.example.com") as r:
...     for chunk in r.iter_raw():
...         print(chunk)
```

If you're using streaming responses in any of these ways then the `response.content` and `response.text` attributes will

not be available, and will raise errors if accessed. However you can also use the response streaming functionality to conditionally load the response body:

```
>>> with httpx.stream("GET", "https://www.example.com") as r:
...     if int(r.headers['Content-Length']) < TOO_LONG:
...         r.read()
...         print(r.text)
```

## Cookies

Any cookies that are set on the response can be easily accessed:

```
>>> r = httpx.get('https://httpbin.org/cookies/set?chocolate=chip')
>>> r.cookies['chocolate']
'chip'
```

To include cookies in an outgoing request, use the `cookies` parameter:

```
>>> cookies = {"peanut": "butter"}
>>> r = httpx.get('https://httpbin.org/cookies', cookies=cookies)
>>> r.json()
{'cookies': {'peanut': 'butter'}}
```

Cookies are returned in a `Cookies` instance, which is a dict-like data structure with additional API for accessing cookies by their domain or path.

```
>>> cookies = httpx.Cookies()
>>> cookies.set('cookie_on_domain', 'hello, there!', domain='httpbin.org')
>>> cookies.set('cookie_off_domain', 'nope.', domain='example.org')
>>> r = httpx.get('http://httpbin.org/cookies', cookies=cookies)
>>> r.json()
{'cookies': {'cookie_on_domain': 'hello, there!'}}
```

## Redirection and History

By default, HTTPX will **not** follow redirects for all HTTP methods, although this can be explicitly enabled.

For example, GitHub redirects all HTTP requests to HTTPS.

```
>>> r = httpx.get('http://github.com/')
>>> r.status_code
301
>>> r.history
[]
>>> r.next_request
<Request('GET', 'https://github.com/')>
```

You can modify the default redirection handling with the `follow_redirects` parameter:

```
>>> r = httpx.get('http://github.com/', follow_redirects=True)
>>> r.url
URL('https://github.com/')
>>> r.status_code
200
>>> r.history
[<Response [301 Moved Permanently]>]
```

The `history` property of the response can be used to inspect any followed redirects. It contains a list of any redirect responses that were followed, in the order in which they were made.

## Timeouts

HTTPX defaults to including reasonable timeouts for all network operations, meaning that if a connection is



not properly established then it should always raise an error rather than hanging indefinitely.

The default timeout for network inactivity is five seconds. You can modify the value to be more or less strict:

```
>>> httpx.get('https://github.com/', timeout=0.001)
```

You can also disable the timeout behavior completely...

```
>>> httpx.get('https://github.com/', timeout=None)
```

For advanced timeout management, see [Timeout fine-tuning](#).

## Authentication

HTTPX supports Basic and Digest HTTP authentication.

To provide Basic authentication credentials, pass a 2-tuple of plaintext `str` or `bytes` objects as the `auth` argument to the request functions:

```
>>> httpx.get("https://example.com", auth=("my_user", "password123"))
```

To provide credentials for Digest authentication you'll need to instantiate a `DigestAuth` object with the plaintext username and password as arguments. This object can be then passed as the `auth` argument to the request methods as above:

```
>>> auth = httpx.DigestAuth("my_user", "password123")
>>> httpx.get("https://example.com", auth=auth)
<Response [200 OK]>
```

## Exceptions

HTTPX will raise exceptions if an error occurs.

The most important exception classes in HTTPX are `RequestError` and `HTTPStatusError`.

The `RequestError` class is a superclass that encompasses any exception that occurs while issuing an HTTP request. These exceptions include a `.request` attribute.

```
try:
    response = httpx.get("https://www.example.com/")
except httpx.RequestError as exc:
    print(f"An error occurred while requesting {exc.request.url!r}.")
```

The `HTTPStatusError` class is raised by `response.raise_for_status()` on responses which are not a 2xx success code. These exceptions include both a `.request` and a `.response` attribute.

```
response = httpx.get("https://www.example.com/")
try:
    response.raise_for_status()
except httpx.HTTPStatusError as exc:
    print(f"Error response {exc.response.status_code} while requesting {exc.request.url!r}.")
```

There is also a base class `HTTPError` that includes both of these categories, and can be used to catch either failed requests, or 4xx and 5xx responses.

You can either use this base class to catch both categories...

```
try:
    response = httpx.get("https://www.example.com/")
    response.raise_for_status()
except httpx.HTTPError as exc:
    print(f"Error while requesting {exc.request.url!r}.")
```

Or handle each case explicitly...

```
try:
    response = httpx.get("https://www.example.com/")
    response.raise_for_status()
except httpx.RequestError as exc:
    print(f"An error occurred while requesting {exc.request.url!r}.")
except httpx.HTTPStatusError as exc:
    print(f"Error response {exc.response.status_code} while requesting {exc.request.url!r}.")
```

For a full list of available exceptions, see [Exceptions \(API Reference\)](#).

Made with Material for MkDocs



## HTTPX

# Troubleshooting

This page lists some common problems or issues you could encounter while developing with HTTPX, as well as possible solutions.

## Proxies

"The handshake operation timed out" on HTTPS requests when using a proxy

**Description:** When using a proxy and making an HTTPS request, you see an exception looking like this:

```
httpx.ProxyError: _ssl.c:1091: The handshake operation timed out
```

**Similar issues:** [encode/httpx#1412](#), [encode/httpx#1433](#)

**Resolution:** it is likely that you've set up your proxies like this...

```
proxies = {  
    "http://": "http://myproxy.org",  
    "https://": "https://myproxy.org",  
}
```

Using this setup, you're telling HTTPX to connect to the proxy using HTTP for HTTP requests, and using HTTPS for HTTPS requests.

But if you get the error above, it is likely that your proxy doesn't support connecting via HTTPS. Don't worry: that's a common gotcha.

Change the scheme of your HTTPS proxy to `http://...` instead of `https://...`:

```
proxies = {  
    "http://": "http://myproxy.org",  
    "https://": "http://myproxy.org",  
}
```

This can be simplified to:

```
proxies = "http://myproxy.org"
```

For more information, see [Proxies: FORWARD vs TUNNEL](#).

## Error when making requests to an HTTPS proxy

**Description:** your proxy *does* support connecting via HTTPS, but you are seeing errors along the lines of...

```
httpx.ProxyError: [SSL: PRE_MAC_LENGTH_TOO_LONG] invalid alert (_ssl.c:1091)
```

**Similar issues:** [encode/httpx#1424](#).

**Resolution:** HTTPX does not properly support HTTPS proxies at this time. If that's something you're interested in having, please see [encode/httpx#1434](https://github.com/encode/httpx/issues/1434) and consider lending a hand there.

Made with Material for MkDocs



## HTTPX



# HTTPX

 Test Suite passing  pypi package 0.25.1

*A next-generation HTTP client for Python.*

HTTPX is a fully featured HTTP client for Python 3, which provides sync and async APIs, and support for both HTTP/1.1 and HTTP/2.

Install HTTPX using pip:

```
$ pip install httpx
```

Now, let's get started:

```
>>> import httpx
>>> r = httpx.get('https://www.example.org/')
>>> r
<Response [200 OK]>
>>> r.status_code
200
>>> r.headers['content-type']
'text/html; charset=UTF-8'
>>> r.text
'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>...'
```

Or, using the command-line client.

```
# The command line client is an optional dependency.
$ pip install 'httpx[cli]'
```

Which now allows us to use HTTPX directly from the command-line...

```
$ httpx --help

HTTPX 🦋

A next generation HTTP client.

Usage: httpx <URL> [OPTIONS]

-m, --method METHOD      Request method, such as GET, POST, PUT, PATCH, DELETE, OPTIONS, HEAD.
                        [Default: GET, or POST if a request body is included]

-p, --params <NAME VALUE> ... Query parameters to include in the request URL.

-c, --content TEXT       Byte content to include in the request body.

-d, --data <NAME VALUE> ... Form data to include in the request body.

-f, --files <NAME FILENAME> ... Form files to include in the request body.

-j, --json TEXT          JSON data to include in the request body.

-h, --headers <NAME VALUE> ... Include additional HTTP headers in the request.

--cookies <NAME VALUE> ... Cookies to include in the request.

--auth <USER PASS>       Username and password to include in the request. Specify '-' for the password to use a
                        password prompt. Note that using --verbose/-v will expose the Authorization header,
                        including the password encoding in a trivially reversible format.

--proxy URL              Send the request via a proxy. Should be the URL giving the proxy address.

--timeout FLOAT          Timeout value to use for network operations, such as establishing the connection,
                        reading some data, etc... [Default: 5.0]

--follow-redirects       Automatically follow redirects.

--no-verify              Disable SSL verification.

--http2                 Send the request using HTTP/2, if the remote server supports it.

--download FILE          Save the response content as a file, rather than displaying it.

-v, --verbose            Verbose output. Show request as well as response.

--help                  Show this message and exit.

$
```

Sending a request...

```
httpx
$ httpx http://httpbin.org/json
HTTP/1.1 200 OK
Date: Mon, 13 Sep 2021 09:59:51 GMT
Content-Type: application/json
Content-Length: 429
Connection: keep-alive
Server: gunicorn/19.9.0
Access-Control-Allow-Origin: *
Access-Control-Allow-Credentials: true

{
  "slideshow": {
    "author": "Yours Truly",
    "date": "date of publication",
    "slides": [
      {
        "title": "Wake up to WonderWidgets!",
        "type": "all"
      },
      {
        "items": [
          "Why <em>WonderWidgets</em> are great",
          "Who <em>buys</em> WonderWidgets"
        ],
        "title": "Overview",
        "type": "all"
      }
    ],
    "title": "Sample Slide Show"
  }
}
```

## Features

HTTPX builds on the well-established usability of `requests`, and gives you:

- A broadly requests-compatible API.
- Standard synchronous interface, but with async support if you need it.
- HTTP/1.1 and HTTP/2 support.
- Ability to make requests directly to WSGI applications or ASGI applications.
- Strict timeouts everywhere.
- Fully type annotated.
- 100% test coverage.

Plus all the standard features of `requests` ...

- International Domains and URLs
- Keep-Alive & Connection Pooling
- Sessions with Cookie Persistence
- Browser-style SSL Verification
- Basic/Digest Authentication

- Elegant Key/Value Cookies
- Automatic Decompression
- Automatic Content Decoding
- Unicode Response Bodies
- Multipart File Uploads
- HTTP(S) Proxy Support
- Connection Timeouts
- Streaming Downloads
- .netrc Support
- Chunked Requests

## Documentation

For a run-through of all the basics, head over to the QuickStart.

For more advanced topics, see the Advanced Usage section, the async support section, or the HTTP/2 section.

The Developer Interface provides a comprehensive API reference.

To find out about tools that integrate with HTTPX, see Third Party Packages.

## Dependencies

The HTTPX project relies on these excellent libraries:

- `httpcore` - The underlying transport implementation for `httpx`.
- `h11` - HTTP/1.1 support.
- `certifi` - SSL certificates.
- `idna` - Internationalized domain name support.
- `sniffio` - Async library autodetection.

As well as these optional installs:

- `h2` - HTTP/2 support. (*Optional, with `httpx[http2]`*)
- `socksio` - SOCKS proxy support. (*Optional, with `httpx[socks]`*)
- `rich` - Rich terminal support. (*Optional, with `httpx[cli]`*)
- `click` - Command line client support. (*Optional, with `httpx[cli]`*)
- `brrotli` or `brrotlicffi` - Decoding for "brrotli" compressed responses. (*Optional, with `httpx[brrotli]`*)

A huge amount of credit is due to `requests` for the API layout that much of this work follows, as well as to `urllib3` for plenty of design inspiration around the lower-level networking details.

## Installation

Install with pip:

```
$ pip install httpx
```



Or, to include the optional HTTP/2 support, use:

```
$ pip install httpx[http2]
```

To include the optional brotli decoder support, use:

```
$ pip install httpx[brotli]
```

HTTPX requires Python 3.8+

Made with Material for MkDocs



## HTTPX

# Developer Interface

## Helper Functions

### Note

Only use these functions if you're testing HTTPX in a console or making a small number of requests. Using a `Client` will enable HTTP/2 and connection pooling for more efficient and long-lived connections.

```
httpx.request(method, url, *, params=None, content=None, data=None, files=None, json=None,
headers=None, cookies=None, auth=None, proxies=None, timeout=Timeout(timeout=5.0),
follow_redirects=False, verify=True, cert=None, trust_env=True)
```

Sends an HTTP request.

### Parameters:

- **method** - HTTP method for the new `Request` object: `GET`, `OPTIONS`, `HEAD`, `POST`, `PUT`, `PATCH`, or `DELETE`.
- **url** - URL for the new `Request` object.
- **params** - *(optional)* Query parameters to include in the URL, as a string, dictionary, or sequence of two-tuples.
- **content** - *(optional)* Binary content to include in the body of the request, as bytes or a byte iterator.
- **data** - *(optional)* Form data to include in the body of the request, as a dictionary.
- **files** - *(optional)* A dictionary of upload files to include in the body of the request.
- **json** - *(optional)* A JSON serializable object to include in the body of the request.
- **headers** - *(optional)* Dictionary of HTTP headers to include in the request.
- **cookies** - *(optional)* Dictionary of Cookie items to include in the request.
- **auth** - *(optional)* An authentication class to use when sending the request.
- **proxies** - *(optional)* A dictionary mapping proxy keys to proxy URLs.
- **timeout** - *(optional)* The timeout configuration to use when sending the request.
- **follow\_redirects** - *(optional)* Enables or disables HTTP redirects.
- **verify** - *(optional)* SSL certificates (a.k.a CA bundle) used to verify the identity of requested hosts. Either `True` (default CA bundle), a path to an SSL certificate file, an `ssl.SSLContext`, or `False` (which will disable verification).
- **cert** - *(optional)* An SSL certificate used by the requested host to authenticate the client. Either a path to an SSL certificate file, or two-tuple of (certificate file, key file), or a three-tuple of (certificate file, key file, password).
- **trust\_env** - *(optional)* Enables or disables usage of environment variables for configuration.

**Returns:** `Response`

Usage:

```
>>> import httpx
>>> response = httpx.request('GET', 'https://httpbin.org/get')
>>> response
<Response [200 OK]>
```

```
httpx.get(url, *, params=None, headers=None, cookies=None, auth=None, proxies=None,
follow_redirects=False, cert=None, verify=True, timeout=Timeout(timeout=5.0), trust_env=True)
```

Sends a GET request.

**Parameters:** See `httpx.request`.

Note that the `data`, `files`, `json` and `content` parameters are not available on this function, as GET requests should not include a request body.

```
httpx.options(url, *, params=None, headers=None, cookies=None, auth=None, proxies=None,
follow_redirects=False, cert=None, verify=True, timeout=Timeout(timeout=5.0), trust_env=True)
```

Sends an OPTIONS request.

**Parameters:** See `httpx.request`.

Note that the `data`, `files`, `json` and `content` parameters are not available on this function, as OPTIONS requests should not include a request body.

```
httpx.head(url, *, params=None, headers=None, cookies=None, auth=None, proxies=None,
follow_redirects=False, cert=None, verify=True, timeout=Timeout(timeout=5.0), trust_env=True)
```

Sends a HEAD request.

**Parameters:** See `httpx.request`.

Note that the `data`, `files`, `json` and `content` parameters are not available on this function, as HEAD requests should not include a request body.

```
httpx.post(url, *, content=None, data=None, files=None, json=None, params=None, headers=None,
cookies=None, auth=None, proxies=None, follow_redirects=False, cert=None, verify=True,
timeout=Timeout(timeout=5.0), trust_env=True)
```

Sends a POST request.

**Parameters:** See `httpx.request`.

```
httpx.put(url, *, content=None, data=None, files=None, json=None, params=None, headers=None,
cookies=None, auth=None, proxies=None, follow_redirects=False, cert=None, verify=True,
timeout=Timeout(timeout=5.0), trust_env=True)
```

Sends a PUT request.

**Parameters:** See `httpx.request`.

```
httpx.patch(url, *, content=None, data=None, files=None, json=None, params=None, headers=None,
cookies=None, auth=None, proxies=None, follow_redirects=False, cert=None, verify=True,
timeout=Timeout(timeout=5.0), trust_env=True)
```

Sends a PATCH request.

**Parameters:** See `httpx.request`.

`httpx.delete(url, *, params=None, headers=None, cookies=None, auth=None, proxies=None, follow_redirects=False, cert=None, verify=True, timeout=Timeout(timeout=5.0), trust_env=True)`

Sends a `DELETE` request.

**Parameters:** See `httpx.request`.

Note that the `data`, `files`, `json` and `content` parameters are not available on this function, as `DELETE` requests should not include a request body.

`httpx.stream(method, url, *, params=None, content=None, data=None, files=None, json=None, headers=None, cookies=None, auth=None, proxies=None, timeout=Timeout(timeout=5.0), follow_redirects=False, verify=True, cert=None, trust_env=True)`

Alternative to `httpx.request()` that streams the response body instead of loading it into memory at once.

**Parameters:** See `httpx.request`.

See also: Streaming Responses

## Client

`class httpx.Client(*, auth=None, params=None, headers=None, cookies=None, verify=True, cert=None, http1=True, http2=False, proxies=None, mounts=None, timeout=Timeout(timeout=5.0), follow_redirects=False, limits=Limits(max_connections=100, max_keepalive_connections=20, keepalive_expiry=5.0), max_redirects=20, event_hooks=None, base_url="", transport=None, app=None, trust_env=True, default_encoding='utf-8')`

An HTTP client, with connection pooling, HTTP/2, redirects, cookie persistence, etc.

It can be shared between threads.

Usage:

```
>>> client = httpx.Client()
>>> response = client.get('https://example.org')
```

### Parameters:

- **auth** - *(optional)* An authentication class to use when sending requests.
- **params** - *(optional)* Query parameters to include in request URLs, as a string, dictionary, or sequence of two-tuples.
- **headers** - *(optional)* Dictionary of HTTP headers to include when sending requests.
- **cookies** - *(optional)* Dictionary of Cookie items to include when sending requests.
- **verify** - *(optional)* SSL certificates (a.k.a CA bundle) used to verify the identity of requested hosts. Either `True` (default CA bundle), a path to an SSL certificate file, an `ssl.SSLContext`, or `False` (which will disable verification).
- **cert** - *(optional)* An SSL certificate used by the requested host to authenticate the client. Either a path to an SSL certificate file, or two-tuple of (certificate file, key file), or a three-tuple of (certificate file, key file, password).
- **proxies** - *(optional)* A dictionary mapping proxy keys to proxy URLs.
- **timeout** - *(optional)* The timeout configuration to use when sending requests.

- **limits** - *(optional)* The limits configuration to use.
- **max\_redirects** - *(optional)* The maximum number of redirect responses that should be followed.
- **base\_url** - *(optional)* A URL to use as the base when building request URLs.
- **transport** - *(optional)* A transport class to use for sending requests over the network.
- **app** - *(optional)* An WSGI application to send requests to, rather than sending actual network requests.
- **trust\_env** - *(optional)* Enables or disables usage of environment variables for configuration.
- **default\_encoding** - *(optional)* The default encoding to use for decoding response text, if no charset information is included in a response Content-Type header. Set to a callable for automatic character set detection. Default: "utf-8".

#### headers

HTTP headers to include when sending requests.

#### cookies

Cookie values to include when sending requests.

#### params

Query parameters to include in the URL when sending requests.

#### auth

Authentication class used when none is passed at the request-level.

See also Authentication.

**request** (*self, method, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None*)

Build and send a request.

Equivalent to:

```
request = client.build_request(...)
response = client.send(request, ...)
```

See `Client.build_request()`, `Client.send()` and Merging of configuration for how the various parameters are merged with client-level configuration.

**get** (*self, url, \*, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None*)

Send a `GET` request.

**Parameters:** See `httpx.request`.

**head** (*self, url, \*, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None*)

Send a `HEAD` request.

**Parameters:** See `httpx.request` .

`options(self, url, *, params=None, headers=None, cookies=None, auth=, follow_redirects=, timeout=, extensions=None)`

Send an `OPTIONS` request.

**Parameters:** See `httpx.request` .

`post(self, url, *, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow_redirects=, timeout=, extensions=None)`

Send a `POST` request.

**Parameters:** See `httpx.request` .

`put(self, url, *, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow_redirects=, timeout=, extensions=None)`

Send a `PUT` request.

**Parameters:** See `httpx.request` .

`patch(self, url, *, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow_redirects=, timeout=, extensions=None)`

Send a `PATCH` request.

**Parameters:** See `httpx.request` .

`delete(self, url, *, params=None, headers=None, cookies=None, auth=, follow_redirects=, timeout=, extensions=None)`

Send a `DELETE` request.

**Parameters:** See `httpx.request` .

`stream(self, method, url, *, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow_redirects=, timeout=, extensions=None)`

Alternative to `httpx.request()` that streams the response body instead of loading it into memory at once.

**Parameters:** See `httpx.request` .

See also: Streaming Responses

`build_request(self, method, url, *, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, timeout=, extensions=None)`

Build and return a request instance.

- The `params`, `headers` and `cookies` arguments are merged with any values set on the client.
- The `url` argument is merged with any `base_url` set on the client.

See also: Request instances

`send (self, request, *, stream=False, auth=, follow_redirects=)`

Send a request.

The request is sent as-is, unmodified.

Typically you'll want to build one with `Client.build_request()` so that any client-level configuration is merged into the request, but passing an explicit `httpx.Request()` is supported as well.

See also: Request instances

`close (self)`

Close transport and proxies.

## AsyncClient

```
class httpx.AsyncClient (*, auth=None, params=None, headers=None, cookies=None, verify=True, cert=None,
http1=True, http2=False, proxies=None, mounts=None, timeout=Timeout(timeout=5.0),
follow_redirects=False, limits=Limits(max_connections=100, max_keepalive_connections=20,
keepalive_expiry=5.0), max_redirects=20, event_hooks=None, base_url='', transport=None, app=None,
trust_env=True, default_encoding='utf-8')
```

An asynchronous HTTP client, with connection pooling, HTTP/2, redirects, cookie persistence, etc.

Usage:

```
>>> async with httpx.AsyncClient() as client:
>>> response = await client.get('https://example.org')
```

### Parameters:

- **auth** - *(optional)* An authentication class to use when sending requests.
- **params** - *(optional)* Query parameters to include in request URLs, as a string, dictionary, or sequence of two-tuples.
- **headers** - *(optional)* Dictionary of HTTP headers to include when sending requests.
- **cookies** - *(optional)* Dictionary of Cookie items to include when sending requests.
- **verify** - *(optional)* SSL certificates (a.k.a CA bundle) used to verify the identity of requested hosts. Either `True` (default CA bundle), a path to an SSL certificate file, an `ssl.SSLContext`, or `False` (which will disable verification).
- **cert** - *(optional)* An SSL certificate used by the requested host to authenticate the client. Either a path to an SSL certificate file, or two-tuple of (certificate file, key file), or a three-tuple of (certificate file, key file, password).
- **http2** - *(optional)* A boolean indicating if HTTP/2 support should be enabled. Defaults to `False`.
- **proxies** - *(optional)* A dictionary mapping HTTP protocols to proxy URLs.
- **timeout** - *(optional)* The timeout configuration to use when sending requests.
- **limits** - *(optional)* The limits configuration to use.
- **max\_redirects** - *(optional)* The maximum number of redirect responses that should be followed.
- **base\_url** - *(optional)* A URL to use as the base when building request URLs.
- **transport** - *(optional)* A transport class to use for sending requests over the network.
- **app** - *(optional)* An ASGI application to send requests to, rather than sending actual network requests.

- **trust\_env** - *(optional)* Enables or disables usage of environment variables for configuration.
- **default\_encoding** - *(optional)* The default encoding to use for decoding response text, if no charset information is included in a response Content-Type header. Set to a callable for automatic character set detection. Default: "utf-8".

#### headers

HTTP headers to include when sending requests.

#### cookies

Cookie values to include when sending requests.

#### params

Query parameters to include in the URL when sending requests.

#### auth

Authentication class used when none is passed at the request-level.

See also Authentication.

*async request(self, method, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Build and send a request.

Equivalent to:

```
request = client.build_request(...)
response = await client.send(request, ...)
```

See `AsyncClient.build_request()`, `AsyncClient.send()` and Merging of configuration for how the various parameters are merged with client-level configuration.

*async get(self, url, \*, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send a GET request.

**Parameters:** See `httpx.request`.

*async head(self, url, \*, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send a HEAD request.

**Parameters:** See `httpx.request`.

*async options(self, url, \*, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send an OPTIONS request.

**Parameters:** See `httpx.request`.



*async post(self, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send a POST request.

**Parameters:** See `httpx.request` .

*async put(self, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send a PUT request.

**Parameters:** See `httpx.request` .

*async patch(self, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send a PATCH request.

**Parameters:** See `httpx.request` .

*async delete(self, url, \*, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Send a DELETE request.

**Parameters:** See `httpx.request` .

*stream(self, method, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, auth=, follow\_redirects=, timeout=, extensions=None)*

Alternative to `httpx.request()` that streams the response body instead of loading it into memory at once.

**Parameters:** See `httpx.request` .

See also: Streaming Responses

*build\_request(self, method, url, \*, content=None, data=None, files=None, json=None, params=None, headers=None, cookies=None, timeout=, extensions=None)*

Build and return a request instance.

- The `params` , `headers` and `cookies` arguments are merged with any values set on the client.
- The `url` argument is merged with any `base_url` set on the client.

See also: Request instances

*async send(self, request, \*, stream=False, auth=, follow\_redirects=)*

Send a request.

The request is sent as-is, unmodified.

Typically you'll want to build one with `AsyncClient.build_request()` so that any client-level configuration is merged into the request, but passing an explicit `httpx.Request()` is supported as well.

See also: Request instances

*async* `aclose(self)`

Close transport and proxies.

## Response

*An HTTP response.*

- `def __init__(...)`
- `.status_code` - **int**
- `.reason_phrase` - **str**
- `.http_version` - "HTTP/2" OR "HTTP/1.1"
- `.url` - **URL**
- `.headers` - **Headers**
- `.content` - **bytes**
- `.text` - **str**
- `.encoding` - **str**
- `.is_redirect` - **bool**
- `.request` - **Request**
- `.next_request` - **Optional[Request]**
- `.cookies` - **Cookies**
- `.history` - **List[Response]**
- `.elapsed` - **timedelta**
- The amount of time elapsed between sending the request and calling `close()` on the corresponding response received for that request. `total_seconds()` to correctly get the total elapsed seconds.
- `def .raise_for_status()` - **Response**
- `def .json()` - **Any**
- `def .read()` - **bytes**
- `def .iter_raw([chunk_size])` - **bytes iterator**
- `def .iter_bytes([chunk_size])` - **bytes iterator**
- `def .iter_text([chunk_size])` - **text iterator**
- `def .iter_lines()` - **text iterator**
- `def .close()` - **None**
- `def .next()` - **Response**
- `def .aread()` - **bytes**
- `def .aiter_raw([chunk_size])` - **async bytes iterator**
- `def .aiter_bytes([chunk_size])` - **async bytes iterator**
- `def .aiter_text([chunk_size])` - **async text iterator**
- `def .aiter_lines()` - **async text iterator**
- `def .aclose()` - **None**
- `def .anext()` - **Response**

## Request

*An HTTP request. Can be constructed explicitly for more control over exactly what gets sent over the wire.*

```
>>> request = httpx.Request("GET", "https://example.org", headers={'host': 'example.org'})
>>> response = client.send(request)
```

- `def __init__(method, url, [params], [headers], [cookies], [content], [data], [files], [json], [stream])`
- `.method` - **str**
- `.url` - **URL**
- `.content` - **byte, byte iterator**, or **byte async iterator**
- `.headers` - **Headers**
- `.cookies` - **Cookies**

## URL

*A normalized, IDNA supporting URL.*

```
>>> url = URL("https://example.org/")
>>> url.host
'example.org'
```

- `def __init__(url, allow_relative=False, params=None)`
- `.scheme` - **str**
- `.authority` - **str**
- `.host` - **str**
- `.port` - **int**
- `.path` - **str**
- `.query` - **str**
- `.raw_path` - **str**
- `.fragment` - **str**
- `.is_ssl` - **bool**
- `.is_absolute_url` - **bool**
- `.is_relative_url` - **bool**
- `def .copy_with([scheme], [authority], [path], [query], [fragment])` - **URL**

## Headers

*A case-insensitive multi-dict.*

```
>>> headers = Headers({'Content-Type': 'application/json'})
>>> headers['content-type']
'application/json'
```

- `def __init__(self, headers, encoding=None)`
- `def copy()` - **Headers**

## Cookies

*A dict-like cookie store.*

```
>>> cookies = Cookies()
>>> cookies.set("name", "value", domain="example.org")
```

- `def __init__(cookies: [dict, Cookies, CookieJar])`
- `.jar` - **CookieJar**
- `def extract_cookies(response)`
- `def set_cookie_header(request)`
- `def set(name, value, [domain], [path])`
- `def get(name, [domain], [path])`
- `def delete(name, [domain], [path])`
- `def clear([domain], [path])`
- *Standard mutable mapping interface*

Made with Material for MkDocs



## HTTPX

# Contributing

Thank you for being interested in contributing to HTTPX. There are many ways you can contribute to the project:

- Try HTTPX and report bugs/issues you find
- Implement new features
- Review Pull Requests of others
- Write documentation
- Participate in discussions

## Reporting Bugs or Other Issues

Found something that HTTPX should support? Stumbled upon some unexpected behaviour?

Contributions should generally start out with a discussion. Possible bugs may be raised as a "Potential Issue" discussion, feature requests may be raised as an "Ideas" discussion. We can then determine if the discussion needs to be escalated into an "Issue" or not, or if we'd consider a pull request.

Try to be more descriptive as you can and in case of a bug report, provide as much information as possible like:

- OS platform
- Python version
- Installed dependencies and versions ( `python -m pip freeze` )
- Code snippet
- Error traceback

You should always try to reduce any examples to the *simplest possible case* that demonstrates the issue.

Some possibly useful tips for narrowing down potential issues...

- Does the issue exist on HTTP/1.1, or HTTP/2, or both?
- Does the issue exist with `Client` , `AsyncClient` , or both?
- When using `AsyncClient` does the issue exist when using `asyncio` or `trio` , or both?

## Development

To start developing HTTPX create a **fork** of the HTTPX repository on GitHub.

Then clone your fork with the following command replacing `YOUR-USERNAME` with your GitHub username:

```
$ git clone https://github.com/YOUR-USERNAME/httpx
```

You can now install the project and its dependencies using:

```
$ cd httpx
$ scripts/install
```

## Testing and Linting

We use custom shell scripts to automate testing, linting, and documentation building workflow.

To run the tests, use:

```
$ scripts/test
```

### Warning

The test suite spawns testing servers on ports **8000** and **8001**. Make sure these are not in use, so the tests can run properly.

Any additional arguments will be passed to `pytest`. See the `pytest` documentation for more information.

For example, to run a single test script:

```
$ scripts/test tests/test_multipart.py
```

To run the code auto-formatting:

```
$ scripts/lint
```

Lastly, to run code checks separately (they are also run as part of `scripts/test`), run:

```
$ scripts/check
```

## Documenting





Documentation pages are located under the `docs/` folder.

To run the documentation site locally (useful for previewing changes), use:

```
$ scripts/docs
```


## Resolving Build / CI Failures

Once you've submitted your pull request, the test suite will automatically run, and the results will show up in GitHub. If the test suite fails, you'll want to click through to the "Details" link, and try to identify why the test suite failed.

 <b>Some checks were not successful</b> <span>Hide all checks</span>		2 failing and 1 cancelled checks	
✗		Test Suite / Python 3.6 (pull_request)	Failing after 48s — Python 3.6 <span>Required</span> <a href="#">Details</a>
⚠		Test Suite / Python 3.7 (pull_request)	Cancelled after 52s — Python 3.7 <span>Required</span> <a href="#">Details</a>
✗		Test Suite / Python 3.8 (pull_request)	Failing after 45s — Python 3.8 <span>Required</span> <a href="#">Details</a>

Here are some common ways the test suite can fail:

## Check Job Failed


**Test Suite**

on: pull\_request

✗ Python 3.6

⚠ Python 3.7

⚠ Python 3.8

Test Suite / Python 3.6

failed 5 days ago in 49s

Search logs

< > ...

▶ ✓ Set up job

▶ ✓ Run actions/checkout@v2

▶ ✓ Run actions/setup-python@v1

▶ ✓ Install dependencies

▼ ✗ Run linting checks

1 ▶ Run scripts/check

6 + black --check --diff --target-version=py36 httpx tests

7 All done! 🎉 🍌 🎉

8 48 files would be left unchanged.

9 + flake8 httpx tests

10 httpx/\_types.py:7:1: F401 'typing.AnyStr' imported but unused

11 ##[error]Process completed with exit code 1.

○ Build package & docs

○ Run tests

○ Enforce coverage

▶ ✓ Post Run actions/checkout@v2

▶ ✓ Complete job

2s

1s

0s

41s

5s

0s

0s

0s

0s

0s

This job failing means there is either a code formatting issue or type-annotation issue. You can look at the job output to figure out why it's failed or within a shell run:

```
$ scripts/check
```

It may be worth it to run `$ scripts/lint` to attempt auto-formatting the code and if that job succeeds commit the changes.

## Docs Job Failed

This job failing means the documentation failed to build. This can happen for a variety of reasons like invalid markdown or missing configuration within `mkdocs.yml`.

## Python 3.X Job Failed

Test Suite / Python 3.6  
failed 19 days ago in 59s

Search logs

- Set up job 2s
- Run actions/checkout@v2 1s
- Run actions/setup-python@v1 0s
- Install dependencies 34s
- Run linting checks 10s
- Build package & docs 3s
- Run tests 9s

```

35 tests/models/test_headers.py ..... [ 84%]
36 tests/models/test_queryparams.py ..... [ 85%]
37 tests/models/test_requests.py ..... [ 88%]
38 tests/models/test_responses.py ..... [ 96%]
39 tests/models/test_url.py ..... [100%]
40
41 ===== FAILURES =====
42 test_write_timeout[asyncio]
43
44 server = <tests.conftest.TestServer object at 0x7fb137e55940>
45
46 @pytest.mark.usefixtures("async_environment")
47 async def test_write_timeout(server):
48     timeout = httpx.Timeout(write_timeout=1e-6)
49
50     async with httpx.AsyncClient(timeout=timeout) as client:
51         with pytest.raises(httpx.WriteTimeout):
52             data = b"*" * 1024 * 1024 * 100
53             await client.put(server.url.copy_with(path="/slow_response"), data=data)
54 E       Failed: DID NOT RAISE <class 'httpcore._exceptions.WriteTimeout'>
55
56 tests/test_timeouts.py:22: Failed
57 ----- Captured stdout call -----
58 INFO: 127.0.0.1:33240 - "PUT /slow_response HTTP/1.1" 200 OK
59
60 ----- coverage: platform linux, python 3.6.10-final-0 -----
61 Name                               Stmts Miss Cover Missing
62 -----

```

This job failing means the unit tests failed or not all code paths are covered by unit tests.

If tests are failing you will see this message under the coverage report:

```
=== 1 failed, 435 passed, 1 skipped, 1 xfailed in 11.09s ===
```

If tests succeed but coverage doesn't reach our current threshold, you will see this message under the coverage report:

```
FAIL Required test coverage of 100% not reached. Total coverage: 99.00%
```

## Releasing

*This section is targeted at HTTPX maintainers.*

Before releasing a new version, create a pull request that includes:

- **An update to the changelog:**
  - We follow the format from keepachangelog.
  - Compare `master` with the tag of the latest release, and list all entries that are of interest to our users:
    - Things that **must** go in the changelog: added, changed, deprecated or removed features, and bug fixes.
    - Things that **should not** go in the changelog: changes to documentation, tests or tooling.
    - Try sorting entries in descending order of impact / importance.
    - Keep it concise and to-the-point. ☐
- **A version bump:** see `__version__.py`.

For an example, see #1006.



Once the release PR is merged, create a new release including:

- Tag version like `0.13.3`.
- Release title `Version 0.13.3`
- Description copied from the changelog.

Once created this release will be automatically uploaded to PyPI.

If something goes wrong with the PyPI job the release can be published using the `scripts/publish` script.

## Development proxy setup

To test and debug requests via a proxy it's best to run a proxy server locally. Any server should do but HTTPCore's test suite uses `mitmproxy` which is written in Python, it's fully featured and has excellent UI and tools for introspection of requests.

You can install `mitmproxy` using `pip install mitmproxy` or several other ways.

`mitmproxy` does require setting up local TLS certificates for HTTPS requests, as its main purpose is to allow developers to inspect requests that pass through it. We can set them up follows:

1. `pip install trustme-cli`.
2. `trustme-cli -i example.org www.example.org`, assuming you want to test connecting to that domain, this will create three files: `server.pem`, `server.key` and `client.pem`.
3. `mitmproxy` requires a PEM file that includes the private key and the certificate so we need to concatenate them: `cat server.key server.pem > server.withkey.pem`.
4. Start the proxy server `mitmproxy --certs server.withkey.pem`, or use the other `mitmproxy` commands with different UI options.

At this point the server is ready to start serving requests, you'll need to configure HTTPX as described in the proxy section and the SSL certificates section, this is where our previously generated `client.pem` comes in:

```
import httpx

proxies = {"all://": "http://127.0.0.1:8080/"}

with httpx.Client(proxies=proxies, verify="/path/to/client.pem") as client:
    response = client.get("https://example.org")
    print(response.status_code) # should print 200
```

Note, however, that HTTPS requests will only succeed to the host specified in the SSL/TLS certificate we generated, HTTPS requests to other hosts will raise an error like:

```
ssl.SSLCertVerificationError: [SSL: CERTIFICATE_VERIFY_FAILED] certificate
verify failed: Hostname mismatch, certificate is not valid for
'duckduckgo.com'. (_ssl.c:1108)
```

If you want to make requests to more hosts you'll need to regenerate the certificates and include all the hosts you intend to connect to in the second step, i.e.

```
trustme-cli -i example.org www.example.org duckduckgo.com www.duckduckgo.com
```



## HTTPX

# Requests Compatibility Guide

HTTPX aims to be broadly compatible with the `requests` API, although there are a few design differences in places.

This documentation outlines places where the API differs...

## Redirects

Unlike `requests`, HTTPX does **not follow redirects by default**.

We differ in behaviour here because auto-redirects can easily mask unnecessary network calls being made.

You can still enable behaviour to automatically follow redirects, but you need to do so explicitly...

```
response = client.get(url, follow_redirects=True)
```

Or else instantiate a client, with redirect following enabled by default...

```
client = httpx.Client(follow_redirects=True)
```

## Client instances

The HTTPX equivalent of `requests.Session` is `httpx.Client`.

```
session = requests.Session(**kwargs)
```

is generally equivalent to

```
client = httpx.Client(**kwargs)
```

## Request URLs

Accessing `response.url` will return a `URL` instance, rather than a string.

Use `str(response.url)` if you need a string instance.

## Determining the next redirect request

The `requests` library exposes an attribute `response.next`, which can be used to obtain the next redirect request.

```
session = requests.Session()
request = requests.Request("GET", ...).prepare()
while request is not None:
    response = session.send(request, allow_redirects=False)
    request = response.next
```

In HTTPX, this attribute is instead named `response.next_request`. For example:

```
client = httpx.Client()
request = client.build_request("GET", ...)
while request is not None:
    response = client.send(request)
    request = response.next_request
```

## Request Content

For uploading raw text or binary content we prefer to use a `content` parameter, in order to better separate this usage from the case of uploading form data.

For example, using `content=...` to upload raw content:

```
# Uploading text, bytes, or a bytes iterator.
httpx.post(..., content=b"Hello, world")
```

And using `data=...` to send form data:

```
# Uploading form data.
httpx.post(..., data={"message": "Hello, world"})
```

Using the `data=<text/byte content>` will raise a deprecation warning, and is expected to be fully removed with the HTTPX 1.0 release.

## Upload files

HTTPX strictly enforces that upload files must be opened in binary mode, in order to avoid character encoding issues that can result from attempting to upload files opened in text mode.

## Content encoding

HTTPX uses `utf-8` for encoding `str` request bodies. For example, when using `content=<str>` the request body will be encoded to `utf-8` before being sent over the wire. This differs from Requests which uses `latin1`. If you need an explicit encoding, pass encoded bytes explicitly, e.g. `content=<str>.encode("latin1")`. For response bodies, assuming the server didn't send an explicit encoding then HTTPX will do its best to figure out an appropriate encoding. HTTPX makes a guess at the encoding to use for decoding the response using `charset_normalizer`. Fallback to that or any content with less than 32 octets will be decoded using `utf-8` with the `error="replace"` decoder strategy.

## Cookies

If using a client instance, then cookies should always be set on the client rather than on a per-request basis.

This usage is supported:

```
client = httpx.Client(cookies=...)
client.post(...)
```

This usage is **not** supported:

```
client = httpx.Client()
client.post(..., cookies=...)
```

We prefer enforcing a stricter API here because it provides clearer expectations around cookie persistence, particularly when redirects occur.

## Status Codes

In our documentation we prefer the uppercased versions, such as `codes.NOT_FOUND`, but also provide lower-cased versions for API compatibility with `requests`.

Requests includes various synonyms for status codes that HTTPX does not support.

## Streaming responses

HTTPX provides a `.stream()` interface rather than using `stream=True`. This ensures that streaming responses are always properly closed outside of the stream block, and makes it visually clearer at which points streaming I/O APIs may be used with a response.

For example:

```
with httpx.stream("GET", "https://www.example.com") as response:
    ...
```

Within a `stream()` block request data is made available with:

- `.iter_bytes()` - Instead of `response.iter_content()`
- `.iter_text()` - Instead of `response.iter_content(decode_unicode=True)`
- `.iter_lines()` - Corresponding to `response.iter_lines()`
- `.iter_raw()` - Use this instead of `response.raw`
- `.read()` - Read the entire response body, making `request.text` and `response.content` available.

## Timeouts

HTTPX defaults to including reasonable timeouts for all network operations, while Requests has no timeouts by default.

To get the same behavior as Requests, set the `timeout` parameter to `None`:

```
httpx.get('https://www.example.com', timeout=None)
```

## Proxy keys

When using `httpx.Client(proxies={...})` to map to a selection of different proxies, we use full URL schemes, such as `proxies={"http://": ..., "https://": ...}`.

This is different to the `requests` usage of `proxies={"http": ..., "https": ...}`.

This change is for better consistency with more complex mappings, that might also include domain names, such as `proxies={"all://": ..., "all://www.example.com": None}` which maps all requests onto a proxy, except for requests to "www.example.com" which have an explicit exclusion.

Also note that `requests.Session.request(...)` allows a `proxies=...` parameter, whereas `httpx.Client.request(...)` does not.

## SSL configuration

When using a `Client` instance, the `trust_env`, `verify`, and `cert` arguments should always be passed on client instantiation, rather than passed to the request method.

If you need more than one different SSL configuration, you should use different client instances for each SSL configuration.

Requests supports `REQUESTS_CA_BUNDLE` which points to either a file or a directory. HTTPX supports the `SSL_CERT_FILE` (for a file) and `SSL_CERT_DIR` (for a directory) OpenSSL variables instead.

## Request body on HTTP methods

The HTTP `GET`, `DELETE`, `HEAD`, and `OPTIONS` methods are specified as not supporting a request body. To stay in line with this, the `.get`, `.delete`, `.head` and `.options` functions do not support `content`, `files`, `data`, or `json` arguments.

If you really do need to send request data using these http methods you should use the generic `.request` function instead.

```
httpx.request(  
    method="DELETE",  
    url="https://www.example.com/",  
    content=b'A request body on a DELETE request.'  
)
```

## Checking for success and failure responses

We don't support `response.is_ok` since the naming is ambiguous there, and might incorrectly imply an equivalence to `response.status_code == codes.OK`. Instead we provide the `response.is_success` property, which can be used to check for a 2xx response.

## Request instantiation

There is no notion of prepared requests in HTTPX. If you need to customize request instantiation, see [Request instances](#).

Besides, `httpx.Request()` does not support the `auth`, `timeout`, `follow_redirects`, `proxies`, `verify` and `cert` parameters. However these are available in `httpx.request`, `httpx.get`, `httpx.post` etc., as well as on `Client` instances.

## Mocking

If you need to mock HTTPX the same way that test utilities like `responses` and `requests-mock` does for `requests`, see [RESPX](#).

## Caching

If you use `cachecontrol` or `requests-cache` to add HTTP Caching support to the `requests` library, you can use [Hishel](#) for HTTPX.

## Networking layer

`requests` defers most of its HTTP networking code to the excellent `urllib3` library.

On the other hand, HTTPX uses `HTTPCore` as its core HTTP networking layer, which is a different project than `urllib3`.

# Query Parameters

`requests` omits `params` whose values are `None` (e.g. `requests.get(..., params={"foo": None})`). This is not supported by HTTPX.

For both query params (`params=`) and form data (`data=`), `requests` supports sending a list of tuples (e.g. `requests.get(..., params=[('key1', 'value1'), ('key1', 'value2')])`). This is not supported by HTTPX. Instead, use a dictionary with lists as values. E.g.: `httpx.get(..., params={'key1': ['value1', 'value2']})` or with form data: `httpx.post(..., data={'key1': ['value1', 'value2']})`.

## Event Hooks

`requests` allows event hooks to mutate `Request` and `Response` objects. See examples given in the documentation for `requests`.

In HTTPX, event hooks may access properties of requests and responses, but event hook callbacks cannot mutate the original request/response.

If you are looking for more control, consider checking out Custom Transports.

Made with Material for MkDocs



## HTTPX

# Advanced Usage

## Client Instances

### Hint

If you are coming from Requests, `httpx.Client()` is what you can use instead of `requests.Session()`.

## Why use a Client?

### TL;DR

If you do anything more than experimentation, one-off scripts, or prototypes, then you should use a `Client` instance.

### More efficient usage of network resources

When you make requests using the top-level API as documented in the Quickstart guide, HTTPX has to establish a new connection *for every single request* (connections are not reused). As the number of requests to a host increases, this quickly becomes inefficient.

On the other hand, a `Client` instance uses HTTP connection pooling. This means that when you make several requests to the same host, the `Client` will reuse the underlying TCP connection, instead of recreating one for every single request.

This can bring **significant performance improvements** compared to using the top-level API, including:

- Reduced latency across requests (no handshaking).
- Reduced CPU usage and round-trips.
- Reduced network congestion.

### Extra features

`Client` instances also support features that aren't available at the top-level API, such as:

- Cookie persistence across requests.
- Applying configuration across all outgoing requests.
- Sending requests through HTTP proxies.
- Using HTTP/2.

The other sections on this page go into further detail about what you can do with a `Client` instance.

## Usage

The recommended way to use a `Client` is as a context manager. This will ensure that connections are properly cleaned up when leaving the `with` block:

```
with httpx.Client() as client:
    ...
```

Alternatively, you can explicitly close the connection pool without block-usage using `.close()` :

```
client = httpx.Client()
try:
    ...
finally:
    client.close()
```

## Making requests

Once you have a `Client`, you can send requests using `.get()`, `.post()`, etc. For example:

```
>>> with httpx.Client() as client:
...     r = client.get('https://example.com')
...
>>> r
<Response [200 OK]>
```

These methods accept the same arguments as `httpx.get()`, `httpx.post()`, etc. This means that all features documented in the Quickstart guide are also available at the client level.

For example, to send a request with custom headers:

```
>>> with httpx.Client() as client:
...     headers = {'X-Custom': 'value'}
...     r = client.get('https://example.com', headers=headers)
...
>>> r.request.headers['X-Custom']
'value'
```

## Sharing configuration across requests

Clients allow you to apply configuration to all outgoing requests by passing parameters to the `Client` constructor.

For example, to apply a set of custom headers *on every request*:

```
>>> url = 'http://httpbin.org/headers'
>>> headers = {'user-agent': 'my-app/0.0.1'}
>>> with httpx.Client(headers=headers) as client:
...     r = client.get(url)
...
>>> r.json()['headers']['User-Agent']
'my-app/0.0.1'
```

## Merging of configuration

When a configuration option is provided at both the client-level and request-level, one of two things can happen:

- For headers, query parameters and cookies, the values are combined together. For example:



```
>>> headers = {'X-Auth': 'from-client'}
>>> params = {'client_id': 'client1'}
>>> with httpx.Client(headers=headers, params=params) as client:
...     headers = {'X-Custom': 'from-request'}
...     params = {'request_id': 'request1'}
...     r = client.get('https://example.com', headers=headers, params=params)
...
>>> r.request.url
URL('https://example.com?client_id=client1&request_id=request1')
>>> r.request.headers['X-Auth']
'from-client'
>>> r.request.headers['X-Custom']
'from-request'
```

- For all other parameters, the request-level value takes priority. For example:

```
>>> with httpx.Client(auth=('tom', 'mot123')) as client:
...     r = client.get('https://example.com', auth=('alice', 'ecila123'))
...
>>> _, _, auth = r.request.headers['Authorization'].partition(' ')
>>> import base64
>>> base64.b64decode(auth)
b'alice:ecila123'
```

If you need finer-grained control on the merging of client-level and request-level parameters, see [Request instances](#).

## Other Client-only configuration options

Additionally, `Client` accepts some configuration options that aren't available at the request level.

For example, `base_url` allows you to prepend an URL to all outgoing requests:

```
>>> with httpx.Client(base_url='http://httpbin.org') as client:
...     r = client.get('/headers')
...
>>> r.request.url
URL('http://httpbin.org/headers')
```

For a list of all available client parameters, see the [Client API reference](#).

## Character set encodings and auto-detection

When accessing `response.text`, we need to decode the response bytes into a unicode text representation.

By default `httpx` will use "charset" information included in the response `Content-Type` header to determine how the response bytes should be decoded into text.

In cases where no charset information is included on the response, the default behaviour is to assume "utf-8" encoding, which is by far the most widely used text encoding on the internet.

## Using the default encoding

To understand this better let's start by looking at the default behaviour for text decoding...

```
import httpx
# Instantiate a client with the default configuration.
client = httpx.Client()
# Using the client...
response = client.get(...)
print(response.encoding) # This will either print the charset given in
                        # the Content-Type charset, or else "utf-8".
print(response.text) # The text will either be decoded with the Content-Type
                    # charset, or using "utf-8".
```

This is normally absolutely fine. Most servers will respond with a properly formatted Content-Type header, including a charset encoding. And in most cases where no charset encoding is included, UTF-8 is very likely to be used, since it is so widely adopted.

## Using an explicit encoding

In some cases we might be making requests to a site where no character set information is being set explicitly by the server, but we know what the encoding is. In this case it's best to set the default encoding explicitly on the client.

```
import httpx
# Instantiate a client with a Japanese character set as the default encoding.
client = httpx.Client(default_encoding="shift-jis")
# Using the client...
response = client.get(...)
print(response.encoding) # This will either print the charset given in
                        # the Content-Type charset, or else "shift-jis".
print(response.text) # The text will either be decoded with the Content-Type
                    # charset, or using "shift-jis".
```

## Using character set auto-detection

In cases where the server is not reliably including character set information, and where we don't know what encoding is being used, we can enable auto-detection to make a best-guess attempt when decoding from bytes to text.

To use auto-detection you need to set the `default_encoding` argument to a callable instead of a string. This callable should be a function which takes the input bytes as an argument and returns the character set to use for decoding those bytes to text.

There are two widely used Python packages which both handle this functionality:

- `chardet` - This is a well established package, and is a port of the auto-detection code in Mozilla.
- `charset-normalizer` - A newer package, motivated by `chardet`, with a different approach.

Let's take a look at installing autodetection using one of these packages...

```
shell $ pip install httpx $ pip install chardet
```

Once `chardet` is installed, we can configure a client to use character-set autodetection.

```
import httpx
import chardet

def autodetect(content):
    return chardet.detect(content).get("encoding")

# Using a client with character-set autodetection enabled.
client = httpx.Client(default_encoding=autodetect)
response = client.get(...)
print(response.encoding) # This will either print the charset given in
                        # the Content-Type charset, or else the auto-detected
                        # character set.
print(response.text)
```

## Calling into Python Web Apps

You can configure an `httpx` client to call directly into a Python web application using the WSGI protocol.

This is particularly useful for two main use-cases:

- Using `httpx` as a client inside test cases.
- Mocking out external services during tests or in dev/staging environments.

Here's an example of integrating against a Flask application:

```
from flask import Flask
import httpx

app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

with httpx.Client(app=app, base_url="http://testserver") as client:
    r = client.get("/")
    assert r.status_code == 200
    assert r.text == "Hello World!"
```

For some more complex cases you might need to customize the WSGI transport. This allows you to:

- Inspect 500 error responses rather than raise exceptions by setting `raise_app_exceptions=False`.
- Mount the WSGI application at a subpath by setting `script_name` (WSGI).
- Use a given client address for requests by setting `remote_addr` (WSGI).

For example:

```
# Instantiate a client that makes WSGI requests with a client IP of "1.2.3.4".
transport = httpx.WSGITransport(app=app, remote_addr="1.2.3.4")
with httpx.Client(transport=transport, base_url="http://testserver") as client:
    ...
```

## Request instances

For maximum control on what gets sent over the wire, HTTPX supports building explicit `Request` instances:

```
request = httpx.Request("GET", "https://example.com")
```

To dispatch a `Request` instance across to the network, create a `Client` instance and use `.send()`:

```
with httpx.Client() as client:
    response = client.send(request)
    ...
```

If you need to mix client-level and request-level options in a way that is not supported by the default Merging of parameters, you can use `.build_request()` and then make arbitrary modifications to the `Request` instance. For example:

```
headers = {"X-API-Key": "...", "X-Client-ID": "ABC123"}

with httpx.Client(headers=headers) as client:
    request = client.build_request("GET", "https://api.example.com")

    print(request.headers["X-Client-ID"]) # "ABC123"

    # Don't send the API key for this particular request.
    del request.headers["X-API-Key"]

    response = client.send(request)
    ...
```

## Event Hooks

HTTPX allows you to register "event hooks" with the client, that are called every time a particular type of

event takes place.

There are currently two event hooks:

- `request` - Called after a request is fully prepared, but before it is sent to the network. Passed the `request` instance.
- `response` - Called after the response has been fetched from the network, but before it is returned to the caller. Passed the `response` instance.

These allow you to install client-wide functionality such as logging, monitoring or tracing.

```
def log_request(request):
    print(f"Request event hook: {request.method} {request.url} - Waiting for response")

def log_response(response):
    request = response.request
    print(f"Response event hook: {request.method} {request.url} - Status {response.status_code}")

client = httpx.Client(event_hooks={'request': [log_request], 'response': [log_response]})
```

You can also use these hooks to install response processing code, such as this example, which creates a client instance that always raises `httpx.HTTPStatusError` on 4xx and 5xx responses.

```
def raise_on_4xx_5xx(response):
    response.raise_for_status()

client = httpx.Client(event_hooks={'response': [raise_on_4xx_5xx]})
```

#### Note

Response event hooks are called before determining if the response body should be read or not.

If you need access to the response body inside an event hook, you'll need to call `response.read()`, or for `AsyncClients`, `response.aread()`.

The hooks are also allowed to modify `request` and `response` objects.

```
def add_timestamp(request):
    request.headers['x-request-timestamp'] = datetime.now(tz=datetime.utcnow()).isoformat()

client = httpx.Client(event_hooks={'request': [add_timestamp]})
```

Event hooks must always be set as a **list of callables**, and you may register multiple event hooks for each type of event.

As well as being able to set event hooks on instantiating the client, there is also an `.event_hooks` property, that allows you to inspect and modify the installed hooks.

```
client = httpx.Client()
client.event_hooks['request'] = [log_request]
client.event_hooks['response'] = [log_response, raise_on_4xx_5xx]
```

#### Note

If you are using HTTPX's async support, then you need to be aware that hooks registered with `httpx.AsyncClient` **MUST** be async functions, rather than plain functions.

## Monitoring download progress

If you need to monitor download progress of large responses, you can use response streaming and inspect the `response.num_bytes_downloaded` property.

This interface is required for properly determining download progress, because the total number of bytes returned by `response.content` or `response.iter_content()` will not always correspond with the raw content length of the response if HTTP response compression is being used.

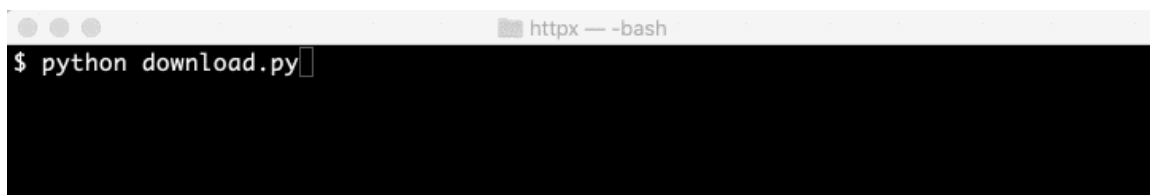
For example, showing a progress bar using the `tqdm` library while a response is being downloaded could be done like this...

```
import tempfile

import httpx
from tqdm import tqdm

with tempfile.NamedTemporaryFile() as download_file:
    url = "https://speed.hetzner.de/100MB.bin"
    with httpx.stream("GET", url) as response:
        total = int(response.headers["Content-Length"])

    with tqdm(total=total, unit_scale=True, unit_divisor=1024, unit="B") as progress:
        num_bytes_downloaded = response.num_bytes_downloaded
        for chunk in response.iter_bytes():
            download_file.write(chunk)
            progress.update(response.num_bytes_downloaded - num_bytes_downloaded)
            num_bytes_downloaded = response.num_bytes_downloaded
```

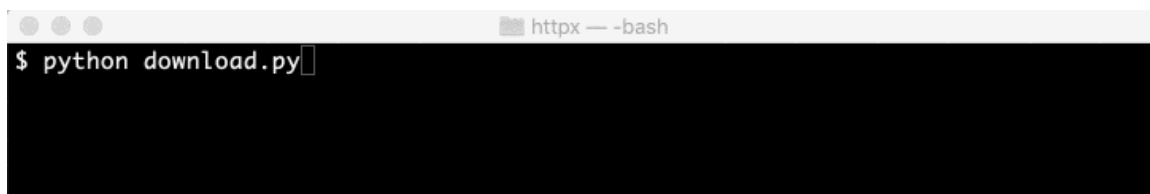


Or an alternate example, this time using the `rich` library...

```
import tempfile
import httpx
import rich.progress

with tempfile.NamedTemporaryFile() as download_file:
    url = "https://speed.hetzner.de/100MB.bin"
    with httpx.stream("GET", url) as response:
        total = int(response.headers["Content-Length"])

    with rich.progress.Progress(
        "[progress.percentage]{task.percentage:>3.0f}% ",
        rich.progress.BarColumn(bar_width=None),
        rich.progress.DownloadColumn(),
        rich.progress.TransferSpeedColumn(),
    ) as progress:
        download_task = progress.add_task("Download", total=total)
        for chunk in response.iter_bytes():
            download_file.write(chunk)
            progress.update(download_task, completed=response.num_bytes_downloaded)
```



## Monitoring upload progress

If you need to monitor upload progress of large responses, you can use request content generator streaming.

For example, showing a progress bar using the `tqdm` library.

```

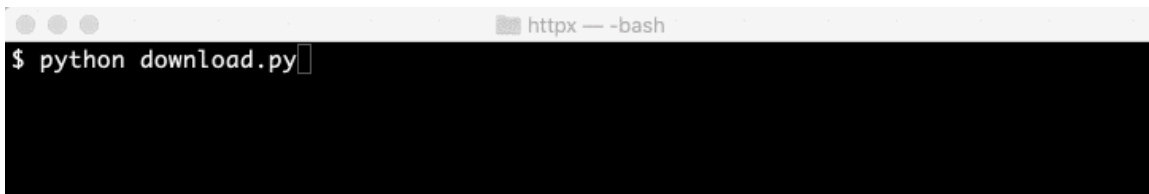
import io
import random

import httpx
from tqdm import tqdm

def gen():
    """
    this is a complete example with generated random bytes.
    you can replace `io.BytesIO` with real file object.
    """
    total = 32 * 1024 * 1024 # 32m
    with tqdm(ascii=True, unit_scale=True, unit='B', unit_divisor=1024, total=total) as bar:
        with io.BytesIO(random.randbytes(total)) as f:
            while data := f.read(1024):
                yield data
                bar.update(len(data))

httpx.post("https://httpbin.org/post", content=gen())

```



## .netrc Support

HTTPX can be configured to use a `.netrc` config file for authentication.

The `.netrc` config file allows authentication credentials to be associated with specified hosts. When a request is made to a host that is found in the netrc file, the username and password will be included using HTTP basic auth.

Example `.netrc` file:

```

machine example.org
login example-username
password example-password

machine python-httpx.org
login other-username
password other-password

```

Some examples of configuring `.netrc` authentication with `httpx`.

Use the default `.netrc` file in the users home directory:

```

>>> auth = httpx.NetRCAuth()
>>> client = httpx.Client(auth=auth)

```

Use an explicit path to a `.netrc` file:

```

>>> auth = httpx.NetRCAuth(file="/path/to/.netrc")
>>> client = httpx.Client(auth=auth)

```

Use the `NETRC` environment variable to configure a path to the `.netrc` file, or fallback to the default.

```

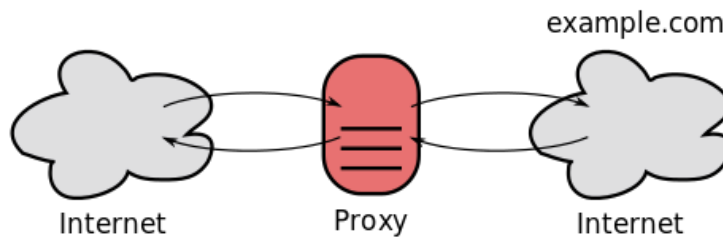
>>> auth = httpx.NetRCAuth(file=os.environ.get("NETRC"))
>>> client = httpx.Client(auth=auth)

```

The `NetRCAuth()` class uses the `netrc.netrc()` function from the Python standard library. See the documentation there for more details on exceptions that may be raised if the netrc file is not found, or cannot be parsed.

# HTTP Proxying

HTTPX supports setting up HTTP proxies via the `proxies` parameter to be passed on client initialization or top-level API functions like `httpx.get(..., proxies=...)`.



*Diagram of how a proxy works (source: Wikipedia). The left hand side "Internet" blob may be your HTTPX client requesting `example.com` through a proxy.*

## Example

To route all traffic (HTTP and HTTPS) to a proxy located at `http://localhost:8030`, pass the proxy URL to the client...

with `httpx.Client(proxies="http://localhost:8030")` as client:

...

For more advanced use cases, pass a `proxies` dict. For example, to route HTTP and HTTPS requests to 2 different proxies, respectively located at `http://localhost:8030`, and `http://localhost:8031`, pass a dict of proxy URLs:

```
proxies = {
    "http://": "http://localhost:8030",
    "https://": "http://localhost:8031",
}
```

with `httpx.Client(proxies=proxies)` as client:

...

For detailed information about proxy routing, see the Routing section.

### Gotcha

In most cases, the proxy URL for the `https://` key *should* use the `http://` scheme (that's not a typo!).

This is because HTTP proxying requires initiating a connection with the proxy server. While it's possible that your proxy supports doing it via HTTPS, most proxies only support doing it via HTTP.

For more information, see [FORWARD vs TUNNEL](#)

## Authentication

Proxy credentials can be passed as the `userinfo` section of the proxy URL. For example:

```
proxies = {
    "http://": "http://username:password@localhost:8030",
    # ...
}
```

## Routing

HTTPX provides fine-grained controls for deciding which requests should go through a proxy, and which

shouldn't. This process is known as proxy routing.

The `proxies` dictionary maps URL patterns ("proxy keys") to proxy URLs. HTTPX matches requested URLs against proxy keys to decide which proxy should be used, if any. Matching is done from most specific proxy keys (e.g. `https://<domain>:<port>`) to least specific ones (e.g. `https://`).

HTTPX supports routing proxies based on **scheme**, **domain**, **port**, or a combination of these.

## Wildcard routing

Route everything through a proxy...

```
proxies = {
    "all://": "http://localhost:8030",
}
```

## Scheme routing

Route HTTP requests through one proxy, and HTTPS requests through another...

```
proxies = {
    "http://": "http://localhost:8030",
    "https://": "http://localhost:8031",
}
```

## Domain routing

Proxy all requests on domain "example.com", let other requests pass through...

```
proxies = {
    "all://example.com": "http://localhost:8030",
}
```

Proxy HTTP requests on domain "example.com", let HTTPS and other requests pass through...

```
proxies = {
    "http://example.com": "http://localhost:8030",
}
```

Proxy all requests to "example.com" and its subdomains, let other requests pass through...

```
proxies = {
    "all://*example.com": "http://localhost:8030",
}
```

Proxy all requests to strict subdomains of "example.com", let "example.com" and other requests pass through...

```
proxies = {
    "all://*.example.com": "http://localhost:8030",
}
```

## Port routing

Proxy HTTPS requests on port 1234 to "example.com"...

```
proxies = {
    "https://example.com:1234": "http://localhost:8030",
}
```

Proxy all requests on port 1234...

```
proxies = {
    "all://*:1234": "http://localhost:8030",
}
```



## No-proxy support

It is also possible to define requests that *shouldn't* be routed through proxies.

To do so, pass `None` as the proxy URL. For example...

```
proxies = {  
    # Route requests through a proxy by default...  
    "all://": "http://localhost:8031",  
    # Except those for "example.com".  
    "all://example.com": None,  
}
```

## Complex configuration example

You can combine the routing features outlined above to build complex proxy routing configurations. For example...

```
proxies = {  
    # Route all traffic through a proxy by default...  
    "all://": "http://localhost:8030",  
    # But don't use proxies for HTTPS requests to "domain.io"...  
    "https://domain.io": None,  
    # And use another proxy for requests to "example.com" and its subdomains...  
    "all://*example.com": "http://localhost:8031",  
    # And yet another proxy if HTTP is used,  
    # and the "internal" subdomain on port 5550 is requested...  
    "http://internal.example.com:5550": "http://localhost:8032",  
}
```

## Environment variables

HTTP proxying can also be configured through environment variables, although with less fine-grained control.

See documentation on `HTTP_PROXY`, `HTTPS_PROXY`, `ALL_PROXY` for more information.

## Proxy mechanisms

### Note

This section describes **advanced** proxy concepts and functionality.

## FORWARD vs TUNNEL

In general, the flow for making an HTTP request through a proxy is as follows:

1. The client connects to the proxy (initial connection request).
2. The proxy transfers data to the server on your behalf.

How exactly step 2/ is performed depends on which of two proxying mechanisms is used:

- **Forwarding:** the proxy makes the request for you, and sends back the response it obtained from the server.
- **Tunnelling:** the proxy establishes a TCP connection to the server on your behalf, and the client reuses this connection to send the request and receive the response. This is known as an HTTP Tunnel. This mechanism is how you can access websites that use HTTPS from an HTTP proxy (the client "upgrades" the connection to HTTPS by performing the TLS handshake with the server over the TCP connection provided by the proxy).

## Troubleshooting proxies

If you encounter issues when setting up proxies, please refer to our [Troubleshooting guide](#).

## SOCKS

In addition to HTTP proxies, `httpcore` also supports proxies using the SOCKS protocol. This is an optional feature that requires an additional third-party library be installed before use.

You can install SOCKS support using `pip`:

```
$ pip install httpx[socks]
```

You can now configure a client to make requests via a proxy using the SOCKS protocol:

```
httpx.Client(proxies='socks5://user:pass@host:port')
```

## Timeout Configuration

HTTPX is careful to enforce timeouts everywhere by default.

The default behavior is to raise a `TimeoutException` after 5 seconds of network inactivity.

### Setting and disabling timeouts

You can set timeouts for an individual request:

```
# Using the top-level API:
httpx.get('http://example.com/api/v1/example', timeout=10.0)

# Using a client instance:
with httpx.Client() as client:
    client.get("http://example.com/api/v1/example", timeout=10.0)
```

Or disable timeouts for an individual request:

```
# Using the top-level API:
httpx.get('http://example.com/api/v1/example', timeout=None)

# Using a client instance:
with httpx.Client() as client:
    client.get("http://example.com/api/v1/example", timeout=None)
```

### Setting a default timeout on a client

You can set a timeout on a client instance, which results in the given `timeout` being used as the default for requests made with this client:

```
client = httpx.Client() # Use a default 5s timeout everywhere.
client = httpx.Client(timeout=10.0) # Use a default 10s timeout everywhere.
client = httpx.Client(timeout=None) # Disable all timeouts by default.
```

### Fine tuning the configuration

HTTPX also allows you to specify the timeout behavior in more fine grained detail.

There are four different types of timeouts that may occur. These are **connect**, **read**, **write**, and **pool** timeouts.

- The **connect** timeout specifies the maximum amount of time to wait until a socket connection to the

requested host is established. If HTTPX is unable to connect within this time frame, a `ConnectTimeout` exception is raised.

- The **read** timeout specifies the maximum duration to wait for a chunk of data to be received (for example, a chunk of the response body). If HTTPX is unable to receive data within this time frame, a `ReadTimeout` exception is raised.
- The **write** timeout specifies the maximum duration to wait for a chunk of data to be sent (for example, a chunk of the request body). If HTTPX is unable to send data within this time frame, a `WriteTimeout` exception is raised.
- The **pool** timeout specifies the maximum duration to wait for acquiring a connection from the connection pool. If HTTPX is unable to acquire a connection within this time frame, a `PoolTimeout` exception is raised. A related configuration here is the maximum number of allowable connections in the connection pool, which is configured by the `limits` argument.

You can configure the timeout behavior for any of these values...

```
# A client with a 60s timeout for connecting, and a 10s timeout elsewhere.
timeout = httpx.Timeout(10.0, connect=60.0)
client = httpx.Client(timeout=timeout)

response = client.get('http://example.com/')
```

## Pool limit configuration

You can control the connection pool size using the `limits` keyword argument on the client. It takes instances of `httpx.Limits` which define:

- `max_keepalive_connections`, number of allowable keep-alive connections, or `None` to always allow. (Defaults 20)
- `max_connections`, maximum number of allowable connections, or `None` for no limits. (Default 100)
- `keepalive_expiry`, time limit on idle keep-alive connections in seconds, or `None` for no limits. (Default 5)

```
limits = httpx.Limits(max_keepalive_connections=5, max_connections=10)
client = httpx.Client(limits=limits)
```

## Multipart file encoding

As mentioned in the quickstart multipart file encoding is available by passing a dictionary with the name of the payloads as keys and either tuple of elements or a file-like object or a string as values.

```
>>> files = {'upload-file': ('report.xls', open('report.xls', 'rb'), 'application/vnd.ms-excel')}
>>> r = httpx.post("https://httpbin.org/post", files=files)
>>> print(r.text)
{
  ...
  "files": {
    "upload-file": "<... binary content ...>"
  },
  ...
}
```

More specifically, if a tuple is used as a value, it must have between 2 and 3 elements:

- The first element is an optional file name which can be set to `None`.
- The second element may be a file-like object or a string which will be automatically encoded in UTF-8.
- An optional third element can be used to specify the MIME type of the file being uploaded. If not specified HTTPX will attempt to guess the MIME type based on the file name, with unknown file extensions defaulting to "application/octet-stream". If the file name is explicitly set to `None` then HTTPX

will not include a content-type MIME header field.

```
>>> files = {'upload-file': (None, 'text content', 'text/plain')}
>>> r = httpx.post("https://httpbin.org/post", files=files)
>>> print(r.text)
{
  ...
  "files": {},
  "form": {
    "upload-file": "text-content"
  },
  ...
}
```

#### Tip

It is safe to upload large files this way. File uploads are streaming by default, meaning that only one chunk will be loaded into memory at a time.

Non-file data fields can be included in the multipart form using by passing them to `data=...`.

You can also send multiple files in one go with a multiple file field form. To do that, pass a list of `(field, <file>)` items instead of a dictionary, allowing you to pass multiple items with the same `field`. For instance this request sends 2 files, `foo.png` and `bar.png` in one request on the `images` form field:

```
>>> files = [('images', ('foo.png', open('foo.png', 'rb'), 'image/png')),
             ('images', ('bar.png', open('bar.png', 'rb'), 'image/png'))]
>>> r = httpx.post("https://httpbin.org/post", files=files)
```

## Customizing authentication

When issuing requests or instantiating a client, the `auth` argument can be used to pass an authentication scheme to use. The `auth` argument may be one of the following...

- A two-tuple of `username / password`, to be used with basic authentication.
- An instance of `httpx.BasicAuth()`, `httpx.DigestAuth()`, OR `httpx.NetRCAuth()`.
- A callable, accepting a request and returning an authenticated request instance.
- An instance of subclasses of `httpx.Auth`.

The most involved of these is the last, which allows you to create authentication flows involving one or more requests. A subclass of `httpx.Auth` should implement `def auth_flow(request)`, and yield any requests that need to be made...

```
class MyCustomAuth(httpx.Auth):
    def __init__(self, token):
        self.token = token

    def auth_flow(self, request):
        # Send the request, with a custom `X-Authentication` header.
        request.headers['X-Authentication'] = self.token
        yield request
```

If the auth flow requires more than one request, you can issue multiple yields, and obtain the response in each case...

```
class MyCustomAuth(httpx.Auth):
    def __init__(self, token):
        self.token = token

    def auth_flow(self, request):
        response = yield request
        if response.status_code == 401:
            # If the server issues a 401 response then resend the request,
            # with a custom `X-Authentication` header.
            request.headers['X-Authentication'] = self.token
            yield request
```

Custom authentication classes are designed to not perform any I/O, so that they may be used with both sync and async client instances. If you are implementing an authentication scheme that requires the request body, then you need to indicate this on the class using a `requires_request_body` property.

You will then be able to access `request.content` inside the `.auth_flow()` method.

```
class MyCustomAuth(httpx.Auth):
    requires_request_body = True

    def __init__(self, token):
        self.token = token

    def auth_flow(self, request):
        response = yield request
        if response.status_code == 401:
            # If the server issues a 401 response then resend the request,
            # with a custom `X-Authentication` header.
            request.headers['X-Authentication'] = self.sign_request(...)
            yield request

    def sign_request(self, request):
        # Create a request signature, based on `request.method`, `request.url`,
        # `request.headers`, and `request.content`.
        ...
```

Similarly, if you are implementing a scheme that requires access to the response body, then use the `requires_response_body` property. You will then be able to access response body properties and methods such as `response.content`, `response.text`, `response.json()`, etc.

```
class MyCustomAuth(httpx.Auth):
    requires_response_body = True

    def __init__(self, access_token, refresh_token, refresh_url):
        self.access_token = access_token
        self.refresh_token = refresh_token
        self.refresh_url = refresh_url

    def auth_flow(self, request):
        request.headers["X-Authentication"] = self.access_token
        response = yield request

        if response.status_code == 401:
            # If the server issues a 401 response, then issue a request to
            # refresh tokens, and resend the request.
            refresh_response = yield self.build_refresh_request()
            self.update_tokens(refresh_response)

            request.headers["X-Authentication"] = self.access_token
            yield request

    def build_refresh_request(self):
        # Return an `httpx.Request` for refreshing tokens.
        ...

    def update_tokens(self, response):
        # Update the `access_token` and `refresh_token` tokens
        # based on a refresh response.
        data = response.json()
        ...
```

If you *do* need to perform I/O other than HTTP requests, such as accessing a disk-based cache, or you need to use concurrency primitives, such as locks, then you should override `.sync_auth_flow()` and `.async_auth_flow()` (instead of `.auth_flow()`). The former will be used by `httpx.Client`, while the latter will be used by `httpx.AsyncClient`.

```

import asyncio
import threading
import httpx

class MyCustomAuth(httpx.Auth):
    def __init__(self):
        self._sync_lock = threading.RLock()
        self._async_lock = asyncio.Lock()

    def sync_get_token(self):
        with self._sync_lock:
            ...

    def sync_auth_flow(self, request):
        token = self.sync_get_token()
        request.headers["Authorization"] = f"Token {token}"
        yield request

    async def async_get_token(self):
        async with self._async_lock:
            ...

    async def async_auth_flow(self, request):
        token = await self.async_get_token()
        request.headers["Authorization"] = f"Token {token}"
        yield request

```

If you only want to support one of the two methods, then you should still override it, but raise an explicit `RuntimeError` .

```

import httpx
import sync_only_library

class MyCustomAuth(httpx.Auth):
    def sync_auth_flow(self, request):
        token = sync_only_library.get_token(...)
        request.headers["Authorization"] = f"Token {token}"
        yield request

    async def async_auth_flow(self, request):
        raise RuntimeError("Cannot use a sync authentication class with httpx.AsyncClient")

```

## SSL certificates

When making a request over HTTPS, HTTPX needs to verify the identity of the requested host. To do this, it uses a bundle of SSL certificates (a.k.a. CA bundle) delivered by a trusted certificate authority (CA).

### Changing the verification defaults

By default, HTTPX uses the CA bundle provided by Certifi. This is what you want in most cases, even though some advanced situations may require you to use a different set of certificates.

If you'd like to use a custom CA bundle, you can use the `verify` parameter.

```

import httpx

r = httpx.get("https://example.org", verify="path/to/client.pem")

```

Alternatively, you can pass a standard library `ssl.SSLContext` .

```

>>> import ssl
>>> import httpx
>>> context = ssl.create_default_context()
>>> context.load_verify_locations(cafile="/tmp/client.pem")
>>> httpx.get('https://example.org', verify=context)
<Response [200 OK]>

```

We also include a helper function for creating properly configured `SSLContext` instances.

```
>>> context = httpx.create_ssl_context()
```

The `create_ssl_context` function accepts the same set of SSL configuration arguments (`trust_env`, `verify`, `cert` and `http2` arguments) as `httpx.Client` or `httpx.AsyncClient`

```
>>> import httpx
>>> context = httpx.create_ssl_context(verify="/tmp/client.pem")
>>> httpx.get("https://example.org", verify=context)
<Response [200 OK]>
```

Or you can also disable the SSL verification entirely, which is *not* recommended.

```
import httpx

r = httpx.get("https://example.org", verify=False)
```

## SSL configuration on client instances

If you're using a `Client()` instance, then you should pass any SSL settings when instantiating the client.

```
client = httpx.Client(verify=False)
```

The `client.get(...)` method and other request methods *do not* support changing the SSL settings on a per-request basis. If you need different SSL settings in different cases you should use more than one client instance, with different settings on each. Each client will then be using an isolated connection pool with a specific fixed SSL configuration on all connections within that pool.

## Client Side Certificates

You can also specify a local cert to use as a client-side certificate, either a path to an SSL certificate file, or two-tuple of (certificate file, key file), or a three-tuple of (certificate file, key file, password)

```
import httpx

r = httpx.get("https://example.org", cert="path/to/client.pem")
```

Alternatively,

```
>>> cert = ("path/to/client.pem", "path/to/client.key")
>>> httpx.get("https://example.org", cert=cert)
<Response [200 OK]>
```

or

```
>>> cert = ("path/to/client.pem", "path/to/client.key", "password")
>>> httpx.get("https://example.org", cert=cert)
<Response [200 OK]>
```

## Making HTTPS requests to a local server

When making requests to local servers, such as a development server running on `localhost`, you will typically be using unencrypted HTTP connections.

If you do need to make HTTPS connections to a local server, for example to test an HTTPS-only service, you will need to create and use your own certificates. Here's one way to do it:

1. Use `trustme` to generate a pair of server key/cert files, and a client cert file.
2. Pass the server key/cert files when starting your local server. (This depends on the particular web server you're using. For example, `Uvicorn` provides the `--ssl-keyfile` and `--ssl-certfile` options.)

3. Tell HTTPX to use the certificates stored in `client.pem` :

```
>>> import httpx
>>> r = httpx.get("https://localhost:8000", verify="/tmp/client.pem")
>>> r
Response <200 OK>
```

## Custom Transports

HTTPX's `Client` also accepts a `transport` argument. This argument allows you to provide a custom Transport object that will be used to perform the actual sending of the requests.

### Usage

For some advanced configuration you might need to instantiate a transport class directly, and pass it to the client instance. One example is the `local_address` configuration which is only available via this low-level API.

```
>>> import httpx
>>> transport = httpx.HTTPTransport(local_address="0.0.0.0")
>>> client = httpx.Client(transport=transport)
```

Connection retries are also available via this interface. Requests will be retried the given number of times in case an `httpx.ConnectError` or an `httpx.ConnectTimeout` occurs, allowing smoother operation under flaky networks. If you need other forms of retry behaviors, such as handling read/write errors or reacting to `503 Service Unavailable`, consider general-purpose tools such as `tenacity`.

```
>>> import httpx
>>> transport = httpx.HTTPTransport(retries=1)
>>> client = httpx.Client(transport=transport)
```

Similarly, instantiating a transport directly provides a `uds` option for connecting via a Unix Domain Socket that is only available via this low-level API:

```
>>> import httpx
>>> # Connect to the Docker API via a Unix Socket.
>>> transport = httpx.HTTPTransport(uds="/var/run/docker.sock")
>>> client = httpx.Client(transport=transport)
>>> response = client.get("http://docker/info")
>>> response.json()
{"ID": "...", "Containers": 4, "Images": 74, ...}
```

### urllib3 transport

This public gist provides a transport that uses the excellent `urllib3` library, and can be used with the sync `Client` ...

```
>>> import httpx
>>> from urllib3_transport import URLLib3Transport
>>> client = httpx.Client(transport=URLLib3Transport())
>>> client.get("https://example.org")
<Response [200 OK]>
```

### Writing custom transports

A transport instance must implement the low-level Transport API, which deals with sending a single request, and returning a response. You should either subclass `httpx.BaseTransport` to implement a transport to use with `Client`, or subclass `httpx.AsyncBaseTransport` to implement a transport to use with `AsyncClient`.

At the layer of the transport API we're using the familiar `Request` and `Response` models.



See the `handle_request` and `handle_async_request` docstrings for more details on the specifics of the Transport API.

A complete example of a custom transport implementation would be:

```
import json
import httpx

class HelloWorldTransport(httpx.BaseTransport):
    """
    A mock transport that always returns a JSON "Hello, world!" response.
    """

    def handle_request(self, request):
        message = {"text": "Hello, world!"}
        content = json.dumps(message).encode("utf-8")
        stream = httpx.ByteStream(content)
        headers = [(b"content-type", b"application/json")]
        return httpx.Response(200, headers=headers, stream=stream)
```

Which we can use in the same way:

```
>>> import httpx
>>> client = httpx.Client(transport=HelloWorldTransport())
>>> response = client.get("https://example.org/")
>>> response.json()
{"text": "Hello, world!"}
```

## Mock transports

During testing it can often be useful to be able to mock out a transport, and return pre-determined responses, rather than making actual network requests.

The `httpx.MockTransport` class accepts a handler function, which can be used to map requests onto pre-determined responses:

```
def handler(request):
    return httpx.Response(200, json={"text": "Hello, world!"})

# Switch to a mock transport, if the TESTING environment variable is set.
if os.environ.get('TESTING', '').upper() == "TRUE":
    transport = httpx.MockTransport(handler)
else:
    transport = httpx.HTTPTransport()

client = httpx.Client(transport=transport)
```

For more advanced use-cases you might want to take a look at either the third-party mocking library, `RESPX`, or the `pytest-httpx` library.

## Mounting transports

You can also mount transports against given schemes or domains, to control which transport an outgoing request should be routed via, with the same style used for specifying proxy routing.

```
import httpx

class HTTPSRedirectTransport(httpx.BaseTransport):
    """
    A transport that always redirects to HTTPS.
    """

    def handle_request(self, method, url, headers, stream, extensions):
        scheme, host, port, path = url
        if port is None:
            location = b"https://%s%s" % (host, path)
        else:
            location = b"https://%s:%d%s" % (host, port, path)
        stream = httpx.ByteStream(b"")
        headers = [(b"location", location)]
        extensions = {}
        return 303, headers, stream, extensions

# A client where any `http` requests are always redirected to `https`
mounts = {'http://': HTTPSRedirectTransport()}
client = httpx.Client(mounts=mounts)
```

A couple of other sketches of how you might take advantage of mounted transports...

Disabling HTTP/2 on a single given domain...

```
mounts = {
    "all://": httpx.HTTPTransport(http2=True),
    "all://example.org": httpx.HTTPTransport()
}
client = httpx.Client(mounts=mounts)
```

Mocking requests to a given domain:

```
# All requests to "example.org" should be mocked out.
# Other requests occur as usual.
def handler(request):
    return httpx.Response(200, json={"text": "Hello, World!"})

mounts = {"all://example.org": httpx.MockTransport(handler)}
client = httpx.Client(mounts=mounts)
```

Adding support for custom schemes:

```
# Support URLs like "file:///Users/sylvia_green/websites/new_client/index.html"
mounts = {"file://": FileSystemTransport()}
client = httpx.Client(mounts=mounts)
```

Made with Material for MkDocs



## HTTPX

# Exceptions

This page lists exceptions that may be raised when using HTTPX.

For an overview of how to work with HTTPX exceptions, see [Exceptions \(Quickstart\)](#).

## The exception hierarchy

- `HTTPError`
  - `RequestError`
    - `TransportError`
      - `TimeoutException`
        - `ConnectTimeout`
        - `ReadTimeout`
        - `WriteTimeout`
        - `PoolTimeout`
      - `NetworkError`
        - `ConnectError`
        - `ReadError`
        - `WriteError`
        - `CloseError`
      - `ProtocolError`
        - `LocalProtocolError`
        - `RemoteProtocolError`
      - `ProxyError`
      - `UnsupportedProtocol`
    - `DecodingError`
    - `TooManyRedirects`
  - `HTTPStatusError`
  - `InvalidURL`
  - `CookieConflict`
  - `StreamError`
    - `StreamConsumed`
    - `ResponseNotRead`
    - `RequestNotRead`
    - `StreamClosed`

# Exception classes

*class* `httpx.HTTPError` (*message*)

Base class for `RequestError` and `HTTPStatusError`.

Useful for `try...except` blocks when issuing a request, and then calling `.raise_for_status()`.

For example:

```
try:
    response = httpx.get("https://www.example.com")
    response.raise_for_status()
except httpx.HTTPError as exc:
    print(f"HTTP Exception for {exc.request.url} - {exc}")
```

*class* `httpx.RequestError` (*message*, \*, *request=None*)

Base class for all exceptions that may occur when issuing a `.request()`.

*class* `httpx.TransportError` (*message*, \*, *request=None*)

Base class for all exceptions that occur at the level of the Transport API.

*class* `httpx.TimeoutException` (*message*, \*, *request=None*)

The base class for timeout errors.

An operation has timed out.

*class* `httpx.ConnectTimeout` (*message*, \*, *request=None*)

Timed out while connecting to the host.

*class* `httpx.ReadTimeout` (*message*, \*, *request=None*)

Timed out while receiving data from the host.

*class* `httpx.WriteTimeout` (*message*, \*, *request=None*)

Timed out while sending data to the host.

*class* `httpx.PoolTimeout` (*message*, \*, *request=None*)

Timed out waiting to acquire a connection from the pool.

*class* `httpx.NetworkError` (*message*, \*, *request=None*)

The base class for network-related errors.

An error occurred while interacting with the network.

*class* `httpx.ConnectError` (*message*, \*, *request=None*)

Failed to establish a connection.

*class* `httpx.ReadError (message, *, request=None)`

Failed to receive data from the network.

*class* `httpx.WriteError (message, *, request=None)`

Failed to send data through the network.

*class* `httpx.CloseError (message, *, request=None)`

Failed to close a connection.

*class* `httpx.ProtocolError (message, *, request=None)`

The protocol was violated.

*class* `httpx.LocalProtocolError (message, *, request=None)`

A protocol was violated by the client.

For example if the user instantiated a `Request` instance explicitly, failed to include the mandatory `Host` header, and then issued it directly using `client.send()` .

*class* `httpx.RemoteProtocolError (message, *, request=None)`

The protocol was violated by the server.

For example, returning malformed HTTP.

*class* `httpx.ProxyError (message, *, request=None)`

An error occurred while establishing a proxy connection.

*class* `httpx.UnsupportedProtocol (message, *, request=None)`

Attempted to make a request to an unsupported protocol.

For example issuing a request to `ftp://www.example.com` .

*class* `httpx.DecodingError (message, *, request=None)`

Decoding of the response failed, due to a malformed encoding.

*class* `httpx.TooManyRedirects (message, *, request=None)`

Too many redirects.

*class* `httpx.HTTPStatusError (message, *, request, response)`

The response had an error HTTP status of 4xx or 5xx.

May be raised when calling `response.raise_for_status()`

*class* `httpx.InvalidURL (message)`

URL is improperly formed or cannot be parsed.

*class* httpx.**CookieConflict** (*message*)

Attempted to lookup a cookie by name, but multiple cookies existed.

Can occur when calling `response.cookies.get(...)` .

*class* httpx.**StreamError** (*message*)

The base class for stream exceptions.

The developer made an error in accessing the request stream in an invalid way.

*class* httpx.**StreamConsumed** ()

Attempted to read or stream content, but the content has already been streamed.

*class* httpx.**StreamClosed** ()

Attempted to read or stream response content, but the request has been closed.

*class* httpx.**ResponseNotRead** ()

Attempted to access streaming response content, without having called `read()` .

*class* httpx.**RequestNotRead** ()

Attempted to access streaming request content, without having called `read()` .

Made with Material for MkDocs



## HTTPX

# Third Party Packages

As HTTPX usage grows, there is an expanding community of developers building tools and libraries that integrate with HTTPX, or depend on HTTPX. Here are some of them.

## Plugins

### Hishel

[GitHub - Documentation](#)

An elegant HTTP Cache implementation for HTTPX and HTTP Core.

### Authlib

[GitHub - Documentation](#)

The ultimate Python library in building OAuth and OpenID Connect clients and servers. Includes an OAuth HTTPX client.

### Gidgethub

[GitHub - Documentation](#)

An asynchronous GitHub API library. Includes HTTPX support.

### HTTPX-Auth

[GitHub - Documentation](#)

Provides authentication classes to be used with HTTPX authentication parameter.

### pytest-HTTPX

[GitHub - Documentation](#)

Provides `httpx_mock` pytest fixture to mock HTTPX within test cases.

### RESPX

[GitHub - Documentation](#)

A utility for mocking out the Python HTTPX library.

### rpc.py

[Github - Documentation](#)

An fast and powerful RPC framework based on ASGI/WSGI. Use HTTPX as the client of the RPC service.

## VCR.py

[GitHub - Documentation](#)

A utility for record and repeat an http request.

## httpx-caching

[Github](#)

This package adds caching functionality to HTTPX

## httpx-sse

[GitHub](#)

Allows consuming Server-Sent Events (SSE) with HTTPX.

## robox

[Github](#)

A library for scraping the web built on top of HTTPX.

## Gists

### urllib3-transport

[GitHub](#)

This public gist provides an example implementation for a custom transport implementation on top of the battle-tested `urllib3` library.

Made with Material for MkDocs





## HTTPX

# Logging

If you need to inspect the internal behaviour of `httpx`, you can use Python's standard logging to output information about the underlying network behaviour.

For example, the following configuration...

```
import logging
import httpx

logging.basicConfig(
    format="%(levelname)s [%(asctime)s] %(name)s - %(message)s",
    datefmt="%Y-%m-%d %H:%M:%S",
    level=logging.DEBUG
)

httpx.get("https://www.example.com")
```

Will send debug level output to the console, or wherever `stdout` is directed too...

```
DEBUG [2023-03-16 14:36:20] httpx - load_ssl_context verify=True cert=None trust_env=True http2=False
DEBUG [2023-03-16 14:36:20] httpx - load_verify_locations cafile='/Users/tomchristie/GitHub/encode/httpx/venv/lib/python3.10/site-packages/certifi/cacert.pem'
DEBUG [2023-03-16 14:36:20] httpcore - connection.connect_tcp.started host='www.example.com' port=443 local_address=None timeout=5.0
DEBUG [2023-03-16 14:36:20] httpcore - connection.connect_tcp.complete return_value=<httpcore.backends.sync.SyncStream object at 0x1068fd270>
DEBUG [2023-03-16 14:36:20] httpcore - connection.start_tls.started ssl_context=<ssl.SSLContext object at 0x10689aa40> server_hostname='www.example.com'
DEBUG [2023-03-16 14:36:20] httpcore - connection.start_tls.complete return_value=<httpcore.backends.sync.SyncStream object at 0x1068fd240>
DEBUG [2023-03-16 14:36:20] httpcore - http11.send_request_headers.started request=<Request [b'GET']>
DEBUG [2023-03-16 14:36:20] httpcore - http11.send_request_headers.complete
DEBUG [2023-03-16 14:36:20] httpcore - http11.send_request_body.started request=<Request [b'GET']>
DEBUG [2023-03-16 14:36:20] httpcore - http11.send_request_body.complete
DEBUG [2023-03-16 14:36:20] httpcore - http11.receive_response_headers.started request=<Request [b'GET']>
DEBUG [2023-03-16 14:36:21] httpcore - http11.receive_response_headers.complete return_value=(b'HTTP/1.1', 200, b'OK', [(b'Content-Encoding', b'gzip'), (b'Accept-
INFO [2023-03-16 14:36:21] httpx - HTTP Request: GET https://www.example.com "HTTP/1.1 200 OK"
DEBUG [2023-03-16 14:36:21] httpcore - http11.receive_response_body.started request=<Request [b'GET']>
DEBUG [2023-03-16 14:36:21] httpcore - http11.receive_response_body.complete
DEBUG [2023-03-16 14:36:21] httpcore - http11.response_closed.started
DEBUG [2023-03-16 14:36:21] httpcore - http11.response_closed.complete
DEBUG [2023-03-16 14:36:21] httpcore - connection.close.started
DEBUG [2023-03-16 14:36:21] httpcore - connection.close.complete
```

Logging output includes information from both the high-level `httpx` logger, and the network-level `httpcore` logger, which can be configured separately.

For handling more complex logging configurations you might want to use the dictionary configuration style...

```

import logging.config
import httpx

LOGGING_CONFIG = {
    "version": 1,
    "handlers": {
        "default": {
            "class": "logging.StreamHandler",
            "formatter": "http",
            "stream": "ext://sys.stderr"
        }
    },
    "formatters": {
        "http": {
            "format": "%(levelname)s [%(asctime)s] %(name)s - %(message)s",
            "datefmt": "%Y-%m-%d %H:%M:%S",
        }
    },
    'loggers': {
        'httpx': {
            'handlers': ['default'],
            'level': 'DEBUG',
        },
        'httpcore': {
            'handlers': ['default'],
            'level': 'DEBUG',
        },
    },
}

logging.config.dictConfig(LOGGING_CONFIG)
httpx.get('https://www.example.com')

```

The exact formatting of the debug logging may be subject to change across different versions of `httpx` and `httpcore`. If you need to rely on a particular format it is recommended that you pin installation of these packages to fixed versions.

Made with Material for MkDocs



## HTTPX

# Code of Conduct

We expect contributors to our projects and online spaces to follow the Python Software Foundation's Code of Conduct.

The Python community is made up of members from around the globe with a diverse set of skills, personalities, and experiences. It is through these differences that our community experiences great successes and continued growth. When you're working with members of the community, this Code of Conduct will help steer your interactions and keep Python a positive, successful, and growing community.

## Our Community

Members of the Python community are **open, considerate, and respectful**. Behaviours that reinforce these values contribute to a positive environment, and include:

- **Being open.** Members of the community are open to collaboration, whether it's on PEPs, patches, problems, or otherwise.
- **Focusing on what is best for the community.** We're respectful of the processes set forth in the community, and we work within them.
- **Acknowledging time and effort.** We're respectful of the volunteer efforts that permeate the Python community. We're thoughtful when addressing the efforts of others, keeping in mind that often times the labor was completed simply for the good of the community.
- **Being respectful of differing viewpoints and experiences.** We're receptive to constructive comments and criticism, as the experiences and skill sets of other members contribute to the whole of our efforts.
- **Showing empathy towards other community members.** We're attentive in our communications, whether in person or online, and we're tactful when approaching differing views.
- **Being considerate.** Members of the community are considerate of their peers -- other Python users.
- **Being respectful.** We're respectful of others, their positions, their skills, their commitments, and their efforts.
- **Gracefully accepting constructive criticism.** When we disagree, we are courteous in raising our issues.
- **Using welcoming and inclusive language.** We're accepting of all who wish to take part in our activities, fostering an environment where anyone can participate and everyone can make a difference.

## Our Standards

Every member of our community has the right to have their identity respected. The Python community is dedicated to providing a positive experience for everyone, regardless of age, gender identity and expression, sexual orientation, disability, physical appearance, body size, ethnicity, nationality, race, or religion (or lack thereof), education, or socio-economic status.

# Inappropriate Behavior

Examples of unacceptable behavior by participants include:

- Harassment of any participants in any form
- Deliberate intimidation, stalking, or following
- Logging or taking screenshots of online activity for harassment purposes
- Publishing others' private information, such as a physical or electronic address, without explicit permission
- Violent threats or language directed against another person
- Incitement of violence or harassment towards any individual, including encouraging a person to commit suicide or to engage in self-harm
- Creating additional online accounts in order to harass another person or circumvent a ban
- Sexual language and imagery in online communities or in any conference venue, including talks
- Insults, put downs, or jokes that are based upon stereotypes, that are exclusionary, or that hold others up for ridicule
- Excessive swearing
- Unwelcome sexual attention or advances
- Unwelcome physical contact, including simulated physical contact (eg, textual descriptions like "hug" or "backrub") without consent or after a request to stop
- Pattern of inappropriate social contact, such as requesting/assuming inappropriate levels of intimacy with others
- Sustained disruption of online community discussions, in-person presentations, or other in-person events
- Continued one-on-one communication after requests to cease
- Other conduct that is inappropriate for a professional audience including people of many different backgrounds

Community members asked to stop any inappropriate behavior are expected to comply immediately.

## Enforcement

We take Code of Conduct violations seriously, and will act to ensure our spaces are welcoming, inclusive, and professional environments to communicate in.

If you need to raise a Code of Conduct report, you may do so privately by email to [tom@tomchristie.com](mailto:tom@tomchristie.com).

Reports will be treated confidentially.

Alternately you may make a report to the Python Software Foundation.



## HTTPX

# HTTP/2

HTTP/2 is a major new iteration of the HTTP protocol, that provides a far more efficient transport, with potential performance benefits. HTTP/2 does not change the core semantics of the request or response, but alters the way that data is sent to and from the server.

Rather than the text format that HTTP/1.1 uses, HTTP/2 is a binary format. The binary format provides full request and response multiplexing, and efficient compression of HTTP headers. The stream multiplexing means that where HTTP/1.1 requires one TCP stream for each concurrent request, HTTP/2 allows a single TCP stream to handle multiple concurrent requests.

HTTP/2 also provides support for functionality such as response prioritization, and server push.

For a comprehensive guide to HTTP/2 you may want to check out "http2 explained".

## Enabling HTTP/2

When using the `httpx` client, HTTP/2 support is not enabled by default, because HTTP/1.1 is a mature, battle-hardened transport layer, and our HTTP/1.1 implementation may be considered the more robust option at this point in time. It is possible that a future version of `httpx` may enable HTTP/2 support by default.

If you're issuing highly concurrent requests you might want to consider trying out our HTTP/2 support. You can do so by first making sure to install the optional HTTP/2 dependencies...

```
$ pip install httpx[http2]
```

And then instantiating a client with HTTP/2 support enabled:

```
client = httpx.AsyncClient(http2=True)
...
```

You can also instantiate a client as a context manager, to ensure that all HTTP connections are nicely scoped, and will be closed once the context block is exited.

```
async with httpx.AsyncClient(http2=True) as client:
    ...
```

HTTP/2 support is available on both `Client` and `AsyncClient`, although it's typically more useful in async contexts if you're issuing lots of concurrent requests.

## Inspecting the HTTP version

Enabling HTTP/2 support on the client does not *necessarily* mean that your requests and responses will be transported over HTTP/2, since both the client *and* the server need to support HTTP/2. If you connect to a server that only supports HTTP/1.1 the client will use a standard HTTP/1.1 connection instead.

You can determine which version of the HTTP protocol was used by examining the `.http_version` property on the response.

```
client = httpx.AsyncClient(http2=True)
response = await client.get(...)
print(response.http_version) # "HTTP/1.0", "HTTP/1.1", or "HTTP/2".
```

Made with Material for MkDocs



## HTTPX

# Async Support

HTTPX offers a standard synchronous API by default, but also gives you the option of an async client if you need it.

Async is a concurrency model that is far more efficient than multi-threading, and can provide significant performance benefits and enable the use of long-lived network connections such as WebSockets.

If you're working with an async web framework then you'll also want to use an async client for sending outgoing HTTP requests.

## Making Async requests

To make asynchronous requests, you'll need an `AsyncClient`.

```
>>> async with httpx.AsyncClient() as client:
...     r = await client.get('https://www.example.com/')
...
>>> r
<Response [200 OK]>
```

### Tip

Use IPython or Python 3.8+ with `python -m asyncio` to try this code interactively, as they support executing `async/await` expressions in the console.

## API Differences

If you're using an async client then there are a few bits of API that use async methods.

### Making requests

The request methods are all async, so you should use `response = await client.get(...)` style for all of the following:

- `AsyncClient.get(url, ...)`
- `AsyncClient.options(url, ...)`
- `AsyncClient.head(url, ...)`
- `AsyncClient.post(url, ...)`
- `AsyncClient.put(url, ...)`
- `AsyncClient.patch(url, ...)`
- `AsyncClient.delete(url, ...)`
- `AsyncClient.request(method, url, ...)`
- `AsyncClient.send(request, ...)`

## Opening and closing clients

Use `async with httpx.AsyncClient()` if you want a context-managed client...

async with `httpx.AsyncClient()` as client:  
...

### Warning

In order to get the most benefit from connection pooling, make sure you're not instantiating multiple client instances - for example by using `async with` inside a "hot loop". This can be achieved either by having a single scoped client that's passed throughout wherever it's needed, or by having a single global client instance.

Alternatively, use `await client.aclose()` if you want to close a client explicitly:

```
client = httpx.AsyncClient()
...
await client.aclose()
```

## Streaming responses

The `AsyncClient.stream(method, url, ...)` method is an async context block.

```
>>> client = httpx.AsyncClient()
>>> async with client.stream('GET', 'https://www.example.com/') as response:
...     async for chunk in response.aiter_bytes():
...         ...
```

The async response streaming methods are:

- `Response.aread()` - For conditionally reading a response inside a stream block.
- `Response.aiter_bytes()` - For streaming the response content as bytes.
- `Response.aiter_text()` - For streaming the response content as text.
- `Response.aiter_lines()` - For streaming the response content as lines of text.
- `Response.aiter_raw()` - For streaming the raw response bytes, without applying content decoding.
- `Response.aclose()` - For closing the response. You don't usually need this, since `.stream` block closes the response automatically on exit.

For situations when context block usage is not practical, it is possible to enter "manual mode" by sending a Request instance using `client.send(..., stream=True)`.

Example in the context of forwarding the response to a streaming web endpoint with Starlette:

```
import httpx
from starlette.background import BackgroundTask
from starlette.responses import StreamingResponse

client = httpx.AsyncClient()

async def home(request):
    req = client.build_request("GET", "https://www.example.com/")
    r = await client.send(req, stream=True)
    return StreamingResponse(r.aiter_text(), background=BackgroundTask(r.aclose()))
```

### Warning

When using this "manual streaming mode", it is your duty as a developer to make sure that `Response.aclose()` is called eventually. Failing to do so would leave connections open, most likely resulting in resource leaks down the line.



## Streaming requests

When sending a streaming request body with an `AsyncClient` instance, you should use an async bytes generator instead of a bytes generator:

```
async def upload_bytes():
    ... # yield byte content

await client.post(url, content=upload_bytes())
```

## Explicit transport instances

When instantiating a transport instance directly, you need to use `httpx.AsyncHTTPTransport`.

For instance:

```
>>> import httpx
>>> transport = httpx.AsyncHTTPTransport(retries=1)
>>> async with httpx.AsyncClient(transport=transport) as client:
>>> ...
```

## Supported async environments

HTTPX supports either `asyncio` or `trio` as an async environment.

It will auto-detect which of those two to use as the backend for socket operations and concurrency primitives.

## AsyncIO

AsyncIO is Python's built-in library for writing concurrent code with the `async/await` syntax.

```
import asyncio
import httpx

async def main():
    async with httpx.AsyncClient() as client:
        response = await client.get('https://www.example.com/')
        print(response)

asyncio.run(main())
```

## Trio

Trio is an alternative async library, designed around the the principles of structured concurrency.

```
import httpx
import trio

async def main():
    async with httpx.AsyncClient() as client:
        response = await client.get('https://www.example.com/')
        print(response)

trio.run(main)
```

### Important

The `trio` package must be installed to use the Trio backend.

## AnyIO

AnyIO is an asynchronous networking and concurrency library that works on top of either `asyncio` or `trio`. It blends in with native libraries of your chosen backend (defaults to `asyncio`).

```
import httpx
import anyio

async def main():
    async with httpx.AsyncClient() as client:
        response = await client.get('https://www.example.com/')
        print(response)

anyio.run(main, backend='trio')
```

## Calling into Python Web Apps

Just as `httpx.Client` allows you to call directly into WSGI web applications, the `httpx.AsyncClient` class allows you to call directly into ASGI web applications.

Let's take this Starlette application as an example:

```
from starlette.applications import Starlette
from starlette.responses import HTMLResponse
from starlette.routing import Route

async def hello(request):
    return HTMLResponse("Hello World!")

app = Starlette(routes=[Route("/", hello)])
```

We can make requests directly against the application, like so:

```
>>> import httpx
>>> async with httpx.AsyncClient(app=app, base_url="http://testserver") as client:
...     r = await client.get("/")
...     assert r.status_code == 200
...     assert r.text == "Hello World!"
```

For some more complex cases you might need to customise the ASGI transport. This allows you to:

- Inspect 500 error responses rather than raise exceptions by setting `raise_app_exceptions=False`.
- Mount the ASGI application at a subpath by setting `root_path`.
- Use a given client address for requests by setting `client`.

For example:

```
# Instantiate a client that makes ASGI requests with a client IP of "1.2.3.4",
# on port 123.
transport = httpx.ASGITransport(app=app, client=("1.2.3.4", 123))
async with httpx.AsyncClient(transport=transport, base_url="http://testserver") as client:
    ...
```

See the ASGI documentation for more details on the `client` and `root_path` keys.

## Startup/shutdown of ASGI apps

It is not in the scope of HTTPX to trigger lifespan events of your app.

However it is suggested to use `LifespanManager` from `asgi-lifespan` in pair with `AsyncClient`.