aiohttp Documentation

Release 0.19.0-

KeepSafe

November 25, 2015

Contents

HTTP client/server for *asyncio* (**PEP 3156**).

CHAPTER 1

Features

- Supports both *HTTP Client* and *HTTP Server*.
- Supports both *Server WebSockets* and *Client WebSockets* out-of-the-box.
- Web-server has *Middlewares*, *Signals* and pluggable routing.

CHAPTER 2

Library Installation

```
$ pip install aiohttp
```

You may want to install *optional* *cchardet* library as faster replacement for *chardet*:

```
$ pip install cchardet
```

CHAPTER 3

Getting Started

Client example:

```
import asyncio
import aiohttp

async def fetch_page(client, url):
    async with client.get(url) as response:
        assert response.status == 200
        return await response.read()

loop = asyncio.get_event_loop()
client = aiohttp.ClientSession(loop=loop)
content = loop.run_until_complete(
    fetch_page(client, 'http://python.org'))
print(content)
client.close()
```

Server example:

```
import asyncio
from aiohttp import web

async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(body=text.encode('utf-8'))

async def init(loop):
    app = web.Application(loop=loop)
    app.router.add_route('GET', '/{name}', handle)

    srv = await loop.create_server(app.make_handler(),
                                   '127.0.0.1', 8080)
    print("Server started at http://127.0.0.1:8080")
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
```

CHAPTER 4

Source code

The project is hosted on GitHub

Please feel free to file an issue on the bug tracker if you have found a bug or have some suggestion in order to improve the library.

The library uses Travis for Continuous Integration.

CHAPTER 5

Dependencies

- Python Python 3.4.1+
- *chardet* library
- *Optional* *cchardet* library as faster replacement for *chardet*.

  Install it explicitly via:

```
$ pip install cchardet
```

CHAPTER 6

Discussion list

aio-libs google group: https://groups.google.com/forum/#!forum/aio-libs

Feel free to post your questions and ideas here.

CHAPTER 7

Contributing

Please read the *instructions for contributors* before making a Pull Request.

CHAPTER 8

Authors and License

The `aiohttp` package is written mostly by Nikolay Kim and Andrew Svetlov.

It's *Apache 2* licensed and freely available.

Feel free to improve this package and send a pull request to GitHub.

CHAPTER 9

Contents

9.1 HTTP Client

9.1.1 Make a Request

Begin by importing the aiohttp module:

```
import aiohttp
```

Now, let's try to get a web-page. For example let's get GitHub's public time-line

```
r = await aiohttp.get('https://api.github.com/events')
```

Now, we have a *ClientResponse* object called `r`. We can get all the information we need from this object. The mandatory parameter of *get()* coroutine is an HTTP url.

In order to make an HTTP POST request use *post()* coroutine:

```
r = await aiohttp.post('http://httpbin.org/post', data=b'data')
```

Other HTTP methods are available as well:

```
r = await aiohttp.put('http://httpbin.org/put', data=b'data')
r = await aiohttp.delete('http://httpbin.org/delete')
r = await aiohttp.head('http://httpbin.org/get')
r = await aiohttp.options('http://httpbin.org/get')
r = await aiohttp.patch('http://httpbin.org/patch', data=b'data')
```

9.1.2 Passing Parameters In URLs

You often want to send some sort of data in the URL's query string. If you were constructing the URL by hand, this data would be given as key/value pairs in the URL after a question mark, e.g. `httpbin.org/get?key=val`. Requests allows you to provide these arguments as a dictionary, using the `params` keyword argument. As an example, if you wanted to pass `key1=value1` and `key2=value2` to `httpbin.org/get`, you would use the following code:

```
payload = {'key1': 'value1', 'key2': 'value2'}
async with aiohttp.get('http://httpbin.org/get',
                       params=payload) as r:
    assert r.url == 'http://httpbin.org/get?key2=value2&key1=value1'
```

You can see that the URL has been correctly encoded by printing the URL.

It is also possible to pass a list of 2 item tuples as parameters, in that case you can specify multiple values for each key:

```
payload = [('key', 'value1'), ('key', 'value2')]
async with aiohttp.get('http://httpbin.org/get',
                       params=payload) as r:
    assert r.url == 'http://httpbin.org/get?key=value2&key=value1'
```

You can also pass `str` content as param, but beware - content is not encoded by library. Note that + is not encoded:

```
async with aiohttp.get('http://httpbin.org/get',
                        params='key=value+1') as r:
        assert r.url = 'http://httpbin.org/get?key=value+1'
```

9.1.3 Response Content

We can read the content of the server's response. Consider the GitHub time-line again:

```
r = await aiohttp.get('https://api.github.com/events')
print(await r.text())
```

will printout something like:

```
'[{"created_at":"2015-06-12T14:06:22Z","public":true,"actor":{...
```

`aiohttp` will automatically decode the content from the server. You can specify custom encoding for the *`text()`* method:

```
await r.text(encoding='windows-1251')
```

9.1.4 Binary Response Content

You can also access the response body as bytes, for non-text requests:

```
print(await r.read())
```

```
b'[{"created_at":"2015-06-12T14:06:22Z","public":true,"actor":{...
```

The `gzip` and `deflate` transfer-encodings are automatically decoded for you.

9.1.5 JSON Response Content

There's also a built-in JSON decoder, in case you're dealing with JSON data:

```
async with aiohttp.get('https://api.github.com/events') as r:
    print(await r.json())
```

In case that JSON decoding fails, *`json()`* will raise an exception. It is possible to specify custom encoding and decoder functions for the *`json()`* call.

9.1.6 Streaming Response Content

While methods *read()*, *json()* and *text()* are very convenient you should use them carefully. All these methods load the whole response in memory. For example if you want to download several gigabyte sized files, these methods will load all the data in memory. Instead you can use the *content* attribute. It is an instance of the `aiohttp.StreamReader` class. The `gzip` and `deflate` transfer-encodings are automatically decoded for you:

```
async with aiohttp.get('https://api.github.com/events') as r:
    await r.content.read(10)
```

In general, however, you should use a pattern like this to save what is being streamed to a file:

```
with open(filename, 'wb') as fd:
    while True:
        chunk = await r.content.read(chunk_size)
        if not chunk:
            break
        fd.write(chunk)
```

It is not possible to use *read()*, *json()* and *text()* after explicit reading from *content*.

9.1.7 Releasing Response

Don't forget to release response after use. This will ensure explicit behavior and proper connection pooling.

The easiest way to correctly response releasing is `async with` statement:

```
async with client.get(url) as resp:
    pass
```

But explicit *release()* call also may be used:

```
await r.release()
```

But it's not necessary if you use *read()*, *json()* and *text()* methods. They do release connection internally but better don't rely on that behavior.

9.1.8 Custom Headers

If you need to add HTTP headers to a request, pass them in a `dict` to the *headers* parameter.

For example, if you want to specify the content-type for the previous example:

```
import json
url = 'https://api.github.com/some/endpoint'
payload = {'some': 'data'}
headers = {'content-type': 'application/json'}

await aiohttp.post(url,
                   data=json.dumps(payload),
                   headers=headers)
```

9.1.9 Custom Cookies

To send your own cookies to the server, you can use the *cookies* parameter:

```
url = 'http://httpbin.org/cookies'
cookies = dict(cookies_are='working')

async with aiohttp.get(url, cookies=cookies) as r:
    assert await r.json() == {"cookies": {"cookies_are": "working"}}
```

9.1.10 More complicated POST requests

Typically, you want to send some form-encoded data — much like an HTML form. To do this, simply pass a dictionary to the *data* argument. Your dictionary of data will automatically be form-encoded when the request is made:

```
payload = {'key1': 'value1', 'key2': 'value2'}
async with aiohttp.post('http://httpbin.org/post',
                        data=payload) as r:
    print(await r.text())
```

```
{
  ...
  "form": {
    "key2": "value2",
    "key1": "value1"
  },
  ...
}
```

If you want to send data that is not form-encoded you can do it by passing a `str` instead of a `dict`. This data will be posted directly.

For example, the GitHub API v3 accepts JSON-Encoded POST/PATCH data:

```
import json
url = 'https://api.github.com/some/endpoint'
payload = {'some': 'data'}

r = await aiohttp.post(url, data=json.dumps(payload))
```

9.1.11 POST a Multipart-Encoded File

To upload Multipart-encoded files:

```
url = 'http://httpbin.org/post'
files = {'file': open('report.xls', 'rb')}

await aiohttp.post(url, data=files)
```

You can set the filename, content_type explicitly:

```
url = 'http://httpbin.org/post'
data = FormData()
data.add_field('file',
               open('report.xls', 'rb'),
               filename='report.xls',
               content_type='application/vnd.ms-excel')

await aiohttp.post(url, data=data)
```

If you pass a file object as data parameter, aiohttp will stream it to the server automatically. Check *StreamReader* for supported format information.

See also:

Working with Multipart

9.1.12 Streaming uploads

aiohttp supports multiple types of streaming uploads, which allows you to send large files without reading them into memory.

As a simple case, simply provide a file-like object for your body:

```
with open('massive-body', 'rb') as f:
   await aiohttp.post('http://some.url/streamed', data=f)
```

Or you can provide an coroutine that yields bytes objects:

```
@asyncio.coroutine
def my_coroutine():
   chunk = yield from read_some_data_from_somewhere()
   if not chunk:
      return
   yield chunk
```

Warning: `yield` expression is forbidden inside `async def`.

Note: It is not a standard coroutine as it yields values so it can not be used like `yield from my_coroutine()`. *aiohttp* internally handles such coroutines.

Also it is possible to use a *StreamReader* object. Lets say we want to upload a file from another request and calculate the file SHA1 hash:

```
async def feed_stream(resp, stream):
    h = hashlib.sha256()

    while True:
        chunk = await resp.content.readany()
        if not chunk:
            break
        h.update(chunk)
        s.feed_data(chunk)

    return h.hexdigest()

resp = aiohttp.get('http://httpbin.org/post')
stream = StreamReader()
loop.create_task(aiohttp.post('http://httpbin.org/post', data=stream))

file_hash = await feed_stream(resp, stream)
```

Because the response content attribute is a *StreamReader*, you can chain get and post requests together (aka HTTP pipelining):

```
r = await aiohttp.request('get', 'http://python.org')
await aiohttp.post('http://httpbin.org/post',
                   data=r.content)
```

9.1.13 Uploading pre-compressed data

To upload data that is already compressed before passing it to aiohttp, call the request function with `compress=False` and set the used compression algorithm name (usually deflate or zlib) as the value of the `Content-Encoding` header:

```
@asyncio.coroutine
def my_coroutine( my_data):
    data = zlib.compress(my_data)
    headers = {'Content-Encoding': 'deflate'}
    yield from aiohttp.post(
        'http://httpbin.org/post', data=data, headers=headers,
        compress=False)
```

9.1.14 Keep-Alive, connection pooling and cookie sharing

To share cookies between multiple requests you can create an `ClientSession` object:

```
session = aiohttp.ClientSession()
await session.post(
     'http://httpbin.org/cookies/set/my_cookie/my_value')
async with session.get('http://httpbin.org/cookies') as r:
    json = await r.json()
    assert json['cookies']['my_cookie'] == 'my_value'
```

You also can set default headers for all session requests:

```
session = aiohttp.ClientSession(
    headers={"Authorization": "Basic bG9naW46cGFzcw=="})
async with s.get("http://httpbin.org/headers") as r:
    json = yield from r.json()
    assert json['headers']['Authorization'] == 'Basic bG9naW46cGFzcw=='
```

By default aiohttp does not use connection pooling. In other words multiple calls to `request()` will start a new connection to host each. `ClientSession` object will do connection pooling for you.

9.1.15 Connectors

To tweak or change *transport* layer of requests you can pass a custom **Connector** to *aiohttp.request()* and family. For example:

```
conn = aiohttp.TCPConnector()
r = await aiohttp.get('http://python.org', connector=conn)
```

ClientSession constructor also accepts *connector* instance:

```
session = aiohttp.ClientSession(connector=aiohttp.TCPConnector())
```

9.1.16 Limiting connection pool size

To limit amount of simultaneously opened connection to the same endpoint (`(host, port, is_ssl)` triple) you can pass *limit* parameter to **connector**:

```
conn = aiohttp.TCPConnector(limit=30)
```

The example limits amount of parallel connections to *30*.

9.1.17 SSL control for TCP sockets

`aiohttp.connector.TCPConnector` constructor accepts mutually exclusive *verify_ssl* and *ssl_context* params.

By default it uses strict checks for HTTPS protocol. Certification checks can be relaxed by passing `verify_ssl=False`:

```
conn = aiohttp.TCPConnector(verify_ssl=False)
session = aiohttp.ClientSession(connector=conn)
r = await session.get('https://example.com')
```

If you need to setup custom ssl parameters (use own certification files for example) you can create a `ssl.SSLContext` instance and pass it into the connector:

```
sslcontext = ssl.create_default_context(cafile='/path/to/ca-bundle.crt')
conn = aiohttp.TCPConnector(ssl_context=sslcontext)
session = aiohttp.ClientSession(connector=conn)
r = await session.get('https://example.com')
```

You may also verify certificates via MD5, SHA1, or SHA256 fingerprint:

```
# Attempt to connect to https://www.python.org
# with a pin to a bogus certificate:
bad_md5 = b'\xa2\x06G\xad\xaa\xf5\xd8\\J\x99^by;\x06='
conn = aiohttp.TCPConnector(fingerprint=bad_md5)
session = aiohttp.ClientSession(connector=conn)
exc = None
try:
    r = yield from session.get('https://www.python.org')
except FingerprintMismatch as e:
    exc = e
assert exc is not None
assert exc.expected == bad_md5

# www.python.org cert's actual md5
assert exc.got == b'\xca;I\x9cuv\x8es\x138N$?\x15\xca\xcb'
```

Note that this is the fingerprint of the DER-encoded certificate. If you have the certificate in PEM format, you can convert it to DER with e.g. `openssl x509 -in crt.pem -inform PEM -outform DER > crt.der`.

Tip: to convert from a hexadecimal digest to a binary byte-string, you can use `binascii.unhexlify`:

```
md5_hex = 'ca3b499c75768e7313384e243f15cacb'
from binascii import unhexlify
assert unhexlify(md5_hex) == b'\xca;I\x9cuv\x8es\x138N$?\x15\xca\xcb'
```

9.1.18 Unix domain sockets

If your HTTP server uses UNIX domain sockets you can use `aiohttp.connector.UnixConnector`:

```
conn = aiohttp.UnixConnector(path='/path/to/socket')
r = await aiohttp.get('http://python.org', connector=conn)
```

9.1.19 Proxy support

aiohttp supports proxy. You have to use `aiohttp.connector.ProxyConnector`:

```
conn = aiohttp.ProxyConnector(proxy="http://some.proxy.com")
r = await aiohttp.get('http://python.org',
                      connector=conn)
```

`ProxyConnector` also supports proxy authorization:

```
conn = aiohttp.ProxyConnector(
   proxy="http://some.proxy.com",
   proxy_auth=aiohttp.BasicAuth('user', 'pass'))
session = aiohttp.ClientSession(connector=conn)
async with session.get('http://python.org') as r:
    assert r.status == 200
```

Authentication credentials can be passed in proxy URL:

```
conn = aiohttp.ProxyConnector(
    proxy="http://user:pass@some.proxy.com")
session = aiohttp.ClientSession(connector=conn)
async with session.get('http://python.org') as r:
    assert r.status == 200
```

9.1.20 Response Status Codes

We can check the response status code:

```
async with aiohttp.get('http://httpbin.org/get') as r:
    assert r.status == 200
```

9.1.21 Response Headers

We can view the server's response headers using a multidict:

```
>>> r.headers
{'ACCESS-CONTROL-ALLOW-ORIGIN': '*',
 'CONTENT-TYPE': 'application/json',
 'DATE': 'Tue, 15 Jul 2014 16:49:51 GMT',
 'SERVER': 'gunicorn/18.0',
 'CONTENT-LENGTH': '331',
 'CONNECTION': 'keep-alive'}
```

The dictionary is special, though: it's made just for HTTP headers. According to RFC 7230, HTTP Header names are case-insensitive. It also supports multiple values for the same key as HTTP protocol does.

So, we can access the headers using any capitalization we want:

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

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

9.1.22 Response Cookies

If a response contains some Cookies, you can quickly access them:

```
url = 'http://example.com/some/cookie/setting/url'
async with aiohttp.get(url) as r:
    print(r.cookies['example_cookie_name'])
```

Note: Response cookies contain only values, that were in `Set-Cookie` headers of the **last** request in redirection chain. To gather cookies between all redirection requests you can use *aiohttp.ClientSession* object.

9.1.23 Response History

If a request was redirected, it is possible to view previous responses using the *history* attribute:

```
>>> r = await aiohttp.get('http://example.com/some/redirect/')
>>> r
<ClientResponse(http://example.com/some/other/url/) [200]>
>>> r.history
(<ClientResponse(http://example.com/some/redirect/) [301]>,)
```

If no redirects occurred or `allow_redirects` is set to `False`, history will be an empty sequence.

9.1.24 Timeouts

You should use `asyncio.wait_for()` coroutine if you want to limit time to wait for a response from a server:

```
>>> asyncio.wait_for(aiohttp.get('http://github.com'),
...                              0.001)
Traceback (most recent call last)\:
  File "<stdin>", line 1, in <module>
asyncio.TimeoutError()
```

Or wrap your client call in `Timeout` context manager:

```
with aiohttp.Timeout(0.001):
    async with aiohttp.get('https://github.com') as r:
        await r.text()
```

Warning: *timeout* is not a time limit on the entire response download; rather, an exception is raised if the server has not issued a response for *timeout* seconds (more precisely, if no bytes have been received on the underlying socket for *timeout* seconds).

9.2 HTTP Client Reference

9.2.1 Client Session

Client session is the recommended interface for making HTTP requests.

Session encapsulates *connection pool* (*connector* instance) and supports keepalives by default.

Usage example:

```
import aiohttp
import asyncio

async def fetch(client):
    async with client.get('http://python.org') as resp:
        assert resp.status == 200
        print(await resp.text())

with aiohttp.ClientSession() as client:
    asyncio.get_event_loop().run_until_complete(fetch(client))
```

New in version 0.17.

The client session supports context manager protocol for self closing.

class `aiohttp.`**`ClientSession`**(*, *connector=None*, *loop=None*, *cookies=None*, *headers=None*, *skip_auto_headers=None*, *auth=None*, *request_class=ClientRequest*, *response_class=ClientResponse*, *ws_response_class=ClientWebSocketResponse*)

The class for creating client sessions and making requests.

Parameters

- **`connector`** (*aiohttp.connector.BaseConnector*) – BaseConnector sub-class instance to support connection pooling.
- **`loop`** – event loop used for processing HTTP requests.

  If *loop* is `None` the constructor borrows it from *connector* if specified.

  `asyncio.get_event_loop()` is used for getting default event loop otherwise.
- **`cookies`** (*dict*) – Cookies to send with the request (optional)
- **`headers`** – HTTP Headers to send with the request (optional).

  May be either *iterable of key-value pairs* or `Mapping` (e.g. `dict`, *`CIMultiDict`*).
- **`skip_auto_headers`** – set of headers for which autogeneration should be skipped.

  aiohttp autogenerates headers like `User-Agent` or `Content-Type` if these headers are not explicitly passed. Using `skip_auto_headers` parameter allows to skip that generation. Note that `Content-Length` autogeneration can't be skipped.

  Iterable of `str` or *`upstr`* (optional)
- **`auth`** (aiohttp.BasicAuth) – an object that represents HTTP Basic Authorization (optional)
- **`request_class`** – Request class implementation. `ClientRequest` by default.
- **`response_class`** – Response class implementation. *`ClientResponse`* by default.

- **ws_response_class** – WebSocketResponse class implementation. `ClientWebSocketResponse` by default.

  New in version 0.16.

Changed in version 0.16: *request_class* default changed from `None` to `ClientRequest`

Changed in version 0.16: *response_class* default changed from `None` to *ClientResponse*

closed
`True` if the session has been closed, `False` otherwise.

A read-only property.

connector
`aiohttp.connector.BaseConnector` derived instance used for the session.

A read-only property.

cookies
The session cookies, `http.cookies.SimpleCookie` instance.

A read-only property. Overriding *session.cookies = new_val* is forbidden, but you may modify the object in-place if needed.

coroutine request(*method*, *url*, *, *params=None*, *data=None*, *headers=None*, *skip_auto_headers=None*, *auth=None*, *allow_redirects=True*, *max_redirects=10*, *encoding='utf-8'*, *version=HttpVersion(major=1, minor=1)*, *compress=None*, *chunked=None*, *expect100=False*, *read_until_eof=True*)
Performs an asynchronous HTTP request. Returns a response object.

Parameters

- **method** (*str*) – HTTP method
- **url** (*str*) – Request URL
- **params** – Mapping, iterable of tuple of *key*/*value* pairs or string to be sent as parameters in the query string of the new request (optional)

  Allowed values are:

  – `collections.abc.Mapping` e.g. `dict`, *aiohttp.MultiDict* or *aiohttp.MultiDictProxy*

  – `collections.abc.Iterable` e.g. `tuple` or `list`

  – `str` with preferably url-encoded content (**Warning:** content will not be encoded by *aiohttp*)

- **data** – Dictionary, bytes, or file-like object to send in the body of the request (optional)
- **headers** (*dict*) – HTTP Headers to send with the request (optional)
- **skip_auto_headers** – set of headers for which autogeneration should be skipped.

  aiohttp autogenerates headers like `User-Agent` or `Content-Type` if these headers are not explicitly passed. Using `skip_auto_headers` parameter allows to skip that generation.

  Iterable of `str` or *upstr* (optional)

- **auth** (aiohttp.BasicAuth) – an object that represents HTTP Basic Authorization (optional)
- **allow_redirects** (*bool*) – If set to `False`, do not follow redirects. `True` by default (optional).

- **version** (aiohttp.protocol.HttpVersion) – Request HTTP version (optional)
- **compress** (*bool*) – Set to `True` if request has to be compressed with deflate encoding. `None` by default (optional).
- **chunked** (*int*) – Set to chunk size for chunked transfer encoding. `None` by default (optional).
- **expect100** (*bool*) – Expect 100-continue response from server. `False` by default (optional).
- **read_until_eof** (*bool*) – Read response until EOF if response does not have Content-Length header. `True` by default (optional).

Return ClientResponse a *client response* object.

coroutine get(*url*, *, *allow_redirects=True*, ***kwargs*)

Perform a `GET` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters

- **url** (*str*) – Request URL
- **allow_redirects** (*bool*) – If set to `False`, do not follow redirects. `True` by default (optional).

Return ClientResponse a *client response* object.

coroutine post(*url*, *, *data=None*, ***kwargs*)

Perform a `POST` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters

- **url** (*str*) – Request URL
- **data** – Dictionary, bytes, or file-like object to send in the body of the request (optional)

Return ClientResponse a *client response* object.

coroutine put(*url*, *, *data=None*, ***kwargs*)

Perform a `PUT` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters

- **url** (*str*) – Request URL
- **data** – Dictionary, bytes, or file-like object to send in the body of the request (optional)

Return ClientResponse a *client response* object.

coroutine delete(*url*, ***kwargs*)

Perform a `DELETE` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters **url** (*str*) – Request URL

Return ClientResponse a *client response* object.

coroutine head(*url*, *, *allow_redirects=False*, ***kwargs*)

Perform a `HEAD` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters

- **`url`** (*str*) – Request URL
- **`allow_redirects`** (*bool*) – If set to `False`, do not follow redirects. `False` by default (optional).

Return ClientResponse a *client response* object.

coroutine `options`(*url*, *, *allow_redirects=True*, **kwargs*)
Perform an `OPTIONS` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters

- **`url`** (*str*) – Request URL
- **`allow_redirects`** (*bool*) – If set to `False`, do not follow redirects. `True` by default (optional).

Return ClientResponse a *client response* object.

coroutine `patch`(*url*, *, *data=None*, **kwargs*)
Perform a `PATCH` request.

In order to modify inner `request` parameters, provide *kwargs*.

Parameters

- **`url`** (*str*) – Request URL
- **`data`** – Dictionary, bytes, or file-like object to send in the body of the request (optional)

Return ClientResponse a *client response* object.

coroutine `ws_connect`(*url*, *, *protocols=()*, *timeout=10.0*, *auth=None*, *autoclose=True*, *autoping=True*, *origin=None*)
Create a websocket connection. Returns a *ClientWebSocketResponse* object.

Parameters

- **`url`** (*str*) – Websocket server url
- **`protocols`** (*tuple*) – Websocket protocols
- **`timeout`** (*float*) – Timeout for websocket read. 10 seconds by default
- **`auth`** (aiohttp.BasicAuth) – an object that represents HTTP Basic Authorization (optional)
- **`autoclose`** (*bool*) – Automatically close websocket connection on close message from server. If *autoclose* is False them close procedure has to be handled manually
- **`autoping`** (*bool*) – automatically send *pong* on *ping* message from server
- **`origin`** (*str*) – Origin header to send to server

New in version 0.16: Add *ws_connect()*.

New in version 0.18: Add *auth* parameter.

New in version 0.19: Add *origin* parameter.

close()
Close underlying connector.

Release all acquired resources.

detach()
Detach connector from session without closing the former.

Session is switched to closed state anyway.

9.2.2 Basic API

While we encourage `ClientSession` usage we also provide simple coroutines for making HTTP requests.

Basic API is good for performing simple HTTP requests without keepaliving, cookies and complex connection stuff like properly configured SSL certification chaining.

coroutine aiohttp.**request**(*method*, *url*, ***, *params=None*, *data=None*, *headers=None*, *cookies=None*, *auth=None*, *allow_redirects=True*, *max_redirects=10*, *encoding='utf-8'*, *version=HttpVersion(major=1, minor=1)*, *compress=None*, *chunked=None*, *expect100=False*, *connector=None*, *loop=None*, *read_until_eof=True*, *request_class=None*, *response_class=None*)
Perform an asynchronous HTTP request. Return a response object (`ClientResponse` or derived from).

Parameters

- **method** (*str*) – HTTP method
- **url** (*str*) – Requested URL
- **params** (*dict*) – Parameters to be sent in the query string of the new request (optional)
- **data** – Dictionary, bytes, or file-like object to send in the body of the request (optional)
- **headers** (*dict*) – HTTP Headers to send with the request (optional)
- **cookies** (*dict*) – Cookies to send with the request (optional)
- **auth** (aiohttp.BasicAuth) – an object that represents HTTP Basic Authorization (optional)
- **allow_redirects** (*bool*) – If set to `False`, do not follow redirects. `True` by default (optional).
- **version** (aiohttp.protocol.HttpVersion) – Request HTTP version (optional)
- **compress** (*bool*) – Set to `True` if request has to be compressed with deflate encoding. `False` instructs aiohttp to not compress data even if the Content-Encoding header is set. Use it when sending pre-compressed data. `None` by default (optional).
- **chunked** (*int*) – Set to chunk size for chunked transfer encoding. `None` by default (optional).
- **expect100** (*bool*) – Expect 100-continue response from server. `False` by default (optional).
- **connector** (*aiohttp.connector.BaseConnector*) – BaseConnector sub-class instance to support connection pooling.
- **read_until_eof** (*bool*) – Read response until EOF if response does not have Content-Length header. `True` by default (optional).
- **request_class** – Custom Request class implementation (optional)
- **response_class** – Custom Response class implementation (optional)

- **loop** – event loop used for processing HTTP requests. If param is `None`, `asyncio.get_event_loop()` is used for getting default event loop, but we strongly recommend to use explicit loops everywhere. (optional)

Return ClientResponse a *client response* object.

Usage:

```
import aiohttp

async def fetch():
    async with aiohttp.request('GET', 'http://python.org/') as resp:
        assert resp.status == 200
        print(await resp.text())
```

coroutine `aiohttp.`**`get`**(*url*, ***kwargs*)
Perform a GET request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

coroutine `aiohttp.`**`options`**(*url*, ***kwargs*)
Perform a OPTIONS request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

coroutine `aiohttp.`**`head`**(*url*, ***kwargs*)
Perform a HEAD request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

coroutine `aiohttp.`**`delete`**(*url*, ***kwargs*)
Perform a DELETE request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

coroutine `aiohttp.`**`post`**(*url*, *, *data=None*, ***kwargs*)
Perform a POST request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

coroutine aiohttp.**put**(*url*, *, *data=None*, **kwargs*)
Perform a PUT request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

coroutine aiohttp.**patch**(*url*, *, *data=None*, **kwargs*)
Perform a PATCH request.

Parameters

- **url** (*str*) – Requested URL.
- ****kwargs** – Optional arguments that *request()* takes.

Returns *ClientResponse* or derived from

9.2.3 Connectors

Connectors are transports for aiohttp client API.

There are standard connectors:

1. *TCPConnector* for regular *TCP sockets* (both *HTTP* and *HTTPS* schemes supported).
2. *ProxyConnector* for connecting via HTTP proxy.
3. *UnixConnector* for connecting via UNIX socket (it's used mostly for testing purposes).

All connector classes should be derived from *BaseConnector*.

By default all *connectors* except *ProxyConnector* support *keep-alive connections* (behavior is controlled by *force_close* constructor's parameter).

BaseConnector

class aiohttp.**BaseConnector**(*, *conn_timeout=None*, *keepalive_timeout=30*, *limit=None*, *share_cookies=False*, *force_close=False*, *loop=None*)
Base class for all connectors.

Parameters

- **conn_timeout** (*float*) – timeout for connection establishing (optional). Values `0` or `None` mean no timeout.
- **keepalive_timeout** (*float*) – timeout for connection reusing after releasing (optional). Values `0` or `None` mean no timeout.
- **limit** (*int*) – limit for simultaneous connections to the same endpoint. Endpoints are the same if they are have equal `(host, port, is_ssl)` triple. If *limit* is `None` the connector has no limit.
- **share_cookies** (*bool*) – update `cookies` on connection processing (optional, deprecated).
- **force_close** (*bool*) – do close underlying sockets after connection releasing (optional).

- **loop** – event loop used for handling connections. If param is `None`, `asyncio.get_event_loop()` is used for getting default event loop, but we strongly recommend to use explicit loops everywhere. (optional)

Deprecated since version 0.15.2: *share_cookies* parameter is deprecated, use `ClientSession` for handling cookies for client connections.

closed
Read-only property, `True` if connector is closed.

force_close
Read-only property, `True` if connector should ultimately close connections on releasing.

New in version 0.16.

limit
The limit for simultaneous connections to the same endpoint.

Endpoints are the same if they are have equal `(host, port, is_ssl)` triple.

If *limit* is `None` the connector has no limit (default).

Read-only property.

New in version 0.16.

close()
Close all opened connections.

coroutine connect(*request*)
Get a free connection from pool or create new one if connection is absent in the pool.

The call may be paused if *limit* is exhausted until used connections returns to pool.

Parameters **request** (*aiohttp.client.ClientRequest*) – request object which is connection initiator.

Returns *Connection* object.

coroutine _create_connection(*req*)
Abstract method for actual connection establishing, should be overridden in subclasses.

TCPConnector

class aiohttp.**TCPConnector**(*, *verify_ssl=True*, *fingerprint=None*, *use_dns_cache=False*, *family=0*, *ssl_context=None*, *conn_timeout=None*, *keepalive_timeout=30*, *limit=None*, *share_cookies=False*, *force_close=False*, *loop=None*)
Connector for working with *HTTP* and *HTTPS* via *TCP* sockets.

The most common transport. When you don't know what connector type to use, use a *TCPConnector* instance.

TCPConnector inherits from *BaseConnector*.

Constructor accepts all parameters suitable for *BaseConnector* plus several TCP-specific ones:

Parameters

- **verify_ssl** (*bool*) – Perform SSL certificate validation for *HTTPS* requests (enabled by default). May be disabled to skip validation for sites with invalid certificates.
- **fingerprint** (*bytes*) – Pass the binary MD5, SHA1, or SHA256 digest of the expected certificate in DER format to verify that the certificate the server presents matches. Useful for certificate pinning.

New in version 0.16.

- **use_dns_cache** (*bool*) – use internal cache for DNS lookups, `False` by default.

  Enabling an option *may* speedup connection establishing a bit but may introduce some *side effects* also.

  New in version 0.17.

- **resolve** (*bool*) – alias for *use_dns_cache* parameter.

  Deprecated since version 0.17.

- **family** (*int*) –

  TCP socket family, both IPv4 and IPv6 by default. For *IPv4* only use `socket.AF_INET`, for *IPv6* only – `socket.AF_INET6`.

  Changed in version 0.18: *family* is *0* by default, that means both IPv4 and IPv6 are accepted. To specify only concrete version please pass `socket.AF_INET` or `socket.AF_INET6` explicitly.

- **ssl_context** (*ssl.SSLContext*) – ssl context used for processing *HTTPS* requests (optional).

  ssl_context may be used for configuring certification authority channel, supported SSL options etc.

verify_ssl
Check *ssl certifications* if `True`.

Read-only `bool` property.

ssl_context
`ssl.SSLContext` instance for *https* requests, read-only property.

family
TCP socket family e.g. `socket.AF_INET` or `socket.AF_INET6`

Read-only property.

dns_cache
Use quick lookup in internal *DNS* cache for host names if `True`.

Read-only `bool` property.

New in version 0.17.

resolve
Alias for *dns_cache*.

Deprecated since version 0.17.

cached_hosts
The cache of resolved hosts if *dns_cache* is enabled.

Read-only `types.MappingProxyType` property.

New in version 0.17.

resolved_hosts
Alias for *cached_hosts*

Deprecated since version 0.17.

fingerprint
MD5, SHA1, or SHA256 hash of the expected certificate in DER format, or `None` if no certificate fingerprint check required.

Read-only `bytes` property.

New in version 0.16.

clear_dns_cache(*self*, *host=None*, *port=None*)
Clear internal *DNS* cache.

Remove specific entry if both *host* and *port* are specified, clear all cache otherwise.

New in version 0.17.

clear_resolved_hosts(*self*, *host=None*, *port=None*)
Alias for *clear_dns_cache()*.

Deprecated since version 0.17.

ProxyConnector

class aiohttp.**ProxyConnector**(*proxy*, ***, *proxy_auth=None*, *conn_timeout=None*, *keepalive_timeout=30*, *limit=None*, *share_cookies=False*, *force_close=True*, *loop=None*)
HTTP Proxy connector.

Use *ProxyConnector* for sending *HTTP/HTTPS* requests through *HTTP proxy*.

ProxyConnector is inherited from *TCPConnector*.

Usage:

```
conn == ProxyConnector(proxy="http://some.proxy.com")
session = ClientSession(connector=conn)
async with session.get('http://python.org') as resp:
    assert resp.status == 200
```

Constructor accepts all parameters suitable for *TCPConnector* plus several proxy-specific ones:

Parameters

- **proxy** (*str*) – URL for proxy, e.g. `"http://some.proxy.com"`.
- **proxy_auth** (aiohttp.BasicAuth) – basic authentication info used for proxies with authorization.

Note: *ProxyConnector* in opposite to all other connectors **doesn't** support *keep-alives* by default (`force_close` is `True`).

Changed in version 0.16: *force_close* parameter changed to `True` by default.

proxy
Proxy *URL*, read-only `str` property.

proxy_auth
Proxy authentication info, read-only *BasicAuth* property or `None` for proxy without authentication.

New in version 0.16.

UnixConnector

class aiohttp.**UnixConnector**(*path*, ***, *conn_timeout=None*, *keepalive_timeout=30*, *limit=None*, *share_cookies=False*, *force_close=False*, *loop=None*)

Unix socket connector.

Use *ProxyConnector* for sending *HTTP/HTTPS* requests through *UNIX Sockets* as underlying transport.

UNIX sockets are handy for writing tests and making very fast connections between processes on the same host.

UnixConnector is inherited from *BaseConnector*.

Usage:

```
conn = UnixConnector(path='/path/to/socket')
session = ClientSession(connector=conn)
async with session.get('http://python.org') as resp:
    ...
```

Constructor accepts all parameters suitable for *BaseConnector* plus UNIX-specific one:

Parameters **path** (*str*) – Unix socket path

path

Path to *UNIX socket*, read-only str property.

Connection

class aiohttp.**Connection**

Encapsulates single connection in connector object.

End user should never create *Connection* instances manually but get it by *BaseConnector.connect()* coroutine.

closed

bool read-only property, True if connection was closed, released or detached.

loop

Event loop used for connection

close()

Close connection with forcibly closing underlying socket.

release()

Release connection back to connector.

Underlying socket is not closed, the connection may be reused later if timeout (30 seconds by default) for connection was not expired.

detach()

Detach underlying socket from connection.

Underlying socket is not closed, next *close()* or *release()* calls don't return socket to free pool.

9.2.4 Response object

class aiohttp.**ClientResponse**

Client response returned be *ClientSession.request()* and family.

User never creates the instance of ClientResponse class but gets it from API calls.

ClientResponse supports async context manager protocol, e.g.:

```
resp = await client_session.get(url)
async with resp:
    assert resp.status == 200
```

After exiting from `async with` block response object will be *released* (see *release()* coroutine).

New in version 0.18: Support for `async with`.

version
Response's version, `HttpVersion` instance.

status
HTTP status code of response (`int`), e.g. `200`.

reason
HTTP status reason of response (`str`), e.g. `"OK"`.

connection
Connection used for handling response.

content
Payload stream, contains response's BODY (`StreamReader` compatible instance, most likely `FlowControlStreamReader` one).

cookies
HTTP cookies of response (*Set-Cookie* HTTP header, `SimpleCookie`).

headers
HTTP headers of response, *CIMultiDictProxy*.

history
A `Sequence` of *ClientResponse* objects of preceding requests if there were redirects, an empty sequence otherwise.

close()
Close response and underlying connection.

For *keep-alive* support see *release()*.

coroutine **read**()
Read the whole response's body as `bytes`.

Close underlying connection if data reading gets an error, release connection otherwise.

Return bytes read *BODY*.

See also:

close(), *release()*.

coroutine **release**()
Finish response processing, release underlying connection and return it into free connection pool for re-usage by next upcoming request.

coroutine **text**(*encoding=None*)
Read response's body and return decoded `str` using specified *encoding* parameter.

If *encoding* is `None` content encoding is autocalculated using *cchardet* or *chardet* as fallback if *cchardet* is not available.

Close underlying connection if data reading gets an error, release connection otherwise.

Parameters **encoding** (*str*) – text encoding used for *BODY* decoding, or `None` for encoding autodetection (default).

Return str decoded *BODY*

coroutine **json**(*encoding=None*, *loads=json.loads*)

Read response's body as *JSON*, return `dict` using specified *encoding* and *loader*.

If *encoding* is `None` content encoding is autocalculated using *cchardet* or *chardet* as fallback if *cchardet* is not available.

Close underlying connection if data reading gets an error, release connection otherwise.

Parameters

- **encoding** (*str*) – text encoding used for *BODY* decoding, or `None` for encoding autodetection (default).
- **loads** (*callable*) – `callable()` used for loading *JSON* data, `json.loads()` by default.

Returns *BODY* as *JSON* data parsed by *loads* parameter or `None` if *BODY* is empty or contains white-spaces only.

9.2.5 Utilities

BasicAuth

class `aiohttp.`**BasicAuth**(*login*, *password=''*, *encoding='latin1'*)

HTTP basic authentication helper.

Parameters

- **login** (*str*) – login
- **password** (*str*) – password
- **encoding** (*str*) – encoding (*'latin1'* by default)

Should be used for specifying authorization data in client API, e.g. *auth* parameter for *`ClientSession.request()`*.

encode()

Encode credentials into string suitable for `Authorization` header etc.

Returns encoded authentication data, `str`.

9.3 WebSockets Client

New in version 0.15.

`aiohttp` works with client websockets out-of-the-box.

You have to use the *`aiohttp.ClientSession.ws_connect()`* coroutine for client websocket connection. It accepts a *url* as a first parameter and returns *`ClientWebSocketResponse`*, with that object you can communicate with websocket server using response's methods:

```
session = aiohttp.ClientSession()
async with session.ws_connect('http://example.org/websocket') as ws:

    async for msg in ws:
        if msg.tp == aiohttp.MsgType.text:
            if msg.data == 'close cmd':
                await ws.close()
                break
            else:
                ws.send_str(msg.data + '/answer')
        elif msg.tp == aiohttp.MsgType.closed:
            break
        elif msg.tp == aiohttp.MsgType.error:
            break
```

If you prefer to establish *websocket client connection* without explicit *ClientSession* instance please use *ws_connect()*:

```
async with aiohttp.ws_connect('http://example.org/websocket') as ws:
    ...
```

You **must** use the only websocket task for both reading (e.g `await ws.receive()` or `async for msg in ws:`) and writing but may have multiple writer tasks which can only send data asynchronously (by `ws.send_str('data')` for example).

9.3.1 ws_connect

To connect to a websocket server you have to use the *aiohttp.ws_connect()* or *aiohttp.ClientSession.ws_connect()* coroutines, do not create an instance of class *ClientWebSocketResponse* manually.

coroutine aiohttp.**ws_connect**(*url*, ***, *protocols=()*, *timeout=10.0*, *connector=None*, *auth=None*, *ws_response_class=ClientWebSocketResponse*, *autoclose=True*, *autoping=True*, *loop=None*, *origin=None*)

This function creates a websocket connection, checks the response and returns a *ClientWebSocketResponse* object. In case of failure it may raise a *WSServerHandshakeError* exception.

Parameters

- **url** (*str*) – Websocket server url
- **protocols** (*tuple*) – Websocket protocols
- **timeout** (*float*) – Timeout for websocket read. 10 seconds by default
- **connector** (*obj*) – object *TCPConnector*
- **ws_response_class** – WebSocketResponse class implementation. `ClientWebSocketResponse` by default.

  New in version 0.16.
- **autoclose** (*bool*) – Automatically close websocket connection on close message from server. If *autoclose* is False them close procedure has to be handled manually
- **autoping** (*bool*) – Automatically send *pong* on *ping* message from server
- **auth** (*aiohttp.helpers.BasicAuth*) – BasicAuth named tuple that represents HTTP Basic Authorization (optional)

- **loop** – event loop used for processing HTTP requests.

  If param is `None` `asyncio.get_event_loop()` used for getting default event loop, but we strongly recommend to use explicit loops everywhere.

- **origin** (*str*) – Origin header to send to server

New in version 0.18: Add *auth* parameter.

New in version 0.19: Add *origin* parameter.

9.3.2 ClientWebSocketResponse

class `aiohttp.`**ClientWebSocketResponse**

Class for handling client-side websockets.

closed

Read-only property, `True` if *`close()`* has been called of `MSG_CLOSE` message has been received from peer.

protocol

Websocket *subprotocol* chosen after `start()` call.

May be `None` if server and client protocols are not overlapping.

exception()

Returns exception if any occurs or returns None.

ping(*message=b''*)

Send `MSG_PING` to peer.

Parameters **message** – optional payload of *ping* message, `str` (converted to *UTF-8* encoded bytes) or `bytes`.

send_str(*data*)

Send *data* to peer as `MSG_TEXT` message.

Parameters **data** (*str*) – data to send.

Raises **TypeError** if data is not `str`

send_bytes(*data*)

Send *data* to peer as `MSG_BINARY` message.

Parameters **data** – data to send.

Raises **TypeError** if data is not `bytes`, `bytearray` or `memoryview`.

coroutine close(*, *code=1000*, *message=b''*)

A coroutine that initiates closing handshake by sending `MSG_CLOSE` message. It waits for close response from server. It add timeout to *close()* call just wrap call with *asyncio.wait()* or *asyncio.wait_for()*.

Parameters

- **code** (*int*) – closing code
- **message** – optional payload of *pong* message, `str` (converted to *UTF-8* encoded bytes) or `bytes`.

coroutine receive()

A coroutine that waits upcoming *data* message from peer and returns it.

The coroutine implicitly handles `MSG_PING`, `MSG_PONG` and `MSG_CLOSE` without returning the message.

It process *ping-pong game* and performs *closing handshake* internally.

Returns *Message*, *tp* is types of *~aiohttp.MsgType*

9.4 HTTP Server Usage

Changed in version 0.12: The module was deeply refactored which makes it backward incompatible.

9.4.1 Run a simple web server

In order to implement a web server, first create a *request handler*.

Handler is a coroutine or a regular function that accepts only *request* parameters of type *Request* and returns *Response* instance:

```
import asyncio
from aiohttp import web

async def hello(request):
    return web.Response(body=b"Hello, world")
```

Next, you have to create a *Application* instance and register *handler* in the application's router pointing *HTTP method*, *path* and *handler*:

```
app = web.Application()
app.router.add_route('GET', '/', hello)
```

After that, create a server and run the *asyncio loop* as usual:

```
loop = asyncio.get_event_loop()
handler = app.make_handler()
f = loop.create_server(handler, '0.0.0.0', 8080)
srv = loop.run_until_complete(f)
print('serving on', srv.sockets[0].getsockname())
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    loop.run_until_complete(handler.finish_connections(1.0))
    srv.close()
    loop.run_until_complete(srv.wait_closed())
    loop.run_until_complete(app.finish())
loop.close()
```

That's it.

9.4.2 Handler

Handler is an any *callable* that accepts a single *Request* argument and returns a *StreamResponse* derived (e.g. *Response*) instance.

Handler **may** be a coroutine, *aiohttp.web* will **unyield** returned result by applying `await` to the handler.

Handlers are connected to the *Application* via routes:

```
handler = Handler()
app.router.add_route('GET', '/', handler)
```

Variable routes

You can also use *variable routes*. If route contains strings like `'/a/{name}/c'` that means the route matches to the path like `'/a/b/c'` or `'/a/1/c'`.

Parsed *path part* will be available in the *request handler* as `request.match_info['name']`:

```
async def variable_handler(request):
    return web.Response(
        text="Hello, {}".format(request.match_info['name']))

app.router.add_route('GET', '/{name}', variable_handler)
```

You can also specify regex for variable route in the form `{name:regex}`:

```
app.router.add_route('GET', r'/{name:\d+}', variable_handler)
```

By default regex is `[^{}/]+`.

New in version 0.13: Support for custom regexs in variable routes.

Named routes and url reverse constructing

Routes may have a *name*:

```
app.router.add_route('GET', '/root', handler, name='root')
```

In web-handler you may build *URL* for that route:

```
>>> request.app.router['root'].url(query={"a": "b", "c": "d"})
'/root?a=b&c=d'
```

More interesting example is building *URL* for *variable router*:

```
app.router.add_route('GET', r'/{user}/info',
                     variable_handler, name='handler')
```

In this case you can pass route parts also:

```
>>> request.app.router['handler'].url(
...     parts={'user': 'john_doe'},
...     query="?a=b")
'/john_doe/info?a=b'
```

Using plain coroutines and classes for web-handlers

Handlers *may* be first-class functions, e.g.:

```
async def hello(request):
    return web.Response(body=b"Hello, world")

app.router.add_route('GET', '/', hello)
```

But sometimes you would like to group logically coupled handlers into a python class.

aiohttp.web doesn't dictate any implementation details, so application developer can use classes if he wants:

```
class Handler:

    def __init__(self):
        pass

    def handle_intro(self, request):
        return web.Response(body=b"Hello, world")

    async def handle_greeting(self, request):
        name = request.match_info.get('name', "Anonymous")
        txt = "Hello, {}".format(name)
        return web.Response(text=txt)

handler = Handler()
app.router.add_route('GET', '/intro', handler.handle_intro)
app.router.add_route('GET', '/greet/{name}', handler.handle_greeting)
```

New in version 0.15.2: *UrlDispatcher.add_route()* supports wildcard as *HTTP method*:

```
app.router.add_route('*', '/path', handler)
```

That means the handler for `'/path'` is applied for every HTTP method.

Route views

New in version 0.18.

For look on *all* routes in the router you may use *UrlDispatcher.routes()* method.

You can iterate over routes in the router table:

```
for route in app.router.routes():
    print(route)
```

or get router table size:

```
len(app.router.routes())
```

9.4.3 Custom conditions for routes lookup

Sometimes you need to distinguish *web-handlers* on more complex criteria than *HTTP method* and *path*.

While *UrlDispatcher* doesn't accept extra criterias there is an easy way to do the task by implementing the second routing layer by hand.

The next example shows custom processing based on *HTTP Accept* header:

```
class AcceptChooser:

    def __init__(self):
        self._accepts = {}

    async def do_route(self, request):
        for accept in request.headers.getall('ACCEPT', []):
            acceptor = self._accepts.get(accept)
```

```
            if acceptor is not None:
                return (await acceptor(request))
        raise HTTPNotAcceptable()

    def reg_acceptor(self, accept, handler):
        self._accepts[accept] = handler


async def handle_json(request):
    # do json handling

async def handle_xml(request):
    # do xml handling

chooser = AcceptChooser()
app.router.add_route('GET', '/', chooser.do_route)

chooser.reg_acceptor('application/json', handle_json)
chooser.reg_acceptor('application/xml', handle_xml)
```

9.4.4 Template rendering

aiohttp.web has no support for template rendering out-of-the-box.

But there is third-party library `aiohttp_jinja2` which is supported by *aiohttp* authors.

The usage is simple: create dictionary with data and pass it into template renderer.

Before template rendering you have to setup *jinja2 environment* first (`aiohttp_jinja2.setup()` call):

```
app = web.Application(loop=self.loop)
aiohttp_jinja2.setup(app,
    loader=jinja2.FileSystemLoader('/path/to/templates/folder'))
```

After that you may use template engine in your *web-handlers*. The most convenient way is to use `aiohttp_jinja2.template()` decorator:

```
@aiohttp_jinja2.template('tmpl.jinja2')
def handler(request):
    return {'name': 'Andrew', 'surname': 'Svetlov'}
```

If you prefer Mako template engine please take a look on aiohttp_mako library.

9.4.5 User sessions

Often you need a container for storing per-user data. The concept is usually called *session*.

aiohttp.web has no *sessions* but there is third-party `aiohttp_session` library for that:

```
import asyncio
import time
from aiohttp import web
from aiohttp_session import get_session, session_middleware
from aiohttp_session.cookie_storage import EncryptedCookieStorage

async def handler(request):
    session = await get_session(request)
```

```
    session['last_visit'] = time.time()
    return web.Response(body=b'OK')

async def init(loop):
    app = web.Application(middlewares=[session_middleware(
        EncryptedCookieStorage(b'Sixteen byte key'))])
    app.router.add_route('GET', '/', handler)
    srv = await loop.create_server(
        app.make_handler(), '0.0.0.0', 8080)
    return srv

loop = asyncio.get_event_loop()
loop.run_until_complete(init(loop))
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
```

9.4.6 *Expect* header support

New in version 0.15.

aiohttp.web supports *Expect* header. By default it responds with an *HTTP/1.1 100 Continue* status code. It is possible to specify custom *Expect* header handler on per route basis. This handler gets called after receiving all headers and before processing application middlewares *Middlewares* and route handler. Handler can return *None*, in that case the request processing continues as usual. If handler returns an instance of class *StreamResponse*, *request handler* uses it as response. Custom handler *must* write *HTTP/1.1 100 Continue* status if all checks pass.

This example shows custom handler for *Except* header:

```
async def check_auth(request):
    if request.version != aiohttp.HttpVersion11:
        return

    if request.headers.get('AUTHORIZATION') is None:
        return web.HTTPForbidden()

    request.transport.write(b"HTTP/1.1 100 Continue\r\n\r\n")

async def hello(request):
    return web.Response(body=b"Hello, world")

app = web.Application()
app.router.add_route('GET', '/', hello, expect_handler=check_auth)
```

9.4.7 File Uploads

There are two steps necessary for handling file uploads. The first is to make sure that you have a form that has been setup correctly to accept files. This means adding the *enctype* attribute to your form element with the value of *multipart/form-data*. A very simple example would be a form that accepts a mp3 file. Notice, we have set up the form as previously explained and also added the *input* element of the *file* type:

```
<form action="/store_mp3" method="post" accept-charset="utf-8"
      enctype="multipart/form-data">

    <label for="mp3">Mp3</label>
```

```
    <input id="mp3" name="mp3" type="file" value="" />

    <input type="submit" value="submit" />
</form>
```

The second step is handling the file upload in your *request handler* (here assumed to answer on */store_mp3*). The uploaded file is added to the request object as a *FileField* object accessible through the *Request.post()* coroutine. The two properties we are interested in are *file* and *filename* and we will use those to read a file's name and a content:

```
async def store_mp3_view(request):

    data = await request.post()

    # filename contains the name of the file in string format.
    filename = data['mp3'].filename

    # input_file contains the actual file data which needs to be
    # stored somewhere.

    input_file = data['mp3'].file

    content = input_file.read()

    return web.Response(body=content,
                        headers=MultiDict(
                            {'CONTENT-DISPOSITION': input_file})
```

9.4.8 WebSockets

New in version 0.14.

aiohttp.web works with websockets out-of-the-box.

You have to create *WebSocketResponse* in *web-handler* and communicate with peer using response's methods:

```
async def websocket_handler(request):

    ws = web.WebSocketResponse()
    await ws.prepare(request)

    async for msg in ws:
        if msg.tp == aiohttp.MsgType.text:
            if msg.data == 'close':
                await ws.close()
            else:
                ws.send_str(msg.data + '/answer')
        elif msg.tp == aiohttp.MsgType.error:
            print('ws connection closed with exception %s' %
                  ws.exception())

    print('websocket connection closed')

    return ws
```

You **must** use the only websocket task for both reading (e.g `await ws.receive()`) and writing but may have multiple writer tasks which can only send data asynchronously (by `ws.send_str('data')` for example).

Note: While *aiohttp.web* itself supports websockets only without downgrading to LONG-POLLING etc. our team supports SockJS aiohttp-based library for implementing SockJS-compatible server code.

9.4.9 Exceptions

aiohttp.web defines exceptions for list of *HTTP status codes*.

Each class relates to a single HTTP status code. Each class is a subclass of the `HTTPException`.

Those exceptions are derived from *Response* too, so you can either return exception object from *Handler* or raise it.

The following snippets are the same:

```
async def handler(request):
    return aiohttp.web.HTTPFound('/redirect')
```

and:

```
async def handler(request):
    raise aiohttp.web.HTTPFound('/redirect')
```

Each exception class has a status code according to **RFC 2068**: codes with 100-300 are not really errors; 400s are client errors, and 500s are server errors.

HTTP Exception hierarchy chart:

```
Exception
  HTTPException
    HTTPSuccessful
      * 200 - HTTPOk
      * 201 - HTTPCreated
      * 202 - HTTPAccepted
      * 203 - HTTPNonAuthoritativeInformation
      * 204 - HTTPNoContent
      * 205 - HTTPResetContent
      * 206 - HTTPPartialContent
    HTTPRedirection
      * 300 - HTTPMultipleChoices
      * 301 - HTTPMovedPermanently
      * 302 - HTTPFound
      * 303 - HTTPSeeOther
      * 304 - HTTPNotModified
      * 305 - HTTPUseProxy
      * 307 - HTTPTemporaryRedirect
    HTTPError
      HTTPClientError
        * 400 - HTTPBadRequest
        * 401 - HTTPUnauthorized
        * 402 - HTTPPaymentRequired
        * 403 - HTTPForbidden
        * 404 - HTTPNotFound
        * 405 - HTTPMethodNotAllowed
        * 406 - HTTPNotAcceptable
        * 407 - HTTPProxyAuthenticationRequired
        * 408 - HTTPRequestTimeout
        * 409 - HTTPConflict
        * 410 - HTTPGone
        * 411 - HTTPLengthRequired
        * 412 - HTTPPreconditionFailed
```

```
        * 413 - HTTPRequestEntityTooLarge
        * 414 - HTTPRequestURITooLong
        * 415 - HTTPUnsupportedMediaType
        * 416 - HTTPRequestRangeNotSatisfiable
        * 417 - HTTPExpectationFailed
      HTTPServerError
        * 500 - HTTPInternalServerError
        * 501 - HTTPNotImplemented
        * 502 - HTTPBadGateway
        * 503 - HTTPServiceUnavailable
        * 504 - HTTPGatewayTimeout
        * 505 - HTTPVersionNotSupported
```

All HTTP exceptions have the same constructor:

```
HTTPNotFound(*, headers=None, reason=None,
             body=None, text=None, content_type=None)
```

if other not directly specified. *headers* will be added to *default response headers*.

Classes `HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPFound`, `HTTPSeeOther`, `HTTPUseProxy`, `HTTPTemporaryRedirect` has constructor signature like:

```
HTTPFound(location, *, headers=None, reason=None,
          body=None, text=None, content_type=None)
```

where *location* is value for *Location HTTP header*.

`HTTPMethodNotAllowed` constructed with pointing trial method and list of allowed methods:

```
HTTPMethodNotAllowed(method, allowed_methods, *,
                     headers=None, reason=None,
                     body=None, text=None, content_type=None)
```

9.4.10 Data sharing

aiohttp discourages the use of *global variables*, aka *singletons*.

Every variable should have it's own context that is *not global*.

Thus, *aiohttp.web.Application* and *aiohttp.web.Request* support a `collections.abc.MutableMapping` interface (i.e. they are dict-like objects), allowing them to be used as data stores.

For storing *global-like* variables, feel free to save them in an *Application* instance:

```
app['my_private_key'] = data
```

and get it back in the *web-handler*:

```
async def handler(request):
    data = request.app['my_private_key']
```

Variables that are only needed for the lifetime of a *Request*, can be stored in a *Request*:

```
async def handler(request):
  request['my_private_key'] = "data"
  ...
```

This is mostly useful for *Middlewares* and *Signals* handlers to store data for further processing by the next handlers in the chain.

To avoid clashing with other *aiohttp* users and third-party libraries, please choose a unique key name for storing data.

If your code is published on PyPI, then the project name is most likely unique and safe to use as the key. Otherwise, something based on your company name/url would be satisfactory (i.e `org.company.app`).

9.4.11 Middlewares

New in version 0.13.

`Application` accepts optional *middlewares* keyword-only parameter, which should be a sequence of *middleware factories*, e.g:

```
app = web.Application(middlewares=[middleware_factory_1,
                                   middleware_factory_2])
```

The most trivial *middleware factory* example:

```
async def middleware_factory(app, handler):
    async def middleware(request):
        return await handler(request)
    return middleware
```

Every factory is a coroutine that accepts two parameters: *app* (`Application` instance) and *handler* (next handler in middleware chain).

The last handler is *web-handler* selected by routing itself (`resolve()` call).

Middleware should return a new coroutine by wrapping *handler* parameter. Signature of returned handler should be the same as for *web-handler*: accept single *request* parameter, return *response* or raise exception.

The factory is a coroutine, thus it can do extra `await` calls on making new handler, e.g. call database etc.

After constructing outermost handler by applying middleware chain to *web-handler* in reversed order `RequestHandler` executes the outermost handler as regular *web-handler*.

Middleware usually calls an inner handler, but may do something other, like displaying *403 Forbidden page* or raising `HTTPForbidden` exception if user has no permissions to access underlying resource. Also middleware may render errors raised by handler, do some pre- and post- processing and so on.

Changed in version 0.14: Middleware accepts route exceptions (`HTTPNotFound` and `HTTPMethodNotAllowed`).

9.4.12 Signals

New in version 0.18.

While *middlewares* give very powerful tool for customizing *web handler* processing we also need another machinery called signals.

For example middleware may change HTTP headers for *unprepared* response only (see `prepare()`).

But sometimes we need a hook for changing HTTP headers for streamed responses and websockets. That can be done by subscribing on `on_response_prepare` signal:

```
async def on_prepare(request, response):
    response.headers['My-Header'] = 'value'

app.on_response_prepare.append(on_prepare)
```

Signal handlers should not return a value but may modify incoming mutable parameters.

Warning: Signals has provisional status.
That means API may be changed in future releases.
Most likely signal subscription/sending will be the same but signal object creation is subject for changing. Unless you don't create new signals but reuse existing only you are not affected.

9.4.13 CORS support

aiohttp.web itself has no support for Cross-Origin Resource Sharing but there is aiohttp plugin for it: aiohttp_cors.

9.4.14 Debug toolbar

aiohttp_debugtoolbar is very useful library that provides debug toolbar while you're developing *aiohttp.web* application.

Install it via `pip` tool:

```
$ pip install aiohttp_debugtoolbar
```

After that attach middleware to your *aiohttp.web.Application* and call `aiohttp_debugtoolbar.setup`:

```
import aiohttp_debugtoolbar
from aiohttp_debugtoolbar import toolbar_middleware_factory

app = web.Application(loop=loop,
                      middlewares=[toolbar_middleware_factory])
aiohttp_debugtoolbar.setup(app)
```

Debug toolbar is ready to use. Enjoy!!!

9.5 HTTP Server Reference

Changed in version 0.12: The module was deeply refactored in backward incompatible manner.

9.5.1 Request

The Request object contains all the information about an incoming HTTP request.

Every *handler* accepts a request instance as the first positional parameter.

A *Request* is a `dict`-like object, allowing it to be used for *sharing data* among *Middlewares* and *Signals* handlers.

Although *Request* is `dict`-like object, it can't be duplicated like one using `Request.copy()`.

Note: You should never create the *Request* instance manually – *aiohttp.web* does it for you.

class `aiohttp.web.`**`Request`**

scheme
A string representing the scheme of the request.

The scheme is `'https'` if transport for request handling is *SSL* or `secure_proxy_ssl_header` is matching.

`'http'` otherwise.

Read-only `str` property.

method
HTTP method, read-only property.

The value is upper-cased `str` like `"GET"`, `"POST"`, `"PUT"` etc.

version
HTTP version of request, Read-only property.

Returns *aiohttp.protocol.HttpVersion* instance.

host
HOST header of request, Read-only property.

Returns `str` or `None` if HTTP request has no *HOST* header.

path_qs
The URL including PATH_INFO and the query string. e.g, `/app/blog?id=10`

Read-only `str` property.

path
The URL including *PATH INFO* without the host or scheme. e.g., `/app/blog`. The path is URL-unquoted. For raw path info see *raw_path*.

Read-only `str` property.

raw_path
The URL including raw *PATH INFO* without the host or scheme. Warning, the path may be quoted and may contains non valid URL characters, e.g. `/my%2Fpath%7Cwith%21some%25strange%24characters`.

For unquoted version please take a look on *path*.

Read-only `str` property.

query_string
The query string in the URL, e.g., `id=10`

Read-only `str` property.

GET
A multidict with all the variables in the query string.

Read-only *MultiDictProxy* lazy property.

Changed in version 0.17: A multidict contains empty items for query string like `?arg=`.

POST
A multidict with all the variables in the POST parameters. POST property available only after *Request.post()* coroutine call.

Read-only *MultiDictProxy*.

Raises **RuntimeError** if *Request.post()* was not called before accessing the property.

headers
A case-insensitive multidict proxy with all headers.

Read-only *CIMultiDictProxy* property.

keep_alive
`True` if keep-alive connection enabled by HTTP client and protocol version supports it, otherwise `False`.

Read-only `bool` property.

match_info
Read-only property with `AbstractMatchInfo` instance for result of route resolving.

Note: Exact type of property depends on used router. If `app.router` is *UrlDispatcher* the property contains *UrlMappingMatchInfo* instance.

app
An *Application* instance used to call *request handler*, Read-only property.

transport
An transport used to process request, Read-only property.

The property can be used, for example, for getting IP address of client's peer:

```
peername = request.transport.get_extra_info('peername')
if peername is not None:
    host, port = peername
```

cookies
A multidict of all request's cookies.

Read-only *MultiDictProxy* lazy property.

content
A *FlowControlStreamReader* instance, input stream for reading request's *BODY*.

Read-only property.

New in version 0.15.

has_body
Return `True` if request has *HTTP BODY*, `False` otherwise.

Read-only `bool` property.

New in version 0.16.

payload
A *FlowControlStreamReader* instance, input stream for reading request's *BODY*.

Read-only property.

Deprecated since version 0.15: Use *content* instead.

content_type
Read-only property with *content* part of *Content-Type* header.

Returns `str` like `'text/html'`

Note: Returns value is `'application/octet-stream'` if no Content-Type header present in HTTP headers according to **RFC 2616**

charset
Read-only property that specifies the *encoding* for the request's BODY.

The value is parsed from the *Content-Type* HTTP header.

Returns `str` like `'utf-8'` or `None` if *Content-Type* has no charset information.

content_length
Read-only property that returns length of the request's BODY.

The value is parsed from the *Content-Length* HTTP header.

Returns `int` or `None` if *Content-Length* is absent.

if_modified_since
Read-only property that returns the date specified in the *If-Modified-Since* header.

Returns `datetime.datetime` or `None` if *If-Modified-Since* header is absent or is not a valid HTTP date.

coroutine read()
Read request body, returns `bytes` object with body content.

Note: The method **does** store read data internally, subsequent *read()* call will return the same value.

coroutine text()
Read request body, decode it using *charset* encoding or `UTF-8` if no encoding was specified in *MIME-type*.

Returns `str` with body content.

Note: The method **does** store read data internally, subsequent *text()* call will return the same value.

coroutine json(*, *loader=json.loads*)
Read request body decoded as *json*.

The method is just a boilerplate coroutine implemented as:

```
async def json(self, *, loader=json.loads):
    body = await self.text()
    return loader(body)
```

Parameters **loader** (*callable*) – any *callable* that accepts `str` and returns `dict` with parsed JSON (`json.loads()` by default).

Note: The method **does** store read data internally, subsequent *json()* call will return the same value.

coroutine post()
A coroutine that reads POST parameters from request body.

Returns *MultiDictProxy* instance filled with parsed data.

If *method* is not *POST*, *PUT* or *PATCH* or *content_type* is not empty or *application/x-www-form-urlencoded* or *multipart/form-data* returns empty multidict.

Note: The method **does** store read data internally, subsequent *post()* call will return the same value.

coroutine release()
Release request.

Eat unread part of HTTP BODY if present.

Note: User code may never call *release()*, all required work will be processed by *aiohttp.web* internal machinery.

9.5.2 Response classes

For now, *aiohttp.web* has two classes for the *HTTP response*: *StreamResponse* and *Response*.

Usually you need to use the second one. *StreamResponse* is intended for streaming data, while *Response* contains *HTTP BODY* as an attribute and sends own content as single piece with the correct *Content-Length HTTP header*.

For sake of design decisions *Response* is derived from *StreamResponse* parent class.

The response supports *keep-alive* handling out-of-the-box if *request* supports it.

You can disable *keep-alive* by *force_close()* though.

The common case for sending an answer from *web-handler* is returning a *Response* instance:

```
def handler(request):
    return Response("All right!")
```

StreamResponse

class aiohttp.web.**StreamResponse**(*, *status=200*, *reason=None*)

The base class for the *HTTP response* handling.

Contains methods for setting *HTTP response headers*, *cookies*, *response status code*, writing *HTTP response BODY* and so on.

The most important thing you should know about *response* — it is *Finite State Machine*.

That means you can do any manipulations with *headers*, *cookies* and *status code* only before *prepare()* coroutine is called.

Once you call *prepare()* any change of the *HTTP header* part will raise RuntimeError exception.

Any *write()* call after *write_eof()* is also forbidden.

Parameters

- **status** (*int*) – HTTP status code, 200 by default.
- **reason** (*str*) – HTTP reason. If param is None reason will be calculated basing on *status* parameter. Otherwise pass str with arbitrary *status* explanation..

prepared

Read-only bool property, True if *prepare()* has been called, False otherwise.

New in version 0.18.

started

Deprecated alias for *prepared*.

Deprecated since version 0.18.

status

Read-only property for *HTTP response status code*, int.

200 (OK) by default.

reason
Read-only property for *HTTP response reason*, `str`.

set_status(*status*, *reason=None*)
Set *status* and *reason*.

reason value is auto calculated if not specified (`None`).

keep_alive
Read-only property, copy of *Request.keep_alive* by default.

Can be switched to `False` by *force_close()* call.

force_close()
Disable *keep_alive* for connection. There are no ways to enable it back.

compression
Read-only `bool` property, `True` if compression is enabled.

`False` by default.

New in version 0.14.

See also:

enable_compression()

enable_compression(*force=None*)
Enable compression.

When *force* is unset compression encoding is selected based on the request's *Accept-Encoding* header.

Accept-Encoding is not checked if *force* is set to a *ContentCoding*.

New in version 0.14.

See also:

compression

chunked
Read-only property, indicates if chunked encoding is on.

Can be enabled by *enable_chunked_encoding()* call.

New in version 0.14.

See also:

enable_chunked_encoding

enable_chunked_encoding()
Enables *chunked* encoding for response. There are no ways to disable it back. With enabled *chunked* encoding each *write()* operation encoded in separate chunk.

New in version 0.14.

> **Warning:** chunked encoding can be enabled for `HTTP/1.1` only.
> Setting up both *content_length* and chunked encoding is mutually exclusive.

See also:

chunked

headers
CIMultiDict instance for *outgoing HTTP headers*.

cookies
An instance of `http.cookies.SimpleCookie` for *outgoing* cookies.

> **Warning:** Direct setting up *Set-Cookie* header may be overwritten by explicit calls to cookie manipulation.
> We are encourage using of *cookies* and *set_cookie()*, *del_cookie()* for cookie manipulations.

set_cookie(*name*, *value*, ***, *path='/'*, *expires=None*, *domain=None*, *max_age=None*, *secure=None*, *httponly=None*, *version=None*)
Convenient way for setting *cookies*, allows to specify some additional properties like *max_age* in a single call.

Parameters

- **name** (*str*) – cookie name
- **value** (*str*) – cookie value (will be converted to `str` if value has another type).
- **expires** – expiration date (optional)
- **domain** (*str*) – cookie domain (optional)
- **max_age** (*int*) – defines the lifetime of the cookie, in seconds. The delta-seconds value is a decimal non- negative integer. After delta-seconds seconds elapse, the client should discard the cookie. A value of zero means the cookie should be discarded immediately. (optional)
- **path** (*str*) – specifies the subset of URLs to which this cookie applies. (optional, `'/'` by default)
- **secure** (*bool*) – attribute (with no value) directs the user agent to use only (unspecified) secure means to contact the origin server whenever it sends back this cookie. The user agent (possibly under the user's control) may determine what level of security it considers appropriate for "secure" cookies. The *secure* should be considered security advice from the server to the user agent, indicating that it is in the session's interest to protect the cookie contents. (optional)
- **httponly** (*bool*) – `True` if the cookie HTTP only (optional)
- **version** (*int*) – a decimal integer, identifies to which version of the state management specification the cookie conforms. (Optional, *version=1* by default)

Changed in version 0.14.3: Default value for *path* changed from `None` to `'/'`.

del_cookie(*name*, ***, *path='/'*, *domain=None*)
Deletes cookie.

Parameters

- **name** (*str*) – cookie name
- **domain** (*str*) – optional cookie domain
- **path** (*str*) – optional cookie path, `'/'` by default

Changed in version 0.14.3: Default value for *path* changed from `None` to `'/'`.

content_length
Content-Length for outgoing response.

content_type
Content part of *Content-Type* for outgoing response.

charset
Charset aka *encoding* part of *Content-Type* for outgoing response.

The value converted to lower-case on attribute assigning.

last_modified
Last-Modified header for outgoing response.

This property accepts raw `str` values, `datetime.datetime` objects, Unix timestamps specified as an `int` or a `float` object, and the value `None` to unset the header.

start (*request*)

Parameters **request** (aiohttp.web.Request) – HTTP request object, that the response answers.

Send *HTTP header*. You should not change any header data after calling this method.

Deprecated since version 0.18: Use *prepare()* instead.

Warning: The method doesn't call `web.Application.on_response_prepare` signal, use *prepare()* instead.

coroutine prepare (*request*)

Parameters **request** (aiohttp.web.Request) – HTTP request object, that the response answers.

Send *HTTP header*. You should not change any header data after calling this method.

The coroutine calls `web.Application.on_response_prepare` signal handlers.

New in version 0.18.

write (*data*)
Send byte-ish data as the part of *response BODY*.

prepare() must be called before.

Raises `TypeError` if data is not `bytes`, `bytearray` or `memoryview` instance.

Raises `RuntimeError` if *prepare()* has not been called.

Raises `RuntimeError` if *write_eof()* has been called.

coroutine drain ()
A coroutine to let the write buffer of the underlying transport a chance to be flushed.

The intended use is to write:

```
resp.write(data)
await resp.drain()
```

Yielding from *drain()* gives the opportunity for the loop to schedule the write operation and flush the buffer. It should especially be used when a possibly large amount of data is written to the transport, and the coroutine does not yield-from between calls to *write()*.

New in version 0.14.

coroutine write_eof ()
A coroutine *may* be called as a mark of the *HTTP response* processing finish.

Internal machinery will call this method at the end of the request processing if needed.

After *write_eof()* call any manipulations with the *response* object are forbidden.

Response

class aiohttp.web.**Response**(*, *status=200*, *headers=None*, *content_type=None*, *charset=None*, *body=None*, *text=None*)

The most usable response class, inherited from *StreamResponse*.

Accepts *body* argument for setting the *HTTP response BODY*.

The actual *body* sending happens in overridden *write_eof()*.

Parameters

- **body** (*bytes*) – response's BODY
- **status** (*int*) – HTTP status code, 200 OK by default.
- **headers** (*collections.abc.Mapping*) – HTTP headers that should be added to response's ones.
- **text** (*str*) – response's BODY
- **content_type** (*str*) – response's content type. `'text/plain'` if *text* is passed also, `'application/octet-stream'` otherwise.
- **charset** (*str*) – response's charset. `'utf-8'` if *text* is passed also, `None` otherwise.

body

Read-write attribute for storing response's content aka BODY, `bytes`.

Setting *body* also recalculates *content_length* value.

Resetting *body* (assigning `None`) sets *content_length* to `None` too, dropping *Content-Length* HTTP header.

text

Read-write attribute for storing response's content, represented as str, `str`.

Setting `str` also recalculates *content_length* value and `body` value

Resetting *body* (assigning `None`) sets *content_length* to `None` too, dropping *Content-Length* HTTP header.

WebSocketResponse

class aiohttp.web.**WebSocketResponse**(*, *timeout=10.0*, *autoclose=True*, *autoping=True*, *protocols=()*)

Class for handling server-side websockets.

After starting (by *prepare()* call) the response you cannot use *write()* method but should to communicate with websocket client by *send_str()*, *receive()* and others.

New in version 0.19: The class supports `async for` statement for iterating over incoming messages:

```
ws = web.WebSocketResponse()
await ws.prepare(request)

async for msg in ws:
    print(msg.data)
```

coroutine prepare(*request*)

Starts websocket. After the call you can use websocket methods.

Parameters **request** (aiohttp.web.Request) – HTTP request object, that the response answers.

Raises HTTPException if websocket handshake has failed.

New in version 0.18.

start(*request*)
Starts websocket. After the call you can use websocket methods.

Parameters **request** (aiohttp.web.Request) – HTTP request object, that the response answers.

Raises HTTPException if websocket handshake has failed.

Deprecated since version 0.18: Use *prepare()* instead.

can_prepare(*request*)
Performs checks for *request* data to figure out if websocket can be started on the request.

If *can_prepare()* call is success then *prepare()* will success too.

Parameters **request** (aiohttp.web.Request) – HTTP request object, that the response answers.

Returns `(ok, protocol)` pair, *ok* is `True` on success, *protocol* is websocket subprotocol which is passed by client and accepted by server (one of *protocols* sequence from *WebSocketResponse* ctor). *protocol* may be `None` if client and server subprotocols are nit overlapping.

Note: The method never raises exception.

---

can_start(*request*)
Deprecated alias for *can_prepare()*

Deprecated since version 0.18.

closed
Read-only property, `True` if connection has been closed or in process of closing. `MSG_CLOSE` message has been received from peer.

close_code
Read-only property, close code from peer. It is set to `None` on opened connection.

protocol
Websocket *subprotocol* chosen after *start()* call.

May be `None` if server and client protocols are not overlapping.

exception()
Returns last occurred exception or None.

ping(*message=b''*)
Send `MSG_PING` to peer.

Parameters **message** – optional payload of *ping* message, `str` (converted to *UTF-8* encoded bytes) or `bytes`.

Raises **RuntimeError** if connections is not started or closing.

pong(*message=b''*)
Send *unsolicited* `MSG_PONG` to peer.

Parameters **message** – optional payload of *pong* message, `str` (converted to *UTF-8* encoded bytes) or `bytes`.

Raises **RuntimeError** if connections is not started or closing.

send_str(*data*)
Send *data* to peer as `MSG_TEXT` message.

Parameters **data** (*str*) – data to send.

Raises

- **RuntimeError** – if connection is not started or closing
- **TypeError** – if data is not `str`

send_bytes(*data*)

Send *data* to peer as `MSG_BINARY` message.

Parameters **data** – data to send.

Raises

- **RuntimeError** – if connection is not started or closing
- **TypeError** – if data is not `bytes`, `bytearray` or `memoryview`.

coroutine close(*, *code=1000*, *message=b''*)

A coroutine that initiates closing handshake by sending `MSG_CLOSE` message.

Parameters

- **code** (*int*) – closing code
- **message** – optional payload of *pong* message, `str` (converted to *UTF-8* encoded bytes) or `bytes`.

Raises **RuntimeError** if connection is not started or closing

coroutine receive()

A coroutine that waits upcoming *data* message from peer and returns it.

The coroutine implicitly handles `MSG_PING`, `MSG_PONG` and `MSG_CLOSE` without returning the message.

It process *ping-pong game* and performs *closing handshake* internally.

After websocket closing raises *WSClientDisconnectedError* with connection closing data.

Returns *Message*

Raises **RuntimeError** if connection is not started

Raise *WSClientDisconnectedError* on closing.

coroutine receive_str()

A coroutine that calls `receive_mgs()` but also asserts the message type is `MSG_TEXT`.

Return str peer's message content.

Raises **TypeError** if message is `MSG_BINARY`.

coroutine receive_bytes()

A coroutine that calls `receive_mgs()` but also asserts the message type is `MSG_BINARY`.

Return bytes peer's message content.

Raises **TypeError** if message is `MSG_TEXT`.

New in version 0.14.

See also:

WebSockets handling

9.5.3 json_response

aiohttp.web.**json_response**([*data*], *, *text=None*, *body=None*, *status=200*, *reason=None*, *headers=None*, *content_type='application/json'*, *dumps=json.dumps*)

Return *Response* with predefined `'application/json'` content type and *data* encoded by *dumps* parameter (`json.dumps()` by default).

9.5.4 Application and Router

Application

Application is a synonym for web-server.

To get fully working example, you have to make *application*, register supported urls in *router* and create a *server socket* with `aiohttp.RequestHandlerFactory` as a *protocol factory*. *RequestHandlerFactory* could be constructed with `make_handler()`.

Application contains a *router* instance and a list of callbacks that will be called during application finishing.

Application is a `dict`-like object, so you can use it for *sharing data* globally by storing arbitrary properties for later access from a *handler* via the *Request.app* property:

```
app = Application(loop=loop)
app['database'] = await aiopg.create_engine(**db_config)

async def handler(request):
    with (await request.app['database']) as conn:
        conn.execute("DELETE * FROM table")
```

Although *Application* is a `dict`-like object, it can't be duplicated like one using `Application.copy()`.

class aiohttp.web.**Application**(*, *loop=None*, *router=None*, *logger=<default>*, *middlewares=()*, ***kwargs*)

The class inherits `dict`.

Parameters

- **loop** – event loop used for processing HTTP requests.

  If param is `None` `asyncio.get_event_loop()` used for getting default event loop, but we strongly recommend to use explicit loops everywhere.

- **router** – `aiohttp.abc.AbstractRouter` instance, the system creates *UrlDispatcher* by default if *router* is `None`.

- **logger** – `logging.Logger` instance for storing application logs.

  By default the value is `logging.getLogger("aiohttp.web")`

- **middlewares** – `list` of middleware factories, see *Middlewares* for details.

  New in version 0.13.

router
Read-only property that returns *router instance*.

logger
`logging.Logger` instance for storing application logs.

loop
event loop used for processing HTTP requests.

on_response_prepare
A *Signal* that is fired at the beginning of *StreamResponse.prepare()* with parameters *request* and *response*. It can be used, for example, to add custom headers to each response before sending.

Signal handlers should have the following signature:

```
async def handler(request, response):
    pass
```

make_handler(***kwargs*)
Creates HTTP protocol factory for handling requests.

Parameters **kwargs** – additional parameters for `RequestHandlerFactory` constructor.

You should pass result of the method as *protocol_factory* to `create_server()`, e.g.:

```
loop = asyncio.get_event_loop()

app = Application(loop=loop)

# setup route table
# app.router.add_route(...)

await loop.create_server(app.make_handler(),
                         '0.0.0.0', 8080)
```

coroutine finish()
A coroutine that should be called after server stopping.

This method executes functions registered by `register_on_finish()` in LIFO order.

If callback raises an exception, the error will be stored by `call_exception_handler()` with keys: *message*, *exception*, *application*.

register_on_finish(self, func, *args, **kwargs):
Register *func* as a function to be executed at termination. Any optional arguments that are to be passed to *func* must be passed as arguments to `register_on_finish()`. It is possible to register the same function and arguments more than once.

During the call of *finish()* all functions registered are called in last in, first out order.

func may be either regular function or coroutine, *finish()* will un-yield (*await*) the later.

Note: Application object has *router* attribute but has no `add_route()` method. The reason is: we want to support different router implementations (even maybe not url-matching based but traversal ones).

For sake of that fact we have very trivial ABC for `AbstractRouter`: it should have only `AbstractRouter.resolve()` coroutine.

No methods for adding routes or route reversing (getting URL by route name). All those are router implementation details (but, sure, you need to deal with that methods after choosing the router for your application).

RequestHandlerFactory

RequestHandlerFactory is responsible for creating HTTP protocol objects that can handle HTTP connections.

aiohttp.web.**connections**
List of all currently opened connections.

aiohttp.web.**finish_connections**(*timeout*)
A coroutine that should be called to close all opened connections.

Router

For dispatching URLs to *handlers* `aiohttp.web` uses *routers*.

Router is any object that implements `AbstractRouter` interface.

`aiohttp.web` provides an implementation called `UrlDispatcher`.

`Application` uses `UrlDispatcher` as `router()` by default.

class `aiohttp.web.`**`UrlDispatcher`**

Straightforward url-matching router, implements `collections.abc.Mapping` for access to *named routes*.

Before running `Application` you should fill *route table* first by calling `add_route()` and `add_static()`.

Handler lookup is performed by iterating on added *routes* in FIFO order. The first matching *route* will be used to call corresponding *handler*.

If on route creation you specify *name* parameter the result is *named route*.

Named route can be retrieved by `app.router[name]` call, checked for existence by `name in app.router` etc.

See also:

Route classes

`add_route`(*method*, *path*, *handler*, ***, *name=None*, *expect_handler=None*)

Append *handler* to the end of route table.

***path* may be either *constant* string like `'/a/b/c'` or** *variable rule* like `'/a/{var}'` (see *handling variable pathes*)

Pay attention please: *handler* is converted to coroutine internally when it is a regular function.

Parameters

- **`method`** (*str*) – HTTP method for route. Should be one of `'GET'`, `'POST'`, `'PUT'`, `'DELETE'`, `'PATCH'`, `'HEAD'`, `'OPTIONS'` or `'*'` for any method.

  The parameter is case-insensitive, e.g. you can push `'get'` as well as `'GET'`.

- **`path`** (*str*) – route path. Should be started with slash (`'/'`).
- **`handler`** (*callable*) – route handler.
- **`name`** (*str*) – optional route name.
- **`expect_handler`** (*coroutine*) – optional *expect* header handler.

Returns new `PlainRoute` or `DynamicRoute` instance.

`add_static`(*prefix*, *path*, ***, *name=None*, *expect_handler=None*, *chunk_size=256*1024*, *response_factory=StreamResponse*)

Adds a router and a handler for returning static files.

Useful for serving static content like images, javascript and css files.

On platforms that support it, the handler will transfer files more efficiently using the `sendfile` system call.

In some situations it might be necessary to avoid using the `sendfile` system call even if the platform supports it. This can be accomplished by by setting environment variable `AIOHTTP_NOSENDFILE=1`.

Warning: Use *add_static()* for development only. In production, static content should be processed by web servers like *nginx* or *apache*.

Changed in version 0.18.0: Transfer files using the `sendfile` system call on supported platforms.

Changed in version 0.19.0: Disable `sendfile` by setting environment variable `AIOHTTP_NOSENDFILE=1`

Parameters

- **prefix** (*str*) – URL path prefix for handled static files
- **path** (*str*) – path to the folder in file system that contains handled static files.
- **name** (*str*) – optional route name.
- **expect_handler** (*coroutine*) – optional *expect* header handler.
- **chunk_size** (*int*) – size of single chunk for file downloading, 256Kb by default.

  Increasing *chunk_size* parameter to, say, 1Mb may increase file downloading speed but consumes more memory.

  New in version 0.16.
- **response_factory** (*callable*) – factory to use to generate a new response, defaults to *StreamResponse* and should expose a compatible API.

  New in version 0.17.

Returns new *StaticRoute* instance.

coroutine resolve(*requst*)

A coroutine that returns `AbstractMatchInfo` for *request*.

The method never raises exception, but returns `AbstractMatchInfo` instance with:

1. `route` assigned to *SystemRoute* instance
2. `handler` which raises `HTTPNotFound` or `HTTPMethodNotAllowed` on handler's execution if there is no registered route for *request*.

   Middlewares can process that exceptions to render pretty-looking error page for example.

Used by internal machinery, end user unlikely need to call the method.

Note: The method uses *Request.raw_path* for pattern matching against registered routes.

Changed in version 0.14: The method don't raise `HTTPNotFound` and `HTTPMethodNotAllowed` anymore.

routes()

The method returns a *view* for *all* registered routes.

The view is an object that allows to:

1. Get size of the router table:

```
len(app.router.routes())
```

2. Iterate over registered routes:

```
for route in app.router.routes():
    print(route)
```

3.Make a check if the route is registered in the router table:

```
route in app.router.routes()
```

New in version 0.18.

named_routes()

Returns a dict-like types.MappingProxyType *view* over *all* named routes.

The view maps every named route's *Route.name* attribute to the *Route*. It supports the usual dict-like operations, except for any mutable operations (i.e. it's **read-only**):

```
len(app.router.named_routes())

for name, route in app.router.named_routes().items():
    print(name, route)

"route_name" in app.router.named_routes()

app.router.named_routes()["route_name"]
```

New in version 0.19.

Route

Default router *UrlDispatcher* operates with *routes*.

User should not instantiate route classes by hand but can give *named route instance* by `router[name]` if he have added route by *UrlDispatcher.add_route()* or *UrlDispatcher.add_static()* calls with non-empty *name* parameter.

The main usage of *named routes* is constructing URL by route name for passing it into *template engine* for example:

```
url = app.router['route_name'].url(query={'a': 1, 'b': 2})
```

There are three concrete route classes:

- *PlainRoute* for urls without *variable pathes* spec.
- *DynamicRoute* for urls with *variable pathes* spec.
- *StaticRoute* for static file handlers.

class aiohttp.web.**Route**

Base class for routes served by *UrlDispatcher*.

method

HTTP method handled by the route, e.g. *GET*, *POST* etc.

handler

handler that processes the route.

name

Name of the route.

match(*path*)

Abstract method, accepts *URL path* and returns `dict` with parsed *path parts* for *`UrlMappingMatchInfo`* or `None` if the route cannot handle given *path*.

The method exists for internal usage, end user unlikely need to call it.

`url`(*, *query=None*, ***kwargs*)

Abstract method for constructing url handled by the route.

query is a mapping or list of *(name, value)* pairs for specifying *query* part of url (parameter is processed by `urlencode()`).

Other available parameters depends on concrete route class and described in descendant classes.

class `aiohttp.web.`**`PlainRoute`**

The route class for handling plain *URL path*, e.g. `"/a/b/c"`

`url`(*, *parts*, *query=None*)

Construct url, doesn't accepts extra parameters:

```
>>> route.url(query={'d': 1, 'e': 2})
'/a/b/c/?d=1&e=2'
```

class `aiohttp.web.`**`DynamicRoute`**

The route class for handling *variable path*, e.g. `"/a/{name1}/{name2}"`

`url`(*, *parts*, *query=None*)

Construct url with given *dynamic parts*:

```
>>> route.url(parts={'name1': 'b', 'name2': 'c'},
              query={'d': 1, 'e': 2})
'/a/b/c/?d=1&e=2'
```

class `aiohttp.web.`**`StaticRoute`**

The route class for handling static files, created by *`UrlDispatcher.add_static()`* call.

`url`(*, *filename*, *query=None*)

Construct url for given *filename*:

```
>>> route.url(filename='img/logo.png', query={'param': 1})
'/path/to/static/img/logo.png?param=1'
```

class `aiohttp.web.`**`SystemRoute`**

The route class for internal purposes.

Now it has used for handling *404: Not Found* and *405: Method Not Allowed*.

`url`()

Always raises `RuntimeError`, *`SystemRoute`* should not be used in url construction expressions.

MatchInfo

After route matching web application calls found handler if any.

Matching result can be accessible from handler as *`Request.match_info`* attribute.

In general the result may be any object derived from `AbstractMatchInfo` (*`UrlMappingMatchInfo`* for default *`UrlDispatcher`* router).

class aiohttp.web.**UrlMappingMatchInfo**
Inherited from dict and AbstractMatchInfo. Dict items are given from *Route.match()* call return value.

route

Route instance for url matching.

9.5.5 Utilities

class aiohttp.web.**FileField**
A namedtuple() that is returned as multidict value by *Request.POST()* if field is uploaded file.

name
Field name

filename
File name as specified by uploading (client) side.

file
An io.IOBase instance with content of uploaded file.

content_type
MIME type of uploaded file, 'text/plain' by default.

See also:

File Uploads

9.5.6 Constants

class aiohttp.web.**ContentCoding**
An enum.Enum class of available Content Codings.

deflate

gzip

identity

9.6 Low-level HTTP Server

Note: This topic describes the low-level HTTP support. For high-level interface please take a look on *aiohttp.web*.

9.6.1 Run a basic server

Start implementing the basic server by inheriting the *ServerHttpProtocol* object. Your class should implement the only method *ServerHttpProtocol.handle_request()* which must be a coroutine to handle requests asynchronously

```
from urllib.parse import urlparse, parse_qsl

import aiohttp
import aiohttp.server
```

```
    from aiohttp import MultiDict


    import asyncio

    class HttpRequestHandler(aiohttp.server.ServerHttpProtocol):

      async def handle_request(self, message, payload):
          response = aiohttp.Response(
              self.writer, 200, http_version=message.version
          )
          response.add_header('Content-Type', 'text/html')
          response.add_header('Content-Length', '18')
          response.send_headers()
          response.write(b'<h1>It Works!</h1>')
          await response.write_eof()
```

The next step is to create a loop and register your handler within a server. `KeyboardInterrupt` exception handling is necessary so you can stop your server with Ctrl+C at any time.

```
    if __name__ == '__main__':
        loop = asyncio.get_event_loop()
        f = loop.create_server(
            lambda: HttpRequestHandler(debug=True, keep_alive=75),
            '0.0.0.0', '8080')
        srv = loop.run_until_complete(f)
        print('serving on', srv.sockets[0].getsockname())
        try:
            loop.run_forever()
        except KeyboardInterrupt:
            pass
```

9.6.2 Headers

Data is passed to the handler in the `message`, while request body is passed in `payload` param. HTTP headers are accessed through `headers` member of the message. To check what the current method of the request is use the `method` member of the `message`. It should be one of `GET`, `POST`, `PUT` or `DELETE` strings.

9.6.3 Handling GET params

Currently aiohttp does not provide automatic parsing of incoming GET params. However aiohttp does provide a nice `MulitiDict` wrapper for already parsed params.

```
    from urllib.parse import urlparse, parse_qsl

    from aiohttp import MultiDict

    class HttpRequestHandler(aiohttp.server.ServerHttpProtocol):

        async def handle_request(self, message, payload):
            response = aiohttp.Response(
                self.writer, 200, http_version=message.version
            )
            get_params = MultiDict(parse_qsl(urlparse(message.path).query))
            print("Passed in GET", get_params)
```

9.6.4 Handling POST data

POST data is accessed through the `payload.read()` generator method. If you have form data in the request body, you can parse it in the same way as GET params.

```
from urllib.parse import urlparse, parse_qsl

from aiohttp import MultiDict

class HttpRequestHandler(aiohttp.server.ServerHttpProtocol):

    async def handle_request(self, message, payload):
        response = aiohttp.Response(
            self.writer, 200, http_version=message.version
        )
        data = await payload.read()
        post_params = MultiDict(parse_qsl(data))
        print("Passed in POST", post_params)
```

9.6.5 SSL

To use asyncio's SSL support, just pass an SSLContext object to the `asyncio.BaseEventLoop.create_server()` method of the loop.

```
import ssl

sslcontext = ssl.SSLContext(ssl.PROTOCOL_SSLv23)
sslcontext.load_cert_chain('sample.crt', 'sample.key')

loop = asyncio.get_event_loop()
loop.create_server(lambda: handler, "0.0.0.0", "8080", ssl=sslcontext)
```

9.6.6 Reference

simple http server.

class `aiohttp.server.`**`ServerHttpProtocol`**(**, loop=None, keep_alive=75, keep_alive_on=True, timeout=0, logger=<logging.Logger object>, access_log=None, access_log_format='%a %l %u %t "%r" %s %b "%{Referrer}i" "%{User-Agent}i"', debug=False, log=None, **kwargs*)

Bases: *aiohttp.parsers.StreamProtocol*

Simple http protocol implementation.

ServerHttpProtocol handles incoming http request. It reads request line, request headers and request payload and calls handle_request() method. By default it always returns with 404 response.

ServerHttpProtocol handles errors in incoming request, like bad status line, bad headers or incomplete payload. If any error occurs, connection gets closed.

Parameters

- **`keep_alive`** (*int or None*) – number of seconds before closing keep-alive connection
- **`keep_alive_on`** (*bool*) – keep-alive is o, default is on
- **`timeout`** (*int*) – slow request timeout

- **allowed_methods** (*tuple*) – (optional) List of allowed request methods. Set to empty list to allow all methods.
- **debug** (*bool*) – enable debug mode
- **logger** (*aiohttp.log.server_logger*) – custom logger object
- **access_log** (*aiohttp.log.server_logger*) – custom logging object
- **access_log_format** (*str*) – access log format string
- **loop** – Optional event loop

cancel_slow_request()

closing(*timeout=15.0*)
Worker process is about to exit, we need cleanup everything and stop accepting requests. It is especially important for keep-alive connections.

connection_lost(*exc*)

connection_made(*transport*)

data_received(*data*)

handle_error(*status=500*, *message=None*, *payload=None*, *exc=None*, *headers=None*, *reason=None*)
Handle errors.

Returns http response with specific status code. Logs additional information. It always closes current connection.

handle_request(*message*, *payload*)
Handle a single http request.

Subclass should override this method. By default it always returns 404 response.

Parameters

- **message** (aiohttp.protocol.HttpRequestParser) – Request headers
- **payload** (aiohttp.streams.FlowControlStreamReader) – Request payload

keep_alive(*val*)
Set keep-alive connection mode.

Parameters **val** (*bool*) – new state.

keep_alive_timeout

log_access(*message*, *environ*, *response*, *time*)

log_debug(**args*, ***kw*)

log_exception(**args*, ***kw*)

start()
Start processing of incoming requests.

It reads request line, request headers and request payload, then calls handle_request() method. Subclass has to override handle_request(). start() handles various exceptions in request or response handling. Connection is being closed always unless keep_alive(True) specified.

9.7 Multidicts

HTTP Headers and *URL query string* require specific data structure: *multidict*. It behaves mostly like a `dict` but it can have several *values* for the same *key*.

aiohttp has four multidict classes: *MultiDict*, *MultiDictProxy*, *CIMultiDict* and *CIMultiDictProxy*.

Immutable proxies (*MultiDictProxy* and *CIMultiDictProxy*) provide a dynamic view on the proxied multidict, the view reflects the multidict changes. They implement the `Mapping` interface.

Regular mutable (*MultiDict* and *CIMultiDict*) classes implement `MutableMapping` and allows to change their own content.

Case insensitive (*CIMultiDict* and *CIMultiDictProxy*) ones assumes the *keys* are case insensitive, e.g.:

```
>>> dct = CIMultiDict(a='val')
>>> 'A' in dct
True
>>> dct['A']
'val'
```

Keys should be a `str`.

9.7.1 MultiDict

class aiohttp.**MultiDict**(***kwargs*)
class aiohttp.**MultiDict**(*mapping*, ***kwargs*)
class aiohttp.**MultiDict**(*iterable*, ***kwargs*)

Creates a mutable multidict instance.

Accepted parameters are the same as for `dict`.

If the same key appears several times it will be added, e.g.:

```
>>> d = MultiDict([('a', 1), ('b', 2), ('a', 3)])
>>> d
<MultiDict {'a': 1, 'b': 2, 'a': 3}>
```

len(*d*)
Return the number of items in multidict *d*.

d[key]
Return the **first** item of *d* with key *key*.

Raises a `KeyError` if key is not in the multidict.

d[key] = value
Set `d[key]` to *value*.

Replace all items where key is equal to *key* with single item `(key, value)`.

del d[key]
Remove all items where key is equal to *key* from *d*. Raises a `KeyError` if *key* is not in the map.

key in d
Return `True` if d has a key *key*, else `False`.

key not in d
Equivalent to `not (key in d)`

iter(*d*)
Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

add(*key*, *value*)
Append `(key, value)` pair to the dictionary.

clear()
Remove all items from the dictionary.

copy()
Return a shallow copy of the dictionary.

extend(\[*other*\])
Extend the dictionary with the key/value pairs from *other*, overwriting existing keys. Return `None`.

extend() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then extended with those key/value pairs: `d.extend(red=1, blue=2)`.

getone(*key*\[, *default*\])
Return the **first** value for *key* if *key* is in the dictionary, else *default*.

Raises `KeyError` if *default* is not given and *key* is not found.

`d[key]` is equivalent to `d.getone(key)`.

getall(*key*\[, *default*\])
Return a list of all values for *key* if *key* is in the dictionary, else *default*.

Raises `KeyError` if *default* is not given and *key* is not found.

get(*key*\[, *default*\])
Return the **first** value for *key* if *key* is in the dictionary, else *default*.

If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

`d.get(key)` is equivalent to `d.getone(key, None)`.

keys()
Return a new view of the dictionary's keys.

View contains all keys, possibly with duplicates.

items()
Return a new view of the dictionary's items (`(key, value)` pairs).

View contains all items, multiple items can have the same key.

values()
Return a new view of the dictionary's values.

View contains all values.

pop(*key*\[, *default*\])
If *key* is in the dictionary, remove it and return its the **first** value, else return *default*.

If *default* is not given and *key* is not in the dictionary, a `KeyError` is raised.

popitem()
Remove and return an arbitrary `(key, value)` pair from the dictionary.

popitem() is useful to destructively iterate over a dictionary, as often used in set algorithms.

If the dictionary is empty, calling *popitem()* raises a `KeyError`.

setdefault(*key*[, *default*])
If *key* is in the dictionary, return its the **first** value. If not, insert *key* with a value of *default* and return *default*. *default* defaults to `None`.

update([*other*])
Update the dictionary with the key/value pairs from *other*, overwriting existing keys.

Return `None`.

update() accepts either another dictionary object or an iterable of key/value pairs (as tuples or other iterables of length two). If keyword arguments are specified, the dictionary is then updated with those key/value pairs: `d.update(red=1, blue=2)`.

See also:

MultiDictProxy can be used to create a read-only view of a *MultiDict*.

9.7.2 CIMultiDict

class aiohttp.**CIMultiDict**(***kwargs*)
class aiohttp.**CIMultiDict**(*mapping*, ***kwargs*)
class aiohttp.**CIMultiDict**(*iterable*, ***kwargs*)
Create a case insensitive multidict instance.

The behavior is the same as of *MultiDict* but key comparisons are case insensitive, e.g.:

```
>>> dct = CIMultiDict(a='val')
>>> 'A' in dct
True
>>> dct['A']
'val'
>>> dct['a']
'val'
>>> dct['b'] = 'new val'
>>> dct['B']
'new val'
```

The class is inherited from *MultiDict*.

See also:

CIMultiDictProxy can be used to create a read-only view of a *CIMultiDict*.

9.7.3 MultiDictProxy

class aiohttp.**MultiDictProxy**(*multidict*)
Create an immutable multidict proxy.

It provides a dynamic view on the multidict's entries, which means that when the multidict changes, the view reflects these changes.

Raises `TypeError` is *multidict* is not *MultiDict* instance.

len(*d*)
Return number of items in multidict *d*.

d[key]
Return the **first** item of *d* with key *key*.

Raises a `KeyError` if key is not in the multidict.

key in d
Return `True` if d has a key *key*, else `False`.

key not in d
Equivalent to `not (key in d)`

iter(*d*)
Return an iterator over the keys of the dictionary. This is a shortcut for `iter(d.keys())`.

copy()
Return a shallow copy of the underlying multidict.

getone(*key*[, *default*])
Return the **first** value for *key* if *key* is in the dictionary, else *default*.

Raises `KeyError` if *default* is not given and *key* is not found.

`d[key]` is equivalent to `d.getone(key)`.

getall(*key*[, *default*])
Return a list of all values for *key* if *key* is in the dictionary, else *default*.

Raises `KeyError` if *default* is not given and *key* is not found.

get(*key*[, *default*])
Return the **first** value for *key* if *key* is in the dictionary, else *default*.

If *default* is not given, it defaults to `None`, so that this method never raises a `KeyError`.

`d.get(key)` is equivalent to `d.getone(key, None)`.

keys()
Return a new view of the dictionary's keys.

View contains all keys, possibly with duplicates.

items()
Return a new view of the dictionary's items (`(key, value)` pairs).

View contains all items, multiple items can have the same key.

values()
Return a new view of the dictionary's values.

View contains all values.

9.7.4 CIMultiDictProxy

class aiohttp.**CIMultiDictProxy**(*multidict*)
Case insensitive version of *MultiDictProxy*.

Raises `TypeError` is *multidict* is not *CIMultiDict* instance.

The class is inherited from *MultiDict*.

9.7.5 upstr

CIMultiDict accepts `str` as *key* argument for dict lookups but converts it to upper case internally.

For more effective processing it should know if the *key* is already upper cased.

To skip the `upper()` call you may want to create upper cased strings by hand, e.g:

```
>>> key = upstr('Key')
>>> key
'KEY'
>>> mdict = CIMultiDict(key='value')
>>> key in mdict
True
>>> mdict[key]
'value'
```

For performance you should create *upstr* strings once and store them globally, like `aiohttp.hdrs` does.

class `aiohttp.`**`upstr`**`(`*object=''*`)`
class `aiohttp.`**`upstr`**`(`*bytes_or_buffer*`[`, *encoding*`[`, *errors*`]]``)`

Create a new **upper cased** string object from the given *object*. If *encoding* or *errors* are specified, then the object must expose a data buffer that will be decoded using the given encoding and error handler.

Otherwise, returns the result of `object.__str__()` (if defined) or `repr(object)`.

encoding defaults to `sys.getdefaultencoding()`.

errors defaults to `'strict'`.

The class is inherited from `str` and has all regular string methods.

9.8 Working with Multipart

aiohttp supports a full featured multipart reader and writer. Both are designed with steaming processing in mind to avoid unwanted footprint which may be significant if you're dealing with large payloads, but this also means that most I/O operation are only possible to be executed a single time.

9.8.1 Reading Multipart Responses

Assume you made a request, as usual, and want to process the response multipart data:

```
async with aiohttp.request(...) as resp:
    pass
```

First, you need to wrap the response with a *MultipartReader.from_response()*. This needs to keep the implementation of *MultipartReader* separated from the response and the connection routines which makes it more portable:

```
reader = aiohttp.MultipartReader.from_response(resp)
```

Let's assume with this response you'd received some JSON document and multiple files for it, but you don't need all of them, just a specific one.

So first you need to enter into a loop where the multipart body will be processed:

```
metadata = None
filedata = None
while True:
    part = await reader.next()
```

The returned type depends on what the next part is: if it's a simple body part then you'll get *BodyPartReader* instance here, otherwise, it will be another *MultipartReader* instance for the nested multipart. Remember, that multipart format is recursive and supports multiple levels of nested body parts. When there are no more parts left to fetch, `None` value will be returned - that's the signal to break the loop:

```
if part is None:
    break
```

Both *BodyPartReader* and *MultipartReader* provides access to body part headers: this allows you to filter parts by their attributes:

```
if part.headers[aiohttp.hdrs.CONTENT-TYPE] == 'application/json':
    metadata = await part.json()
    continue
```

Nor *BodyPartReader* or *MultipartReader* instances doesn't read the whole body part data without explicitly asking for. *BodyPartReader* provides a set of helpers methods to fetch popular content types in friendly way:

- *BodyPartReader.text()* for plain text data;
- *BodyPartReader.json()* for JSON;
- *BodyPartReader.form()* for *application/www-urlform-encode*

Each of these methods automatically recognizes if content is compressed by using *gzip* and *deflate* encoding (while it respects *identity* one), or if transfer encoding is base64 or *quoted-printable* - in each case the result will get automatically decoded. But in case you need to access to raw binary data as it is, there are *BodyPartReader.read()* and *BodyPartReader.read_chunk()* coroutine methods as well to read raw binary data as it is all-in-single-shot or by chunks respectively.

When you have to deal with multipart files, the *BodyPartReader.filename* property comes to help. It's a very smart helper which handles *Content-Disposition* handler right and extracts the right filename attribute from it:

```
if part.filename != 'secret.txt':
    continue
```

If current body part doesn't matches your expectation and you want to skip it - just continue a loop to start a next iteration of it. Here is where magic happens. Before fetching the next body part `await reader.next()` it ensures that the previous one was read completely. If it wasn't, all its content sends to the void in term to fetch the next part. So you don't have to care about cleanup routines while you're within a loop.

Once you'd found a part for the file you'd searched for, just read it. Let's handle it as it is without applying any decoding magic:

```
filedata = await part.read(decode=False)
```

Later you may decide to decode the data. It's still simple and possible to do:

```
filedata = part.decode(filedata)
```

Once you are done with multipart processing, just break a loop:

```
break
```

9.8.2 Sending Multipart Requests

MultipartWriter provides an interface to build multipart payload from the Python data and serialize it into chunked binary stream. Since multipart format is recursive and supports deeply nesting, you can use `with` statement to design your multipart data closer to how it will be:

```
with aiohttp.MultipartWriter('mixed') as mpwriter:
    ...
    with aiohttp.MultipartWriter('related') as subwriter:
        ...
    mpwriter.append(subwriter)
```

```
    with aiohttp.MultipartWriter('related') as subwriter:
        ...
        with aiohttp.MultipartWriter('related') as subsubwriter:
            ...
        subwriter.append(subsubwriter)
    mpwriter.append(subwriter)

    with aiohttp.MultipartWriter('related') as subwriter:
        ...
    mpwriter.append(subwriter)
```

The *MultipartWriter.append()* is used to join new body parts into a single stream. It accepts various inputs and determines what default headers should be used for.

For text data default *Content-Type* is *text/plain; charset=utf-8*:

```
mpwriter.append('hello')
```

For binary data *application/octet-stream* is used:

```
mpwriter.append(b'aiohttp')
```

You can always override these default by passing your own headers with the second argument:

```
mpwriter.append(io.BytesIO(b'GIF89a...'),
                {'CONTENT-TYPE': 'image/gif'})
```

For file objects *Content-Type* will be determined by using Python's mimetypes module and additionally *Content-Disposition* header will include the file's basename:

```
part = root.append(open(__file__, 'rb'))
```

If you want to send a file with a different name, just handle the *BodyPartWriter* instance which *MultipartWriter.append()* will always return and set *Content-Disposition* explicitly by using the *BodyPartWriter.set_content_disposition()* helper:

```
part.set_content_disposition('attachment', filename='secret.txt')
```

Additionally, you may want to set other headers here:

```
part.headers[aiohttp.hdrs.CONTENT_ID] = 'X-12345'
```

If you'd set *Content-Encoding*, it will be automatically applied to the data on serialization (see below):

```
part.headers[aiohttp.hdrs.CONTENT_ENCODING] = 'gzip'
```

There are also *MultipartWriter.append_json()* and *MultipartWriter.append_form()* helpers which are useful to work with JSON and form urlencoded data, so you don't have to encode it every time manually:

```
mpwriter.append_json({'test': 'passed'})
mpwriter.append_form([('key', 'value')])
```

When it's done, to make a request just pass a root *MultipartWriter* instance as `aiohttp.client.request()` *data* argument:

```
await aiohttp.post('http://example.com', data=mpwriter)
```

Behind the scenes *MultipartWriter.serialize()* will yield chunks of every part and if body part has *Content-Encoding* or *Content-Transfer-Encoding* they will be applied on streaming content.

Please note, that on *MultipartWriter.serialize()* all the file objects will be read until the end and there is no way to repeat a request without rewinding their pointers to the start.

9.8.3 Hacking Multipart

The Internet is full of terror and sometimes you may find a server which implements multipart support in strange ways when an oblivious solution doesn't work.

For instance, is server used cgi.FieldStorage then you have to ensure that no body part contains a *Content-Length* header:

```
for part in mpwriter:
    part.headers.pop(aiohttp.hdrs.CONTENT_LENGTH, None)
```

On the other hand, some server may require to specify *Content-Length* for the whole multipart request. *aiohttp* doesn't do that since it sends multipart using chunked transfer encoding by default. To overcome this issue, you have to serialize a *MultipartWriter* by our own in the way to calculate its size:

```
body = b''.join(mpwriter.serialize())
await aiohttp.post('http://example.com',
                   data=body, headers=mpwriter.headers)
```

Sometimes the server response may not be well formed: it may or may not contains nested parts. For instance, we request a resource which returns JSON documents with the files attached to it. If the document has any attachments, they are returned as a nested multipart. If it has not it responds as plain body parts:

```
CONTENT-TYPE: multipart/mixed; boundary=--:

--:
CONTENT-TYPE: application/json

{"_id": "foo"}
--:
CONTENT-TYPE: multipart/related; boundary=----:

----:
CONTENT-TYPE: application/json

{"_id": "bar"}
----:
CONTENT-TYPE: text/plain
CONTENT-DISPOSITION: attachment; filename=bar.txt

bar! bar! bar!
----:--
--:
CONTENT-TYPE: application/json

{"_id": "boo"}
--:
CONTENT-TYPE: multipart/related; boundary=----:

----:
CONTENT-TYPE: application/json

{"_id": "baz"}
----:
CONTENT-TYPE: text/plain
```

```
CONTENT-DISPOSITION: attachment; filename=baz.txt

baz! baz! baz!
----:--
--:--
```

Reading such kind of data in single stream is possible, but is not clean at all:

```
result = []
while True:
    part = await reader.next()

    if part is None:
        break

    if isinstance(part, aiohttp.MultipartReader):
        # Fetching files
        while True:
            filepart = await part.next()
            if filepart is None:
                break
            result[-1].append((await filepart.read()))

    else:
        # Fetching document
        result.append([(await part.json())])
```

Let's hack a reader in the way to return pairs of document and reader of the related files on each iteration:

```
class PairsMultipartReader(aiohttp.MultipartReader):

    # keep reference on the original reader
    multipart_reader_cls = aiohttp.MultipartReader

    async def next(self):
        """Emits a tuple of document object (:class:`dict`) and multipart
        reader of the followed attachments (if any).

        :rtype: tuple
        """
        reader = await super().next()

        if self._at_eof:
            return None, None

        if isinstance(reader, self.multipart_reader_cls):
            part = await reader.next()
            doc = await part.json()
        else:
            doc = await reader.json()

        return doc, reader
```

And this gives us a more cleaner solution:

```
reader = PairsMultipartReader.from_response(resp)
result = []
while True:
    doc, files_reader = await reader.next()
```

```
    if doc is None:
        break

    files = []
    while True:
        filepart = await files_reader.next()
        if file.part is None:
            break
        files.append((await filepart.read()))

    result.append((doc, files))
```

See also:

Multipart API in *Helpers API* section.

9.9 Helpers API

All public names from submodules `errors`, `multipart`, `parsers`, `protocol`, `utils`, `websocket` and `wsgi` are exported into `aiohttp` namespace.

9.9.1 aiohttp.errors module

http related errors.

exception `aiohttp.errors.`**DisconnectedError**
Bases: `Exception`

Disconnected.

exception `aiohttp.errors.`**ClientDisconnectedError**
Bases: *aiohttp.errors.DisconnectedError*

Client disconnected.

exception `aiohttp.errors.`**ServerDisconnectedError**
Bases: *aiohttp.errors.DisconnectedError*

Server disconnected.

exception `aiohttp.errors.`**HttpProcessingError**(*, *code=None*, *message=''*, *headers=None*)
Bases: `Exception`

Http error.

Shortcut for raising http errors with custom code, message and headers.

Parameters

- **code** (*int*) – HTTP Error code.
- **message** (*str*) – (optional) Error message.
- **of [tuple] headers** (*list*) – (optional) Headers to be sent in response.

code = 0

headers = None

message = ''

exception aiohttp.errors.**BadHttpMessage**(*message*, *, *headers=None*)
Bases: *aiohttp.errors.HttpProcessingError*

code = 400

message = 'Bad Request'

exception aiohttp.errors.**HttpMethodNotAllowed**(*, *code=None*, *message=''*, *headers=None*)
Bases: *aiohttp.errors.HttpProcessingError*

code = 405

message = 'Method Not Allowed'

exception aiohttp.errors.**HttpBadRequest**(*message*, *, *headers=None*)
Bases: *aiohttp.errors.BadHttpMessage*

code = 400

message = 'Bad Request'

exception aiohttp.errors.**HttpProxyError**(*, *code=None*, *message=''*, *headers=None*)
Bases: *aiohttp.errors.HttpProcessingError*

Http proxy error.

Raised in aiohttp.connector.ProxyConnector if proxy responds with status other than 200 OK on CONNECT request.

exception aiohttp.errors.**BadStatusLine**(*line=''*)
Bases: *aiohttp.errors.BadHttpMessage*

exception aiohttp.errors.**LineTooLong**(*line*, *limit='Unknown'*)
Bases: *aiohttp.errors.BadHttpMessage*

exception aiohttp.errors.**InvalidHeader**(*hdr*)
Bases: *aiohttp.errors.BadHttpMessage*

exception aiohttp.errors.**ClientError**
Bases: Exception

Base class for client connection errors.

exception aiohttp.errors.**ClientHttpProcessingError**
Bases: *aiohttp.errors.ClientError*

Base class for client http processing errors.

exception aiohttp.errors.**ClientConnectionError**
Bases: *aiohttp.errors.ClientError*

Base class for client socket errors.

exception aiohttp.errors.**ClientOSError**
Bases: *aiohttp.errors.ClientConnectionError*, OSError

OSError error.

exception aiohttp.errors.**ClientTimeoutError**
Bases: *aiohttp.errors.ClientConnectionError*, concurrent.futures._base.TimeoutError

Client connection timeout error.

exception aiohttp.errors.**ProxyConnectionError**
Bases: *aiohttp.errors.ClientConnectionError*

Proxy connection error.

Raised in `aiohttp.connector.ProxyConnector` if connection to proxy can not be established.

exception `aiohttp.errors.`**`ClientRequestError`**
Bases: *aiohttp.errors.ClientHttpProcessingError*

Connection error during sending request.

exception `aiohttp.errors.`**`ClientResponseError`**
Bases: *aiohttp.errors.ClientHttpProcessingError*

Connection error during reading response.

exception `aiohttp.errors.`**`FingerprintMismatch`**(*expected*, *got*, *host*, *port*)
Bases: *aiohttp.errors.ClientConnectionError*

SSL certificate does not match expected fingerprint.

exception `aiohttp.errors.`**`WSServerHandshakeError`**(*message*, *, *headers=None*)
Bases: *aiohttp.errors.HttpProcessingError*

websocket server handshake error.

exception `aiohttp.errors.`**`WSClientDisconnectedError`**
Bases: *aiohttp.errors.ClientDisconnectedError*

Deprecated.

9.9.2 aiohttp.helpers module

Various helper functions

class `aiohttp.helpers.`**`FormData`**(*fields=()*)
Bases: `object`

Helper class for multipart/form-data and application/x-www-form-urlencoded body generation.

`add_field`(*name*, *value*, *, *content_type=None*, *filename=None*, *content_transfer_encoding=None*)

`add_fields`(**fields*)

`content_type`

`is_multipart`

`aiohttp.helpers.`**`parse_mimetype`**(*mimetype*)
Parses a MIME type into its components.

Parameters **`mimetype`** (*str*) – MIME type

Returns 4 element tuple for MIME type, subtype, suffix and parameters

Return type tuple

Example:

```
>>> parse_mimetype('text/html; charset=utf-8')
('text', 'html', '', {'charset': 'utf-8'})
```

class `aiohttp.helpers.`**`Timeout`**(*timeout*, *, *loop=None*)
Bases: `object`

Timeout context manager.

Useful in cases when you want to apply timeout logic around block of code or in cases when asyncio.wait_for is not suitable. For example:

```
>>> with aiohttp.Timeout(0.001):
>>>     async with aiohttp.get('https://github.com') as r:
>>>         await r.text()
```

Parameters

- **timeout** – timeout value in seconds
- **loop** – asyncio compatible event loop

9.9.3 aiohttp.multipart module

class aiohttp.multipart.**MultipartReader**(*headers*, *content*)

Bases: object

Multipart body reader.

at_eof()

Returns True if the final boundary was reached or False otherwise.

Return type bool

fetch_next_part()

Returns the next body part reader.

classmethod from_response(*response*)

Constructs reader instance from HTTP response.

Parameters **response** – ClientResponse instance

multipart_reader_cls = None

Multipart reader class, used to handle multipart/* body parts. None points to type(self)

next()

Emits the next multipart body part.

part_reader_cls

Body part reader class for non multipart/* content types.

alias of *BodyPartReader*

release()

Reads all the body parts to the void till the final boundary.

response_wrapper_cls

Response wrapper, used when multipart readers constructs from response.

alias of MultipartResponseWrapper

class aiohttp.multipart.**MultipartWriter**(*subtype='mixed'*, *boundary=None*)

Bases: object

Multipart body writer.

append(*obj*, *headers=None*)

Adds a new body part to multipart writer.

append_form(*obj*, *headers=None*)

Helper to append form urlencoded part.

append_json(*obj*, *headers=None*)

Helper to append JSON part.

boundary

part_writer_cls
Body part reader class for non multipart/* content types.

alias of *BodyPartWriter*

serialize()
Yields multipart byte chunks.

class aiohttp.multipart.**BodyPartReader**(*boundary*, *headers*, *content*)
Bases: object

Multipart reader for single body part.

at_eof()
Returns True if the boundary was reached or False otherwise.

Return type bool

chunk_size = 8192

decode(*data*)
Decodes data according the specified *Content-Encoding* or *Content-Transfer-Encoding* headers value.

Supports gzip, deflate and identity encodings for *Content-Encoding* header.

Supports base64, quoted-printable encodings for *Content-Transfer-Encoding* header.

Parameters **data** (*bytearray*) – Data to decode.

Raises RuntimeError - if encoding is unknown.

Return type bytes

filename
Returns filename specified in Content-Disposition header or None if missed or header is malformed.

form(*, *encoding=None*)
Lke *read()*, but assumes that body parts contains form urlencoded data.

Parameters **encoding** (*str*) – Custom form encoding. Overrides specified in charset param of *Content-Type* header

get_charset(*default=None*)
Returns charset parameter from Content-Type header or default.

json(*, *encoding=None*)
Lke *read()*, but assumes that body parts contains JSON data.

Parameters **encoding** (*str*) – Custom JSON encoding. Overrides specified in charset param of *Content-Type* header

next()

read(*, *decode=False*)
Reads body part data.

Parameters **decode** (*bool*) – Decodes data following by encoding method from *Content-Encoding* header. If it missed data remains untouched

Return type bytearray

read_chunk(*size=8192*)
Reads body part content chunk of the specified size. The body part must has *Content-Length* header with proper value.

Parameters **size** (*int*) – chunk size

Return type bytearray

readline()
Reads body part by line by line.

Return type bytearray

release()
Lke *read()*, but reads all the data to the void.

Return type None

text(*, *encoding=None*)
Lke *read()*, but assumes that body part contains text data.

Parameters **encoding** (*str*) – Custom text encoding. Overrides specified in charset param of *Content-Type* header

Return type str

class aiohttp.multipart.**BodyPartWriter**(*obj*, *headers=None*, *, *chunk_size=8192*)
Bases: object

Multipart writer for single body part.

filename
Returns filename specified in Content-Disposition header or None if missed.

serialize()
Yields byte chunks for body part.

set_content_disposition(*disptype*, **params*)
Sets Content-Disposition header.

Parameters

- **disptype** (*str*) – Disposition type: inline, attachment, form-data. Should be valid extension token (see RFC 2183)
- **params** (*dict*) – Disposition params

exception aiohttp.multipart.**BadContentDispositionHeader**
Bases: RuntimeWarning

exception aiohttp.multipart.**BadContentDispositionParam**
Bases: RuntimeWarning

aiohttp.multipart.**parse_content_disposition**(*header*)

aiohttp.multipart.**content_disposition_filename**(*params*)

9.9.4 aiohttp.parsers module

Parser is a generator function (NOT coroutine).

Parser receives data with generator's send() method and sends data to destination DataQueue. Parser receives ParserBuffer and DataQueue objects as a parameters of the parser call, all subsequent send() calls should send bytes objects. Parser sends parsed *term* to destination buffer with DataQueue.feed_data() method. DataQueue object should implement two methods. feed_data() - parser uses this method to send parsed protocol data. feed_eof() - parser uses this method for indication of end of parsing stream. To indicate end of incoming data stream EofStream exception should be sent into parser. Parser could throw exceptions.

There are three stages:

- Data flow chain:
    1. Application creates StreamParser object for storing incoming data.
    2. StreamParser creates ParserBuffer as internal data buffer.
    3. Application create parser and set it into stream buffer:

        parser = HttpRequestParser() data_queue = stream.set_parser(parser)

    3. At this stage StreamParser creates DataQueue object and passes it and internal buffer into parser as an arguments.

        def set_parser(self, parser): output = DataQueue() self.p = parser(output, self._input) return output

    4. Application waits data on output.read()

        while True: msg = yield from output.read() ...

- Data flow:
    1. asyncio's transport reads data from socket and sends data to protocol with data_received() call.
    2. Protocol sends data to StreamParser with feed_data() call.
    3. StreamParser sends data into parser with generator's send() method.
    4. Parser processes incoming data and sends parsed data to DataQueue with feed_data()
    5. Application received parsed data from DataQueue.read()
- Eof:
    1. StreamParser receives eof with feed_eof() call.
    2. StreamParser throws EofStream exception into parser.
    3. Then it unsets parser.

_SocketSocketTransport -> -> "protocol" -> StreamParser -> "parser" -> DataQueue <- "application"

exception aiohttp.parsers.**EofStream**
Bases: Exception

eof stream indication.

class aiohttp.parsers.**StreamParser**(*, *loop=None*, *buf=None*, *limit=65536*, *eof_exc_class=<class 'RuntimeError'>*, ***kwargs*)
Bases: object

StreamParser manages incoming bytes stream and protocol parsers.

StreamParser uses ParserBuffer as internal buffer.

set_parser() sets current parser, it creates DataQueue object and sends ParserBuffer and DataQueue into parser generator.

unset_parser() sends EofStream into parser and then removes it.

at_eof()

exception()

feed_data(*data*)
send data to current parser or store in buffer.

feed_eof()
send eof to all parsers, recursively.

output

set_exception(*exc*)

set_parser(*parser*, *output=None*)
set parser to stream. return parser's DataQueue.

set_transport(*transport*)

unset_parser()
unset parser, send eof to the parser and then remove it.

class aiohttp.parsers.**StreamProtocol**(*, *loop=None*, *disconnect_error=<class 'RuntimeError'>*, ***kwargs*)
Bases: asyncio.streams.FlowControlMixin, asyncio.protocols.Protocol

Helper class to adapt between Protocol and StreamReader.

connection_lost(*exc*)

connection_made(*transport*)

data_received(*data*)

eof_received()

is_connected()

class aiohttp.parsers.**ParserBuffer**(**args*)
Bases: object

ParserBuffer is NOT a bytearray extension anymore.

ParserBuffer provides helper methods for parsers.

exception()

extend(*data*)

feed_data(*data*)

read(*size*)
read() reads specified amount of bytes.

readsome(*size=None*)
reads size of less amount of bytes.

readuntil(*stop*, *limit=None*)

set_exception(*exc*)

skip(*size*)
skip() skips specified amount of bytes.

skipuntil(*stop*)
skipuntil() reads until *stop* bytes sequence.

wait(*size*)
wait() waits for specified amount of bytes then returns data without changing internal buffer.

waituntil(*stop*, *limit=None*)
waituntil() reads until *stop* bytes sequence.

class aiohttp.parsers.**LinesParser**(*limit=65536*)
Bases: object

Lines parser.

Lines parser splits a bytes stream into a chunks of data, each chunk ends with n symbol.

class aiohttp.parsers.**ChunksParser**(*size=8192*)
Bases: object

Chunks parser.

Chunks parser splits a bytes stream into a specified size chunks of data.

9.9.5 aiohttp.protocol module

Http related parsers and protocol.

class aiohttp.protocol.**HttpMessage**(*transport*, *version*, *close*)
Bases: object

HttpMessage allows to write headers and payload to a stream.

For example, lets say we want to read file then compress it with deflate compression and then send it with chunked transfer encoding, code may look like this:

```
>>> response = aiohttp.Response(transport, 200)
```

We have to use deflate compression first:

```
>>> response.add_compression_filter('deflate')
```

Then we want to split output stream into chunks of 1024 bytes size:

```
>>> response.add_chunking_filter(1024)
```

We can add headers to response with add_headers() method. add_headers() does not send data to transport, send_headers() sends request/response line and then sends headers:

```
>>> response.add_headers(
...     ('Content-Disposition', 'attachment; filename="..."'))
>>> response.send_headers()
```

Now we can use chunked writer to write stream to a network stream. First call to write() method sends response status line and headers, add_header() and add_headers() method unavailable at this stage:

```
>>> with open('...', 'rb') as f:
...     chunk = fp.read(8192)
...     while chunk:
...         response.write(chunk)
...         chunk = fp.read(8192)
```

```
>>> response.write_eof()
```

HOP_HEADERS = None

SERVER_SOFTWARE = 'Python/3.4 aiohttp/0.19.0'

add_chunking_filter(*chunk_size=16384*, ***, *EOF_MARKER=<object object>*, *EOL_MARKER=<object object>*)
Split incoming stream into chunks.

add_compression_filter(*encoding='deflate'*, ***, *EOF_MARKER=<object object>*, *EOL_MARKER=<object object>*)
Compress incoming stream with deflate or gzip encoding.

add_header(*name*, *value*)
Analyze headers. Calculate content length, removes hop headers, etc.

add_headers(**headers*)
Adds headers to a http message.

body_length

enable_chunked_encoding()

filter = None

force_close()

has_chunked_hdr = False

is_headers_sent()

keep_alive()

send_headers(*_sep=': '*, *_end='\r\n'*)
Writes headers to a stream. Constructs payload writer.

status = None

status_line = b''

upgrade = False

websocket = False

write(*chunk*, ***, *drain=False*, *EOF_MARKER=<object object>*, *EOL_MARKER=<object object>*)
Writes chunk of data to a stream by using different writers.

writer uses filter to modify chunk of data. write_eof() indicates end of stream. writer can't be used after write_eof() method being called. write() return drain future.

write_eof()

writer = None

class aiohttp.protocol.**Request**(*transport*, *method*, *path*, *http_version=HttpVersion(major=1, minor=1)*, *close=False*)
Bases: *aiohttp.protocol.HttpMessage*

HOP_HEADERS = ()

class aiohttp.protocol.**Response**(*transport*, *status*, *http_version=HttpVersion(major=1, minor=1)*, *close=False*, *reason=None*)
Bases: *aiohttp.protocol.HttpMessage*

Create http response message.

Transport is a socket stream transport. status is a response status code, status has to be integer value. http_version is a tuple that represents http version, (1, 0) stands for HTTP/1.0 and (1, 1) is for HTTP/1.1

HOP_HEADERS = ()

static **calc_reason**(*status*, ***, *_RESPONSES={428: ('Precondition Required', 'The origin server requires the request to be conditional.'), 301: ('Moved Permanently', 'Object moved permanently – see URI list'), 400: ('Bad Request', 'Bad request syntax or unsupported method'), 401: ('Unauthorized', 'No permission – see authorization schemes'), 402: ('Payment Required', 'No payment – see charging schemes'), 403: ('Forbidden', 'Request forbidden – authorization will not help'), 404: ('Not Found', 'Nothing matches the given URI'), 405: ('Method Not Allowed', 'Specified method is invalid for this resource.'), 406: ('Not Acceptable', 'URI not available in preferred format.'), 407: ('Proxy Authentication Required', 'You must authenticate with this proxy before proceeding.'), 408: ('Request Timeout', 'Request timed out; try again later.'), 409: ('Conflict', 'Request conflict.'), 410: ('Gone', 'URI no longer exists and has been permanently removed.'), 411: ('Length Required', 'Client must specify Content-Length.'), 412: ('Precondition Failed', 'Precondition in headers is false.'), 413: ('Request Entity Too Large', 'Entity is too large.'), 414: ('Request-URI Too Long', 'URI is too long.'), 415: ('Unsupported Media Type', 'Entity body in unsupported format.'), 416: ('Requested Range Not Satisfiable', 'Cannot satisfy request range.'), 417: ('Expectation Failed', 'Expect condition could not be satisfied.'), 300: ('Multiple Choices', 'Object has several resources – see URI list'), 429: ('Too Many Requests', 'The user has sent too many requests in a given amount of time ("rate limiting").'), 302: ('Found', 'Object moved temporarily – see URI list'), 431: ('Request Header Fields Too Large', 'The server is unwilling to process the request because its header fields are too large.'), 304: ('Not Modified', 'Document has not changed since given time'), 305: ('Use Proxy', 'You must use proxy specified in Location to access this resource.'), 307: ('Temporary Redirect', 'Object moved temporarily – see URI list'), 200: ('OK', 'Request fulfilled, document follows'), 201: ('Created', 'Document created, URL follows'), 202: ('Accepted', 'Request accepted, processing continues off-line'), 203: ('Non-Authoritative Information', 'Request fulfilled from cache'), 204: ('No Content', 'Request fulfilled, nothing follows'), 205: ('Reset Content', 'Clear input form for further input.'), 206: ('Partial Content', 'Partial content follows.'), 303: ('See Other', 'Object moved – see Method and URL list'), 100: ('Continue', 'Request received, please continue'), 101: ('Switching Protocols', 'Switching to new protocol; obey Upgrade header'), 500: ('Internal Server Error', 'Server got itself in trouble'), 501: ('Not Implemented', 'Server does not support this operation'), 502: ('Bad Gateway', 'Invalid responses from another server/proxy.'), 503: ('Service Unavailable', 'The server cannot process the request due to a high load'), 504: ('Gateway Timeout', 'The gateway server did not receive a timely response'), 505: ('HTTP Version Not Supported', 'Cannot fulfill request.'), 511: ('Network Authentication Required', 'The client needs to authenticate to gain network access.')}*)

class aiohttp.protocol.**HttpVersion**(*major*, *minor*)

Bases: tuple

major

Alias for field number 0

minor

Alias for field number 1

class aiohttp.protocol.**RawRequestMessage**(*method*, *path*, *version*, *headers*, *should_close*, *compression*)

Bases: tuple

compression

Alias for field number 5

headers
Alias for field number 3

method
Alias for field number 0

path
Alias for field number 1

should_close
Alias for field number 4

version
Alias for field number 2

class aiohttp.protocol.**RawResponseMessage**(*version*, *code*, *reason*, *headers*, *should_close*, *compression*)
Bases: tuple

code
Alias for field number 1

compression
Alias for field number 5

headers
Alias for field number 3

reason
Alias for field number 2

should_close
Alias for field number 4

version
Alias for field number 0

class aiohttp.protocol.**HttpPrefixParser**(*allowed_methods=()*)
Bases: object

Waits for 'HTTP' prefix (non destructive)

class aiohttp.protocol.**HttpRequestParser**(*max_line_size=8190*, *max_headers=32768*, *max_field_size=8190*)
Bases: aiohttp.protocol.HttpParser

Read request status line. Exception errors.BadStatusLine could be raised in case of any errors in status line. Returns RawRequestMessage.

class aiohttp.protocol.**HttpResponseParser**(*max_line_size=8190*, *max_headers=32768*, *max_field_size=8190*)
Bases: aiohttp.protocol.HttpParser

Read response status line and headers.

BadStatusLine could be raised in case of any errors in status line. Returns RawResponseMessage

class aiohttp.protocol.**HttpPayloadParser**(*message*, *length=None*, *compression=True*, *readall=False*, *response_with_body=True*)
Bases: object

parse_chunked_payload(*out*, *buf*)
Chunked transfer encoding parser.

parse_eof_payload(*out*, *buf*)
Read all bytes until eof.

parse_length_payload(*out*, *buf*, *length=0*)
Read specified amount of bytes.

9.9.6 aiohttp.signals module

class aiohttp.signals.**BaseSignal**
Bases: list

copy()

sort()

class aiohttp.signals.**DebugSignal**
Bases: *aiohttp.signals.BaseSignal*

send(*ordinal*, *name*, **args*, ***kwargs*)

class aiohttp.signals.**PostSignal**
Bases: *aiohttp.signals.DebugSignal*

class aiohttp.signals.**PreSignal**
Bases: *aiohttp.signals.DebugSignal*

ordinal()

class aiohttp.signals.**Signal**(*app*)
Bases: *aiohttp.signals.BaseSignal*

Coroutine-based signal implementation.

To connect a callback to a signal, use any list method.

Signals are fired using the *send()* coroutine, which takes named arguments.

send(**args*, ***kwargs*)
Sends data to all registered receivers.

9.9.7 aiohttp.streams module

exception aiohttp.streams.**EofStream**
Bases: Exception

eof stream indication.

class aiohttp.streams.**StreamReader**(*limit=65536*, *loop=None*)
Bases: asyncio.streams.StreamReader, aiohttp.streams.AsyncStreamReaderMixin

An enhancement of asyncio.StreamReader.

Supports asynchronous iteration by line, chunk or as available:

```
async for line in reader:
    ...
async for chunk in reader.iter_chunked(1024):
    ...
async for slice in reader.iter_any():
    ...
```

at_eof()
Return True if the buffer is empty and 'feed_eof' was called.

exception()

feed_data(*data*)

feed_eof()

is_eof()
Return True if 'feed_eof' was called.

read(*n=-1*)

read_nowait()

readany()

readexactly(*n*)

readline()

set_exception(*exc*)

total_bytes = 0

wait_eof()

class aiohttp.streams.**DataQueue**(*, *loop=None*)
Bases: object

DataQueue is a general-purpose blocking queue with one reader.

at_eof()

exception()

feed_data(*data*, *size=0*)

feed_eof()

is_eof()

read()

set_exception(*exc*)

class aiohttp.streams.**ChunksQueue**(*, *loop=None*)
Bases: *aiohttp.streams.DataQueue*

Like a *DataQueue*, but for binary chunked data transfer.

read()

readany()

class aiohttp.streams.**FlowControlStreamReader**(*stream*, *limit=65536*, **args*, ***kwargs*)
Bases: *aiohttp.streams.StreamReader*

feed_data(*data*, *size=0*)

read(*n=-1*)

readany()

readexactly(*n*)

readline()

class aiohttp.streams.**FlowControlDataQueue**(*stream*, ***, *limit=65536*, *loop=None*)
Bases: *aiohttp.streams.DataQueue*

FlowControlDataQueue resumes and pauses an underlying stream.

It is a destination for parsed data.

feed_data(*data*, *size*)

read()

class aiohttp.streams.**FlowControlChunksQueue**(*stream*, ***, *limit=65536*, *loop=None*)
Bases: *aiohttp.streams.FlowControlDataQueue*

read()

readany()

9.9.8 aiohttp.websocket module

WebSocket protocol versions 13 and 8.

aiohttp.websocket.**WebSocketParser**(*out*, *buf*)

class aiohttp.websocket.**WebSocketWriter**(*writer*, ***, *use_mask=False*, *random=<random.Random object at 0x25d5498>*)
Bases: object

close(*code=1000*, *message=b''*)
Close the websocket, sending the specified code and message.

ping(*message=b''*)
Send ping message.

pong(*message=b''*)
Send pong message.

send(*message*, *binary=False*)
Send a frame over the websocket with message as its payload.

aiohttp.websocket.**do_handshake**(*method*, *headers*, *transport*, *protocols=()*)
Prepare WebSocket handshake.

It return http response code, response headers, websocket parser, websocket writer. It does not perform any IO.

protocols is a sequence of known protocols. On successful handshake, the returned response headers contain the first protocol in this list which the server also knows.

class aiohttp.websocket.**Message**(*tp*, *data*, *extra*)
Bases: tuple

data
Alias for field number 1

extra
Alias for field number 2

tp
Alias for field number 0

exception aiohttp.websocket.**WebSocketError**(*code*, *message*)
Bases: Exception

WebSocket protocol parser error.

9.9.9 aiohttp.wsgi module

wsgi server.

TODO:

- proxy protocol
- x-forward security
- wsgi file support (os.sendfile)

class aiohttp.wsgi.**WSGIServerHttpProtocol**(*app*, *readpayload=False*, *is_ssl=False*, **args*, ***kw*)

Bases: *aiohttp.server.ServerHttpProtocol*

HTTP Server that implements the Python WSGI protocol.

It uses 'wsgi.async' of 'True'. 'wsgi.input' can behave differently depends on 'readpayload' constructor parameter. If readpayload is set to True, wsgi server reads all incoming data into BytesIO object and sends it as 'wsgi.input' environ var. If readpayload is set to false 'wsgi.input' is a StreamReader and application should read incoming data with "yield from environ['wsgi.input'].read()". It defaults to False.

SCRIPT_NAME = ''

create_wsgi_environ(*message*, *payload*)

create_wsgi_response(*message*)

handle_request(*message*, *payload*)
Handle a single HTTP request

9.10 Logging

aiohttp uses standard `logging` for tracking the library activity.

We have the following loggers enumerated by names:

- `'aiohttp.client'`
- `'aiohttp.internal'`
- `'aiohttp.server'`
- `'aiohttp.web'`
- `'aiohttp.websocket'`

You may subscribe to these loggers for getting logging messages. The page does not provide instructions for logging subscribing while the most friendly method is `logging.config.dictConfig()` for configuring whole loggers in your application.

9.10.1 Access logs

Access log is enabled by specifying *access_log* parameter (`logging.Logger` instance) on *aiohttp.web.Application.make_handler()* call.

Optional *access_log_format* parameter may be used for specifying log format (see below).

Note: Access log is disabled by default.

Format specification.

The library provides custom micro-language to specifying info about request and response:

Option	Meaning
`%%`	The percent sign
`%a`	Remote IP-address (IP-address of proxy if using reverse proxy)
`%t`	Time when the request was started to process
`%P`	The process ID of the child that serviced the request
`%r`	First line of request
`%s`	Response status code
`%b`	Size of response in bytes, excluding HTTP headers
`%O`	Bytes sent, including headers
`%T`	The time taken to serve the request, in seconds
`%D`	The time taken to serve the request, in microseconds
`%{FOO}i`	`request.headers['FOO']`
`%{FOO}o`	`response.headers['FOO']`
`%{FOO}e`	`os.environ['FOO']`

Default access log format is:

```
'%a %l %u %t "%r" %s %b "%{Referrer}i" "%{User-Agent}i"'
```

9.10.2 Error logs

aiohttp.web uses logger named `'aiohttp.server'` to store errors given on web requests handling.

The log is always enabled.

To use different logger name please specify *logger* parameter (`logging.Logger` instance) onmake *`aiohttp.web.Application.make_handler()`* call.

9.11 Deployment using Gunicorn

aiohttp can be deployed using Gunicorn, which is based on a pre-fork worker model. Gunicorn launches your app as worker processes for handling incoming requests.

9.11.1 Prepare environment

You firstly need to setup your deployment environment. This example is based on Ubuntu 14.04.

Create a directory for your application:

```
>> mkdir myapp
>> cd myapp
```

Ubuntu has a bug in pyenv, so to create virtualenv you need to do some extra manipulation:

```
>> pyvenv-3.4 --without-pip venv
>> source venv/bin/activate
>> curl https://bootstrap.pypa.io/get-pip.py | python
>> deactivate
>> source venv/bin/activate
```

Now that the virtual environment is ready, we'll proceed to install aiohttp and gunicorn:

```
>> pip install gunicorn
>> pip install -e git+https://github.com/KeepSafe/aiohttp.git#egg=aiohttp
```

9.11.2 Application

Lets write a simple application, which we will save to file. We'll name this file *my_app_module.py*:

```
from aiohttp import web

def index(request):
    return web.Response(text="Welcome home!")


my_web_app = web.Application()
my_web_app.router.add_route('GET', '/', index)
```

9.11.3 Start Gunicorn

When Running Gunicorn, you provide the name of the module, i.e. *my_app_module*, and the name of the app, i.e. *my_web_app*, along with other Gunicorn Settings provided as command line flags or in your config file.

In this case, we will use:

- the *'–bind'* flag to set the server's socket address;
- the *'–worker-class'* flag to tell Gunicorn that we want to use a custom worker subclass instead of one of the Gunicorn default worker types;
- you may also want to use the *'–workers'* flag to tell Gunicorn how many worker processes to use for handling requests. (See the documentation for recommendations on How Many Workers?)

The custom worker subclass is defined in *aiohttp.worker.GunicornWebWorker* and should be used instead of the *gaiohttp* worker provided by Gunicorn, which supports only aiohttp.wsgi applications:

```
>> gunicorn my_app_module:my_web_app --bind localhost:8080 --worker-class aiohttp.worker.GunicornWebW
[2015-03-11 18:27:21 +0000] [1249] [INFO] Starting gunicorn 19.3.0
[2015-03-11 18:27:21 +0000] [1249] [INFO] Listening at: http://127.0.0.1:8080 (1249)
[2015-03-11 18:27:21 +0000] [1249] [INFO] Using worker: aiohttp.worker.GunicornWebWorker
[2015-03-11 18:27:21 +0000] [1253] [INFO] Booting worker with pid: 1253
```

Gunicorn is now running and ready to serve requests to your app's worker processes.

9.11.4 More information

The Gunicorn documentation recommends deploying Gunicorn behind a Nginx proxy server. See the official documentation for more information about suggested nginx configuration.

9.12 Contributing

9.12.1 Instructions for contributors

In order to make a clone of the GitHub repo: open the link and press the "Fork" button on the upper-right menu of the web page.

I hope everybody knows how to work with git and github nowadays :)

Workflow is pretty straightforward:

1. Clone the GitHub repo
2. Make a change
3. Make sure all tests passed
4. Commit changes to own aiohttp clone
5. Make pull request from github page for your clone

Note: If your PR has long history or many commits please rebase it from main repo before creating PR.

9.12.2 Preconditions for running aiohttp test suite

We expect you to use a python virtual environment to run our tests.

There are several ways to make a virtual environment.

If you like to use *virtualenv* please run:

```
$ cd aiohttp
$ virtualenv --python=`which python3` venv
```

For standard python *venv*:

```
$ cd aiohttp
$ python3 -m venv venv
```

For *virtualenvwrapper* (my choice):

```
$ cd aiohttp
$ mkvirtualenv --python=`which python3` aiohttp
```

There are other tools like *pyvenv* but you know the rule of thumb now: create a python3 virtual environment and activate it.

After that please install libraries required for development:

```
$ pip install -r requirements-dev.txt
```

We also recommend to install *ipdb* but it's on your own:

```
$ pip install ipdb
```

Congratulations, you are ready to run the test suite

9.12.3 Run aiohttp test suite

After all the preconditions are met you can run tests typing the next command:

```
$ make test
```

The command at first will run the *flake8* tool (sorry, we don't accept pull requests with pep8 or pyflakes errors).

On *flake8* success the tests will be run.

Please take a look on the produced output.

Any extra texts (print statements and so on) should be removed.

9.12.4 Tests coverage

We are trying hard to have good test coverage; please don't make it worse.

Use:

```
$ make cov
```

to run test suite and collect coverage information. Once the command has finished check your coverage at the file that appears in the last line of the output: `open file:///.../aiohttp/coverage/index.html`

Please go to the link and make sure that your code change is covered.

9.12.5 Documentation

We encourage documentation improvements.

Please before making a Pull Request about documentation changes run:

```
$ make doc
```

Once it finishes it will output the index html page `open file:///.../aiohttp/docs/_build/html/index.html`..

Go to the link and make sure your doc changes looks good.

9.12.6 The End

After finishing all steps make a GitHub Pull Request, thanks.

9.13 CHANGES

9.13.1 0.19.0 (11-25-2015)

- Memory leak in ParserBuffer #579
- Suppport gunicorn's *max_requests* settings in gunicorn worker
- Fix wsgi environment building #573
- Improve access logging #572
- Drop unused host and port from low-level server #586
- Add Python 3.5 *async for* implementation to server websocket #543
- Add Python 3.5 *async for* implementation to client websocket
- Add Python 3.5 *async with* implementation to client websocket
- Add charset parameter to web.Response constructor #593
- Forbid passing both Content-Type header and content_type or charset params into web.Response constructor
- Forbid duplicating of web.Application and web.Request #602
- Add an option to pass Origin header in ws_connect #607

- Add json_response function #592
- Make concurrent connections respect limits #581
- Collect history of responses if redirects occur #614
- Enable passing pre-compressed data in requests #621
- Expose named routes via UrlDispatcher.named_routes() #622
- Allow disabling sendfile by environment variable AIOHTTP_NOSENDFILE #629
- Use ensure_future if available
- Always quote params for Content-Disposition #641
- Support async for in multipart reader #640
- Add Timeout context manager #611

9.13.2 0.18.4 (13-11-2015)

- Relax rule for router names again by adding dash to allowed characters: they may contain identifiers, dashes, dots and columns

9.13.3 0.18.3 (25-10-2015)

- Fix formatting for _RequestContextManager helper #590

9.13.4 0.18.2 (22-10-2015)

- Fix regression for OpenSSL < 1.0.0 #583

9.13.5 0.18.1 (20-10-2015)

- Relax rule for router names: they may contain dots and columns starting from now

9.13.6 0.18.0 (19-10-2015)

- Use errors.HttpProcessingError.message as HTTP error reason and message #459
- Optimize cythonized multidict a bit
- Change repr's of multidicts and multidict views
- default headers in ClientSession are now case-insensitive
- Make '=' char and 'wss://' schema safe in urls #477
- *ClientResponse.close()* forces connection closing by default from now #479

  N.B. Backward incompatible change: was *.close(force=False) Using 'force* parameter for the method is deprecated: use *.release()* instead.
- Properly requote URL's path #480
- add *skip_auto_headers* parameter for client API #486
- Properly parse URL path in aiohttp.web.Request #489

- Raise RuntimeError when chunked enabled and HTTP is 1.0 #488
- Fix a bug with processing io.BytesIO as data parameter for client API #500
- Skip auto-generation of Content-Type header #507
- Use sendfile facility for static file handling #503
- Default *response_factory* in *app.router.add_static* now is *StreamResponse*, not *None*. The functionality is not changed if default is not specified.
- Drop *ClientResponse.message* attribute, it was always implementation detail.
- Streams are optimized for speed and mostly memory in case of a big HTTP message sizes #496
- Fix a bug for server-side cookies for dropping cookie and setting it again without Max-Age parameter.
- Don't trim redirect URL in client API #499
- Extend precision of access log "D" to milliseconds #527
- Deprecate *StreamResponse.start()* method in favor of *StreamResponse.prepare()* coroutine #525

  .start() is still supported but responses begun with *.start()* doesn't call signal for response preparing to be sent.
- Add *StreamReader.__repr__*
- Drop Python 3.3 support, from now minimal required version is Python 3.4.1 #541
- Add *async with* support for *ClientSession.request()* and family #536
- Ignore message body on 204 and 304 responses #505
- *TCPConnector* processed both IPv4 and IPv6 by default #559
- Add *.routes()* view for urldispatcher #519
- Route name should be a valid identifier name from now #567
- Implement server signals #562
- Drop an year-old deprecated *files* parameter from client API.
- Added *async for* support for aiohttp stream #542

9.13.7 0.17.4 (09-29-2015)

- Properly parse URL path in aiohttp.web.Request #489
- Add missing coroutine decorator, the client api is await-compatible now

9.13.8 0.17.3 (08-28-2015)

- Remove Content-Length header on compressed responses #450
- Support Python 3.5
- Improve performance of transport in-use list #472
- Fix connection pooling #473

9.13.9 0.17.2 (08-11-2015)

- Don't forget to pass *data* argument forward #462
- Fix multipart read bytes count #463

9.13.10 0.17.1 (08-10-2015)

- Fix multidict comparsion to arbitrary abc.Mapping

9.13.11 0.17.0 (08-04-2015)

- Make StaticRoute support Last-Modified and If-Modified-Since headers #386
- Add Request.if_modified_since and Stream.Response.last_modified properties
- Fix deflate compression when writing a chunked response #395
- Request's content-length header is cleared now after redirect from POST method #391
- Return a 400 if server received a non HTTP content #405
- Fix keep-alive support for aiohttp clients #406
- Allow gzip compression in high-level server response interface #403
- Rename TCPConnector.resolve and family to dns_cache #415
- Make UrlDispatcher ignore quoted characters during url matching #414 Backward-compatibility warning: this may change the url matched by your queries if they send quoted character (like %2F for /) #414
- Use optional cchardet accelerator if present #418
- Borrow loop from Connector in ClientSession if loop is not set
- Add context manager support to ClientSession for session closing.
- Add toplevel get(), post(), put(), head(), delete(), options(), patch() coroutines.
- Fix IPv6 support for client API #425
- Pass SSL context through proxy connector #421
- Make the rule: path for add_route should start with slash
- Don't process request finishing by low-level server on closed event loop
- Don't override data if multiple files are uploaded with same key #433
- Ensure multipart.BodyPartReader.read_chunk read all the necessary data to avoid false assertions about malformed multipart payload
- Dont sent body for 204, 205 and 304 http exceptions #442
- Correctly skip Cython compilation in MSVC not found #453
- Add response factory to StaticRoute #456
- Don't append trailing CRLF for multipart.BodyPartReader #454

9.13.12 0.16.6 (07-15-2015)

- Skip compilation on Windows if vcvarsall.bat cannot be found #438

9.13.13 0.16.5 (06-13-2015)

- Get rid of all comprehensions and yielding in _multidict #410

9.13.14 0.16.4 (06-13-2015)

- Don't clear current exception in multidict's *__repr__* (cythonized versions) #410

9.13.15 0.16.3 (05-30-2015)

- Fix StaticRoute vulnerability to directory traversal attacks #380

9.13.16 0.16.2 (05-27-2015)

- Update python version required for *__del__* usage: it's actually 3.4.1 instead of 3.4.0
- Add check for presence of loop.is_closed() method before call the former #378

9.13.17 0.16.1 (05-27-2015)

- Fix regression in static file handling #377

9.13.18 0.16.0 (05-26-2015)

- Unset waiter future after cancellation #363
- Update request url with query parameters #372
- Support new *fingerprint* param of TCPConnector to enable verifying SSL certificates via MD5, SHA1, or SHA256 digest #366
- Setup uploaded filename if field value is binary and transfer encoding is not specified #349
- Implement *ClientSession.close()* method
- Implement *connector.closed* readonly property
- Implement *ClientSession.closed* readonly property
- Implement *ClientSession.connector* readonly property
- Implement *ClientSession.detach* method
- Add *__del__* to client-side objects: sessions, connectors, connections, requests, responses.
- Refactor connections cleanup by connector #357
- Add *limit* parameter to connector constructor #358
- Add *request.has_body* property #364
- Add *response_class* parameter to *ws_connect()* #367
- *ProxyConnector* doesn't support keep-alive requests by default starting from now #368
- Add *connector.force_close* property
- Add ws_connect to ClientSession #374

- Support optional *chunk_size* parameter in *router.add_static()*

9.13.19 0.15.3 (04-22-2015)

- Fix graceful shutdown handling
- Fix *Expect* header handling for not found and not allowed routes #340

9.13.20 0.15.2 (04-19-2015)

- Flow control subsystem refactoring
- HTTP server performace optimizations
- Allow to match any request method with *
- Explicitly call drain on transport #316
- Make chardet module dependency mandatory #318
- Support keep-alive for HTTP 1.0 #325
- Do not chunk single file during upload #327
- Add ClientSession object for cookie storage and default headers #328
- Add *keep_alive_on* argument for HTTP server handler.

9.13.21 0.15.1 (03-31-2015)

- Pass Autobahn Testsuit tests
- Fixed websocket fragmentation
- Fixed websocket close procedure
- Fixed parser buffer limits
- Added *timeout* parameter to WebSocketResponse ctor
- Added *WebSocketResponse.close_code* attribute

9.13.22 0.15.0 (03-27-2015)

- Client WebSockets support
- New Multipart system #273
- Support for "Except" header #287 #267
- Set default Content-Type for post requests #184
- Fix issue with construction dynamic route with regexps and trailing slash #266
- Add repr to web.Request
- Add repr to web.Response
- Add repr for NotFound and NotAllowed match infos
- Add repr for web.Application

- Add repr to UrlMappingMatchInfo #217
- Gunicorn 19.2.x compatibility

9.13.23 0.14.4 (01-29-2015)

- Fix issue with error during constructing of url with regex parts #264

9.13.24 0.14.3 (01-28-2015)

- Use path='/' by default for cookies #261

9.13.25 0.14.2 (01-23-2015)

- Connections leak in BaseConnector #253
- Do not swallow websocket reader exceptions #255
- web.Request's read, text, json are memorized #250

9.13.26 0.14.1 (01-15-2015)

- HttpMessage._add_default_headers does not overwrite existing headers #216
- Expose multidict classes at package level
- add *aiohttp.web.WebSocketResponse*
- According to RFC 6455 websocket subprotocol preference order is provided by client, not by server
- websocket's ping and pong accept optional message parameter
- multidict views do not accept *getall* parameter anymore, it returns the full body anyway.
- multidicts have optional Cython optimization, cythonized version of multidicts is about 5 times faster than pure Python.
- multidict.getall() returns *list*, not *tuple*.
- Backward imcompatible change: now there are two mutable multidicts (*MultiDict*, *CIMultiDict*) and two immutable multidict proxies (*MultiDictProxy* and *CIMultiDictProxy*). Previous edition of multidicts was not a part of public API BTW.
- Router refactoring to push Not Allowed and Not Found in middleware processing
- Convert *ConnectionError* to *aiohttp.DisconnectedError* and don't eat *ConnectionError* exceptions from web handlers.
- Remove hop headers from Response class, wsgi response still uses hop headers.
- Allow to send raw chunked encoded response.
- Allow to encode output bytes stream into chunked encoding.
- Allow to compress output bytes stream with *deflate* encoding.
- Server has 75 seconds keepalive timeout now, was non-keepalive by default.
- Application doesn't accept ***kwargs* anymore (#243).

- Request is inherited from dict now for making per-request storage to middlewares (#242).

9.13.27 0.13.1 (12-31-2014)

- Add *aiohttp.web.StreamResponse.started* property #213
- Html escape traceback text in *ServerHttpProtocol.handle_error*
- Mention handler and middlewares in *aiohttp.web.RequestHandler.handle_request* on error (#218)

9.13.28 0.13.0 (12-29-2014)

- *StreamResponse.charset* converts value to lower-case on assigning.
- Chain exceptions when raise *ClientRequestError*.
- Support custom regexps in route variables #204
- Fixed graceful shutdown, disable keep-alive on connection closing.
- Decode HTTP message with *utf-8* encoding, some servers send headers in utf-8 encoding #207
- Support *aiohtt.web* middlewares #209
- Add ssl_context to TCPConnector #206

9.13.29 0.12.0 (12-12-2014)

- Deep refactoring of *aiohttp.web* in backward-incompatible manner. Sorry, we have to do this.
- Automatically force aiohttp.web handlers to coroutines in *UrlDispatcher.add_route()* #186
- Rename *Request.POST()* function to *Request.post()*
- Added POST attribute
- Response processing refactoring: constructor doesn't accept Request instance anymore.
- Pass application instance to finish callback
- Exceptions refactoring
- Do not unquote query string in *aiohttp.web.Request*
- Fix concurrent access to payload in *RequestHandle.handle_request()*
- Add access logging to *aiohttp.web*
- Gunicorn worker for *aiohttp.web*
- Removed deprecated *AsyncGunicornWorker*
- Removed deprecated HttpClient

9.13.30 0.11.0 (11-29-2014)

- Support named routes in *aiohttp.web.UrlDispatcher* #179
- Make websocket subprotocols conform to spec #181

9.13.31 0.10.2 (11-19-2014)

- Don't unquote *environ['PATH_INFO']* in wsgi.py #177

9.13.32 0.10.1 (11-17-2014)

- aiohttp.web.HTTPException and descendants now files response body with string like *404: NotFound*
- Fix multidict *__iter__*, the method should iterate over keys, not (key, value) pairs.

9.13.33 0.10.0 (11-13-2014)

- Add aiohttp.web subpackage for highlevel HTTP server support.
- Add *reason* optional parameter to aiohttp.protocol.Response ctor.
- Fix aiohttp.client bug for sending file without content-type.
- Change error text for connection closed between server responses from 'Can not read status line' to explicit 'Connection closed by server'
- Drop closed connections from connector #173
- Set server.transport to None on .closing() #172

9.13.34 0.9.3 (10-30-2014)

- Fix compatibility with asyncio 3.4.1+ #170

9.13.35 0.9.2 (10-16-2014)

- Improve redirect handling #157
- Send raw files as is #153
- Better websocket support #150

9.13.36 0.9.1 (08-30-2014)

- Added MultiDict support for client request params and data #114.
- Fixed parameter type for IncompleteRead exception #118.
- Strictly require ASCII headers names and values #137
- Keep port in ProxyConnector #128.
- Python 3.4.1 compatibility #131.

9.13.37 0.9.0 (07-08-2014)

- Better client basic authentication support #112.
- Fixed incorrect line splitting in HttpRequestParser #97.
- Support StreamReader and DataQueue as request data.
- Client files handling refactoring #20.
- Backward incompatible: Replace DataQueue with StreamReader for request payload #87.

9.13.38 0.8.4 (07-04-2014)

- Change ProxyConnector authorization parameters.

9.13.39 0.8.3 (07-03-2014)

- Publish TCPConnector properties: verify_ssl, family, resolve, resolved_hosts.
- Don't parse message body for HEAD responses.
- Refactor client response decoding.

9.13.40 0.8.2 (06-22-2014)

- Make ProxyConnector.proxy immutable property.
- Make UnixConnector.path immutable property.
- Fix resource leak for aiohttp.request() with implicit connector.
- Rename Connector's reuse_timeout to keepalive_timeout.

9.13.41 0.8.1 (06-18-2014)

- Use case insensitive multidict for server request/response headers.
- MultiDict.getall() accepts default value.
- Catch server ConnectionError.
- Accept MultiDict (and derived) instances in aiohttp.request header argument.
- Proxy 'CONNECT' support.

9.13.42 0.8.0 (06-06-2014)

- Add support for utf-8 values in HTTP headers
- Allow to use custom response class instead of HttpResponse
- Use MultiDict for client request headers
- Use MultiDict for server request/response headers
- Store response headers in ClientResponse.headers attribute
- Get rid of timeout parameter in aiohttp.client API

- Exceptions refactoring

9.13.43 0.7.3 (05-20-2014)

- Simple HTTP proxy support.

9.13.44 0.7.2 (05-14-2014)

- Get rid of *__del__* methods
- Use ResourceWarning instead of logging warning record.

9.13.45 0.7.1 (04-28-2014)

- Do not unquote client request urls.
- Allow multiple waiters on transport drain.
- Do not return client connection to pool in case of exceptions.
- Rename SocketConnector to TCPConnector and UnixSocketConnector to UnixConnector.

9.13.46 0.7.0 (04-16-2014)

- Connection flow control.
- HTTP client session/connection pool refactoring.
- Better handling for bad server requests.

9.13.47 0.6.5 (03-29-2014)

- Added client session reuse timeout.
- Better client request cancellation support.
- Better handling responses without content length.
- Added HttpClient verify_ssl parameter support.

9.13.48 0.6.4 (02-27-2014)

- Log content-length missing warning only for put and post requests.

9.13.49 0.6.3 (02-27-2014)

- Better support for server exit.
- Read response body until EOF if content-length is not defined #14

9.13.50 0.6.2 (02-18-2014)

- Fix trailing char in allowed_methods.
- Start slow request timer for first request.

9.13.51 0.6.1 (02-17-2014)

- Added utility method HttpResponse.read_and_close()
- Added slow request timeout.
- Enable socket SO_KEEPALIVE if available.

9.13.52 0.6.0 (02-12-2014)

- Better handling for process exit.

9.13.53 0.5.0 (01-29-2014)

- Allow to use custom HttpRequest client class.
- Use gunicorn keepalive setting for asynchronous worker.
- Log leaking responses.
- python 3.4 compatibility

9.13.54 0.4.4 (11-15-2013)

- Resolve only AF_INET family, because it is not clear how to pass extra info to asyncio.

9.13.55 0.4.3 (11-15-2013)

- Allow to wait completion of request with *HttpResponse.wait_for_close()*

9.13.56 0.4.2 (11-14-2013)

- Handle exception in client request stream.
- Prevent host resolving for each client request.

9.13.57 0.4.1 (11-12-2013)

- Added client support for *expect: 100-continue* header.

9.13.58 0.4 (11-06-2013)

- Added custom wsgi application close procedure
- Fixed concurrent host failure in HttpClient

9.13.59 0.3 (11-04-2013)

- Added PortMapperWorker
- Added HttpClient
- Added TCP connection timeout to HTTP client
- Better client connection errors handling
- Gracefully handle process exit

9.13.60 0.2

- Fix packaging

9.14 Python 3.3, ..., 3.4.1 support

As of aiohttp **v0.18.0** we dropped support for Python 3.3 up to 3.4.1. The main reason for that is the `object.__del__()` method, which is fully working since Python 3.4.1 and we need it for proper resource closing.

The last Python 3.3, 3.4.0 compatible version of aiohttp is **v0.17.4**.

This should not be an issue for most aiohttp users (for example Ubuntu 14.04.3 LTS provides python upgraded to 3.4.3), however libraries depending on aiohttp should consider this and either freeze aiohttp version or drop Python 3.3 support as well.

9.15 Glossary

asyncio The library for writing single-threaded concurrent code using coroutines, multiplexing I/O access over sockets and other resources, running network clients and servers, and other related primitives.

Reference implementation of **PEP 3156**

https://pypi.python.org/pypi/asyncio/

callable Any object that can be called. Use `callable()` to check that.

chardet The Universal Character Encoding Detector

https://pypi.python.org/pypi/chardet/

cchardet cChardet is high speed universal character encoding detector - binding to charsetdetect.

https://pypi.python.org/pypi/cchardet/

keep-alive A technique for communicating between HTTP client and server when connection is not closed after sending response but keeped open for sending next request through the same socket.

It makes communication faster by getting rid of connection establishment for every request.

web-handler An endpoint that returns HTTP response.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

a

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W