## CONTENTS

1 Key Features 3

2 Library Installation 5
   2.1 Installing speedups altogether 5

3 Getting Started 7
   3.1 Client example 7
   3.2 Server example: 7

4 What’s new in aiohttp 3? 9

5 Tutorial 11

6 Source code 13

7 Dependencies 15

8 Communication channels 17

9 Contributing 19

10 Authors and License 21

11 Policy for Backward Incompatible Changes 23

12 Table Of Contents 25
   12.1 Client 25
   12.2 Server 78
   12.3 Utilities 165
   12.4 FAQ 180
   12.5 Miscellaneous 187
   12.6 Who uses aiohttp? 244
   12.7 Contributing 248

Python Module Index 253

Index 255
Asynchronous HTTP Client/Server for *asyncio* and Python.
Current version is 4.0.0a1.
CHAPTER ONE

KEY FEATURES

• Supports both Client and HTTP Server.
• Supports both Server WebSockets and Client WebSockets out-of-the-box without the Callback Hell.
• Web-server has Middlewares, Signals and plugable routing.
$ pip install aiohttp

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

$ pip install cchardet

For speeding up DNS resolving by client API you may install `aiodns` as well. This option is highly recommended:

$ pip install aiodns

### 2.1 Installing speedups altogether

The following will get you `aiohttp` along with `chardet`, `aiodns` and Brotli in one bundle. No need to type separate commands anymore!

$ pip install aiohttp[speedups]
3.1 Client example

```python
import aiohttp
import asyncio

async def main():
    async with aiohttp.ClientSession() as session:
        async with session.get('http://python.org') as response:
            print("Status:", response.status)
            print("Content-type:", response.headers['content-type'])

            html = await response.text()
            print("Body:", html[:15], "...")

loop = asyncio.get_event_loop()
loop.run_until_complete(main())

This prints:

Status: 200
Content-type: text/html; charset=utf-8
Body: <!doctype html> ...

Coming from requests? Read why we need so many lines.
```

3.2 Server example:

```python
from aiohttp import web

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

app = web.Application()
app.add_routes([web.get('/', handle),
                web.get('/{name}', handle)])

(continues on next page)"
if __name__ == '__main__':
    web.run_app(app)

For more information please visit Client and Server pages.
Go to *What’s new in aiohttp 3.0* page for aiohttp 3.0 major release changes.
Polls tutorial
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 Azure Pipelines for Continuous Integration.
• Python 3.6+
• async_timeout
• attrs
• chardet
• multidict
• yarl

• Optional cchardet as faster replacement for chardet.
  Install it explicitly via:

  $ pip install cchardet

• Optional aiodns for fast DNS resolving. The library is highly recommended.

  $ pip install aiodns
COMMUNICATION CHANNELS

*aio-libs discourse group*: https://aio-libs.discourse.group
Feel free to post your questions and ideas here.

*gitter chat* https://gitter.im/aio-libs/Lobby
We support Stack Overflow. Please add *aiohttp* tag to your question there.
Please read the *instructions for contributors* before making a Pull Request.
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.
POLICY FOR BACKWARD INCOMPATIBLE CHANGES

aiohttp keeps backward compatibility.

After deprecating some Public API (method, class, function argument, etc.) the library guarantees the usage of deprecated API is still allowed at least for a year and half after publishing new release with deprecation.

All deprecations are reflected in documentation and raises DeprecationWarning.

Sometimes we are forced to break our own rule for the sake of very strong reason. Most likely the reason is a critical bug which cannot be solved without major API change, but we are working hard for keeping these changes as rare as possible.
12.1 Client

The page contains all information about aiohttp Client API:

12.1.1 Client Quickstart

Eager to get started? This page gives a good introduction in how to get started with aiohttp client API.

First, make sure that aiohttp is installed and up-to-date

Let’s get started with some simple examples.

Make a Request

Begin by importing the aiohttp module:

```python
import aiohttp
```

Now, let’s try to get a web-page. For example let’s query http://httpbin.org/get:

```python
async with aiohttp.ClientSession() as session:
    async with session.get('http://httpbin.org/get') as resp:
        print(resp.status)
        print(await resp.text())
```

Now, we have a ClientSession called session and a ClientResponse object called resp. We can get all the information we need from the response. The mandatory parameter of ClientSession.get() coroutine is an HTTP url (str or class:yarl.URL instance).

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

```python
session.post('http://httpbin.org/post', data=b'data')
```

Other HTTP methods are available as well:

```python
session.put('http://httpbin.org/put', data=b'data')
session.delete('http://httpbin.org/delete')
session.head('http://httpbin.org/get')
session.options('http://httpbin.org/get')
session.patch('http://httpbin.org/patch', data=b'data')
```
**Note:** Don’t create a session per request. Most likely you need a session per application which performs all requests altogether.

More complex cases may require a session per site, e.g., one for Github and another one for Facebook APIs. Anyway making a session for every request is a **very bad** idea.

A session contains a connection pool inside. Connection reusage and keep-alives (both are on by default) may speed up total performance.

You may find more information about creating persistent sessions in *Persistent session*.

A session context manager usage is not mandatory but `await session.close()` method should be called in this case, e.g.:

```python
session = aiohttp.ClientSession()
async with session.get('...'):
    # ...
await session.close()
```

### 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 `dict`, 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:

```python
params = {'key1': 'value1', 'key2': 'value2'}
async with session.get('http://httpbin.org/get',
    params=params) as resp:
    expect = 'http://httpbin.org/get?key2=value2&key1=value1'
    assert str(resp.url) == expect
```

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

For sending data with multiple values for the same key `MultiDict` may be used as well.

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:

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

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

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

**Note:** `aiohttp` internally performs URL canonization before sending request.

Canonization encodes `host` part by `IDNA` codec and applies `requoting` to `path` and `query` parts.
For example URL('http://example.com/%30?a=%31') is converted to URL('http://example.com/%D0%BF%D1%83%D1%82%D1%8C/0?a=1').

Sometimes canonization is not desirable if server accepts exact representation and does not requote URL itself. To disable canonization use encoded=True parameter for URL construction:

```python
await session.get(
    URL('http://example.com/%30', encoded=True))
```

**Warning:** Passing `params` overrides `encoded=True`, never use both options.

**Response Content and Status Code**

We can read the content of the server’s response and its status code. Consider the GitHub timeline again:

```python
async with session.get('https://api.github.com/events') as resp:
    print(resp.status)
    print(await resp.text())
```

prints out something like:

```
200
['{"created_at":"2015-06-12T14:06:22Z","public":true,"actor":...
```

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

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

**Binary Response Content**

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

```python
print(await resp.read())
```

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

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

You can enable `brotli` transfer-encodings support, just install `Brotli`. 

---

12.1. Client
**JSON Request**

Any of session’s request methods like `request()`, `ClientSession.get()`, `ClientSession.post()` etc. accept `json` parameter:

```python
async with aiohttp.ClientSession() as session:
    await session.post(url, json={'test': 'object'})
```

By default session uses python’s standard `json` module for serialization. But it is possible to use different serializer. `ClientSession` accepts `json_serialize` parameter:

```python
import ujson
async with aiohttp.ClientSession(
    json_serialize=ujson.dumps) as session:
    await session.post(url, json={'test': 'object'})
```

**Note:** `ujson` library is faster than standard `json` but slightly incompatible.

**JSON Response Content**

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

```python
async with session.get('https://api.github.com/events') as resp:
    print(await resp.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.

**Note:** The methods above reads the whole response body into memory. If you are planning on reading lots of data, consider using the streaming response method documented below.

**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:

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

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

```python
with open(filename, 'wb') as fd:
    while True:
        chunk = await resp.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`. 

---

*aiohttp Documentation, Release 4.0.0a1*

Chapter 12. Table Of Contents
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:

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

If you want to send data that is not form-encoded you can do it by passing a `bytes` instead of a `dict`. This data will be posted directly and content-type set to `application/octet-stream` by default:

```python
async with session.post(url, data=b'Binary-data') as resp:
    ...
```

If you want to send JSON data:

```python
async with session.post(url, json={'example': 'test'}) as resp:
    ...
```

To send text with appropriate content-type just use `data` argument:

```python
async with session.post(url, data='') as resp:
    ...
```

POST a Multipart-Encoded File

To upload Multipart-encoded files:

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

You can set the `filename` and `content_type` explicitly:

```python
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 session.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*

**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:

```python
with open('massive-body', 'rb') as f:
    await session.post('http://httpbin.org/post', data=f)
```

Or you can use asynchronous generator:

```python
async def file_sender(file_name=None):
    async with aiofiles.open(file_name, 'rb') as f:
        chunk = await f.read(64*1024)
        while chunk:
            yield chunk
            chunk = await f.read(64*1024)

# Then you can use file_sender as a data provider:

async with session.post('http://httpbin.org/post',
                         data=file_sender(file_name='huge_file')) as resp:
    print(await resp.text())
```

Because the `content` attribute is a `StreamReader` (provides async iterator protocol), you can chain get and post requests together:

```python
resp = await session.get('http://python.org')
await session.post('http://httpbin.org/post',
                   data=resp.content)
```

**WebSockets**

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:

```python
async with session.ws_connect('http://example.org/ws') as ws:
    async for msg in ws:
        if msg.type == aiohttp.WSMsgType.TEXT:
            if msg.data == 'close cmd':
                await ws.close()
            break
        else:
            await ws.send_str(msg.data + '/answer')
    elif msg.type == aiohttp.WSMsgType.ERROR:
        break
```
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 `await ws.send_str('data')`) for example.

### Timeouts

Timeout settings are stored in *ClientTimeout* data structure.

By default *aiohttp* uses a total 300 seconds (5min) timeout, it means that the whole operation should finish in 5 minutes.

The value could be overridden by `timeout` parameter for the session (specified in seconds):

```python
timeout = aiohttp.ClientTimeout(total=60)
async with aiohttp.ClientSession(timeout=timeout) as session:
    ...
```

Timeout could be overridden for a request like `ClientSession.get()`:

```python
async with session.get(url, timeout=timeout) as resp:
    ...
```

Supported *ClientTimeout* fields are:

- **total**
  
  The maximal number of seconds for the whole operation including connection establishment, request sending and response reading.

- **connect**
  
  The maximal number of seconds for connection establishment of a new connection or for waiting for a free connection from a pool if pool connection limits are exceeded.

- **sock_connect**
  
  The maximal number of seconds for connecting to a peer for a new connection, not given from a pool.

- **sock_read**
  
  The maximal number of seconds allowed for period between reading a new data portion from a peer.

All fields are floats, `None` or 0 disables a particular timeout check, see the *ClientTimeout* reference for defaults and additional details.

Thus the default timeout is:

```python
aiohttp.ClientTimeout(total=5*60, connect=None, sock_connect=None, sock_read=None)
```
aiohttp Documentation, Release 4.0.0a1

12.1.2 Advanced Client Usage
Client Session
ClientSession is the heart and the main entry point for all client API operations.
Create the session first, use the instance for performing HTTP requests and initiating WebSocket connections.
The session contains a cookie storage and connection pool, thus cookies and connections are shared between HTTP
requests sent by the same session.
Custom Request 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 directly:
url = 'http://example.com/image'
payload = b'GIF89a\x01\x00\x01\x00\x00\xff\x00,\x00\x00'
b'\x00\x00\x01\x00\x01\x00\x00\x02\x00;'
headers = {'content-type': 'image/gif'}
await session.post(url,
data=payload,
headers=headers)

You also can set default headers for all session requests:
headers={"Authorization": "Basic bG9naW46cGFzcw=="}
async with aiohttp.ClientSession(headers=headers) as session:
async with session.get("http://httpbin.org/headers") as r:
json_body = await r.json()
assert json_body['headers']['Authorization'] == \
'Basic bG9naW46cGFzcw=='

Typical use case is sending JSON body. You can specify content type directly as shown above, but it is more convenient
to use special keyword json:
await session.post(url, json={'example': 'text'})

For text/plain
await session.post(url, data=', !')

Custom Cookies
To send your own cookies to the server, you can use the cookies parameter of ClientSession constructor:
url = 'http://httpbin.org/cookies'
cookies = {'cookies_are': 'working'}
async with ClientSession(cookies=cookies) as session:
async with session.get(url) as resp:
assert await resp.json() == {
"cookies": {"cookies_are": "working"}}

32

Chapter 12. Table Of Contents


Note: httpbin.org/cookies endpoint returns request cookies in JSON-encoded body. To access session cookies see ClientSession.cookie_jar.

ClientSession may be used for sharing cookies between multiple requests:

```
async with aiohttp.ClientSession() as session:
    await session.get('http://httpbin.org/cookies/set?my_cookie=my_value')
    filtered = session.cookie_jar.filter_cookies(URL('http://httpbin.org'))
    assert filtered['my_cookie'].value == 'my_value'
    async with session.get('http://httpbin.org/cookies') as r:
        json_body = await r.json()
        assert json_body['cookies']['my_cookie'] == 'my_value'
```

Response Headers and Cookies

We can view the server’s response ClientResponse.headers using a CIMultiDictProxy:

```
assert resp.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:

```
assert resp.headers['Content-Type'] == 'application/json'
assert resp.headers.get('content-type') == 'application/json'
```

All headers are converted from binary data using UTF-8 with surrogateescape option. That works fine on most cases but sometimes unconverted data is needed if a server uses nonstandard encoding. While these headers are malformed from RFC 7230 perspective they may be retrieved by using ClientResponse.raw_headers property:

```
assert resp.raw_headers == {
    (b'SERVER', b'nginx'),
    (b'DATE', b'Sat, 09 Jan 2016 20:28:40 GMT'),
    (b'CONTENT-TYPE', b'text/html; charset=utf-8'),
    (b'CONTENT-LENGTH', b'12150'),
    (b'CONNECTION', b'keep-alive')
```

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

```
url = 'http://example.com/some/cookie/setting/url'
async with session.get(url) as resp:
    print(resp.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 please use `aiohttp.ClientSession` object.

### Redirection History

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

```python
resp = await session.get('http://example.com/some/redirect/)
assert resp.status == 200
assert resp.url == URL('http://example.com/some/other/url/)
assert len(resp.history) == 1
assert resp.history[0].status == 301
assert resp.history[0].url == URL('http://example.com/some/redirect/')
```

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

### Cookie Jar

#### Cookie Safety

By default `ClientSession` uses strict version of `aiohttp.CookieJar`. **RFC 2109** explicitly forbids cookie accepting from URLs with IP address instead of DNS name (e.g. `http://127.0.0.1:80/cookie`).

It’s good but sometimes for testing we need to enable support for such cookies. It should be done by passing `unsafe=True` to `aiohttp.CookieJar` constructor:

```python
jar = aiohttp.CookieJar(unsafe=True)
session = aiohttp.ClientSession(cookie_jar=jar)
```

#### Dummy Cookie Jar

Sometimes cookie processing is not desirable. For this purpose it’s possible to pass `aiohttp.DummyCookieJar` instance into client session:

```python
jar = aiohttp.DummyCookieJar()
session = aiohttp.ClientSession(cookie_jar=jar)
```

### Uploading pre-compressed data

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

```python
async def my_coroutine(session, headers, my_data):
    data = zlib.compress(my_data)
    headers = {'Content-Encoding': 'deflate'}
    async with session.post('http://httpbin.org/post',
        data=data, headers=headers)
    pass
```
Disabling content type validation for JSON responses

The standard explicitly restricts JSON Content-Type HTTP header to application/json or any extended form, e.g. application/vnd.custom-type+json. Unfortunately, some servers send a wrong type, like text/html.

This can be worked around in two ways:

1. Pass the expected type explicitly (in this case checking will be strict, without the extended form support, so custom/xxx+type won’t be accepted):
   await resp.json(content_type='custom/type').

2. Disable the check entirely:
   await resp.json(content_type=None).

Client Tracing

The execution flow of a specific request can be followed attaching listeners coroutines to the signals provided by the TraceConfig instance, this instance will be used as a parameter for the ClientSession constructor having as a result a client that triggers the different signals supported by the TraceConfig. By default any instance of ClientSession class comes with the signals ability disabled. The following snippet shows how the start and the end signals of a request flow can be followed:

```python
async def on_request_start(session, trace_config_ctx, params):
    print("Starting request")

async def on_request_end(session, trace_config_ctx, params):
    print("Ending request")

trace_config = aiohttp.TraceConfig()
trace_config.on_request_start.append(on_request_start)
trace_config.on_request_end.append(on_request_end)
async with aiohttp.ClientSession(
    trace_configs=[trace_config]) as client:
    client.get('http://example.com/some/redirect/')
```

The `trace_configs` is a list that can contain instances of TraceConfig class that allow run the signals handlers coming from different TraceConfig instances. The following example shows how two different TraceConfig that have a different nature are installed to perform their job in each signal handle:

```python
from mylib.traceconfig import AuditRequest
from mylib.traceconfig import XRay

async with aiohttp.ClientSession(
    trace_configs=[AuditRequest(), XRay()]) as client:
    client.get('http://example.com/some/redirect/')
```

All signals take as a parameters first, the ClientSession instance used by the specific request related to that signals and second, a SimpleNamespace instance called trace_config_ctx. The trace_config_ctx object can be used to share the state through to the different signals that belong to the same request and to the same TraceConfig class, perhaps:

```python
async def on_request_start(
    session, trace_config_ctx, params):
```

(continues on next page)
trace_config_ctx.start = asyncio.get_event_loop().time()

async def on_request_end(session, trace_config_ctx, params):
    elapsed = asyncio.get_event_loop().time() - trace_config_ctx.start
    print("Request took {}".format(elapsed))

The `trace_config_ctx` param is by default a `SimpleNamespace` that is initialized at the begin-
ing of the request flow. However, the factory used to create this object can be overwritten using the
`trace_config_ctx_factory` constructor param of the `TraceConfig` class.

The `trace_request_ctx` param can given at the beginning of the request execution, accepted by all of the HTTP
verbs, and will be passed as a keyword argument for the `trace_config_ctx_factory` factory. This param is
useful to pass data that is only available at request time, perhaps:

async def on_request_start(session, trace_config_ctx, params):
    print(trace_config_ctx.trace_request_ctx)

    session.get('http://example.com/some/redirect/',
        trace_request_ctx={'foo': 'bar'})

See also:
`Tracing Reference` section for more information about the different signals supported.

Connectors

To tweak or change `transport` layer of requests you can pass a custom `connector` to `ClientSession` and family.
For example:

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

**Note:** By default `session` object takes the ownership of the connector, among other things closing the connections
once the `session` is closed. If you are keen on share the same `connector` through different `session` instances you must
give the `connector_owner` parameter as `False` for each `session` instance.

See also:
`Connectors` section for more information about different connector types and configuration options.

Limiting connection pool size

To limit amount of simultaneously opened connections you can pass `limit` parameter to `connector`:

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

The example limits total amount of parallel connections to 30.
The default is 100.
If you explicitly want not to have limits, pass 0. For example:
conn = aiohttp.TCPConnector(limit=0)

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

conn = aiohttp.TCPConnector(limit_per_host=30)

The example limits amount of parallel connections to the same to 30.
The default is 0 (no limit on per host bases).

### Tuning the DNS cache

By default TCPConnector comes with the DNS cache table enabled, and resolutions will be cached by default for 10 seconds. This behavior can be changed either to change of the TTL for a resolution, as can be seen in the following example:

conn = aiohttp.TCPConnector(ttl_dns_cache=300)

or disabling the use of the DNS cache table, meaning that all requests will end up making a DNS resolution, as the following example shows:

conn = aiohttp.TCPConnector(use_dns_cache=False)

### Resolving using custom nameservers

In order to specify the nameservers to when resolving the hostnames, aiodns is required:

```python
from aiohttp.resolver import AsyncResolver
resolver = AsyncResolver(nameservers=['8.8.8.8', '8.8.4.4'])
conn = aiohttp.TCPConnector(resolver=resolver)
```

### Unix domain sockets

If your HTTP server uses UNIX domain sockets you can use UnixConnector:

```python
conn = aiohttp.UnixConnector(path='/path/to/socket')
session = aiohttp.ClientSession(connector=conn)
```

### Named pipes in Windows

If your HTTP server uses Named pipes you can use NamedPipeConnector:

```python
conn = aiohttp.NamedPipeConnector(path=r'\\.\pipe\<name-of-pipe>')
session = aiohttp.ClientSession(connector=conn)
```

It will only work with the ProactorEventLoop
SSL control for TCP sockets

By default aiohttp uses strict checks for HTTPS protocol. Certification checks can be relaxed by setting ssl to False:

```python
r = await session.get('https://example.com', ssl=False)
```

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 proper ClientSession method:

```python
sslcontext = ssl.create_default_context(
cafile='/path/to/ca-bundle.crt')
r = await session.get('https://example.com', ssl=sslcontext)
```

If you need to verify self-signed certificates, you can do the same thing as the previous example, but add another call to ssl.SSLContext.load_cert_chain() with the key pair:

```python
sslcontext = ssl.create_default_context(
cafile='/path/to/ca-bundle.crt')
sslcontext.load_cert_chain('/path/to/client/public/device.pem',
'/path/to/client/private/device.key')
r = await session.get('https://example.com', ssl=sslcontext)
```

There is explicit errors when ssl verification fails

- `aiohttp.ClientConnectorSSLError:
  ```python
  try:
  await session.get('https://expired.badssl.com/')
  except aiohttp.ClientConnectorSSLError as e:
  assert isinstance(e, ssl.SSLError)
  ```

- `aiohttp.ClientConnectorCertificateError:
  ```python
  try:
  await session.get('https://wrong.host.badssl.com/')
  except aiohttp.ClientConnectorCertificateError as e:
  assert isinstance(e, ssl.CertificateError)
  ```

If you need to skip both ssl related errors

- `aiohttp.ClientSSLError:
  ```python
  try:
  await session.get('https://expired.badssl.com/')
  except aiohttp.ClientSSLError as e:
  assert isinstance(e, ssl.SSLError)
  ```

  ```python
  try:
  await session.get('https://wrong.host.badssl.com/')
  except aiohttp.ClientSSLError as e:
  assert isinstance(e, ssl.CertificateError)
  ```

You may also verify certificates via SHA256 fingerprint:

```python
# Attempt to connect to https://www.python.org
# with a pin to a bogus certificate:
bad_fp = b'0'*64
exc = None
```
try:
    r = await session.get('https://www.python.org',
                       ssl=aiohttp.Fingerprint(bad_fp))
except aiohttp.FingerprintMismatch as e:
    exc = e
assert exc is not None
assert exc.expected == bad_fp
# www.python.org cert's actual fingerprint
assert exc.got == b'...'  

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:

```bash
openssl x509 -in crt.pem -inform PEM -outform DER > crt.der
```  

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

The `ssl` parameter could be passed to `TCPConnector` as default, the value from `ClientSession.get()` and others override default.

### Proxy support

`aiohttp` supports plain HTTP proxies and HTTP proxies that can be upgraded to HTTPS via the HTTP CONNECT method. `aiohttp` does not support proxies that must be connected to via `https://`. To connect, use the `proxy` parameter:

```python
async with aiohttp.ClientSession() as session:
    async with session.get("http://python.org",
                          proxy="http://proxy.com") as resp:
        print(resp.status)
```  

It also supports proxy authorization:

```python
async with aiohttp.ClientSession() as session:
    proxy_auth = aiohttp.BasicAuth('user', 'pass')
    async with session.get("http://python.org",
                          proxy="http://proxy.com",
                          proxy_auth=proxy_auth) as resp:
        print(resp.status)
```  

Authentication credentials can be passed in proxy URL:

```python
session.get("http://python.org",
           proxy="http://user:pass@some.proxy.com")
```  

Contrary to the `requests` library, it won’t read environment variables by default. But you can do so by passing `trust_env=True` into `aiohttp.ClientSession` constructor for extracting proxy configuration from `HTTP_PROXY`, `HTTPS_PROXY`, `WS_PROXY` or `WSS_PROXY` environment variables (all are case insensitive):

```python
async with aiohttp.ClientSession(trust_env=True) as session:
    async with session.get("http://python.org") as resp:
        print(resp.status)
```
Proxy credentials are given from ~/.netrc file if present (see aiohttp.ClientSession for more details).

**Persistent session**

Even though creating a session on demand seems like a tempting idea, we advise against it. aiohttp.ClientSession maintains a connection pool. Contained connections can be reused if necessary to gain some performance improvements. If you plan on reusing the session, a.k.a. creating persistent session, you can use either Signals or Cleanup Context. If possible we advise using Cleanup Context, as it results in more compact code:

```python
app.cleanup_ctx.append(persistent_session)

async def persistent_session(app):
    app['PERSISTENT_SESSION'] = session = aiohttp.ClientSession()
    yield
    await session.close()

async def my_request_handler(request):
    session = request.app['PERSISTENT_SESSION']
    async with session.get('http://python.org') as resp:
        print(resp.status)
```

This approach can be successfully used to define numerous of session given certain requirements. It benefits from having a single location where aiohttp.ClientSession instances are created and where artifacts such as aiohttp.connector.BaseConnector can be safely shared between sessions if needed.

In the end all you have to do is to close all sessions after `yield` statement:

```python
async def multiple_sessions(app):
    app['PERSISTENT_SESSION_1'] = session_1 = aiohttp.ClientSession()
    app['PERSISTENT_SESSION_2'] = session_2 = aiohttp.ClientSession()
    app['PERSISTENT_SESSION_3'] = session_3 = aiohttp.ClientSession()

    yield

    await asyncio.gather(*[
        session_1.close(),
        session_2.close(),
        session_3.close(),
    ])
```

**Graceful Shutdown**

When ClientSession closes at the end of an async with block (or through a direct ClientSession.close() call), the underlying connection remains open due to asyncio internal details. In practice, the underlying connection will close after a short while. However, if the event loop is stopped before the underlying connection is closed, an ResourceWarning: unclosed transport warning is emitted (when warnings are enabled).

To avoid this situation, a small delay must be added before closing the event loop to allow any open underlying connections to close.

For a ClientSession without SSL, a simple zero-sleep (await asyncio.sleep(0)) will suffice:

```python
async def read_website():
    async with aiohttp.ClientSession() as session:
        async with session.get('http://example.org/') as resp:
            await resp.read()
```

(continues on next page)
loop = asyncio.get_event_loop()
loop.run_until_complete(read_website())
# Zero-sleep to allow underlying connections to close
loop.run_until_complete(asyncio.sleep(0))
loop.close()

For a `ClientSession` with SSL, the application must wait a short duration before closing:

```
# Wait 250 ms for the underlying SSL connections to close
loop.run_until_complete(asyncio.sleep(0.250))
loop.close()
```

Note that the appropriate amount of time to wait will vary from application to application.

All of this will eventually become obsolete when the asyncio internals are changed so that aiohttp itself can wait on the underlying connection to close. Please follow issue #1925 for the progress on this.

**HTTP Pipelining**

aiohttp does not support HTTP/HTTPS pipelining.

### 12.1.3 Client Reference

**Client Session**

Client session is the recommended interface for making HTTP requests.

Session encapsulates a `connection pool` (`connector` instance) and supports keepalives by default. Unless you are connecting to a large, unknown number of different servers over the lifetime of your application, it is suggested you use a single session for the lifetime of your application to benefit from connection pooling.

Usage example:

```python
import aiohttp
import asyncio

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

async def main():
    async with aiohttp.ClientSession() as client:
        html = await fetch(client)
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

The client session supports the context manager protocol for self closing.

**ClientSession**(*, connector=None, cookies=None, headers=None, skip_auto_headers=None, auth=None, json_serialize=json.dumps, version=aiohttp.HttpVersion11, cookie_jar=None, timeout=sentinel, raise_for_status=False, connector_owner=True, auto_decompress=True, requote_redirect_url=False, trust_env=False, trace_configs=None)

The class for creating client sessions and making requests.
Parameters

- **connector** *(aiohttp.BaseConnector)* – BaseConnector sub-class instance to support connection pooling.
- **cookies** *(dict)* – Cookies to send with the request (optional)
- **headers** – HTTP Headers to send with every 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 *istr* (optional)
- **auth** *(aiohttp.BasicAuth)* – an object that represents HTTP Basic Authorization (optional)
- **version** – supported HTTP version, HTTP 1.1 by default.
- **cookie_jar** – Cookie Jar, *AbstractCookieJar* instance.
  By default every session instance has own private cookie jar for automatic cookies processing but user may redefine this behavior by providing own jar implementation.
  One example is not processing cookies at all when working in proxy mode.
  If no cookie processing is needed, a *aiohttp.DummyCookieJar* instance can be provided.
- **json_serialize** *(callable)* – Json *serializer* callable.
  By default *json.dumps()* function.
- **callable** *(Union[bool,]*) – Automatically call *ClientResponse.raise_for_status()* for each response, *False* by default.
  This parameter can be overridden when you making a request, e.g.:

```python
client_session = aiohttp.ClientSession(raise_for_status=True)
resp = await client_session.get(url, raise_for_status=False)
async with resp:
    assert resp.status == 200
```

Set the parameter to *True* if you need *raise_for_status* for most of cases but override *raise_for_status* for those requests where you need to handle responses with status 400 or higher.

You can also provide a coroutine which takes the response as an argument and can raise an exception based on custom logic, e.g.:

```python
async def custom_check(response):
    if response.status not in (201, 202):
        raise RuntimeError('expected either 201 or 202')
    text = await response.text()
    if 'apple pie' not in text:
        raise RuntimeError('I wanted to see "apple pie" in response')
```

(continues on next page)
As with boolean values, you’re free to set this on the session and/or overwrite it on a per-request basis.

Changed in version 4.0: Async callback support is added.

- **timeout** –
  
  a `ClientTimeout` settings structure, **300 seconds (5min)** total timeout by default.

  New in version 3.3.

- **connector_owner**(bool) – Close connector instance on session closing.

  Setting the parameter to `False` allows to share connection pool between sessions without sharing session state: cookies etc.

- **auto_decompress**(bool) –

  Automatically decompress response body, `True` by default

  New in version 2.3.

- **trust_env**(bool) – Get proxies information from `HTTP_PROXY / HTTPS_PROXY` environment variables if the parameter is `True` (`False` by default).

  Get proxy credentials from `~/.netrc` file if present.

  **See also:**


  New in version 2.3.

  Changed in version 3.0: Added support for `~/.netrc` file.

- **requote_redirect_url**(bool) –

  Apply URL requoting for redirection URLs if automatic redirection is enabled (`True` by default).

  New in version 3.5.

- **trace_configs** – A list of `TraceConfig` instances used for client tracing. `None` (default) is used for request tracing disabling. See **Tracing Reference** for more information.

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

  A read-only property.

  aiohttp.`connector`

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

  A read-only property.

  aiohttp.`cookie_jar`

  The session cookies, `AbstractCookieJar` instance.

  Gives access to cookie jar’s content and modifiers.

  A read-only property.

**12.1. Client**
aiohttp.requote_redirect_url

aiohttp requote’s redirect urls by default, but some servers require exact url from location header. To disable re-quote system create `ClientSession with requote_redirect_url=False`.

aiohttp.timeout

Default client timeouts, `ClientTimeout` instance. The value can be tuned by passing `timeout` parameter to `ClientSession constructor`.

New in version 3.7.

aiohttp.headers

HTTP Headers that sent with every request

May be either iterable of key-value pairs or Mapping (e.g. `dict`, `CIMultiDict`).

New in version 3.7.

aiohttp.skip_auto_headers

Set of headers for which autogeneration skipped.

`frozenset of str` or `istr` (optional)

New in version 3.7.

aiohttp.auth

An object that represents HTTP Basic Authorization.

`BasicAuth` (optional)

New in version 3.7.

aiohttp.json_serialize

Json serializer callable.

By default `json.dumps()` function.

New in version 3.7.

aiohttp.connector_owner

Should connector be closed on session closing

`bool` (optional)

New in version 3.7.

aiohttp.raise_for_status

Should `ClientResponse.raise_for_status()` be called for each response

Either `bool` or callable

New in version 3.7.

aiohttp.auto_decompress

Should the body response be automatically decompressed

`bool` default is True

New in version 3.7.

aiohttp.trust_env

Should get proxies information from HTTP_PROXY / HTTPS_PROXY environment variables or ~/.netrc file if present

`bool` default is False

New in version 3.7.
aiohttp.trace_config
A list of TraceConfig instances used for client tracing. None (default) is used for request tracing disabling. See Tracing Reference for more information.

New in version 3.7.

coroutine async-with aiohttp.request (method, url, *, params=None, data=None, json=None, cookies=None, headers=None, skip_auto_headers=None, auth=None, allow_redirects=True, max_redirects=10, compress=None, chunked=None, expect100=False, raise_for_status=None, read_until_eof=True, proxy=None, proxy_auth=None, timeout=sentinel, ssl=None, proxy_headers=None)

Performs an asynchronous HTTP request. Returns a response object.

Parameters

- **method** *(str)* – HTTP method
- **url** – Request URL, *str* or 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. Ignored for subsequent redirected requests (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** – The data to send in the body of the request. This can be a FormData object or anything that can be passed into FormData, e.g. a dictionary, bytes, or file-like object. (optional)

- **json** – Any json compatible python object (optional). json and data parameters could not be used at the same time.

- **cookies** *(dict)* –

HTTP Cookies to send with the request (optional)

Global session cookies and the explicitly set cookies will be merged when sending the request.

New in version 3.5.

- **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 *istr* (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).

• **max_redirects (int)** – Maximum number of redirects to follow. 10 by default.

• **compress (bool)** – Set to True if request has to be compressed with deflate encoding. If compress can not be combined with a `Content-Encoding` and `Content-Length` headers. None by default (optional).

• **chunked (bool)** – Enable chunked transfer encoding. It is up to the developer to decide how to chunk data streams. If chunking is enabled, aiohttp encodes the provided chunks in the “Transfer-encoding: chunked” format. If chunked is set, then the `Transfer-encoding` and `content-length` headers are disallowed. None by default (optional).

• **expect100 (bool)** – Expect 100-continue response from server. False by default (optional).

• **callable] raise_for_status (Union[bool,])** – Automatically apply a check for failed status codes (usually a code that is greater than or equal to 400). None (default) means that raise_for_status argument of ClientSession constructor controls this behavior.

Set the parameter to True if you need to enforce `ClientResponse.raise_for_status()` call.

Set the argument to False to suppress a HTTP status checker even if ClientSession enables it.

Use an async callback to call user code that accepts a `ClientResponse` and raises an exception to prevent future processing.

The following callback example is a functional equivalent of raise_for_status=True:

```python
async def custom_check(response):
    if 400 <= response.status:
        raise aiohttp.ClientResponseError(
            response.request_info,
            response.history,
            status=response.status,
            message=response.reason,
            headers=response.headers)

client_session = aiohttp.ClientSession(raise_for_status=custom_check)
...
```

Changed in version 4.0.

• **read_until_eof (bool)** – Read response until EOF if response does not have Content-Length header. True by default (optional).

• **proxy** – Proxy URL, str or URL (optional)

• **proxy_auth (aiohttp.BasicAuth)** – an object that represents proxy HTTP Basic Authorization (optional)

• **timeout (int)** – override the session’s timeout.

Changed in version 3.3: The parameter is `ClientTimeout` instance, float is still supported for sake of backward compatibility.
If float is passed it is a total timeout (in seconds).

• **ssl** –

  SSL validation mode. None for default SSL check (ssl.
  create_default_context() is used), False for skip SSL certificate valida-
  tion, aiohttp.Fingerprint for fingerprint validation, ssl.SSLContext
  for custom SSL certificate validation.

 New in version 3.0.

• **proxy_headers** (**abc.Mapping**) – HTTP headers to send to the proxy if the param-
  eter proxy has been provided.

 New in version 2.3.

• **trace_request_ctx** – Object used to give as a kw param for each new
  TraceConfig object instantiated, used to give information to the tracers that is only
  available at request time.

 New in version 3.0.

Return ClientResponse a client response object.

```python
coroutine async-with aiohttp.get(url, *, allow_redirects=True, **kwargs)
```

Perform a GET request.

In order to modify inner request parameters, provide kwargs.

Parameters

• **url** – Request URL, str or URL

• **allow_redirects** (bool) – If set to False, do not follow redirects. True by default
  (optional).

Return ClientResponse a client response object.

```python
coroutine async-with aiohttp.post(url, *, data=None, **kwargs)
```

Perform a POST request.

In order to modify inner request parameters, provide kwargs.

Parameters

• **url** – Request URL, str or URL

• **data** – Data to send in the body of the request; see request for details (optional)

Return ClientResponse a client response object.

```python
coroutine async-with aiohttp.put(url, *, data=None, **kwargs)
```

Perform a PUT request.

In order to modify inner request parameters, provide kwargs.

Parameters

• **url** – Request URL, str or URL

• **data** – Data to send in the body of the request; see request for details (optional)

Return ClientResponse a client response object.

```python
coroutine async-with aiohttp.delete(url, **kwargs)
```

Perform a DELETE request.

In order to modify inner request parameters, provide kwargs.
Parameters `url` – Request URL, str or URL

Return ClientResponse a `client response` object.

coroutine async-with aiohttp.head (`url`, *, `allow_redirects=False`, **kwargs)
Perform a HEAD request.

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

Parameters

• `url` – Request URL, str or URL

• `allow_redirects` (bool) – If set to False, do not follow redirects. False by default (optional).

Return ClientResponse a `client response` object.

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

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

Parameters

• `url` – Request URL, str or URL

• `allow_redirects` (bool) – If set to False, do not follow redirects. True by default (optional).

Return ClientResponse a `client response` object.

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

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

Parameters

• `url` – Request URL, str or URL

• `data` – Data to send in the body of the request; see request for details (optional)

Return ClientResponse a `client response` object.

coroutine async-with aiohttp.ws_connect (`url`, *, `method='GET'`, `protocols=()`, `timeout=sentinel`, `auth=None`, `autoclose=True`, `autoping=True`, `heartbeat=None`, `origin=None`, `headers=None`, `proxy=None`, `proxy_auth=None`, `ssl=None`, `proxy_headers=None`, `compress=0`, `max_msg_size=4194304`)
Create a websocket connection. Returns a `ClientWebSocketResponse` object.

Parameters

• `url` – Websocket server url, str or URL

• `protocols` (tuple) – Websocket protocols

• `timeout` – a `ClientWSTimeout` timeout for websocket. By default, the value `Clie-
tWSTimeout(ws_receive=None, ws_close=10.0)` is used (10.0 seconds for the websocket to close). None means no timeout will be used.

• `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 then close procedure has to be handled manually. *True* by default

• **autoping** (*bool*) – automatically send pong on ping message from server. *True* by default

• **heartbeat** (*float*) – Send ping message every heartbeat seconds and wait pong response, if pong response is not received then close connection. The timer is reset on any data reception.(optional)

• **origin** (*str*) – Origin header to send to server(optional)

• **headers** (*dict*) – HTTP Headers to send with the request (optional)

• **proxy** (*str*) – Proxy URL, *str* or URL (optional)

• **proxy_auth** (*aiohttp.BasicAuth*) – an object that represents proxy HTTP Basic Authorization (optional)

• **ssl** –

SSL validation mode. *None* for default SSL check (*ssl.
create_default_context()* is used), *False* for skip SSL certificate validation, *aiohttp.Fingerprint* for fingerprint validation, *ssl.SSLContext* for custom SSL certificate validation.

New in version 3.0.

• **proxy_headers** (*dict*) – HTTP headers to send to the proxy if the parameter proxy has been provided.

New in version 2.3.

• **compress** (*int*) –

Enable Per-Message Compress Extension support. 0 for disable, 9 to 15 for window bit support. Default value is 0.

New in version 2.3.

• **max_msg_size** (*int*) –

maximum size of read websocket message, 4 MB by default. To disable the size limit use 0.

New in version 3.3.

• **method** (*str*) –

HTTP method to establish WebSocket connection, 'GET' by default.

New in version 3.5.

coroutine **aiohttp.close**()

Close underlying connector.

Release all acquired resources.

**aiohttp.detach**()

Detach connector from session without closing the former.

Session is switched to closed state anyway.
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, json=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, raise_for_status=False, connector=None, read_until_eof=True, timeout=sentinel)*

Async-with

Asynchronous context manager for performing an asynchronous HTTP request. Returns a ClientResponse response object.

Parameters

- **method** *(str)* – HTTP method
- **url** – Requested URL, *str* or *URL*
- **params** *(dict)* – Parameters to be sent in the query string of the new request (optional)
- **data** – The data to send in the body of the request. This can be a FormData object or anything that can be passed into FormData, e.g. a dictionary, bytes, or file-like object. (optional)
- **json** – Any json compatible python object (optional). *json* and *data* parameters could not be used at the same time.
- **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. None by default (optional).
- **chunked** *(int)* – Enables chunked transfer encoding. None by default (optional).
- **expect100** *(bool)* – Expect 100-continue response from server. False by default (optional).
- **raise_for_status** *(bool)* –
  Automatically call ClientResponse.raise_for_status() for response if set to True. If set to None value from ClientSession will be used. None by default (optional).

New in version 3.4.

- **connector** *(aiohttp.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).
• **timeout** – a `ClientTimeout` settings structure, 300 seconds (5min) total timeout by default.

Usage:

```python
import aiohttp

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

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. **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 support *keep-alive connections* (behavior is controlled by `force_close` constructor’s parameter).

**BaseConnector**

```python
class aiohttp.BaseConnector(*, keepalive_timeout=15, force_close=False, limit=100, limit_per_host=0, enable_cleanup_closed=False)
```

Base class for all connectors.

**Parameters**

- **keepalive_timeout** (*float*) – timeout for connection reusing after releasing (optional). Values 0. For disabling keep-alive feature use `force_close=True` flag.
- **limit** (*int*) – total number simultaneous connections. If `limit` is `None` the connector has no limit (default: 100).
- **limit_per_host** (*int*) – limit simultaneous connections to the same endpoint. Endpoints are the same if they are have equal `(host, port, is_ssl)` triple. If `limit` is 0 the connector has no limit (default: 0).
- **force_close** (*bool*) – close underlying sockets after connection releasing (optional).
- **enable_cleanup_closed** (*bool*) – some SSL servers do not properly complete SSL shutdown process, in that case asyncio leaks ssl connections. If this parameter is set to True, aiohttp additionally aborts underlining transport after 2 seconds. It is off by default.

**closed**

Read-only property, True if connector is closed.

**force_close**

Read-only property, True if connector should ultimately close connections on releasing.
**limit**
The total number for simultaneous connections. If limit is 0 the connector has no limit. The default limit size is 100.

**limit_per_host**
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_per_host** is None the connector has no limit per host.
Read-only property.

**coroutine close()**
Close all open connections (and await them to close).

**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.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**

```python
class aiohttp.TCPConnector(*, ssl=None, use_dns_cache=True, ttl_dns_cache=10, family=0, local_addr=None, resolver=None, keepalive_timeout=sentinel, force_close=False, limit=100, limit_per_host=0, enable_cleanup_closed=False)
```

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:

**param ssl**
SSL validation mode. **None** for default SSL check (**ssl.create_default_context()** is used), **False** for skip SSL certificate validation, **aiohttp.Fingerprint** for fingerprint validation, **ssl.SSLContext** for custom SSL certificate validation.

New in version 3.0.

**Parameters**

- **use_dns_cache (bool)** – use internal cache for DNS lookups, **True** by default.
  
  Enabling an option may speedup connection establishing a bit but may introduce some side effects also.

- **ttl_dns_cache (int)** – expire after some seconds the DNS entries, **None** means cached forever. By default 10 seconds (optional).
In some environments the IP addresses related to a specific HOST can change after a specific time. Use this option to keep the DNS cache updated refreshing each entry after N seconds.

- **limit** (*int*) – total number simultaneous connections. If *limit* is *None* the connector has no limit (default: 100).

- **limit_per_host** (*int*) – limit simultaneous connections to the same endpoint. Endpoints are the same if they are have equal `(host, port, is_ssl)` triple. If *limit* is 0 the connector has no limit (default: 0).

- **resolver** (*aiohttp.abc.AbstractResolver*) – custom resolver instance to use. `aiohttp.DefaultResolver` by default.

  Custom resolvers allow to resolve hostnames differently than the way the host is configured. The resolver is `aiohttp.ThreadedResolver` by default. Asynchronous version `aiohttp.AsyncResolver` (requires `aiodns>=1.1`) is pretty robust but might fail in very rare cases.

- **family** (*int*) – TCP socket family, both IPv4 and IPv6 by default. For IPv4 only use socket.AF_INET, for IPv6 only – socket.AF_INET6.

  *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.

- **local_addr** (*tuple*) – tuple of `(local_host, local_port)` used to bind socket locally if specified.

- **force_close** (*bool*) – close underlying sockets after connection releasing (optional).

- **enable_cleanup_closed** (*bool*) – Some ssl servers do not properly complete SSL shutdown process, in that case asyncio leaks SSL connections. If this parameter is set to True, aiohttp additionally aborts underlining transport after 2 seconds. It is off by default.

### 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.

### cached_hosts

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

Read-only `types.MappingProxyType` property.

### clear_dns_cache

Clear internal DNS cache.

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

class aiohttp.UnixConnector(path, *, conn_timeout=None, keepalive_timeout=30, limit=100, force_close=False)

Unix socket connector.

Use UnixConnector 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:

```python
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.

transport
Connection transport

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.
Response object

class aiohttp.ClientResponse
Client response returned by 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.:

```python
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).

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".

method
Request’s method (str).

url
URL of request (URL).

real_url
Unmodified URL of request with URL fragment unstripped (URL).

New in version 3.2.

connection
Connection used for handling response.

content
Payload stream, which contains response’s BODY (StreamReader). It supports various reading methods depending on the expected format. When chunked transfer encoding is used by the server, allows retrieving the actual http chunks.

Reading from the stream may raise aiohttp.ClientPayloadError if the response object is closed before response receives all data or in case if any transfer encoding related errors like misformed chunked encoding of broken compression data.

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

headers
A case-insensitive multidict proxy with HTTP headers of response, CIMultiDictProxy.

raw_headers
Unmodified HTTP headers of response as unconverted bytes, a sequence of (key, value) pairs.

links
Link HTTP header parsed into a MultiDictProxy.

For each link, key is link param rel when it exists, or link url as str otherwise, and value is MultiDictProxy of link params and url at key url as URL instance.

New in version 3.2.
content_type
Read-only property with content part of Content-Type header.

Note: Returns value is 'application/octet-stream' if no Content-Type header present in HTTP headers according to RFC 2616. To make sure Content-Type header is not present in the server reply, use headers or raw_headers, e.g. 'CONTENT-TYPE' not in resp.headers.

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 no Content-Type header present in HTTP headers or it has no charset information.

content_disposition
Read-only property that specified the Content-Disposition HTTP header.

Instance of ContentDisposition or None if no Content-Disposition header present in HTTP headers.

history
A Sequence of ClientResponse objects of preceding requests (earliest request first) 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.

Raise an aiohttp.ClientResponseError if the data can’t be read.

Return bytes read BODY.

See also:
close(), release().

coroutine release()
It is not required to call release on the response object. When the client fully receives the payload, the underlying connection automatically returns back to pool. If the payload is not fully read, the connection is closed.

raise_for_status()
Raise an aiohttp.ClientResponseError if the response status is 400 or higher.

Do nothing for success responses (less than 400).

coroutine text(encoding=None, errors='strict')
Read response’s body and return decoded str using specified encoding parameter.

If encoding is None content encoding is autocalculated using Content-Type HTTP header and chardet tool if the header is not provided by server.

cchardet is used with fallback to chardet 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).
- **errors** *(str)* – error handling scheme, see Error Handlers for more details. 'strict' by default.

Return *str* decoded BODY

Raises **LookupError** – if the encoding detected by chardet or cchardet is unknown by Python (e.g. VISCII).

Note: If response has no charset info in Content-Type HTTP header *cchardet / chardet* is used for content encoding autodetection.

It may hurt performance. If page encoding is known passing explicit *encoding* parameter might help:

```python
await resp.text('ISO-8859-1')
```

coroutine **json**(*, encoding=None, loads=json.loads, content_type='application/json')

Read response's body as JSON, return dict using specified *encoding* and *loader*. If data is not still available a read call will be done,

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

if response's *content-type* does not match *content_type* parameter *aiohttp.ContentTypeError* get raised. To disable content type check pass *None* value.

Parameters

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

By the standard JSON encoding should be UTF-8 but practice beats purity: some servers return non-UTF responses. Autodetection works pretty fine anyway.

- **loads** *(callable)* – *callable()* used for loading JSON data, *json.loads()* by default.

- **content_type** *(str)* – specify response's content-type, if content type does not match raise *aiohttp.ClientResponseError*. To disable content-type check, pass *None* as value. (default: *application/json*).

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

**request_info**

A namedtuple with request URL and headers from *ClientRequest* object, *aiohttp.RequestInfo* instance.

**get_encoding()**

Automatically detect content encoding using charset info in Content-Type HTTP header. If this info is not exists or there are no appropriate codecs for encoding then *cchardet / chardet* is used.

Beware that it is not always safe to use the result of this function to decode a response. Some encodings detected by cchardet are not known by Python (e.g. VISCII).

Raises **RuntimeError** – if called before the body has been read, for *cchardet* usage

New in version 3.0.
ClientWebSocketResponse

To connect to a websocket server `aiohttp.ws_connect()` or `aiohttp.ClientSession.ws_connect()` coroutines should be used, do not create an instance of class `ClientWebSocketResponse` manually.

class aiohttp.ClientWebSocketResponse
Class for handling client-side websockets.

  closed
Read-only property, True if `close()` has been called or 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.

  get_extra_info(name, default=None)
Reads extra info from connection’s transport

  exception()
      Returns exception if any occurs or returns None.

  coroutine ping(message=b'')
Send PING to peer.

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

      Changed in version 3.0: The method is converted into coroutine

  coroutine pong(message=b'')
Send PONG to peer.

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

      Changed in version 3.0: The method is converted into coroutine

  coroutine send_str(data, compress=None)
Send data to peer as TEXT message.

      Parameters

          • data (str) – data to send.

          • compress (int) – sets specific level of compression for single message, None for not overriding per-socket setting.

      Raises TypeError – if data is not str

      Changed in version 3.0: The method is converted into coroutine, compress parameter added.

  coroutine send_bytes(data, compress=None)
Send data to peer as BINARY message.

      Parameters

          • data – data to send.

          • compress (int) – sets specific level of compression for single message, None for not overriding per-socket setting.

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

      Changed in version 3.0: The method is converted into coroutine, compress parameter added.
coroutine send_json (data, compress=None, *, dumps=json.dumps)
Send data to peer as JSON string.

Parameters
- **data** – data to send.
- **compress** *(int)* – sets specific level of compression for single message, None for not overriding per-socket setting.
- **dumps** *(callable)* – any callable that accepts an object and returns a JSON string (json.dumps() by default).

Raises
- **RuntimeError** – if connection is not started or closing
- **ValueError** – if data is not serializable object
- **TypeError** – if value returned by dumps(data) is not str

Changed in version 3.0: The method is converted into coroutine, compress parameter added.

coroutine close (*, code=1000, message=b'*)
A coroutine that initiates closing handshake by sending **CLOSE** message. It waits for close response from server. To add a timeout to close() call just wrap the call with asyncio.wait() or asyncio.wait_for().

Parameters
- **code** *(int)* – closing code
- **message** – optional payload of close 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 **PING, PONG** and **CLOSE** without returning the message.

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

Returns **WSMessage**

coroutine receive_str ()
A coroutine that calls receive() but also asserts the message type is **TEXT**.

Return str peer’s message content.

Raises **TypeError** – if message is **BINARY**.

coroutine receive_bytes ()
A coroutine that calls receive() but also asserts the message type is **BINARY**.

Return bytes peer’s message content.

Raises **TypeError** – if message is **TEXT**.

coroutine receive_json (*, loads=json.loads)
A coroutine that calls receive_str() and loads the JSON string to a Python dict.

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

Return dict loaded JSON content

Raises
- **TypeError** – if message is **BINARY**.
Utilities

ClientTimeout

class aiohttp.ClientTimeout (*, total=None, connect=None, sock_connect, sock_read=None)

A data class for client timeout settings.

See Timeouts for usage examples.

total

Total number of seconds for the whole request.

float, None by default.

connect

Maximal number of seconds for acquiring a connection from pool. The time consists of connection establishment for a new connection or waiting for a free connection from a pool if pool connection limits are exceeded.

For pure socket connection establishment time use sock_connect.

float, None by default.

sock_connect

Maximal number of seconds for connecting to a peer for a new connection, not given from a pool. See also connect.

float, None by default.

sock_read

Maximal number of seconds for reading a portion of data from a peer.

float, None by default.

class aiohttp.ClientWSTimeout (*, ws_receive=None, ws_close=None)

A data class for websocket client timeout settings.

ws_receive

A timeout for websocket to receive a complete message.

float, None by default.

ws_close

A timeout for the websocket to close.

float, 10.0 by default.

New in version 4.0.
RequestInfo

class aiohttp.RequestInfo
A data class with request URL and headers from ClientRequest object, available as ClientResponse.
    request_info attribute.

    url
    Requested url, yarl.URL instance.

    method
    Request HTTP method like 'GET' or 'POST', str.

    headers
    HTTP headers for request, multidict.CIMultiDict instance.

    real_url
    Requested url with URL fragment unstripped, yarl.URL instance.
    New in version 3.2.

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().

classmethod decode(auth_header, encoding='latin1')
Decode HTTP basic authentication credentials.

Parameters
• auth_header (str) – The Authorization header to decode.
• encoding (str) – (optional) encoding ('latin1' by default)

Returns decoded authentication data, BasicAuth.

classmethod from_url(url)
Constructed credentials info from url’s user and password parts.

Returns credentials data, BasicAuth or None is credentials are not provided.
    New in version 2.3.

encode()
Encode credentials into string suitable for Authorization header etc.

Returns encoded authentication data, str.
CookieJar

class aiohttp.CookieJar(*, unsafe=False)
    The cookie jar instance is available as ClientSession.cookie_jar.
    The jar contains Morsel items for storing internal cookie data.
    API provides a count of saved cookies:
    
    ```
    len(session.cookie_jar)
    ```
    
    These cookies may be iterated over:
    
    ```
    for cookie in session.cookie_jar:
        print(cookie.key)
        print(cookie['domain'])
    ```
    
    The class implements collections.abc.Iterable, collections.abc.Sized and aiohttp.AbstractCookieJar interfaces.
    Implements cookie storage adhering to RFC 6265.

    Parameters unsafe (bool) – (optional) Whether to accept cookies from IPs.

    update_cookies (cookies, response_url=None)
        Update cookies returned by server in Set-Cookie header.
        Parameters
        • cookies – a collections.abc.Mapping (e.g. dict, SimpleCookie) or iterable of pairs with cookies returned by server’s response.
        • response_url (str) – URL of response, None for shared cookies. Regular cookies are coupled with server’s URL and are sent only to this server, shared ones are sent in every client request.

    filter_cookies (request_url)
        Return jar’s cookies acceptable for URL and available in Cookie header for sending client requests for given URL.
        Parameters response_url (str) – request’s URL for which cookies are asked.
        Returns http.cookies.SimpleCookie with filtered cookies for given URL.

    save (file_path)
        Write a pickled representation of cookies into the file at provided path.
        Parameters file_path – Path to file where cookies will be serialized, str or pathlib.Path instance.

    load (file_path)
        Load a pickled representation of cookies from the file at provided path.
        Parameters file_path – Path to file from where cookies will be imported, str or pathlib.Path instance.

    class aiohttp.DummyCookieJar
        Dummy cookie jar which does not store cookies but ignores them.
        Could be useful e.g. for web crawlers to iterate over Internet without blowing up with saved cookies information.
        To install dummy cookie jar pass it into session instance:
class aiohttp.Fingerprint(digest)
Fingerprint helper for checking SSL certificates by SHA256 digest.
Useful for certificate pinning.

Parameters digest (bytes) – SHA256 digest for certificate in DER-encoded binary form (see ssl.SSLSocket.getpeercert()).

To check fingerprint pass the object into ClientSession.get() call, e.g.:

```python
import hashlib
with open(path_to_cert, 'rb') as f:
    digest = hashlib.sha256(f.read()).digest()
await session.get(url, ssl=aiohttp.Fingerprint(digest))
```

New in version 3.0.

**FormData**

A FormData object contains the form data and also handles encoding it into a body that is either multipart/form-data or application/x-www-form-urlencoded. multipart/form-data is used if at least one field is an io.IOBase object or was added with at least one optional argument to add_field (content_type, filename, or content_transfer_encoding). Otherwise, application/x-www-form-urlencoded is used.

FormData instances are callable and return a Payload on being called.

class aiohttp.FormData(fields, quote_fields=True, charset=None)
Helper class for multipart/form-data and application/x-www-form-urlencoded body generation.

Parameters fields – A container for the key/value pairs of this form.

Possible types are:

- dict
- tuple or list
- io.IOBase, e.g. a file-like object
- multidict.MultiDict or multidict.MultiDictProxy

If it is a tuple or list, it must be a valid argument for add_fields.

For dict, multidict.MultiDict, and multidict.MultiDictProxy, the keys and values must be valid name and value arguments to add_field, respectively.

add_field (name, value, content_type=None, filename=None, content_transfer_encoding=None)
Add a field to the form.

Parameters

- name (str) – Name of the field
- value – Value of the field

Possible types are:
- str
- bytes, bytearray, or memoryview
- io.IOBase, e.g. a file-like object

- **content_type** (str) – The field’s content-type header (optional)

- **filename** (str) – The field’s filename (optional)

  If this is not set and value is a bytes, bytearray, or memoryview object, the name argument is used as the filename unless content_transfer_encoding is specified.

  If filename is not set and value is an io.IOBase object, the filename is extracted from the object if possible.

- **content_transfer_encoding** (str) – The field’s content-transfer-encoding header (optional)

  **add_fields** *(fields)*

  Add one or more fields to the form.

  **Parameters fields** – An iterable containing:

  - io.IOBase, e.g. a file-like object
  - multidict.MultiDict or multidict.MultiDictProxy
  - tuple or list of length two, containing a name-value pair

### Client exceptions

Exception hierarchy has been significantly modified in version 2.0. aiohttp defines only exceptions that covers connection handling and server response misbehaviors. For developer specific mistakes, aiohttp uses python standard exceptions like `ValueError` or `TypeError`.

Reading a response content may raise a `ClientPayloadError` exception. This exception indicates errors specific to the payload encoding. Such as invalid compressed data, malformed chunked-encoded chunks or not enough data that satisfy the content-length header.

All exceptions are available as members of `aiohttp` module.

**exception** aiohttp.ClientError  
Base class for all client specific exceptions.

Derived from `Exception`

**class** aiohttp.ClientPayloadError  
This exception can only be raised while reading the response payload if one of these errors occurs:

1. invalid compression
2. malformed chunked encoding
3. not enough data that satisfy Content-Length HTTP header.

Derived from `ClientError`

**exception** aiohttp.InvalidURL  
URL used for fetching is malformed, e.g. it does not contain host part.

Derived from `ClientError` and `ValueError`
url
Invalid URL, \texttt{yarl.URL} instance.

class \texttt{aiohttp.ContentDisposition}
Represent Content-Disposition header

\textbf{value}
A \texttt{str} instance. Value of Content-Disposition header itself, e.g. \texttt{attachment}.

\textbf{filename}
A \texttt{str} instance. Content filename extracted from parameters. May be \texttt{None}.

\textbf{parameters}
Read-only mapping contains all parameters.

\textbf{Response errors}

\textbf{exception \texttt{aiohttp.ClientResponseError}}
These exceptions could happen after we get response from server.

Derived from \texttt{ClientError}

\textbf{request\_info}
Instance of \texttt{RequestInfo} object, contains information about request.

\textbf{status}
HTTP status code of response (\texttt{int}), e.g. \texttt{400}.

\textbf{message}
Message of response (\texttt{str}), e.g. "\texttt{OK}".

\textbf{headers}
Headsers in response, a list of pairs.

\textbf{history}
History from failed response, if available, else empty tuple.

A tuple of \texttt{ClientResponse} objects used for handle redirection responses.

\textbf{code}
HTTP status code of response (\texttt{int}), e.g. \texttt{400}.

Deprecated since version 3.1.

class \texttt{aiohttp.WSServerHandshakeError}
Web socket server response error.

Derived from \texttt{ClientResponseError}

class \texttt{aiohttp.ContentTypeError}
Invalid content type.

Derived from \texttt{ClientResponseError}

New in version 2.3.

class \texttt{aiohttp.TooManyRedirects}
Client was redirected too many times.

Maximum number of redirects can be configured by using parameter \texttt{max\_redirects} in \texttt{request}.

Derived from \texttt{ClientResponseError}
New in version 3.2.

**Connection errors**

```python
class aiohttp.ClientConnectionError
    These exceptions related to low-level connection problems.
    Derived from ClientError

class aiohttp.ClientOSError
    Subset of connection errors that are initiated by an OSError exception.
    Derived from ClientConnectionError and OSError

class aiohttp.ClientConnectorError
    Connector related exceptions.
    Derived from ClientOSError

class aiohttp.ClientProxyConnectionError
    Derived from ClientConnectorError

class aiohttp.ServerConnectionError
    Derived from ClientConnectionError

class aiohttp.ClientSSLError
    Derived from ClientConnectorError

class aiohttp.ClientConnectorSSLError
    Response ssl error.
    Derived from ClientSSLError and ssl.SSLError

class aiohttp.ClientConnectorCertificateError
    Response certificate error.
    Derived from ClientSSLError and ssl.CertificateError

class aiohttp.ServerDisconnectedError
    Server disconnected.
    Derived from ServerDisconnectionError

    message
        Partially parsed HTTP message (optional).

class aiohttp.ServerTimeoutError
    Server operation timeout: read timeout, etc.
    Derived from ServerConnectionError and asyncio.TimeoutError

class aiohttp.ServerFingerprintMismatch
    Server fingerprint mismatch.
    Derived from ServerConnectionError
```
Hierarchy of exceptions

- ClientError
  - ClientResponseError
    - ContentTypeError
    - WSServerHandshakeError
    - ClientHttpProxyError
  - ClientConnectionError
    - ClientOSError
      - ClientConnectorError
      - ClientSSLError
      - ClientConnectorCertificateError
      - ClientConnectorSSLError
      - ClientProxyConnectionError
      - ServerConnectionError
      - ServerDisconnectedError
      - ServerTimeoutError
      - ServerFingerprintMismatch
  - ClientPayloadError
  - InvalidURL

12.1.4 Tracing Reference

New in version 3.0.
A reference for client tracing API.

See also:

*Client Tracing* for tracing usage instructions.

Request life cycle

A request goes through the following stages and corresponding fallbacks.
Overview

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>start</td>
<td>on_request_start</td>
</tr>
<tr>
<td>redirect</td>
<td>on_request_redirect</td>
</tr>
<tr>
<td>acquire_connection</td>
<td>Connection acquiring</td>
</tr>
<tr>
<td>headers_received</td>
<td></td>
</tr>
<tr>
<td>exception</td>
<td>on_request_exception</td>
</tr>
<tr>
<td>end</td>
<td>on_request_end</td>
</tr>
<tr>
<td>headers_sent</td>
<td></td>
</tr>
<tr>
<td>chunk_sent</td>
<td>on_request_chunk_sent</td>
</tr>
<tr>
<td>chunk_received</td>
<td>on_response_chunk_received</td>
</tr>
</tbody>
</table>
Connection acquiring

```
begin
  queued_start
    queued_end
      create_start
        resolve_dns
          sock_connect
            exception
              create_end
                end
```

12.1. Client
## DNS resolving

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>queued_start</td>
<td>on_connection_queued_start</td>
</tr>
<tr>
<td>create_start</td>
<td>on_connection_create_start</td>
</tr>
<tr>
<td>reuseconn</td>
<td>on_connection_reuseconn</td>
</tr>
<tr>
<td>queued_end</td>
<td>on_connection_queued_end</td>
</tr>
<tr>
<td>create_end</td>
<td>on_connection_create_end</td>
</tr>
<tr>
<td>exception</td>
<td>Exception raised</td>
</tr>
<tr>
<td>resolve_dns</td>
<td>DNS resolving</td>
</tr>
<tr>
<td>sock_connect</td>
<td>Connection establishment</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin</td>
<td></td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
<tr>
<td>exception</td>
<td>Exception raised</td>
</tr>
<tr>
<td>resolve_end</td>
<td>on_dns_resolvehost_end</td>
</tr>
<tr>
<td>resolve_start</td>
<td>on_dns_resolvehost_start</td>
</tr>
<tr>
<td>cache_hit</td>
<td>on_dns_cache_hit</td>
</tr>
<tr>
<td>cache_miss</td>
<td>on_dns_cache_miss</td>
</tr>
</tbody>
</table>
**TraceConfig**

```python
class aiohttp.TraceConfig(trace_config_ctx_factory=SimpleNamespace)
```

Trace config is the configuration object used to trace requests launched by a `ClientSession` object using different events related to different parts of the request flow. Parameters:
- `trace_config_ctx_factory` — factory used to create trace contexts, default class used `types.SimpleNamespace`

```python
trace_config_ctx(trace_request_ctx=None)
```

Parameters:
- `trace_request_ctx` — Will be used to pass as a kw for the `trace_config_ctx_factory`.

Build a new trace context from the config.

Every signal handler should have the following signature:

```python
async def on_signal(session, context, params): ...
```

where `session` is `ClientSession` instance, `context` is an object returned by `trace_config_ctx()` call and `params` is a data class with signal parameters. The type of `params` depends on subscribed signal and described below.

- **on_request_start**
  
  Property that gives access to the signals that will be executed when a request starts.
  
  `params` is `aiohttp.TraceRequestStartParams` instance.

- **on_request_chunk_sent**
  
  Property that gives access to the signals that will be executed when a chunk of request body is sent.
  
  `params` is `aiohttp.TraceRequestChunkSentParams` instance.

  New in version 3.1.

- **on_response_chunk_received**
  
  Property that gives access to the signals that will be executed when a chunk of response body is received.
  
  `params` is `aiohttp.TraceResponseChunkReceivedParams` instance.

  New in version 3.1.

- **on_request_redirect**
  
  Property that gives access to the signals that will be executed when a redirect happens during a request flow.
  
  `params` is `aiohttp.TraceRequestRedirectParams` instance.

- **on_request_end**
  
  Property that gives access to the signals that will be executed when a request ends.
  
  `params` is `aiohttp.TraceRequestEndParams` instance.

- **on_request_exception**
  
  Property that gives access to the signals that will be executed when a request finishes with an exception.
  
  `params` is `aiohttp.TraceRequestExceptionParams` instance.

- **on_connection_queued_start**
  
  Property that gives access to the signals that will be executed when a request has been queued waiting for an available connection.
  
  `params` is `aiohttp.TraceConnectionQueuedStartParams` instance.
on_connection_queued_end
Property that gives access to the signals that will be executed when a request that was queued already has an available connection.
params is aiohttp.TraceConnectionQueuedEndParams instance.

on_connection_create_start
Property that gives access to the signals that will be executed when a request creates a new connection.
params is aiohttp.TraceConnectionCreateStartParams instance.

on_connection_create_end
Property that gives access to the signals that will be executed when a request that created a new connection finishes its creation.
params is aiohttp.TraceConnectionCreateEndParams instance.

on_connection_reuseconn
Property that gives access to the signals that will be executed when a request reuses a connection.
params is aiohttp.TraceConnectionReuseconnParams instance.

on_dns_resolvehost_start
Property that gives access to the signals that will be executed when a request starts to resolve the domain related with the request.
params is aiohttp.TraceDnsResolveHostStartParams instance.

on_dns_resolvehost_end
Property that gives access to the signals that will be executed when a request finishes to resolve the domain related with the request.
params is aiohttp.TraceDnsResolveHostEndParams instance.

on_dns_cache_hit
Property that gives access to the signals that will be executed when a request was able to use a cached DNS resolution for the domain related with the request.
params is aiohttp.TraceDnsCacheHitParams instance.

on_dns_cache_miss
Property that gives access to the signals that will be executed when a request was not able to use a cached DNS resolution for the domain related with the request.
params is aiohttp.TraceDnsCacheMissParams instance.

TraceRequestStartParams
class aiohttp.TraceRequestStartParams
See TraceConfig.on_request_start for details.

method
Method that will be used to make the request.

url
URL that will be used for the request.

headers
Headers that will be used for the request, can be mutated.
TraceRequestChunkSentParams

class aiohttp.TraceRequestChunkSentParams
   New in version 3.1.
   See `TraceConfig.on_request_chunk_sent` for details.
   chunk
      Bytes of chunk sent

TraceResponseChunkSentParams

class aiohttp.TraceResponseChunkSentParams
   New in version 3.1.
   See `TraceConfig.on_response_chunk_received` for details.
   chunk
      Bytes of chunk received

TraceRequestEndParams

class aiohttp.TraceRequestEndParams
   See `TraceConfig.on_request_end` for details.
   method
      Method used to make the request.
   url
      URL used for the request.
   headers
      Headers used for the request.
   response
      Response `ClientResponse`.

TraceRequestExceptionParams

class aiohttp.TraceRequestExceptionParams
   See `TraceConfig.on_request_exception` for details.
   method
      Method used to make the request.
   url
      URL used for the request.
   headers
      Headers used for the request.
   exception
      Exception raised during the request.
TraceRequestRedirectParams

class aiohttp.TraceRequestRedirectParams
    See TraceConfig.on_request_redirect for details.

    method
        Method used to get this redirect request.

    url
        URL used for this redirect request.

    headers
        Headers used for this redirect.

    response
        Response ClientResponse got from the redirect.

TraceConnectionQueuedStartParams

class aiohttp.TraceConnectionQueuedStartParams
    See TraceConfig.on_connection_queued_start for details.

    There are no attributes right now.

TraceConnectionQueuedEndParams

class aiohttp.TraceConnectionQueuedEndParams
    See TraceConfig.on_connection_queued_end for details.

    There are no attributes right now.

TraceConnectionCreateStartParams

class aiohttp.TraceConnectionCreateStartParams
    See TraceConfig.on_connection_create_start for details.

    There are no attributes right now.

TraceConnectionCreateEndParams

class aiohttp.TraceConnectionCreateEndParams
    See TraceConfig.on_connection_create_end for details.

    There are no attributes right now.
TraceConnectionReuseconnParams

class aiohttp.TraceConnectionReuseconnParams
    See TraceConfig.on_connection_reuseconn for details.
    There are no attributes right now.

TraceDnsResolveHostStartParams

class aiohttp.TraceDnsResolveHostStartParams
    See TraceConfig.on_dns_resolvehost_start for details.
    Host
        Host that will be resolved.

TraceDnsResolveHostEndParams

class aiohttp.TraceDnsResolveHostEndParams
    See TraceConfig.on_dns_resolvehost_end for details.
    Host
        Host that has been resolved.

TraceDnsCacheHitParams

class aiohttp.TraceDnsCacheHitParams
    See TraceConfig.on_dns_cache_hit for details.
    Host
        Host found in the cache.

TraceDnsCacheMissParams

class aiohttp.TraceDnsCacheMissParams
    See TraceConfig.on_dns_cache_miss for details.
    Host
        Host didn’t find the cache.

12.1.5 The aiohttp Request Lifecycle

Why is aiohttp client API that way?

The first time you use aiohttp, you’ll notice that a simple HTTP request is performed not with one, but with up to three steps:

```python
async with aiohttp.ClientSession() as session:
    async with session.get('http://python.org') as response:
        print(await response.text())
```

It’s especially unexpected when coming from other libraries such as the very popular requests, where the “hello world” looks like this:
So why is the aiohttp snippet so verbose?

Because aiohttp is asynchronous, its API is designed to make the most out of non-blocking network operations. In a code like this, requests will block three times, and does it transparently, while aiohttp gives the event loop three opportunities to switch context:

- When doing the `.get()`, both libraries send a GET request to the remote server. For aiohttp, this means asynchronous I/O, which is here marked with an `async with` that gives you the guaranty that not only it doesn’t block, but that it’s cleanly finalized.
- When doing `response.text` in requests, you just read an attribute. The call to `.get()` already preloaded and decoded the entire response payload, in a blocking manner. aiohttp loads only the headers when `.get()` is executed, letting you decide to pay the cost of loading the body afterward, in a second asynchronous operation. Hence the `await response.text()`.
- `async with aiohttp.ClientSession()` does not perform I/O when entering the block, but at the end of it, it will ensure all remaining resources are closed correctly. Again, this is done asynchronously and must be marked as such. The session is also a performance tool, as it manages a pool of connections for you, allowing you to reuse them instead of opening and closing a new one at each request. You can even manage the pool size by passing a connector object.

**Using a session as a best practice**

The requests library does in fact also provides a session system. Indeed, it lets you do:

```python
with requests.session() as session:
    response = session.get('http://python.org')
    print(response.text)
```

It just not the default behavior, nor is it advertised early in the documentation. Because of this, most users take a hit in performances, but can quickly start hacking. And for requests, it’s an understandable trade-off, since its goal is to be “HTTP for humans” and simplicity has always been more important than performance in this context.

However, if one uses aiohttp, one chooses asynchronous programming, a paradigm that makes the opposite trade-off: more verbosity for better performances. And so the library default behavior reflects this, encouraging you to use performant best practices from the start.

**How to use the ClientSession?**

By default the `aiohttp.ClientSession` object will hold a connector with a maximum of 100 connections, putting the rest in a queue. This is quite a big number, this means you must be connected to a hundred different servers (not pages!) concurrently before even having to consider if your task needs resource adjustment.

In fact, you can picture the session object as a user starting and closing a browser: it wouldn’t make sense to do that every time you want to load a new tab.

So you are expected to reuse a session object and make many requests from it. For most scripts and average-sized softwares, this means you can create a single session, and reuse it for the entire execution of the program. You can even pass the session around as a parameter in functions. E.G, the typical “hello world”:

```python
import aiohttp
import asyncio
(continues on next page)```
async def main():
    async with aiohttp.ClientSession() as session:
        async with session.get('http://python.org') as response:
            html = await response.text()
            print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())

Can become this:

```python
import aiohttp
import asyncio

async def fetch(session, url):
    async with session.get(url) as response:
        return await response.text()

async def main():
    async with aiohttp.ClientSession() as session:
        html = await fetch(session, 'http://python.org')
        print(html)

loop = asyncio.get_event_loop()
loop.run_until_complete(main())
```

On more complex code bases, you can even create a central registry to hold the session object from anywhere in the code, or a higher level `Client` class that holds a reference to it.

When to create more than one session object then? It arises when you want more granularity with your resources management:

- you want to group connections by a common configuration. E.G: sessions can set cookies, headers, timeout values, etc. that are shared for all connections they holds.
- you need several threads and want to avoid sharing a mutable object between them.
- you want several connection pools to benefit from different queues and assign priorities. E.G: one session never uses the queue and is for high priority requests, the other one has a small concurrency limit and a very long queue, for non important requests.
12.2 Server

The page contains all information about aiohttp Server API:

12.2.1 Web Server Quickstart

Run a Simple Web Server

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

A request handler must be a coroutine that accepts a Request instance as its only parameter and returns a Response instance:

```python
from aiohttp import web

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

Next, create an Application instance and register the request handler on a particular HTTP method and path:

```python
app = web.Application()
app.add_routes([web.get('/', hello)])
```

After that, run the application by run_app() call:

```python
web.run_app(app)
```

That’s it. Now, head over to http://localhost:8080/ to see the results.

Alternatively if you prefer route decorators create a route table and register a web-handler:

```python
routes = web.RouteTableDef()

@routes.get('/
async def hello(request):
    return web.Response(text="Hello, world")

app = web.Application()
app.add_routes(routes)
web.run_app(app)
```

Both ways essentially do the same work, the difference is only in your taste: do you prefer Django style with famous urls.py or Flask with shiny route decorators.

aiohttp server documentation uses both ways in code snippets to emphasize their equality, switching from one style to another is very trivial.

Note: You can get a powerful aiohttp template by running one command. To do this, simply use our boilerplate for quick start with aiohttp.

See also:
Graceful shutdown section explains what run_app() does and how to implement complex server initialization/finalization from scratch.
**Application runners** for more handling more complex cases like *asynchronous* web application serving and multiple hosts support.

**Command Line Interface (CLI)**

*aiohttp.web* implements a basic CLI for quickly serving an *Application* in *development* over TCP/IP:

```
$ python -m aiohttp.web -H localhost -P 8080 package.module:init_func
```

`package.module:init_func` should be an importable *callable* that accepts a list of any non-parsed command-line arguments and returns an *Application* instance after setting it up:

```python
def init_func(argv):
    app = web.Application()
    app.router.add_get('/', index_handler)
    return app
```

**Handler**

A request handler must be a *coroutine* that accepts a *Request* instance as its only argument and returns a *StreamResponse* derived (e.g. *Response*) instance:

```python
async def handler(request):
    return web.Response()
```

Handlers are setup to handle requests by registering them with the *Application.add_routes()* on a particular route (*HTTP method* and *path* pair) using helpers like *get()* and *post()*:

```python
app.add_routes([web.get('/', handler),
                web.post('/post', post_handler),
                web.put('/put', put_handler)])
```

Or use *route decorators*:

```python
routes = web.RouteTableDef()

@routes.get('/'
async def get_handler(request):
    ...

@routes.post('/post')
async def post_handler(request):
    ...

@routes.put('/put')
async def put_handler(request):
    ...

app.add_routes(routes)
```

Wildcard *HTTP method* is also supported by *route()* or *RouteTableDef.route()* allowing a handler to serve incoming requests on a *path* having any *HTTP method*:

```python
app.add_routes([web.route('*', '/path', all_handler)])
```

12.2. Server
The **HTTP method** can be queried later in the request handler using the `Request.method` property.

By default endpoints added with `GET` method will accept `HEAD` requests and return the same response headers as they would for a `GET` request. You can also deny `HEAD` requests on a route:

```python
web.get('/login', handler, allow_head=False)
```

Here `handler` won’t be called on `HEAD` request and the server will respond with 405: Method Not Allowed.

### Resources and Routes

Internally routes are served by `Application.router` (`UrlDispatcher` instance).

The `router` is a list of **resources**.

Resource is an entry in *route table* which corresponds to requested URL.

Resource in turn has at least one **route**.

Route corresponds to handling **HTTP method** by calling `web handler`.

Thus when you add a **route** the **resource** object is created under the hood.

The library implementation **merges** all subsequent route additions for the same path adding the only resource for all HTTP methods.

Consider two examples:

```python
app.add_routes([web.get('/path1', get_1),
                web.post('/path1', post_1),
                web.get('/path2', get_2),
                web.post('/path2', post_2)]
```

and:

```python
app.add_routes([web.get('/path1', get_1),
                web.get('/path2', get_2),
                web.post('/path2', post_2),
                web.post('/path1', post_1)]
```

First one is **optimized**. You have got the idea.

### Variable Resources

Resource may have **variable path** also. For instance, a resource with the path `/a/{name}/c` would match all incoming requests with paths such as `/a/b/c`, `/a/1/c`, and `/a/etc/c`.

A variable **part** is specified in the form `{identifier}`, where the **identifier** can be used later in a **request handler** to access the matched value for that **part**. This is done by looking up the **identifier** in the `Request.match_info` mapping:

```python
@routes.get('/{name}')
async def variable_handler(request):
    return web.Response(text="Hello, {}\n".format(request.match_info['name']))
```

By default, each **part** matches the regular expression `[^{}]+`.

You can also specify a custom regex in the form `{identifier:regex}`:
Reverse URL Constructing using Named Resources

Routes can also be given a name:

```python
@routes.get('/root', name='root')
async def handler(request):
    ...
```

Which can then be used to access and build a URL for that resource later (e.g. in a request handler):

```python
url = request.app.router['root'].url_for().with_query({"a": "b", "c": "d"})
assert url == URL('/root?a=b&c=d')
```

A more interesting example is building URLs for variable resources:

```python
app.router.add_resource(r'/{user}/info', name='user-info')
```

In this case you can also pass in the parts of the route:

```python
url = request.app.router['user-info'].url_for(user='john_doe')
url_with_qs = url.with_query("a=b")
assert url_with_qs == '/john_doe/info?a=b'
```

Organizing Handlers in Classes

As discussed above, handlers can be first-class coroutines:

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

app.router.add_get('/', hello)
```

But sometimes it’s convenient to group logically similar handlers into a Python class.

Since `aiohttp.web` does not dictate any implementation details, application developers can organize handlers in classes if they so wish:

```python
class Handler:
    def __init__(self):
        pass
    async def handle_intro(self, request):
        return web.Response(text="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()
```

(continues on next page)
app.add_routes([web.get('/intro', handler.handle_intro),
 web.get('/greet/{name}', handler.handle_greeting)])

Class Based Views

`aiohttp.web` has support for class based views.

You can derive from `View` and define methods for handling http requests:

```python
class MyView(web.View):
    async def get(self):
        return await get_resp(self.request)
    async def post(self):
        return await post_resp(self.request)
```

Handlers should be coroutines accepting `self` only and returning response object as regular `web-handler`. Request object can be retrieved by `View.request` property.

After implementing the view (`MyView` from example above) should be registered in application’s router:

```python
app.add_routes([web.view('/path/to', MyView)])
```

or:

```python
@routes.view('/path/to')
class MyView(web.View):
    ...
```

or:

```python
app.router.add_route('*', '/path/to', MyView)
```

Example will process GET and POST requests for `/path/to` but raise `405 Method not allowed` exception for unimplemented HTTP methods.

Resource Views

`All` registered resources in a router can be viewed using the `UrlDispatcher.resources()` method:

```python
for resource in app.router.resources():
    print(resource)
```

A `subset` of the resources that were registered with a `name` can be viewed using the `UrlDispatcher.named_resources()` method:

```python
for name, resource in app.router.named_resources().items():
    print(name, resource)
```
Alternative ways for registering routes

Code examples shown above use imperative style for adding new routes: they call `app.router.add_get(...)` etc.

There are two alternatives: route tables and route decorators.

Route tables look like Django way:

```python
async def handle_get(request):
    ...

async def handle_post(request):
    ...

app.router.add_routes([web.get('/get', handle_get),
                       web.post('/post', handle_post),
                      ])  # The snippet calls add_routes() to register a list of route definitions (aiohttp.web.RouteDef instances) created by aiohttp.web.get() or aiohttp.web.post() functions.
```

See also:

RouteDef and StaticDef reference.

Route decorators are closer to Flask approach:

```python
routes = web.RouteTableDef()

@routes.get('/get')
async def handle_get(request):
    ...

@routes.post('/post')
async def handle_post(request):
    ...

app.router.add_routes(routes)
```

It is also possible to use decorators with class-based views:

```python
routes = web.RouteTableDef()

@routes.view("/view")
class MyView(web.View):
    async def get(self):
        ...

    async def post(self):
        ...

app.router.add_routes(routes)
```

The example creates a aiohttp.web.RouteTableDef container first.

The container is a list-like object with additional decorators `aiohttp.web.RouteTableDef.get()`, `aiohttp.web.RouteTableDef.post()` etc. for registering new routes.

After filling the container `add_routes()` is used for adding registered route definitions into application’s router.
See also:

`RouteTableDef` reference.

All three ways (imperative calls, route tables and decorators) are equivalent, you could use what do you prefer or even mix them on your own.

New in version 2.3.

**JSON Response**

It is a common case to return JSON data in response, `aiohttp.web` provides a shortcut for returning JSON – `aiohttp.web.json_response()`:

```python
async def handler(request):
    data = {'some': 'data'}
    return web.json_response(data)
```

The shortcut method returns `aiohttp.web.Response` instance so you can for example set cookies before returning it from handler.

**User Sessions**

Often you need a container for storing user data across requests. The concept is usually called a session.

`aiohttp.web` has no built-in concept of a session, however, there is a third-party library, `aiohttp_session`, that adds session support:

```python
import asyncio
import time
import base64
from cryptography import fernet
from aiohttp import web
from aiohttp_session import setup, get_session, session_middleware
from aiohttp_session.cookie_storage import EncryptedCookieStorage

async def handler(request):
    session = await get_session(request)
    last_visit = session['last_visit'] if 'last_visit' in session else None
    text = 'Last visited: {}
          '.format(last_visit)
    return web.Response(text=text)

async def make_app():
    app = web.Application()
    # secret_key must be 32 url-safe base64-encoded bytes
    fernet_key = fernet.Fernet.generate_key()
    secret_key = base64.urlsafe_b64decode(fernet_key)
    setup(app, EncryptedCookieStorage(secret_key))
    app.add_routes([web.get('/', handler)])
    return app

web.run_app(make_app())
```
HTTP Forms

HTTP Forms are supported out of the box.

If form's method is "GET" (<form method="get">) use Request.query for getting form data.

To access form data with "POST" method use Request.post() or Request.multipart().

Request.post() accepts both 'application/x-www-form-urlencoded' and 'multipart/form-data' form's data encoding (e.g. <form enctype="multipart/form-data">). It stores files data in temporary directory. If client_max_size is specified post raises ValueError exception. For efficiency use Request.multipart(), It is especially effective for uploading large files (File Uploads).

Values submitted by the following form:

```html
<form action="/login" method="post" accept-charset="utf-8"
    enctype="application/x-www-form-urlencoded">
    <label for="login">Login</label>
    <input id="login" name="login" type="text" value="" autofocus/>
    <label for="password">Password</label>
    <input id="password" name="password" type="password" value=""/>
    <input type="submit" value="login"/>
</form>
```

could be accessed as:

```python
async def do_login(request):
    data = await request.post()
    login = data['login']
    password = data['password']
```

File Uploads

aiohttp.web has built-in support for handling files uploaded from the browser.

First, make sure that the HTML <form> element has its enctype attribute set to enctype="multipart/form-data". As an example, here is a form that accepts an MP3 file:

```html
<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>
```

Then, in the request handler you can access the file input field as a FileField instance. FileField is simply a container for the file as well as some of its metadata:

```python
async def store_mp3_handler(request):
    # WARNING: don't do that if you plan to receive large files!
    data = await request.post()
```

(continues on next page)
mp3 = data['mp3']

#.filename contains the name of the file in string format.
filename = mp3.filename

#.file contains the actual file data that needs to be stored somewhere.
mp3_file = data['mp3'].file

content = mp3_file.read()

return web.Response(body=content, headers=MultiDict(
    {'CONTENT-DISPOSITION': mp3_file}))
WebSockets

*aiohttp.web* supports *WebSockets* out-of-the-box.

To setup a *WebSocket*, create a *WebSocketResponse* in a *request handler* and then use it to communicate with the peer:

```python
async def websocket_handler(request):
    ws = web.WebSocketResponse()
    await ws.prepare(request)

    async for msg in ws:
        # ws.__next__() automatically terminates the loop
        # after ws.close() or ws.exception() is called
        if msg.type == aiohttp.WSMsgType.TEXT:
            if msg.data == 'close':
                await ws.close()
            else:
                await ws.send_str(msg.data + '/answer')
        elif msg.type == aiohttp.WSMsgType.ERROR:
            print('ws connection closed with exception %s' % ws.exception())

    print('websocket connection closed')

    return ws
```

The handler should be registered as HTTP GET processor:

```python
app.add_routes([web.get('/ws', websocket_handler)])
```

Redirects

To redirect user to another endpoint - raise *HTTPFound* with an absolute URL, relative URL or view name (the argument from router):

```python
raise web.HTTPFound('/redirect')
```

The following example shows redirect to view named ‘login’ in routes:

```python
async def handler(request):
    location = request.app.router['login'].url_for()
    raise web.HTTPFound(location=location)

router.add_get('/handler', handler)
router.add_get('/login', login_handler, name='login')
```

Example with login validation:

```python
@aiohttp_jinja2.template('login.html')
async def login(request):
    if request.method == 'POST':
        form = await request.post()
        error = validate_login(form)
```

(continues on next page)
if error:
    return {'error': error}
else:
    # login form is valid
    location = request.app.router['index'].url_for()
    raise web.HTTPFound(location=location)

return {}

app.router.add_get('/', index, name='index')
app.router.add_get('/login', login, name='login')
app.router.add_post('/login', login, name='login')

12.2.2 Web Server Advanced

Unicode support

aiohttp does requoting of incoming request path.

Unicode (non-ASCII) symbols are processed transparently on both route adding and resolving (internally everything is converted to percent-encoding form by yarl library).

But in case of custom regular expressions for Variable Resources please take care that URL is percent encoded: if you pass Unicode patterns they don’t match to requoted path.

Peer disconnection

When a client peer is gone a subsequent reading or writing raises OSError or more specific exception like ConnectionResetError.

The reason for disconnection is vary; it can be a network issue or explicit socket closing on the peer side without reading the whole server response.

aiohttp handles disconnection properly but you can handle it explicitly, e.g.:

async def handler(request):
    try:
        text = await request.text()
    except OSError:
        # disconnected

Passing a coroutine into run_app and Gunicorn

run_app() accepts either application instance or a coroutine for making an application. The coroutine based approach allows to perform async IO before making an app:

async def app_factory():
    await pre_init()
    app = web.Application()
    app.router.add_get(...)
    return app

web.run_app(app_factory())
Gunicorn worker supports a factory as well. For Gunicorn the factory should accept zero parameters:

```python
async def my_web_app():
    app = web.Application()
    app.router.add_get(...)
    return app
```

Start gunicorn:

```
$ gunicorn my_app_module:my_web_app --bind localhost:8080 --worker-class aiohttp.GunicornWebWorker
```

New in version 3.1.

**Custom Routing Criteria**

Sometimes you need to register *handlers* on more complex criteria than simply a *HTTP method* and *path* pair. Although *UrlDispatcher* does not support any extra criteria, routing based on custom conditions can be accomplished by implementing a second layer of routing in your application.

The following example shows custom routing based on the *HTTP Accept* header:

```python
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.add_routes([web.get('/', chooser.do_route)])
chooser.reg_acceptor('application/json', handle_json)
chooser.reg_acceptor('application/xml', handle_xml)
```
Static file handling

The best way to handle static files (images, JavaScripts, CSS files etc.) is using Reverse Proxy like nginx or CDN services.

But for development it’s very convenient to handle static files by aiohttp server itself.

To do it just register a new static route by `RouteTableDef.static()` or `static()` calls:

```python
app.add_routes([web.static('/prefix', path_to_static_folder)])
routes.static('/prefix', path_to_static_folder)
```

When a directory is accessed within a static route then the server responses to client with HTTP/403 Forbidden by default. Displaying folder index instead could be enabled with `show_index` parameter set to True:

```python
web.static('/prefix', path_to_static_folder, show_index=True)
```

When a symlink from the static directory is accessed, the server responds to client with HTTP/404 Not Found by default. To allow the server to follow symlinks, parameter `follow_symlinks` should be set to True:

```python
web.static('/prefix', path_to_static_folder, follow_symlinks=True)
```

When you want to enable cache busting, parameter `append_version` can be set to True

Cache busting is the process of appending some form of file version hash to the filename of resources like JavaScript and CSS files. The performance advantage of doing this is that we can tell the browser to cache these files indefinitely without worrying about the client not getting the latest version when the file changes:

```python
web.static('/prefix', path_to_static_folder, append_version=True)
```

Template Rendering

`aiohttp.web` does not support template rendering out-of-the-box.

However, there is a third-party library, `aiohttp_jinja2`, which is supported by the `aiohttp` authors.

Using it is rather simple. First, setup a jinja2 environment with a call to `aiohttp_jinja2.setup()`:

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

After that you may use the template engine in your `handlers`. The most convenient way is to simply wrap your handlers with the `aiohttp_jinja2.template()` decorator:

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

If you prefer the Mako template engine, please take a look at the `aiohttp_mako` library.

**Warning:** `aiohttp_jinja2.template()` should be applied before `RouteTableDef.get()` decorator and family, e.g. it must be the first (most down decorator in the chain):
```python
@routes.get('/path')
@aiohttp_jinja2.template('tmpl.jinja2')
async def handler(request):
    return {'name': 'Andrew', 'surname': 'Svetlov'}
```

**Reading from the same task in WebSockets**

Reading from the `WebSocket` `(await ws.receive())` **must only** be done inside the request handler task; however, writing `(ws.send_str(...))` to the `WebSocket`, closing `(await ws.close())` and canceling the handler task may be delegated to other tasks. See also [FAQ section](#).

`aiohttp.web` creates an implicit `asyncio.Task` for handling every incoming request.

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

---

**Warning:** Parallel reads from websocket are forbidden, there is no possibility to call `WebSocketResponse.receive()` from two tasks.

See [FAQ section](#) for instructions how to solve the problem.

**Data Sharing aka No Singletons Please**

`aiohttp.web` discourages the use of `global variables`, aka `singletons`. Every variable should have its own context that is **not global**.

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

**Application's config**

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

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

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

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

In case of `nested applications` the desired lookup strategy could be the following:

1. Search the key in the current nested application.
2. If the key is not found continue searching in the parent application(s).

For this please use `Request.config_dict` read-only property:

```python
async def handler(request):
    data = request.config_dict['my_private_key']
```
Request's storage

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

```python
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.

Response's storage

`StreamResponse` and `Response` objects also support `collections.abc.MutableMapping` interface. This is useful when you want to share data with signals and middlewares once all the work in the handler is done:

```python
async def handler(request):
    [ do all the work ]
    response['my_metric'] = 123
    return response
```

Naming hint

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`).

ContextVars support

Starting from Python 3.7 asyncio has Context Variables as a context-local storage (a generalization of thread-local concept that works with asyncio tasks also).

aiohttp server supports it in the following way:

- A server inherits the current task's context used when creating it. `aiohttp.web.run_app()` runs a task for handling all underlying jobs running the app, but alternatively `Application runners` can be used.

- Application initialization / finalization events (`Application.cleanup_ctx`, `Application.on_startup` and `Application.on_shutdown`, `Application.on_cleanup`) are executed inside the same context.

  E.g. all context modifications made on application startup are visible on teardown.

- On every request handling aiohttp creates a context copy. `web-handler` has all variables installed on initialization stage. But the context modification made by a handler or middleware is invisible to another HTTP request handling call.

An example of context vars usage:

```python
from contextvars import ContextVar
from aiohttp import web

VAR = ContextVar('VAR', default='default')
```

(continues on next page)
async def coro():
    return VAR.get()

async def handler(request):
    var = VAR.get()
    VAR.set('handler')
    ret = await coro()
    return web.Response(text='
'.join([var, ret]))

async def on_startup(app):
    print('on_startup', VAR.get())
    VAR.set('on_startup')

async def on_cleanup(app):
    print('on_cleanup', VAR.get())
    VAR.set('on_cleanup')

async def init():
    print('init', VAR.get())
    VAR.set('init')
    app = web.Application()
    app.router.add_get('/', handler)
    app.on_startup.append(on_startup)
    app.on_cleanup.append(on_cleanup)
    return app

web.run_app(init())
print('done', VAR.get())

New in version 3.5.

**Middlewares**

*aiohttp.web* provides a powerful mechanism for customizing request handlers via middlewares.

A middleware is a coroutine that can modify either the request or response. For example, here’s a simple middleware which appends 'wink' to the response:

```python
from aiohttp.web import middleware

async def middleware(request, handler):
    resp = await handler(request)
    resp.text = resp.text + ' wink'
    return resp
```
**Warning:** As of version 4.0.0 “new-style” middleware is default and the @middleware decorator is not required (and is deprecated), you can simply remove the decorator. “Old-style” middleware (a coroutine which returned a coroutine) is no longer supported.

**Note:** The example won’t work with streamed responses or websockets

Every middleware should accept two parameters, a request instance and a handler, and return the response or raise an exception. If the exception is not an instance of HTTPException it is converted to 500 HTTPInternalServerError after processing the middlewares chain.

**Warning:** Second argument should be named handler exactly.

When creating an Application, these middlewares are passed to the keyword-only middlewares parameter:

```python
app = web.Application(middlewares=[middleware_1, middleware_2])
```

Internally, a single request handler is constructed by applying the middleware chain to the original handler in reverse order, and is called by the RequestHandler as a regular handler.

Since middlewares are themselves coroutines, they may perform extra await calls when creating a new handler, e.g. call database etc.

Middlewares usually call the handler, but they may choose to ignore it, e.g. displaying 403 Forbidden page or raising HTTPForbidden exception if the user does not have permissions to access the underlying resource. They may also render errors raised by the handler, perform some pre- or post-processing like handling CORS and so on.

The following code demonstrates middlewares execution order:

```python
from aiohttp import web

async def test(request):
    print('Handler function called')
    return web.Response(text="Hello")

async def middleware1(request, handler):
    print('Middleware 1 called')
    response = await handler(request)
    print('Middleware 1 finished')
    return response

async def middleware2(request, handler):
    print('Middleware 2 called')
    response = await handler(request)
    print('Middleware 2 finished')
    return response

app = web.Application(middlewares=[middleware1, middleware2])
app.router.add_get('/', test)
web.run_app(app)
```

Produced output:
Example

A common use of middlewares is to implement custom error pages. The following example will render 404 errors using a JSON response, as might be appropriate a JSON REST service:

```python
from aiohttp import web

async def error_middleware(request, handler):
    try:
        response = await handler(request)
        if response.status != 404:
            return response
        message = response.message
    except web.HTTPException as ex:
        if ex.status != 404:
            raise
        message = ex.reason
    return web.json_response({'error': message})

app = web.Application(middlewares=[error_middleware])
```

Middleware Factory

A middleware factory is a function that creates a middleware with passed arguments. For example, here’s a trivial middleware factory:

```python
def middleware_factory(text):
    async def sample_middleware(request, handler):
        resp = await handler(request)
        resp.text = resp.text + text
    return resp
    return sample_middleware
```

Note that in contrast to regular middlewares, a middleware factory should return the function, not the value. So when passing a middleware factory to the app you actually need to call it:

```python
app = web.Application(middlewares=[middleware_factory(' wink')])
```
Signals

Although *middlewares* can customize *request handlers* before or after a *Response* has been prepared, they can’t customize a *Response* *while* it’s being prepared. For this *aiohttp.web* provides *signals*.

For example, a middleware can only change HTTP headers for *unprepared* responses (see *StreamResponse.prepare()*) but sometimes we need a hook for changing HTTP headers for streamed responses and WebSockets. This can be accomplished by subscribing to the *Application.on_response_prepare* signal:

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

Additionally, the *Application.on_startup* and *Application.on_cleanup* signals can be subscribed to for application component setup and tear down accordingly.

The following example will properly initialize and dispose an aiopg connection engine:

```python
from aiopg.sa import create_engine

async def create_aiopg(app):
    app['pg_engine'] = await create_engine(
        user='postgre',
        database='postgre',
        host='localhost',
        port=5432,
        password=''
    )

async def dispose_aiopg(app):
    app['pg_engine'].close()
    await app['pg_engine'].wait_closed()

app.on_startup.append(create_aiopg)
app.on_cleanup.append(dispose_aiopg)
```

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

Signal handlers will be run sequentially, in order they were added. All handlers must be asynchronous since *aiohttp* 3.0.

Cleanup Context

Bare *Application.on_startup*/ *Application.on_cleanup* pair still has a pitfall: signals handlers are independent on each other.

E.g. we have *[create_pg, create_redis]* in *startup* signal and *[dispose_pg, dispose_redis]* in *cleanup*.

If, for example, *create_pg(app)* call fails *create_redis(app)* is not called. But on application cleanup both *dispose_pg(app)* and *dispose_redis(app)* are still called: *cleanup signal* has no knowledge about startup/cleanup pairs and their execution state.

The solution is *Application.cleanup_ctx* usage:

```python
async def pg_engine(app):
    app['pg_engine'] = await create_engine(
```

(continues on next page)
User='postgres',
    database='postgres',
    host='localhost',
    port=5432,
    password=''
}

yield
app['pg_engine'].close()
await app['pg_engine'].wait_closed()

app.cleanup_ctx.append(pg_engine)

The attribute is a list of asynchronous generators, a code before yield is an initialization stage (called on startup), a code after yield is executed on cleanup. The generator must have only one yield.

aiohttp guarantees that cleanup code is called if and only if startup code was successfully finished.

New in version 3.1.

**Nested applications**

Sub applications are designed for solving the problem of the big monolithic code base. Let’s assume we have a project with own business logic and tools like administration panel and debug toolbar.

Administration panel is a separate application by its own nature but all toolbar URLs are served by prefix like /admin. Thus we’ll create a totally separate application named admin and connect it to main app with prefix by `Application.add_subapp()`:

```python
admin = web.Application()
# setup admin routes, signals and middlewares
app.add_subapp('/admin/', admin)
```

Middlewares and signals from app and admin are chained.

It means that if URL is '/admin/something' middlewares from app are applied first and admin. middlewares are the next in the call chain.

The same is going for `Application.on_response_prepare` signal – the signal is delivered to both top level app and admin if processing URL is routed to admin sub-application.

Common signals like `Application.on_startup`, `Application.on_shutdown` and `Application.on_cleanup` are delivered to all registered sub-applications. The passed parameter is sub-application instance, not top-level application.

Third level sub-applications can be nested into second level ones – there are no limitation for nesting level.

Url reversing for sub-applications should generate urls with proper prefix.

But for getting URL sub-application’s router should be used:

```python
admin = web.Application()
admin.add_routes([web.get('/resource', handler, name='name')])
app.add_subapp('/admin/', admin)
url = admin.router['name'].url_for()
```
The generated url from example will have a value URL('/admin/resource').
If main application should do URL reversing for sub-application it could use the following explicit technique:

```python
admin = web.Application()
admin.add_routes([web.get('/resource', handler, name='name')])
app.add_subapp('/admin/', admin)
app['admin'] = admin
async def handler(request):
    # main application's handler
    admin = request.app['admin']
    url = admin.router['name'].url_for()
```

**Expect Header**

*aiohttp.web* supports *Expect* header. By default it sends HTTP/1.1 100 Continue line to client, or raises *HTTPExpectationFailed* if header value is not equal to “100-continue”. It is possible to specify custom *Expect* header handler on per route basis. This handler gets called if *Expect* header exist in request after receiving all headers and before processing application's 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. Also handler can raise a subclass of *HTTPException*. In this case all further processing will not happen and client will receive appropriate http response.

**Note:** A server that does not understand or is unable to comply with any of the expectation values in the Expect field of a request MUST respond with appropriate error status. The server MUST respond with a 417 (Expectation Failed) status if any of the expectations cannot be met or, if there are other problems with the request, some other 4xx status.

http://www.w3.org/Protocols/rfc2616/rfc2616-sec14.html#sec14.20

If all checks pass, the custom handler *must* write a HTTP/1.1 100 Continue status code before returning.

The following example shows how to setup a custom handler for the *Expect* header:

```python
async def check_auth(request):
    if request.version != aiohttp.HttpVersion11:
        return
    if request.headers.get('EXPECT') != '100-continue':
        raise HTTPExpectationFailed(text="Unknown Expect: %s" % expect)
    if request.headers.get('AUTHORIZATION') is None:
        raise 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.add_routes([web.add_get('/', hello, expect_handler=check_auth)])
```
Custom resource implementation


Application runners

`run_app()` provides a simple blocking API for running an `Application`.

For starting the application asynchronously or serving on multiple HOST/PORT `AppRunner` exists.

The simple startup code for serving HTTP site on 'localhost', port 8080 looks like:

```python
def main(app):
    runner = web.AppRunner(app)
    await runner.setup()
    site = web.TCPSite(runner, 'localhost', 8080)
    await site.start()
    while True:
        await asyncio.sleep(3600)  # sleep forever

    await runner.cleanup()
```

To stop serving call `AppRunner.cleanup()`:

```python
await runner.cleanup()
```

New in version 3.0.

Graceful shutdown

Stopping `aiohttp web server` by just closing all connections is not always satisfactory.

The problem is: if application supports websockets or data streaming it most likely has open connections at server shutdown time.

The library has no knowledge how to close them gracefully but developer can help by registering `Application.on_shutdown` signal handler and call the signal on `web server` closing.

Developer should keep a list of opened connections (`Application` is a good candidate).

The following `websocket` snippet shows an example for websocket handler:

```python
from aiohttp import web
import weakref

app = web.Application()
app['websockets'] = weakref.WeakSet()

async def websocket_handler(request):
    ws = web.WebSocketResponse()
    await ws.prepare(request)
    request.app['websockets'].add(ws)
    try:
        async for msg in ws:
            ...
    finally:
        request.app['websockets'].discard(ws)
```

(continues on next page)
return ws

Signal handler may look like:

```python
from aiohttp import WSCloseCode

async def on_shutdown(app):
    for ws in set(app['websockets']):
        await ws.close(code=WSCloseCode.GOING_AWAY,
                       message='Server shutdown')

app.on_shutdown.append(on_shutdown)
```

Both `run_app()` and `AppRunner.cleanup()` call shutdown signal handlers.

### Background tasks

Sometimes there’s a need to perform some asynchronous operations just after application start-up.

Even more, in some sophisticated systems there could be a need to run some background tasks in the event loop along with the application’s request handler. Such as listening to message queue or other network message/event sources (e.g. ZeroMQ, Redis Pub/Sub, AMQP, etc.) to react to received messages within the application.

For example the background task could listen to ZeroMQ on `zmq.SUB` socket, process and forward retrieved messages to clients connected via WebSocket that are stored somewhere in the application (e.g. in the `application['websockets']` list).

To run such short and long running background tasks aiohttp provides an ability to register `Application.on_startup` signal handler(s) that will run along with the application’s request handler.

For example there’s a need to run one quick task and two long running tasks that will live till the application is alive. The appropriate background tasks could be registered as an `Application.on_startup` signal handlers as shown in the example below:

```python
async def listen_to_redis(app):
    try:
        sub = await aioredis.create_redis(('localhost', 6379))
        ch, * = await sub.subscribe('news')
        # Forward message to all connected websockets:
        async for msg in ch.iter(encoding='utf-8'):
            # Forward message to all connected websockets:
            for ws in app['websockets']:
                ws.send_str('{}: {}'.format(ch.name, msg))
    except asyncio.CancelledError:
        pass
    finally:
        await sub.unsubscribe(ch.name)
        await sub.quit()

async def start_background_tasks(app):
    app['redis_listener'] = asyncio.create_task(listen_to_redis(app))

async def cleanup_background_tasks(app):
    app['redis_listener'].cancel()
```

(continues on next page)
The task `listen_to_redis()` will run forever. To shut it down correctly, `Application.on_cleanup` signal handler may be used to send a cancellation to it.

**Handling error pages**

Pages like 404 *Not Found* and 500 *Internal Error* could be handled by custom middleware, see polls demo for example.

**Deploying behind a Proxy**

As discussed in *Server Deployment* the preferable way is deploying *aiohttp* web server behind a *Reverse Proxy Server* like *nginx* for production usage.

In this way properties like `BaseRequest.scheme`, `BaseRequest.host` and `BaseRequest.remote` are incorrect.

Real values should be given from proxy server, usually either `Forwarded` or old-fashion `X-Forwarded-For`, `X-Forwarded-Host`, `X-Forwarded-Proto` HTTP headers are used.

*aiohttp* does not take `forwarded` headers into account by default because it produces *security issue*: HTTP client might add these headers too, pushing non-trusted data values.

That’s why *aiohttp* server should setup `forwarded` headers in custom middleware in tight conjunction with *reverse proxy configuration*.

For changing `BaseRequest.scheme`, `BaseRequest.host` and `BaseRequest.remote` the middleware might use `BaseRequest.clone()`.

See also:

https://github.com/aio-libs/aiohttp-remotes provides secure helpers for modifying `scheme`, `host` and `remote` attributes according to `Forwarded` and `X-Forwarded-*` HTTP headers.

**Swagger support**

*aiohttp-swagger* is a library that allow to add Swagger documentation and embed the Swagger-UI into your *aiohttp.web* project.
CORS support

*aiohttp.web* itself does not support Cross-Origin Resource Sharing, but there is an aiohttp plugin for it: *aiohttp_cors*.

Debug Toolbar

*aiohttp-debugtoolbar* is a very useful library that provides a debugging toolbar while you’re developing an *aiohttp.web* application.

Install it with pip:

```bash
$ pip install aiohttp_debugtoolbar
```

Just call *aiohttp_debugtoolbar.setup()*

```python
import aiohttp_debugtoolbar
from aiohttp_debugtoolbar import toolbar_middleware_factory

app = web.Application()
aiohttp_debugtoolbar.setup(app)
```

The toolbar is ready to use. Enjoy!!!

Dev Tools

*aiohttp-devtools* provides a couple of tools to simplify development of *aiohttp.web* applications.

Install with pip:

```bash
$ pip install aiohttp-devtools
```

* ``runserver`` provides a development server with auto-reload, live-reload, static file serving and aiohttp_debugtoolbar integration.
* ``start`` is a `cookiecutter command which does the donkey work of creating new :mod:`aiohttp.web` Applications.

Documentation and a complete tutorial of creating and running an app locally are available at aiohttp-devtools.

**12.2.3 Low Level Server**

This topic describes *aiohttp.web* based low level API.

Abstract

Sometimes user don’t need high-level concepts introduced in *Server*: applications, routers, middlewares and signals. All what is needed is supporting asynchronous callable which accepts a request and returns a response object.

This is done by introducing *aiohttp.web.Server* class which serves a protocol factory role for asyncio. AbstractEventLoop.create_server() and bridges data stream to web handler and sends result back.

Low level *web handler* should accept the single *BaseRequest* parameter and performs one of the following actions:

1. Return a *Response* with the whole HTTP body stored in memory.
2. Create a `StreamResponse`, send headers by `StreamResponse.prepare()` call, send data chunks by `StreamResponse.write()` and return finished response.

3. Raise `HTTPException` derived exception (see `Web Server Exceptions` section).
   
   All other exceptions not derived from `HTTPException` leads to 500 `Internal Server Error` response.

4. Initiate and process Web-Socket connection by `WebSocketResponse` using (see `WebSockets`).

**Run a Basic Low-Level Server**

The following code demonstrates very trivial usage example:

```python
import asyncio
from aiohttp import web

async def handler(request):
    return web.Response(text="OK")

async def main():
    server = web.Server(handler)
    runner = web.ServerRunner(server)
    await runner.setup()
    site = web.TCPSite(runner, 'localhost', 8080)
    await site.start()

    print("===== Serving on http://127.0.0.1:8080/ =====")

    # pause here for very long time by serving HTTP requests and
    # waiting for keyboard interruption
    await asyncio.sleep(100*3600)

loop = asyncio.get_event_loop()

try:
    loop.run_until_complete(main())
except KeyboardInterrupt:
    pass
loop.close()
```

In the snippet we have `handler` which returns a regular `Response` with "OK" in BODY.

This `handler` is processed by `server (Server which acts as protocol factory)`. Network communication is created by `runners API` to serve `http://127.0.0.1:8080/`.

The handler should process every request for every `path`, e.g. `GET, POST, Web-Socket`.

The example is very basic: it always return 200 OK response, real life code is much more complex usually.
12.2.4 Server Reference

Request and Base Request

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

*BaseRequest* is used for *Low-Level Servers* (which have no applications, routers, signals and middlewares). *Request* has an *Request.app* and *Request.match_info* attributes.

A *BaseRequest* / *Request* are *dict* like objects, allowing them to be used for *sharing data* among *Middlewares* and *Signals* handlers.

class aiohttp.web.BaseRequest

  version
    *HTTP version* of request, Read-only property.

  method
    *HTTP method*, read-only property.
    The value is upper-cased *str* like "GET", "POST", "PUT" etc.

  url
    A *URL* instance with absolute URL to resource (*scheme*, *host* and *port* are included).

    **Note:** In case of malformed request (e.g. without "HOST" HTTP header) the absolute url may be unavailable.

  rel_url
    A *URL* instance with relative URL to resource (contains *path*, *query* and *fragment* parts only, *scheme*, *host* and *port* are excluded).
    The property is equal to *url.relative()* but is always present.

    **See also:**
    A note from *url*.

  scheme
    A string representing the scheme of the request.
    The scheme is 'https' if transport for request handling is *SSL*, 'http' otherwise.
    The value could be overridden by *clone()*.
    Read-only *str* property.
    Changed in version 2.3: *Forwarded* and *X-Forwarded-Proto* are not used anymore.
    Call *.clone(scheme=new_scheme)* for setting up the value explicitly.

    **See also:**
    *Deploying behind a Proxy*

  secure
    Shorthand for *request.url.scheme == 'https'*
    Read-only *bool* property.

    **See also:**
scheme

forwarded

A tuple containing all parsed Forwarded header(s).

Makes an effort to parse Forwarded headers as specified by RFC 7239:

• It adds one (immutable) dictionary per Forwarded field-value, i.e., per proxy. The element
  corresponds to the data in the Forwarded field-value added by the first proxy encountered by
  the client. Each subsequent item corresponds to those added by later proxies.
• It checks that every value has valid syntax in general as specified in RFC 7239#section-4: either a
token or a quoted-string.
• It un-escapes quoted-pairs.
• It does NOT validate ‘by’ and ‘for’ contents as specified in RFC 7239#section-6.
• It does NOT validate host contents (Host ABNF).
• It does NOT validate proto contents for valid URI scheme names.

Returns a tuple containing one or more MappingProxy objects

See also:

scheme

See also:

host

host

Host name of the request, resolved in this order:

• Overridden value by clone() call.
• Host HTTP header
• socket.gtfqdn()

Read-only str property.

Changed in version 2.3: Forwarded and X-Forwarded-Host are not used anymore.

Call .clone(host=new_host) for setting up the value explicitly.

See also:

Deploying behind a Proxy

remote

Originating IP address of a client initiated HTTP request.

The IP is resolved through the following headers, in this order:

• Overridden value by clone() call.
• Peer name of opened socket.

Read-only str property.

Call .clone(remote=new_remote) for setting up the value explicitly.

New in version 2.3.

See also:

Deploying behind a Proxy

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-
decoded. 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 URL-encoded and may contain invalid URL characters, e.g. `/my%2Fpath%7Cwith%21some%25strange%24characters`.

For URL-decoded version please take a look on `path`.

Read-only `str` property.

**query**

A multidict with all the variables in the query string.

Read-only `MultiDictProxy` lazy property.

**query_string**

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

Read-only `str` property.

**headers**

A case-insensitive multidict proxy with all headers.

Read-only `CIMultiDictProxy` property.

**raw_headers**

HTTP headers of response as unconverted bytes, a sequence of (key, value) pairs.

**keep_alive**

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

Read-only `bool` 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:

```python
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 `StreamReader` instance, input stream for reading request’s `BODY`.

Read-only property.

**body_exists**

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

Read-only `bool` property.

New in version 2.3.

**can_read_body**

Return `True` if request’s `HTTP BODY` can be read, `False` otherwise.

Read-only `bool` property.
New in version 2.3.

**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.

**http_range**
Read-only property that returns information about `Range` HTTP header.
Returns a `slice` where `.start` is left inclusive bound, `.stop` is right exclusive bound and `.step` is 1.

The property might be used in two manners:
1. Attribute-access style (example assumes that both left and right borders are set, the real logic for case of open bounds is more complex):

   ```python
   rng = request.http_range
   with open(filename, 'rb') as f:
       f.seek(rng.start)
       return f.read(rng.stop - rng.start)
   ```

2. Slice-style:

   ```python
   return buffer[request.http_range]
   ```

**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.

**if_unmodified_since**
Read-only property that returns the date specified in the `If-Unmodified-Since` header.
Returns `datetime.datetime` or `None` if `If-Unmodified-Since` header is absent or is not a valid HTTP date.

New in version 3.1.

**if_range**
Read-only property that returns the date specified in the `If-Range` header.
Returns `datetime.datetime` or `None` if `If-Range` header is absent or is not a valid HTTP date.

New in version 3.1.
clone(*, method=..., rel_url=..., headers=...)  
Clone itself with replacement some attributes.

Creates and returns a new instance of Request object. If no parameters are given, an exact copy is returned. If a parameter is not passed, it will reuse the one from the current request object.

Parameters

- **method**(str) – http method
- **rel_url** – url to use, str or URL
- **headers** – CIMultiDict or compatible headers container.

Returns a cloned Request instance.

get_extra_info(name, default=None)  
Reads extra information from the protocol’s transport. If no value associated with name is found, default is returned.

Parameters

- **name**(str) – The key to look up in the transport extra information.
- **default** – Default value to be used when no value for name is found (default is None).

New in version 3.7.

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(*, loads=json.loads, content_type='application/json')  
Read request body decoded as json. If request’s content-type does not match content_type parameter, web.HTTPBadRequest get raised. To disable content type check pass None value.

Parameters

- **loads**(callable) – any callable that accepts str and returns dict with parsed JSON (json.loads() by default).
- **content_type**(str) – expected value of Content-Type header or None (‘application/json’ by default)

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

coroutine multipart()  
Returns aiohttp.multipart.MultipartReader which processes incoming multipart request.

The method is just a boilerplate coroutine implemented as:
async def multipart(self, *, reader=aiohttp.multipart.MultipartReader):
    return reader(self.headers, self._payload)

This method is a coroutine for consistency with the else reader methods.

**Warning:** The method does not store read data internally. That means once you exhausts multipart reader, you cannot get the request payload one more time.

**See also:**

*Working with Multipart*

Changed in version 3.4: Dropped `reader` parameter.

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

Returns `MultiDictProxy` instance filled with parsed data.

If `method` is not `POST`, `PUT`, `PATCH`, `TRACE` or `DELETE` 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.

coroutine wait_for_disconnection()  
Returns when the connection that sent this request closes

If there is no client disconnection during request handling, this coroutine gets cancelled automatically at the end of this request being handled.

This can be used in handlers as a means of receiving a notification of premature client disconnection.

New in version 4.0.

class aiohttp.web.Request  
A request used for receiving request’s information by `web handler`.

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

The class in derived from `BaseRequest`, shares all parent’s attributes and methods but has a couple of additional properties:

**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.

**config_dict**

A `aiohttp.ChainMapProxy` instance for mapping all properties from the current application returned by `app` property and all its parents.

**See also:**

*Application’s config*

New in version 3.2.

**Note:** You should never create the `Request` instance manually — `aiohttp.web` does it for you. But `clone()` may be used for cloning modified request copy with changed `path`, `method` etc.

### Response classes

For now, `aiohttp.web` has three classes for the HTTP response: `StreamResponse`, `Response` and `FileResponse`.

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:

```python
async def handler(request):
    return Response(text="All right!")
```

Response classes are `dict` like objects, allowing them to be used for sharing data among *Middlewares* and *Signals* handlers:

```python
resp['key'] = value
```

New in version 3.0: Dict-like interface support.

### StreamResponse

```python
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.

**task**

A task that serves HTTP request handling.
May be useful for graceful shutdown of long-running requests (streaming, long polling or web-socket).

**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.

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**.

See also:

**compression**

**chunked**

Read-only property, indicates if chunked encoding is on.
Can be enabled by **enable_chunked_encoding()** call.

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.
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, samesite=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 (unsupported) 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)
- **samesite** *(str)* – Asserts that a cookie must not be sent with cross-origin requests, providing some protection against cross-site request forgery attacks. Generally the value should be one of: `None`, Lax or Strict. (optional)

New in version 3.7.
Warning: In HTTP version 1.1, expires was deprecated and replaced with the easier-to-use max-age, but Internet Explorer (IE6, IE7, and IE8) does not support max-age.

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

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.

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 on_response_prepare signal handlers.

coroutine write(data)
Send byte-ish data as the part of response BODY:

await resp.write(data)

prepare() must be invoked before the call.

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 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 (*, body=None, status=200, reason=None, text=None, headers=None, content_type=None, charset=None, zlib_executor_size=sentinel, zlib_executor=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.
- **zlib_executor_size** (int) – length in bytes which will trigger zlib compression of body to happen in an executor

New in version 3.5.

- **zlib_executor** (int) – executor to use for zlib compression

New in version 3.5.

**body**

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

Setting body also recalculates content_length value.

Assigning str to body will make the body type of aiohttp.payload.StringPayload, which tries to encode the given data based on Content-Type HTTP header, while defaulting to UTF-8.

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 string, str.

Setting text also recalculates content_length value and body value

Resetting text (assigning None) sets content_length to None too, dropping Content-Length HTTP header.
### FileResponse

```python
class aiohttp.web.FileResponse(*, path, chunk_size=256*1024, status=200, reason=None, headers=None)
```

The response class used to send files, inherited from `StreamResponse`.

Supports the Content-Range and If-Range HTTP Headers in requests.

The actual body sending happens in overridden `prepare()`.

**Parameters**
- `path` – Path to file. Accepts both str and `pathlib.Path`.
- `chunk_size` *(int)* – Chunk size in bytes which will be passed into `io.RawIOBase.read()` in the event that the sendfile system call is not supported.
- `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.
- `headers` *(collections.abc.Mapping)* – HTTP headers that should be added to response’s ones. The Content-Type response header will be overridden if provided.

### WebSocketResponse

```python
class aiohttp.web.WebSocketResponse(*, timeout=10.0, receive_timeout=None, auto_close=True, autoping=True, heartbeat=None, protocols=(), compress=True, max_msg_size=4194304)
```

Class for handling server-side websockets, inherited from `StreamResponse`.

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.

To enable back-pressure from slow websocket clients treat methods `ping()`, `pong()`, `send_str()`, `send_bytes()`, `send_json()` as coroutines. By default write buffer size is set to 64k.

**Parameters**
- `autoping` *(bool)* – Automatically send PONG on PING message from client, and handle PONG responses from client. Note that server does not send PING requests, you need to do this explicitly using `ping()` method.
- `heartbeat` *(float)* – Send ping message every heartbeat seconds and wait pong response, close connection if pong response is not received. The timer is reset on any data reception.
- `receive_timeout` *(float)* – Timeout value for receive operations. Default value is `None` (no timeout for receive operation)
- `compress` *(bool)* – Enable per-message deflate extension support. False for disabled, default value is True.
- `max_msg_size` *(int)* – maximum size of read websocket message, 4 MB by default. To disable the size limit use 0.

New in version 3.3.

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 re-
    sponse answers.
    Raises HTTPException – if websocket handshake has failed.

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 re-
    sponse answers.
    Returns
        WebSocketReady instance.
        WebSocketReady.ok is True on success, WebSocketReady.protocol is
        websocket subprotocol which is passed by client and accepted by server (one of proto-
        cols sequence from WebSocketResponse ctor). WebSocketReady.protocol
        may be None if client and server subprotocols are not overlapping.

        Note: The method never raises exception.

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

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

ws_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.

coroutine ping(message=b")
    Send PING to peer.
    Parameters message – optional payload of ping message, str (converted to UTF-8 en-
    coded bytes) or bytes.
    Raises RuntimeError – if connections is not started or closing.
    Changed in version 3.0: The method is converted into coroutine

coroutine pong(message=b")
    Send unsolicited PONG to peer.
    Parameters message – optional payload of pong message, str (converted to UTF-8 en-
    coded bytes) or bytes.
    Raises RuntimeError – if connections is not started or closing.
Changed in version 3.0: The method is converted into coroutine

**coroutine send_str**(data, compress=None)

Send data to peer as TEXT message.

Parameters

- **data** *(str)* – data to send.
- **compress** *(int)* – sets specific level of compression for single message, None for not overriding per-socket setting.

Raises

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

Changed in version 3.0: The method is converted into coroutine, compress parameter added.

**coroutine send_bytes**(data, compress=None)

Send data to peer as BINARY message.

Parameters

- **data** – data to send.
- **compress** *(int)* – sets specific level of compression for single message, None for not overriding per-socket setting.

Raises

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

Changed in version 3.0: The method is converted into coroutine, compress parameter added.

**coroutine send_json**(data, compress=None, *, dumps=json.dumps)

Send data to peer as JSON string.

Parameters

- **data** – data to send.
- **compress** *(int)* – sets specific level of compression for single message, None for not overriding per-socket setting.
- **dumps** *(callable)* – any callable that accepts an object and returns a JSON string (json.dumps() by default).

Raises

- **RuntimeError** – if connection is not started or closing
- **ValueError** – if data is not serializable object
- **TypeError** – if value returned by dumps param is not str

Changed in version 3.0: The method is converted into coroutine, compress parameter added.

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

A coroutine that initiates closing handshake by sending CLOSE message.

It is safe to call close() from different task.

Parameters

- **code** *(int)* – closing code
- **message** – optional payload of close message, str (converted to UTF-8 encoded bytes) or bytes.
Raises `RuntimeError` – if connection is not started

coroutine `receive` *(timeout=None)*
A coroutine that waits upcoming data message from peer and returns it.
The coroutine implicitly handles `PING`, `PONG` and `CLOSE` without returning the message.
It process ping-pong game and performs closing handshake internally.

**Note**: Can only be called by the request handling task.

**Parameters**
- `timeout` – timeout for `receive` operation.
  timeout value overrides response’s receive_timeout attribute.

**Returns** `WSMessage`

**Raises** `RuntimeError` – if connection is not started

coroutine `receive_str` *(*, `timeout=None`)*
A coroutine that calls `receive()` but also asserts the message type is `TEXT`.

**Note**: Can only be called by the request handling task.

**Parameters**
- `timeout` – timeout for `receive` operation.
  timeout value overrides response’s receive_timeout attribute.

**Returns** `str`
peer’s message content.

**Raises** `TypeError` – if message is `BINARY`.

coroutine `receive_bytes` *(*, `timeout=None`)*
A coroutine that calls `receive()` but also asserts the message type is `BINARY`.

**Note**: Can only be called by the request handling task.

**Parameters**
- `timeout` – timeout for `receive` operation.
  timeout value overrides response’s receive_timeout attribute.

**Returns** `bytes`
peer’s message content.

**Raises** `TypeError` – if message is `TEXT`.

coroutine `receive_json` *(*, `loads=json.loads`, `timeout=None`)*
A coroutine that calls `receive_str()` and loads the JSON string to a Python dict.

**Note**: Can only be called by the request handling task.

**Parameters**
- `loads` *(callable)* – any `callable` that accepts `str` and returns `dict` with parsed JSON (`json.loads()` by default).
- `timeout` – timeout for `receive` operation.
  timeout value overrides response’s receive_timeout attribute.
Return dict  loaded JSON content

Raises

- **TypeError** – if message is BINARY.
- **ValueError** – if message is not valid JSON.

See also:

*WebSockets handling*

**WebSocketReady**

```python
class aiohttp.web.WebSocketReady
    A named tuple for returning result from WebSocketResponse.can_prepare().
    Has bool check implemented, e.g.:

    ```python
    if not await ws.can_prepare(...):
        cannot_start_websocket()
    ```

    ok
    True if websocket connection can be established, False otherwise.

    protocol
    str represented selected websocket sub-protocol.

    See also:
    WebSocketResponse.can_prepare()
```

**json_response**

```python
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).

**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 pass it to **aiohttp.web.run_app()** or **aiohttp.web.AppRunner**.

**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:

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

(continues on next page)
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()`.

```python
class aiohttp.web.Application(*, logger=<default>, middlewares=(), handler_args=None, client_max_size=1024**2, debug=...)
```

The class inherits `dict`.

**Parameters**

- `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.
- `handler_args` — dict-like object that overrides keyword arguments of `AppRunner` constructor.
- `client_max_size` — client’s maximum size in a request, in bytes. If a POST request exceeds this value, it raises an `HTTPRequestEntityTooLarge` exception.
- `debug` — Switches debug mode.
  Deprecated since version 3.5: The argument does nothing starting from 4.0, use asyncio Debug Mode instead.

**router**

Read-only property that returns `router instance`.

**logger**

`logging.Logger` instance for storing application logs.

**debug**

Boolean value indicating whether the debug mode is turned on or off.

Deprecated since version 3.5: Use asyncio Debug Mode instead.

**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:

```python
async def on_prepare(request, response):
    pass
```

**on_startup**

A `Signal` that is fired on application start-up.

Subscribers may use the signal to run background tasks in the event loop along with the application’s request handler just after the application start-up.

Signal handlers should have the following signature:

```python
async def on_startup(app):
    pass
```

See also:

* Signals and Cleanup Context.*
on_shutdown
A `Signal` that is fired on application shutdown.

Subscribers may use the signal for gracefully closing long running connections, e.g. websockets and data streaming.

Signal handlers should have the following signature:

```python
async def on_shutdown(app):
    pass
```

It’s up to end user to figure out which `web-handlers` are still alive and how to finish them properly.

We suggest keeping a list of long running handlers in `Application` dictionary.

See also:

*Graceful shutdown* and *on_cleanup*.

on_cleanup
A `Signal` that is fired on application cleanup.

Subscribers may use the signal for gracefully closing connections to database server etc.

Signal handlers should have the following signature:

```python
async def on_cleanup(app):
    pass
```

See also:

*Signals* and *on_shutdown*.

cleanup_ctx
A list of `context generators` for `startup/cleanup` handling.

Signal handlers should have the following signature:

```python
async def context(app):
    # do startup stuff
    yield
    # do cleanup
```

New in version 3.1.

See also:

*Cleanup Context*.

**add_subapp** *(prefix, subapp)*
Register nested sub-application under given path `prefix`.

In resolving process if request’s path starts with `prefix` then further resolving is passed to `subapp`.

**Parameters**

- `prefix` *(str)* – path’s prefix for the resource.
- `subapp` *(Application)* – nested application attached under `prefix`.

**Returns** *a `PrefixedSubAppResource` instance.*

**add_domain** *(domain, subapp)*
Register nested sub-application that serves the domain name or domain name mask.
In resolving process if request.headers['host'] matches the pattern \texttt{domain} then further resolving is passed to \texttt{subapp}.

\textbf{Parameters}

- \texttt{domain (str)} -- domain or mask of domain for the resource.
- \texttt{subapp (Application)} -- nested application.

\textbf{Returns} a \texttt{MatchedSubAppResource} instance.

\texttt{add_routes (routes_table)}

Register route definitions from \texttt{routes_table}.

The table is a \texttt{list} of \texttt{RouteDef} items or \texttt{RouteTableDef}.

\textbf{Returns} list of registered \texttt{AbstractRoute} instances.

The method is a shortcut for \texttt{app.router.add_routes(routes_table), see also UrlDispatcher.add_routes().}

New in version 3.1.

Changed in version 3.7: Return value updated from \texttt{None} to \texttt{list} of \texttt{AbstractRoute} instances.

\texttt{coroutine startup ()}

A coroutine that will be called along with the application’s request handler.

The purpose of the method is calling \texttt{on_startup} signal handlers.

\texttt{coroutine shutdown ()}

A coroutine that should be called on server stopping but before \texttt{cleanup()}.  

The purpose of the method is calling \texttt{on_shutdown} signal handlers.

\texttt{coroutine cleanup ()}

A coroutine that should be called on server stopping but after \texttt{shutdown()}.  

The purpose of the method is calling \texttt{on_cleanup} signal handlers.

\textbf{Note:} Application object has \texttt{router} attribute but has no \texttt{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 \texttt{AbstractRouter}: it should have only \texttt{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).

\underline{Server}

A protocol factory compatible with \texttt{create_server()}.

\texttt{class aiohttp.web.Server}

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

\texttt{connections}

List of all currently opened connections.

\texttt{requests_count}

Amount of processed requests.

\texttt{coroutine shutdown (timeout)}

A coroutine that should be called to close all opened connections.
Router

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

*Routers* is any object that implements `AbstractRouter` interface.

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

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

```python
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_resource(path, *, name=None)

    Append a resource 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 paths)

    Parameters

    • path (str) – resource path spec.
    • name (str) – optional resource name.

    Returns created resource instance (`PlainResource` or `DynamicResource`).

    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 paths)

    Pay attention please: handler must be a coroutine.

    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_routes(routes_table)

    Register route definitions from routes_table.
```
The table is a list of RouteDef items or RouteTableDef.

Returns list of registered AbstractRoute instances.
New in version 2.3.

Changed in version 3.7: Return value updated from None to list of AbstractRoute instances.

add_get (path, handler, *, name=None, allow_head=True, **kwargs)
Shortcut for adding a GET handler. Calls the add_route() with method equals to 'GET'.

If allow_head is True (default) the route for method HEAD is added with the same handler as for GET.

If name is provided the name for HEAD route is suffixed with '-head'. For example router.add_get(path, handler, name='route') call adds two routes: first for GET with name 'route' and second for HEAD with name 'route-head'.

add_post (path, handler, **kwargs)
Shortcut for adding a POST handler. Calls the add_route() with method equals to 'POST'.

add_head (path, handler, **kwargs)
Shortcut for adding a HEAD handler. Calls the add_route() with method equals to 'HEAD'.

add_put (path, handler, **kwargs)
Shortcut for adding a PUT handler. Calls the add_route() with method equals to 'PUT'.

add_patch (path, handler, **kwargs)
Shortcut for adding a PATCH handler. Calls the add_route() with method equals to 'PATCH'.

add_delete (path, handler, **kwargs)
Shortcut for adding a DELETE handler. Calls the add_route() with method equals to 'DELETE'.

add_view (path, handler, **kwargs)
Shortcut for adding a class-based view handler. Calls the add_route() with method equals to '*'.
New in version 3.0.

add_static (prefix, path, *, name=None, expect_handler=None, chunk_size=256*1024, response_factory=StreamResponse, show_index=False, follow_symlinks=False, append_version=False)
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.

If a gzip version of the static content exists at file path + .gz, it will be used for the response.

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

Parameters

- prefix (str) – URL path prefix for handled static files
- path – path to the folder in file system that contains handled static files, str or pathlib.Path.
- 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.

• **show_index** *(bool)* – flag for allowing to show indexes of a directory, by default it’s not allowed and HTTP/403 will be returned on directory access.

• **follow_symlinks** *(bool)* – flag for allowing to follow symlinks from a directory, by default it’s not allowed and HTTP/404 will be returned on access.

• **append_version** *(bool)* – flag for adding file version (hash) to the url query string, this value will be used as default when you call to `StaticRoute.url()` and `StaticRoute.url_for()` methods.

  **Returns** new `StaticRoute` instance.

**coroutine resolve** *(request)*

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

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

1. `http_exception` assigned to `HTTPException` 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.

**resources** *

The method returns a view for all registered resources.

The view is an object that allows to:

1. Get size of the router table:

   ```python
   len(app.router.resources())
   ```

2. Iterate over registered resources:

   ```python
   for resource in app.router.resources():
       print(resource)
   ```

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

   ```python
   route in app.router.resources()
   ```

**routes** *

The method returns a view for all registered routes.

**named_resources** *

Returns a dict-like `types.MappingProxyType` view over all named resources.

The view maps every named resource’s `name` to the `BaseResource` instance. It supports the usual `dict`-like operations, except for any mutable operations (i.e. it’s read-only):
```python
len(app.router.named_resources())

for name, resource in app.router.named_resources().items():
    print(name, resource)

"name" in app.router.named_resources()

app.router.named_resources()["name"]
```

**Resource**

Default router `UrlDispatcher` operates with `resources`.

Resource is an item in `routing table` which has a `path`, an optional unique `name` and at least one `route`.  

`web-handler` lookup is performed in the following way:

1. Router iterates over `resources` one-by-one.
2. If `resource` matches to requested URL the resource iterates over own `routes`.
3. If route matches to requested HTTP method (or '* wildcard) the route's handler is used as found `web-handler`. The lookup is finished.
4. Otherwise router tries next resource from the `routing table`.
5. If the end of `routing table` is reached and no `resource / route` pair found the `router` returns special `AbstractMatchInfo` instance with `AbstractMatchInfo.http_exception` is not `None` but `HTTPException` with either `HTTP 404 Not Found` or `HTTP 405 Method Not Allowed` status code. Registered `AbstractMatchInfo.handler` raises this exception on call.

User should never instantiate resource classes but give it by `UrlDispatcher.add_resource()` call.

After that he may add a `route` by calling `Resource.add_route()`.

`UrlDispatcher.add_route()` is just shortcut for:

```python
router.add_resource(path).add_route(method, handler)
```

Resource with a `name` is called `named resource`. The main purpose of `named resource` is constructing URL by route name for passing it into `template engine` for example:

```python
url = app.router["resource_name"].url_for().with_query({'a': 1, 'b': 2})
```

Resource classes hierarchy:

```yaml
AbstractResource
    Resource
        PlainResource
        DynamicResource
        StaticResource
```

class `aiohttp.web.AbstractResource`

A base class for all resources.

Inherited from `collections.abc.Sized` and `collections.abc.Iterable`.

`len(resource)` returns amount of `routes` belongs to the resource, for `route in resource` allows to iterate over these routes.
name
Read-only *name* of resource or None.

canonical
Read-only *canonical path* associate with the resource. For example `/path/to` or `/path/{to}`
New in version 3.3.

coroutine resolve(request)
Resolve resource by finding appropriate *web-handler* for (method, path) combination.

  Returns
  (match_info, allowed_methods) pair.

  allowed_methods is a set or HTTP methods accepted by resource.

  match_info is either *UrlMappingMatchInfo* if request is resolved or None if no
  route is found.

get_info()
A resource description, e.g. {'path': '/path/to'} or {'formatter': '/path/{to}',
'pattern': re.compile(r'^/path/\{P<to>[a-zA-Z]-[a-zA-Z0-9-]+\}$

url_for(*args, **kwargs)
Construct an URL for route with additional params.

  args and kwargs depend on a parameters list accepted by inherited resource class.

  Returns URL – resulting URL instance.

class aiohttp.web.Resource
A base class for new-style resources, inherits *AbstractResource*.

add_route (method, handler, *, expect_handler=None)
Add a *web-handler* to resource.

  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'.

    The method should be unique for resource.

  * handler (callable) – route handler.

  * expect_handler (coroutine) – optional expect header handler.

  Returns new *ResourceRoute* instance.

class aiohttp.web.PlainResource
A resource, inherited from *Resource*.

The class corresponds to resources with plain-text matching, '/path/to' for example.

canonical
Read-only *canonical path* associate with the resource. Returns the path used to create the PlainResource.
For example /path/to
New in version 3.3.

url_for()
Returns a URL for the resource.

class aiohttp.web.DynamicResource
A resource, inherited from *Resource*.
The class corresponds to resources with variable matching, e.g. '/path/{to}/{param}' etc.

**canonical**
Read-only canonical path associate with the resource. Returns the formatter obtained from the path used to create the DynamicResource. For example, from a path /get/{num:^\d+}, it returns /get/{num}

New in version 3.3.

**url_for(**\*\*params)\**
Returns a URL for the resource.

**Parameters**\*\*params -- a variable substitutions for dynamic resource.

E.g. for '/path/{to}/{param}' pattern the method should be called as resource.url_for(to='val1', param='val2')

---

**class aiohttp.web.StaticResource**

A resource, inherited from Resource.

The class corresponds to resources for static file serving.

**canonical**
Read-only canonical path associate with the resource. Returns the prefix used to create the StaticResource. For example /prefix

New in version 3.3.

**url_for**(filename, append_version=None)\**
Returns a URL for file path under resource prefix.

**Parameters**

- **filename** -- a file name substitution for static file handler.
  
  Accepts both str and pathlib.Path.

  E.g. an URL for '/prefix/dir/file.txt' should be generated as resource.url_for(filename='dir/file.txt')

- **append_version**(bool)=
  -- a flag for adding file version (hash) to the url query string for cache boosting

  By default has value from a constructor (False by default) When set to True -v=FILE_HASH query string param will be added When set to False has no impact

  if file not found has no impact

**class aiohttp.web.PrefixedSubAppResource**

A resource for serving nested applications. The class instance is returned by add_subapp call.

**canonical**
Read-only canonical path associate with the resource. Returns the prefix used to create the PrefixedSubAppResource. For example /prefix

New in version 3.3.

**url_for(**\*\*kwargs)\**

The call is not allowed, it raises RuntimeError.
Route

Route has HTTP method (wildcard ' * ' is an option), web-handler and optional expect handler.

Every route belong to some resource.

Route classes hierarchy:

```
AbstractRoute
  ResourceRoute
  SystemRoute
```

**ResourceRoute** is the route used for resources, **SystemRoute** serves URL resolving errors like 404 Not Found and 405 Method Not Allowed.

**class aiohttp.web.AbstractRoute**
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, always equals to name of resource which owns the route.

- **resource**
  Resource instance which holds the route, None for SystemRoute.

- **url_for(**arg, **kwargs)**
  Abstract method for constructing url handled by the route.
  Actually it’s a shortcut for route.resource.url_for(...).

- **coroutine handle_expect_header(request)**
  100-continue handler.

**class aiohttp.web.ResourceRoute**
The route class for handling different HTTP methods for Resource.

**class aiohttp.web.SystemRoute**
The route class for handling URL resolution errors like like 404 Not Found and 405 Method Not Allowed.

- **status**
  HTTP status code

- **reason**
  HTTP status reason

**RouteDef and StaticDef**

Route definition, a description for not registered yet route.

Could be used for filing route table by providing a list of route definitions (Django style).

The definition is created by functions like get() or post(), list of definitions could be added to router by UrlDispatcher.add_routes() call:
from aiohttp import web

async def handle_get(request):
    ...

async def handle_post(request):
    ...

app.router.add_routes([web.get('/get', handle_get),
                        web.post('/post', handle_post),

class aiohttp.web.AbstractRouteDef
A base class for route definitions.
Inherited from abc.ABC.
New in version 3.1.

    register(router)
    Register itself into UrlDispatcher.
    Abstract method, should be overridden by subclasses.
    Returns list of registered AbstractRoute objects.
    Changed in version 3.7: Return value updated from None to list of AbstractRoute instances.

class aiohttp.web.RouteDef
A definition of not registered yet route.
Implements AbstractRouteDef.
New in version 2.3.
Changed in version 3.1: The class implements AbstractRouteDef interface.

    method
    HTTP method (GET, POST etc.) (str).

    path
    Path to resource, e.g. /path/to. Could contain {} brackets for variable resources (str).

    handler
    An async function to handle HTTP request.

    kwargs
    A dict of additional arguments.

class aiohttp.web.StaticDef
A definition of static file resource.
Implements AbstractRouteDef.
New in version 3.1.

    prefix
    A prefix used for static file handling, e.g. /static.

    path
    File system directory to serve, str or pathlib.Path (e.g. '/home/web-service/path/to/static').
**kwargs
A dict of additional arguments, see :func:`UrlDispatcher.add_static()` for a list of supported options.

```python
aiohttp.web.get(path, handler, *, name=None, allow_head=True, expect_handler=None)
```
Return :class:`RouteDef` for processing GET requests. See :func:`UrlDispatcher.add_get()` for information about parameters.
New in version 2.3.

```python
aiohttp.web.post(path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing POST requests. See :func:`UrlDispatcher.add_post()` for information about parameters.
New in version 2.3.

```python
aiohttp.web.head(path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing HEAD requests. See :func:`UrlDispatcher.add_head()` for information about parameters.
New in version 2.3.

```python
aiohttp.web.put(path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing PUT requests. See :func:`UrlDispatcher.add_put()` for information about parameters.
New in version 2.3.

```python
aiohttp.web.patch(path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing PATCH requests. See :func:`UrlDispatcher.add_patch()` for information about parameters.
New in version 2.3.

```python
aiohttp.web.delete(path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing DELETE requests. See :func:`UrlDispatcher.add_delete()` for information about parameters.
New in version 2.3.

```python
aiohttp.web.view(path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing ANY requests. See :func:`UrlDispatcher.add_view()` for information about parameters.
New in version 3.0.

```python
aiohttp.web.static(prefix, path, *, name=None, expect_handler=None, chunk_size=256*1024, show_index=False, follow_symlinks=False, append_version=False)
```
Return :class:`StaticDef` for processing static files.
See :func:`UrlDispatcher.add_static()` for information about supported parameters.
New in version 3.1.

```python
aiohttp.web.route(method, path, handler, *, name=None, expect_handler=None)
```
Return :class:`RouteDef` for processing requests that decided by method. See :func:`UrlDispatcher.add_route()` for information about parameters.
New in version 2.3.
**RouteTableDef**

A routes table definition used for describing routes by decorators (Flask style):

```python
from aiohttp import web

routes = web.RouteTableDef()

@routes.get('/get')
async def handle_get(request):
    ...

@routes.post('/post')
async def handle_post(request):
    ...

app.router.add_routes(routes)

@routes.view('/view')
class MyView(web.View):
    async def get(self):
        ...

    async def post(self):
        ...
```

**class aiohttp.web.RouteTableDef**

A sequence of RouteDef instances (implements abc.collections.Sequence protocol).

In addition to all standard list methods the class provides also methods like `get()` and `post()` for adding new route definition.

New in version 2.3.

@get(path, *, allow_head=True, name=None, expect_handler=None)
Add a new RouteDef item for registering GET web-handler.

See UrlDispatcher.add_get() for information about parameters.

@post(path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering POST web-handler.

See UrlDispatcher.add_post() for information about parameters.

@head(path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering HEAD web-handler.

See UrlDispatcher.add_head() for information about parameters.

@put(path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering PUT web-handler.

See UrlDispatcher.add_put() for information about parameters.

@patch(path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering PATCH web-handler.

See UrlDispatcher.add_patch() for information about parameters.
@delete(path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering DELETE web-handler.
See UrlDispatcher.add_delete() for information about parameters.

@view(path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering ANY methods against a class-based view.
See UrlDispatcher.add_view() for information about parameters.
New in version 3.0.

@route(method, path, *, name=None, expect_handler=None)
Add a new RouteDef item for registering a web-handler for arbitrary HTTP method.
See UrlDispatcher.add_route() for information about parameters.
New in version 3.1.

static(prefix, path, *, name=None, expect_handler=None, chunk_size=256*1024, show_index=False, follow_symlinks=False, append_version=False)
Add a new StaticDef item for registering static files processor.
See UrlDispatcher.add_static() for information about supported parameters.
New in version 3.1.

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 filled by matching info and is resource-specific.

expect_handler
A coroutine for handling 100-continue.

handler
A coroutine for handling request.

route
Route instance for url matching.

View

class aiohttp.web.View(request)
Inherited from AbstractView.
Base class for class based views. Implementations should derive from View and override methods for handling HTTP verbs like get() or post():

class MyView(View):
    async def get(self):
        resp = await get_response(self.request)
        return resp

(continues on next page)
async def post(self):
    resp = await post_response(self.request)
    return resp

app.router.add_view('/view', MyView)

The view raises 405 Method Not allowed (HTTPMethodNotAllowed) if requested web verb is not supported.

Parameters request – instance of Request that has initiated a view processing.

request
  Request sent to view’s constructor, read-only property.

Overridable coroutine methods: connect(), delete(), get(), head(), options(), patch(), post(), put(), trace().

See also:
Class Based Views

Running Applications

To start web application there is AppRunner and site classes.

Runner is a storage for running application, sites are for running application on specific TCP or Unix socket, e.g.:

```python
runner = web.AppRunner(app)
await runner.setup()
site = web.TCPSite(runner, 'localhost', 8080)
await site.start()
# wait for finish signal
await runner.cleanup()
```

New in version 3.0: AppRunner/ServerRunner and TCPSite/UnixSite/SockSite are added in aiohttp 3.0

class aiohttp.web.BaseRunner

server
  Low-level web Server for handling HTTP requests, read-only attribute.

addresses
  A list of served sockets addresses.

See socket.getsockname() for items type.

New in version 3.3.

sites
  A read-only set of served sites (TCPSite/UnixSite/NamedPipeSite/SockSite instances).

coroutine setup()  
  Initialize the server. Should be called before adding sites.

coroutine cleanup()  
  Stop handling all registered sites and cleanup used resources.
class aiohttp.web.AppRunner(app, *, handle_signals=False, **kwargs)
A runner for Application. Used with conjunction with sites to serve on specific port.

Inherited from BaseRunner.

Parameters

• app (Application) – web application instance to serve.
• handle_signals (bool) – add signal handlers for signal.SIGINT and signal.SIGTERM (False by default).
• kwargs – named parameters to pass into web protocol.

Supported kwargs:

Parameters

• keepalive_timeout (int) – Number of seconds before closing Keep-Alive connection. Default: 75 seconds (NGINX’s default value).
• logger – Custom logger object. Default: aiohttp.log.server_logger.
• access_log – Custom logging object. Default: aiohttp.log.access_logger.
• access_log_format (str) – Access log format string. Default: helpers.AccessLogger.LOG_FORMAT.
• max_line_size (int) – Optional maximum header line size. Default: 8190.
• max_headers (int) – Optional maximum header size. Default: 32768.
• max_field_size (int) – Optional maximum header field size. Default: 8190.
• lingering_time (float) – Maximum time during which the server reads and ignores additional data coming from the client when lingering close is on. Use 0 to disable lingering on server channel closing.

app
Read-only attribute for accessing to Application served instance.

coroutine setup()
Initialize application. Should be called before adding sites.

The method calls Application.on_startup registered signals.

coroutine cleanup()
Stop handling all registered sites and cleanup used resources.

Application.on_shutdown and Application.on_cleanup signals are called internally.

class aiohttp.web.ServerRunner(web_server, *, handle_signals=False, **kwargs)
A runner for low-level Server. Used with conjunction with sites to serve on specific port.

Inherited from BaseRunner.

Parameters

• web_server (Server) – low-level web server instance to serve.
• handle_signals (bool) – add signal handlers for signal.SIGINT and signal.SIGTERM (False by default).
**kwargs – named parameters to pass into web protocol.

See also:

*Low Level Server* demonstrates low-level server usage

**class** `aiohttp.web.BaseSite`

An abstract class for handled sites.

**name**

An identifier for site, read-only `str` property. Could be a handled URL or UNIX socket path.

**coroutine start()**

Start handling a site.

**coroutine stop()**

Stop handling a site.

**class** `aiohttp.web.TCPSite(runner, host=None, port=None, *, shutdown_timeout=60.0, ssl_context=None, backlog=128, reuse_address=None, reuse_port=None)`

Serve a runner on TCP socket.

**Parameters**

- **runner** – a runner to serve.
- **host (str)** – HOST to listen on, `'0.0.0.0'` if `None` (default).
- **port (int)** – PORT to listed on, `8080` if `None` (default).
- **shutdown_timeout (float)** – a timeout for closing opened connections on `BaseSite.stop()` call.
- **ssl_context** – a `ssl.SSLContext` instance for serving SSL/TLS secure server, `None` for plain HTTP server (default).
- **backlog (int)** – a number of unaccepted connections that the system will allow before refusing new connections, see `socket.listen()` for details.
- `128` by default.
- **reuse_address (bool)** – tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire. If not specified will automatically be set to True on UNIX.
- **reuse_port (bool)** – tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.

**class** `aiohttp.web.UnixSite(runner, path, *, shutdown_timeout=60.0, ssl_context=None, backlog=128)`

Serve a runner on UNIX socket.

**Parameters**

- **runner** – a runner to serve.
- **path (str)** – PATH to UNIX socket to listen.
- **shutdown_timeout (float)** – a timeout for closing opened connections on `BaseSite.stop()` call.
- **ssl_context** – a `ssl.SSLContext` instance for serving SSL/TLS secure server, `None` for plain HTTP server (default).
- **backlog (int)** – a number of unaccepted connections that the system will allow before refusing new connections, see `socket.listen()` for details.
128 by default.

```python
class aiohttp.web.NamedPipeSite(runner, path, *, shutdown_timeout=60.0)
```
Serve a runner on Named Pipe in Windows.

**Parameters**
- **runner** – a runner to serve.
- **path (str)** – PATH of named pipe to listen.
- **shutdown_timeout (float)** – a timeout for closing opened connections on `BaseSite.stop()` call.

```python
class aiohttp.web.SockSite(runner, sock, *, shutdown_timeout=60.0, ssl_context=None, backlog=128)
```
Serve a runner on UNIX socket.

**Parameters**
- **runner** – a runner to serve.
- **sock** – `socket.socket` to listen.
- **shutdown_timeout (float)** – a timeout for closing opened connections on `BaseSite.stop()` call.
- **ssl_context** – a `ssl.SSLContext` instance for serving SSL/TLS secure server, `None` for plain HTTP server (default).
- **backlog (int)** – a number of unaccepted connections that the system will allow before refusing new connections, see `socket.listen()` for details.

128 by default.

### Utilities

```python
class aiohttp.web.FileField
```
A namedtuple instance 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**

```python
aiocoro.web.run_app(app, *, host=None, port=None, path=None, sock=None, shutdown_timeout=60.0, ssl_context=None, print=print, backlog=128, access_log_class=aiohttp.helpers.AccessLogger, access_log_format=aiohttp.helpers.AccessLogger.LOG_FORMAT, access_log=aiohttp.log.access_logger, handle_signals=True, reuse_address=None, reuse_port=None)
```
A utility function for running an application, serving it until keyboard interrupt and performing a Graceful shutdown.
Suitable as handy tool for scaffolding aiohttp based projects. Perhaps production config will use more sophisticated runner but it good enough at least at very beginning stage.

The server will listen on any host or Unix domain socket path you supply. If no hosts or paths are supplied, or only a port is supplied, a TCP server listening on 0.0.0.0 (all hosts) will be launched.

Distributing HTTP traffic to multiple hosts or paths on the same application process provides no performance benefit as the requests are handled on the same event loop. See Server Deployment for ways of distributing work for increased performance.

Parameters

- **app** — Application instance to run or a coroutine that returns an application.
- **host** *(str)* — TCP/IP host or a sequence of hosts for HTTP server. Default is '0.0.0.0' if port has been specified or if path is not supplied.
- **port** *(int)* — TCP/IP port for HTTP server. Default is 8080 for plain text HTTP and 8443 for HTTP via SSL (when ssl_context parameter is specified).
- **path** *(str)* — file system path for HTTP server Unix domain socket. A sequence of file system paths can be used to bind multiple domain sockets. Listening on Unix domain sockets is not supported by all operating systems.
- **sock** *(socket)* — a preexisting socket object to accept connections on. A sequence of socket objects can be passed.
- **shutdown_timeout** *(int)* — a delay to wait for graceful server shutdown before disconnecting all open client sockets hard way.

A system with properly Graceful shutdown implemented never waits for this timeout but closes a server in a few milliseconds.

- **ssl_context** — ssl.SSLContext for HTTPS server, None for HTTP connection.
- **print** — a callable compatible with `print()` . May be used to override STDOUT output or suppress it. Passing None disables output.
- **backlog** *(int)* — the number of unaccepted connections that the system will allow before refusing new connections (128 by default).
- **access_log** — logging.Logger instance used for saving access logs. Use None for disabling logs for sake of speedup.
- **access_log_format** — access log format, see Format specification for details.
- **handle_signals** *(bool)* — override signal TERM handling to gracefully exit the application.
- **reuse_address** *(bool)* — tells the kernel to reuse a local socket in TIME_WAIT state, without waiting for its natural timeout to expire. If not specified will automatically be set to True on UNIX.
- **reuse_port** *(bool)* — tells the kernel to allow this endpoint to be bound to the same port as other existing endpoints are bound to, so long as they all set this flag when being created. This option is not supported on Windows.

New in version 3.0: Support access_log_class parameter.

Support reuse_address, reuse_port parameter.
New in version 3.1: Accept a coroutine as `app` parameter.

**Constants**

class aiohttp.web.ContentCoding

An `enum.Enum` class of available Content Codings.

deflate

`DEFLATE` compression

gzip

`GZIP` compression

identity

no compression

**Middlewares**

**Normalize path middleware**

aiohttp.web.normalize_path_middleware(*, append_slash=True, remove_slash=False, merge_slashes=True, redirect_class=HTTPPermanentRedirect)

Middleware factory which produces a middleware that normalizes the path of a request. By normalizing it means:

- Add or remove a trailing slash to the path.
- Double slashes are replaced by one.

The middleware returns as soon as it finds a path that resolves correctly. The order if both merge and append/remove are enabled is:

1. `merge_slashes`
2. `append_slash` or `remove_slash`
3. both `merge_slashes` and `append_slash` or `remove_slash`

If the path resolves with at least one of those conditions, it will redirect to the new path.

Only one of `append_slash` and `remove_slash` can be enabled. If both are `True` the factory will raise an `AssertionError`

If `append_slash` is `True` the middleware will append a slash when needed. If a resource is defined with trailing slash and the request comes without it, it will append it automatically.

If `remove_slash` is `True`, `append_slash` must be `False`. When enabled the middleware will remove trailing slashes and redirect if the resource is defined.

If `merge_slashes` is `True`, merge multiple consecutive slashes in the path into one.

New in version 3.4: Support for `remove_slash`
### 12.2.5 Web Server Exceptions

**Overview**

`aiohttp.web` defines a set of exceptions for every HTTP status code.

Each exception is a subclass of `HTTPException` and relates to a single HTTP status code:

```python
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
            * 308 - HTTPPermanentRedirect
    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 - HTTPRangeNotSatisfiable
            * 417 - HTTPExpectationFailed
            * 421 - HTTPMisdirectedRequest
            * 422 - HTTPUnprocessableEntity
            * 424 - HTTPFailedDependency
```
* 426 - HTTPUpgradeRequired
* 428 - HTTPPreconditionRequired
* 429 - HTTPTooManyRequests
* 431 - HTTPRequestHeaderFieldsTooLarge
* 451 - HTTPUnavailableForLegalReasons

HTTPServerError
* 500 - HTTPInternalServerError
* 501 - HTTPNotImplemented
* 502 - HTTPBadGateway
* 503 - HTTPServiceUnavailable
* 504 - HTTPGatewayTimeout
* 505 - HTTPVersionNotSupported
* 506 - HTTPVariantAlsoNegotiates
* 507 - HTTPInsufficientStorage
* 510 - HTTPNotExtended
* 511 - HTTPNetworkAuthenticationRequired

All HTTP exceptions have the same constructor signature:

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

If not directly specified, `headers` will be added to the default response headers.

Classes `HTTPMultipleChoices`, `HTTPMovedPermanently`, `HTTPFound`, `HTTPSeeOther`, `HTTPUseProxy`, `HTTPTemporaryRedirect` have the following constructor signature:

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

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

`HTTPMethodNotAllowed` is constructed by providing the incoming unsupported method and list of allowed methods:

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

**Base HTTP Exception**

```python
exception aiohttp.web.HTTPException(*, headers=None, reason=None, text=None, content_type=None)
```

The base class for HTTP server exceptions. Inherited from `Exception`.

**Parameters**

- `headers` – HTTP headers (`Mapping`)
- `reason` (`str`) – an optional custom HTTP reason. aiohttp uses default reason string if not specified.
- `text` (`str`) – an optional text used in response body. If not specified `default text` is constructed from status code and reason, e.g. “404: Not Found”.
- `content_type` (`str`) – an optional Content-Type, “text/plain” by default.

**status**

HTTP status code for the exception, `int`
reason
HTTP status reason for the exception, str

text
HTTP status reason for the exception, str or None for HTTP exceptions without body, e.g. “204 No Content”

headers
HTTP headers for the exception, multidict.CIMultiDict

Successful Exceptions

HTTP exceptions for status code in range 200-299. They are not errors but special classes reflected in exceptions hierarchy. E.g. raise web.HTTPNoContent may look strange a little but the construction is absolutely legal.

exception aiohttp.web.HTTPSuccessful
A base class for the category, a subclass of HTTPException.

exception aiohttp.web.HTTPOK
An exception for 200 OK, a subclass of HTTPSuccessful.

exception aiohttp.web.HTTPCreated
An exception for 201 Created, a subclass of HTTPSuccessful.

exception aiohttp.web.HTTPAccepted
An exception for 202 Accepted, a subclass of HTTPSuccessful.

exception aiohttp.web.HTTPNonAuthoritativeInformation
An exception for 203 Non-Authoritative Information, a subclass of HTTPSuccessful.

exception aiohttp.web.HTTPNoContent
An exception for 204 No Content, a subclass of HTTPSuccessful.
Has no HTTP body.

exception aiohttp.web.HTTPResetContent
An exception for 205 Reset Content, a subclass of HTTPSuccessful.
Has no HTTP body.

exception aiohttp.web.HTTPPartialContent
An exception for 206 Partial Content, a subclass of HTTPSuccessful.

Redirections

HTTP exceptions for status code in range 300-399, e.g. raise web. HTTPMovedPermanently(location='/new/path').

exception aiohttp.web.HTTPRedirection
A base class for the category, a subclass of HTTPException.

exception aiohttp.web.HTTPMove(locatin, *, headers=None, reason=None, text=None, content_type=None)
A base class for redirections with implied Location header, all redirections except HTTPNotModified.

Parameters location - a yarl.URL or str used for Location HTTP header.
For other arguments see HTTPException constructor.

location
A Location HTTP header value, yarl.URL.
exception aiohttp.web.HTTPMultipleChoices
    An exception for 300 Multiple Choices, a subclass of HTTPMove.

exception aiohttp.web.HTTPMovedPermanently
    An exception for 301 Moved Permanently, a subclass of HTTPMove.

exception aiohttp.web.HTTPFound
    An exception for 302 Found, a subclass of HTTPMove.

exception aiohttp.web.HTTPSeeOther
    An exception for 303 See Other, a subclass of HTTPMove.

exception aiohttp.web.HTTPNotModified
    An exception for 304 Not Modified, a subclass of HTTPRedirection.
    Has no HTTP body.

exception aiohttp.web.HTTPUseProxy
    An exception for 305 Use Proxy, a subclass of HTTPMove.

exception aiohttp.web.HTTPTemporaryRedirect
    An exception for 307 Temporary Redirect, a subclass of HTTPMove.

exception aiohttp.web.HTTPPermanentRedirect
    An exception for 308 Permanent Redirect, a subclass of HTTPMove.

**Client Errors**

HTTP exceptions for status code in range 400-499, e.g. raise web.HTTPNotFound().

exception aiohttp.web.HTTPClientError
    A base class for the category, a subclass of HTTPException.

exception aiohttp.web.HTTPBadRequest
    An exception for 400 Bad Request, a subclass of HTTPClientError.

exception aiohttp.web.HTTPUnauthorized
    An exception for 401 Unauthorized, a subclass of HTTPClientError.

exception aiohttp.web.HTTPPaymentRequired
    An exception for 402 Payment Required, a subclass of HTTPClientError.

exception aiohttp.web.HTTPForbidden
    An exception for 403 Forbidden, a subclass of HTTPClientError.

exception aiohttp.web.HTTPNotFound
    An exception for 404 Not Found, a subclass of HTTPClientError.

exception aiohttp.web.HTTPMethodNotAllowed
    An exception for 405 Method Not Allowed, a subclass of HTTPClientError.
    Parameters
        • method (str) – requested but not allowed HTTP method.
        • allowed_methods – an iterable of allowed HTTP methods (str), Allow HTTP
          header is constructed from the sequence separated by comma.
    For other arguments see HTTPException constructor.

allowed_methods
    A set of allowed HTTP methods.
method
Requested but not allowed HTTP method.

exception aiohttp.web.HTTPNotAcceptable
An exception for 406 Not Acceptable, a subclass of HTTPClientError.

exception aiohttp.web.HTTPProxyAuthenticationRequired
An exception for 407 Proxy Authentication Required, a subclass of HTTPClientError.

exception aiohttp.web.HTTPRequestTimeout
An exception for 408 Request Timeout, a subclass of HTTPClientError.

exception aiohttp.web.HTTPConflict
An exception for 409 Conflict, a subclass of HTTPClientError.

exception aiohttp.web.HTTPGone
An exception for 410 Gone, a subclass of HTTPClientError.

exception aiohttp.web.HTTPLengthRequired
An exception for 411 Length Required, a subclass of HTTPClientError.

exception aiohttp.web.HTTPPreconditionFailed
An exception for 412 Precondition Failed, a subclass of HTTPClientError.

exception aiohttp.web.HTTPRequestEntityTooLarge(max_size, actual_size, **kwargs)
An exception for 413 Entity Too Large, a subclass of HTTPClientError.

Parameters

• max_size (int) – Maximum allowed request body size
• actual_size (int) – Actual received size
For other acceptable parameters see HTTPException constructor.

exception aiohttp.web.HTTPRequestURITooLong
An exception for 414 URI is too long, a subclass of HTTPClientError.

exception aiohttp.web.HTTPUnsupportedMediaType
An exception for 415 Entity body in unsupported format, a subclass of HTTPClientError.

exception aiohttp.web.HTTPRangeNotSatisfiable
An exception for 416 Cannot satisfy request range, a subclass of HTTPClientError.

exception aiohttp.web.HTTPExpectationFailed
An exception for 417 Expect condition could not be satisfied, a subclass of HTTPClientError.

exception aiohttp.web.HTTPMisdirectedRequest
An exception for 421 Misdirected Request, a subclass of HTTPClientError.

exception aiohttp.web.HTTPUnprocessableEntity
An exception for 422 Unprocessable Entity, a subclass of HTTPClientError.

exception aiohttp.web.HTTPFailedDependency
An exception for 424 Failed Dependency, a subclass of HTTPClientError.

exception aiohttp.web.HTTPUpgradeRequired
An exception for 426 Upgrade Required, a subclass of HTTPClientError.

exception aiohttp.web.HTTPPreconditionRequired
An exception for 428 Precondition Required, a subclass of HTTPClientError.

exception aiohttp.web.HTTPTooManyRequests
An exception for 429 Too Many Requests, a subclass of HTTPClientError.
exception aiohttp.web.HTTPRequestHeaderFieldsTooLarge
   An exception for 431 Requests Header Fields Too Large, a subclass of HTTPClientError.

exception aiohttp.web.HTTPUnavailableForLegalReasons
   An exception for 451 Unavailable For Legal Reasons, a subclass of HTTPClientError.
   Parameters
   Parameters **link** – A link to a resource with information for blocking reason.
   A URL link to a resource with information for blocking reason, read-only property.

Server Errors

HTTP exceptions for status code in range 500-599, e.g. raise web.HTTPBadGateway().

exception aiohttp.web.HTTPServerError
   A base class for the category, a subclass of HTTPException.

exception aiohttp.web.HTTPInternalError
   An exception for 500 Server got itself in trouble, a subclass of HTTPServerError.

exception aiohttp.web.HTTPNotImplemented
   An exception for 501 Server does not support this operation, a subclass of HTTPServerError.

exception aiohttp.web.HTTPBadGateway
   An exception for 502 Invalid responses from another server/proxy, a subclass of HTTPServerError.

exception aiohttp.web.HTTPServiceUnavailable
   An exception for 503 The server cannot process the request due to a high load, a subclass of HTTPServerError.

exception aiohttp.web.HTTPGatewayTimeout
   An exception for 504 The gateway server did not receive a timely response, a subclass of HTTPServerError.

exception aiohttp.web.HTTPVersionNotSupported
   An exception for 505 Cannot fulfill request, a subclass of HTTPServerError.

exception aiohttp.web.HTTPVariantAlsoNegotiates
   An exception for 506 Variant Also Negotiates, a subclass of HTTPServerError.

exception aiohttp.web.HTTPInsufficientStorage
   An exception for 507 Insufficient Storage, a subclass of HTTPServerError.

exception aiohttp.web.HTTPNotExtended
   An exception for 510 Not Extended, a subclass of HTTPServerError.

exception aiohttp.web.HTTPNetworkAuthenticationRequired
   An exception for 511 Network Authentication Required, a subclass of HTTPServerError.
12.2.6 Logging

`aiohttp` uses standard `logging` for tracking the library activity.

We have the following loggers enumerated by names:

- `'aiohttp.access'`
- `'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.

Logging does not work out of the box. It requires at least minimal `logging` configuration. Example of minimal working logger setup:

```python
import logging
from aiohttp import web

app = web.Application()
logging.basicConfig(level=logging.DEBUG)
web.run_app(app, port=5000)
```

New in version 4.0.0.

**Access logs**

Access logs are enabled by default. If the `debug` flag is set, and the default logger `'aiohttp.access'` is used, access logs will be output to `stderr` if no handlers are attached. Furthermore, if the default logger has no log level set, the log level will be set to `logging.DEBUG`.

This logging may be controlled by `aiohttp.web.AppRunner()` and `aiohttp.web.run_app()`.

To override the default logger, pass an instance of `logging.Logger` to override the default logger.

**Note:** Use `web.run_app(app, access_log=None)` to disable access logs.

In addition, `access_log_format` may be used to specify the log format.
Format specification

The library provides custom micro-language to specifying info about request and response:

<table>
<thead>
<tr>
<th>Option</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>%%</td>
<td>The percent sign</td>
</tr>
<tr>
<td>%a</td>
<td>Remote IP-address (IP-address of proxy if using reverse proxy)</td>
</tr>
<tr>
<td>%t</td>
<td>Time when the request was started to process</td>
</tr>
<tr>
<td>%P</td>
<td>The process ID of the child that serviced the request</td>
</tr>
<tr>
<td>%r</td>
<td>First line of request</td>
</tr>
<tr>
<td>%s</td>
<td>Response status code</td>
</tr>
<tr>
<td>%b</td>
<td>Size of response in bytes, including HTTP headers</td>
</tr>
<tr>
<td>%T</td>
<td>The time taken to serve the request, in seconds</td>
</tr>
<tr>
<td>%Tf</td>
<td>The time taken to serve the request, in seconds with fraction in %.06f format</td>
</tr>
<tr>
<td>%D</td>
<td>The time taken to serve the request, in microseconds</td>
</tr>
<tr>
<td>%{FOO}i</td>
<td>request.headers['FOO']</td>
</tr>
<tr>
<td>%{FOO}o</td>
<td>response.headers['FOO']</td>
</tr>
</tbody>
</table>

The default access log format is:

```
'%%a %t "%%r" %s %b "%%{Referer}i" "%%{User-Agent}i"'
```

New in version 2.3.0.

`access_log_class` introduced.

Example of a drop-in replacement for the default access logger:

```python
from aiohttp.abc import AbstractAccessLogger

class AccessLogger(AbstractAccessLogger):
    def log(self, request, response, time):
        self.logger.info(f'
{request.remote}
{request.method} {request.path}
done in {time} s: {response.status}
')
```

New in version 4.0.0.

AccessLogger.log() can now access any exception raised while processing the request with `sys.exc_info()`.

New in version 4.0.0.

If your logging needs to perform IO you can instead inherit from `aiohttp.abc.AbstractAsyncAccessLogger`:

```python
from aiohttp.abc import AbstractAsyncAccessLogger

class AccessLogger(AbstractAsyncAccessLogger):
    async def log(self, request, response, time):
        logging_service = request.app['logging_service']
        await logging_service.log(f'{request.remote} '

This also allows access to the results of coroutines on the request and response, e.g. request.text().
Gunicorn access logs

When Gunicorn is used for deployment, its default access log format will be automatically replaced with the default aiohttp’s access log format.

If Gunicorn’s option access_logformat is specified explicitly, it should use aiohttp’s format specification.

Gunicorn’s access log works only if accesslog is specified explicitly in your config or as a command line option. This configuration can be either a path or '--'. If the application uses a custom logging setup intercepting the 'gunicorn.access' logger, accesslog should be set to '--' to prevent Gunicorn to create an empty access log file upon every startup.

Error logs

aiohttp.web uses a logger named 'aiohttp.server' to store errors given on web requests handling.

This log is enabled by default.

To use a different logger name, pass logger (logging.Logger instance) to the aiohttp.web.AppRunner() constructor.

12.2.7 Testing

Testing aiohttp web servers

aiohttp provides plugin for pytest making writing web server tests extremely easy, it also provides test framework agnostic utilities for testing with other frameworks such as unittest.

Before starting to write your tests, you may also be interested on reading how to write testable services that interact with the loop.

For using pytest plugin please install pytest-aiohttp library:

```
$ pip install pytest-aiohttp
```

If you don’t want to install pytest-aiohttp for some reason you may insert pytest_plugins = 'aiohttp.pytest_plugin' line into conf/test.py instead for the same functionality.

Provisional Status

The module is a provisional.

aiohttp has a year and half period for removing deprecated API (Policy for Backward Incompatible Changes).

But for aiohttp.test_tools the deprecation period could be reduced.

Moreover we may break backward compatibility without deprecation period for some very strong reason.
The Test Client and Servers

`aiohttp` test utils provides a scaffolding for testing aiohttp-based web servers. They consist of two parts: running test server and making HTTP requests to this server. `TestServer` runs `aiohttp.web.Application` based server, `RawTestServer` starts `aiohttp.web.WebServer` low level server.

For performing HTTP requests to these servers you have to create a test client: `TestClient` instance. The client incapsulates `aiohttp.ClientSession` by providing proxy methods to the client for common operations such as `ws_connect`, `get`, `post`, etc.

Pytest

The `aiohttp_client` fixture available from `pytest-aiohttp` plugin allows you to create a client to make requests to test your app. A simple would be:

```
from aiohttp import web

async def hello(request):
    return web.Response(text='Hello, world')

async def test_hello(aiohttp_client):
    app = web.Application()
    app.router.add_get('/', hello)
    client = await aiohttp_client(app)
    resp = await client.get('/')
    assert resp.status == 200
    text = await resp.text()
    assert 'Hello, world' in text
```

It also provides access to the app instance allowing tests to check the state of the app. Tests can be made even more succinct with a fixture to create an app test client:

```
import pytest
from aiohttp import web

async def previous(request):
    if request.method == 'POST':
        request.app['value'] = (await request.post())['value']
    return web.Response(body=b'thanks for the data')

async def test_set_value(cli):
    app = web.Application()
    app.router.add_get('/', previous)
    app.router.add_post('/', previous)
    return loop.run_until_complete(aiohttp_client(app))
```

(continues on next page)
```python
resp = await cli.post('/', data={'value': 'foo'})
assert resp.status == 200
assert await resp.text() == 'thanks for the data'
assert cli.server.app['value'] == 'foo'

async def test_get_value(cli):
    cli.server.app['value'] = 'bar'
    resp = await cli.get('/')
    assert resp.status == 200
    assert await resp.text() == 'value: bar'
```

Pytest tooling has the following fixtures:

```python
pytest_aiohttp.aiohttp_server(app, *, port=None, **kwargs)
A fixture factory that creates TestServer:

async def test_f(aiohttp_server):
    app = web.Application()
    # fill route table
    server = await aiohttp_server(app)
```

The server will be destroyed on exit from test function.

- `app` is the aiohttp.web.Application used to start server.
- `port` optional, port the server is run at, if not provided a random unused port is used.

New in version 3.0.

- `kwargs` are parameters passed to aiohttp.web.AppRunner()
- Changed in version 3.0.
- Deprecated since version 3.2: The fixture was renamed from test_server to aiohttp_server.

```python
pytest_aiohttp.aiohttp_client(app, server_kwargs=None, **kwargs)
pytest_aiohttp.aiohttp_client(server, **kwargs)
pytest_aiohttp.aiohttp_client(raw_server, **kwargs)
```

A fixture factory that creates TestClient for access to tested server:

```python
async def test_f(aiohttp_client):
    app = web.Application()
    # fill route table
    client = await aiohttp_client(app)
    resp = await client.get('/')
```

- `client` and responses are cleaned up after test function finishing.
- `server_kwargs` are parameters passed to the test server if an app is passed, else ignored.
- `kwargs` are parameters passed to aiohttp.test_utils.TestClient constructor.

- Changed in version 3.0: The fixture was renamed from test_client to aiohttp_client.

```python
pytest_aiohttp.aiohttp_raw_server(handler, *, port=None, **kwargs)
```

A fixture factory that creates RawTestServer instance from given web handler.
async def test_f(aiohttp_raw_server, aiohttp_client):
    async def handler(request):
        return web.Response(text="OK")

    raw_server = await aiohttp_raw_server(handler)
    client = await aiohttp_client(raw_server)
    resp = await client.get('/')

handler should be a coroutine which accepts a request and returns response, e.g.

port optional, port the server is run at, if not provided a random unused port is used.

New in version 3.0.

pytest_aiohttp.aiohttp_unused_port
Function to return an unused port number for IPv4 TCP protocol:

async def test_f(aiohttp_client, aiohttp_unused_port):
    port = aiohttp_unused_port()
    app = web.Application()
    # fill route table

    client = await aiohttp_client(app, server_kwargs={'port': port})
    ...

    Changed in version 3.0: The fixture was renamed from unused_port to aiohttp_unused_port.

Unittest

To test applications with the standard library’s unittest or unittest-based functionality, the AioHTTPTestCase is provided:

from aiohttp.test_utils import AioHTTPTestCase, unittest_run_loop
from aiohttp import web

class MyAppTestCase(AioHTTPTestCase):
    async def get_application(self):
        """Override the get_app method to return your application."
        """
        async def hello(request):
            return web.Response(text='Hello, world')

        app = web.Application()
        app.router.add_get('/', hello)
        return app

    # the unittest_run_loop decorator can be used in tandem with
    # the AioHTTPTestCase to simplify running
    # tests that are asynchronous
    @unittest_run_loop
    async def test_example(self):
        resp = await self.client.request("GET", "/")
        assert resp.status == 200
        assert resp.text()
assert "Hello, world" in text

# a vanilla example
def test_example_vanilla(self):
    async def test_get_route():
        url = "/"
        resp = await self.client.request("GET", url)
        assert resp.status == 200
        text = await resp.text()
        assert "Hello, world" in text

    self.loop.run_until_complete(test_get_route())

class aiohttp.test_utils.AioHTTPTestCase
    A base class to allow for unittest web applications using aiohttp.
    Derived from unittest.TestCase
    Provides the following:

    client
        an aiohttp test client, TestClient instance.

    server
        an aiohttp test server, TestServer instance.

        New in version 2.3.

    loop
        The event loop in which the application and server are running.

        Deprecated since version 3.5.

    app
        The application returned by get_app () (aiohttp.web.Application instance).

    coroutine get_client ()
        This async method can be overridden to return the TestClient object used in the test.

        Returns TestClient instance.

        New in version 2.3.

    coroutine get_server ()
        This async method can be overridden to return the TestServer object used in the test.

        Returns TestServer instance.

        New in version 2.3.

    coroutine get_application ()
        This async method should be overridden to return the aiohttp.web.Application object to test.

        Returns aiohttp.web.Application instance.

    coroutine setUpAsync ()
        This async method do nothing by default and can be overridden to execute asynchronous code during the setUp stage of the TestCase.

        New in version 2.3.
**coroutine tearDownAsync()**

This async method do nothing by default and can be overridden to execute asynchronous code during the tearDown stage of the TestCase.

New in version 2.3.

**setUp()**

Standard test initialization method.

**tearDown()**

Standard test finalization method.

**Note:** The TestClient’s methods are asynchronous: you have to execute function on the test client using asynchronous methods.

A basic test class wraps every test method by `unittest_run_loop()` decorator:

```python
class TestA(AioHTTPTestCase):
    @unittest_run_loop
    async def test_f(self):
        resp = await self.client.get('/')
```

**unittest_run_loop:**

A decorator dedicated to use with asynchronous methods of an `AioHTTPTestCase`.

Handles executing an asynchronous function, using the `AioHTTPTestCase.loop` of the `AioHTTPTestCase`.

**Faking request object**

`aiohttp` provides test utility for creating fake `aiohttp.web.Request` objects: `aiohttp.test_utils.make_mocked_request()`, it could be useful in case of simple unit tests, like handler tests, or simulate error conditions that hard to reproduce on real server:

```python
from aiohttp import web
from aiohttp.test_utils import make_mocked_request

def handler(request):
    assert request.headers.get('token') == 'x'
    return web.Response(body=b'data')

def test_handler():
    req = make_mocked_request('GET', '/', headers={'token': 'x'})
    resp = handler(req)
    assert resp.body == b'data'
```

**Warning:** We don’t recommend to apply `make_mocked_request()` everywhere for testing web-handler’s business object – please use test client and real networking via ‘localhost’ as shown in examples before. `make_mocked_request()` exists only for testing complex cases (e.g. emulating network errors) which are extremely hard or even impossible to test by conventional way.
Make mocked web.Request testing purposes.

Useful in unit tests, when spinning full web server is overkill or specific conditions and errors are hard to trigger.

**Parameters**

- `method` *(str)* – str, that represents HTTP method, like; GET, POST.
- `path` *(str)* – str, The URL including **PATH_INFO** without the host or scheme
- `headers` *(dict, multidict.CIMultiDict, list of pairs)* – mapping containing the headers. Can be anything accepted by the multidict.CIMultiDict constructor.
- `match_info` *(dict)* – mapping containing the info to match with url parameters.
- `version` *(aiohttp.protocol.HttpVersion)* – namedtuple with encoded HTTP version
- `closing` *(bool)* – flag indicates that connection should be closed after response.
- `app` *(aiohttp.web.Application)* – the aiohttp.web application attached for fake request
- `writer` *(aiohttp.StreamWriter)* – object for managing outcoming data
- `transport` *(asyncio.transports.Transport)* – asyncio transport instance
- `payload` *(aiohttp.StreamReader)* – raw payload reader object
- `sslcontext` *(ssl.SSLContext)* – ssl.SSLContext object, for HTTPS connection
- `loop` *(asyncio.AbstractEventLoop)* – An event loop instance, mocked loop by default.

**Returns** `aiohttp.web.Request` object.

New in version 2.3: `match_info` parameter.

---

**Framework Agnostic Utilities**

High level test creation:

```python
from aiohttp.test_utils import TestClient, TestServer, loop_context
from aiohttp import request

# loop_context is provided as a utility. You can use any
# asyncio.BaseEventLoop class in its place.
with loop_context() as loop:
    app = _create_example_app()
    with TestClient(TestServer(app), loop=loop) as client:

        async def test_get_route():
            nonlocal client
            resp = await client.get("/")
            assert resp.status == 200
            text = await resp.text()
            assert "Hello, world" in text
```

(continues on next page)
If it’s preferred to handle the creation / teardown on a more granular basis, the TestClient object can be used directly:

```python
from aiohttp.test_utils import TestClient, TestServer

with loop_context() as loop:
    app = _create_example_app()
    client = TestClient(TestServer(app), loop=loop)
    loop.run_until_complete(client.start_server())
    root = "http://127.0.0.1:").format(port)

    async def test_get_route():
        resp = await client.get("/")
        assert resp.status == 200
        text = await resp.text()
        assert "Hello, world" in text

    loop.run_until_complete(test_get_route())
    loop.run_until_complete(client.close())
```

A full list of the utilities provided can be found at the [api reference](api-reference).

### Testing API Reference

#### Test server

Runs given `aiohttp.web.Application` instance on random TCP port.

After creation the server is not started yet, use `start_server()` for actual server starting and `close()` for stopping/cleanup.

Test server usually works in conjunction with `aiohttp.test_utils.TestClient` which provides handy client methods for accessing to the server.

```python
class aiohttp.test_utils.BaseTestServer (*, scheme='http', host='127.0.0.1', port=None)
    Base class for test servers.

    Parameters
    ----------
    • scheme (str) – HTTP scheme, non-protected "http" by default.
    • host (str) – a host for TCP socket, IPv4 local host ('127.0.0.1') by default.
    • port (int) – optional port for TCP socket, if not provided a random unused port is used.

    scheme
    A scheme for tested application, 'http' for non-protected run and 'https' for TLS encrypted server.

    host
    host used to start a test server.

    port
    port used to start the test server.
```

12.2. Server
handler
    aiohttp.web.WebServer used for HTTP requests serving.

server
    asyncio.AbstractServer used for managing accepted connections.

coroutine start_server(**kwargs)
    Start a test server.

coroutine close()
    Stop and finish executed test server.

make_url(path)
    Return an absolute URL for given path.

class aiohttp.test_utils.RawTestServer(handler, *, scheme="http", host='127.0.0.1')
    Low-level test server (derived from BaseTestServer).
    Parameters
    • handler – a coroutine for handling web requests. The handler should accept aiohttp.web.BaseRequest and return a response instance, e.g. StreamResponse or Response.
      The handler could raise HTTPException as a signal for non-200 HTTP response.
    • scheme (str) – HTTP scheme, non-protected "http" by default.
    • host (str) – a host for TCP socket, IPv4 local host ('127.0.0.1') by default.
    • port (int) – optional port for TCP socket, if not provided a random unused port is used.

    New in version 3.0.

class aiohttp.test_utils.TestServer(app, *, scheme="http", host='127.0.0.1')
    Test server (derived from BaseTestServer) for starting Application.
    Parameters
    • app – aiohttp.web.Application instance to run.
    • scheme (str) – HTTP scheme, non-protected "http" by default.
    • host (str) – a host for TCP socket, IPv4 local host ('127.0.0.1') by default.
    • port (int) – optional port for TCP socket, if not provided a random unused port is used.

    New in version 3.0.

app
    aiohttp.web.Application instance to run.

Test Client

class aiohttp.test_utils.TestClient(app_or_server, *, scheme='http', host='127.0.0.1',
    cookie_jar=None, **kwargs)
    A test client used for making calls to tested server.
    Parameters
    • app_or_server – BaseTestServer instance for making client requests to it.

    In order to pass an aiohttp.web.Application you need to convert it first to TestServer first with TestServer(app).
• **cookie_jar** – an optional `aiohttp.CookieJar` instance, may be useful with `CookieJar(unsafe=True)` option.

• **scheme** *(str)* – HTTP scheme, non-protected "http" by default.

• **host** *(str)* – a host for TCP socket, IPv4 *local host* ('127.0.0.1') by default.

**scheme**

A scheme for tested application, 'http' for non-protected run and 'https' for TLS encrypted server.

**host**

host used to start a test server.

**port**

port used to start the server

**server**

*BaseTestServer* test server instance used in conjunction with client.

**app**

An alias for *self.server.app*. return None if *self.server* is not *TestServer* instance(e.g. *RawTestServer* instance for test low-level server).

**session**

An internal `aiohttp.ClientSession`.

Unlike the methods on the *TestClient*, client session requests do not automatically include the host in the url queried, and will require an absolute path to the resource.

**coroutine start_server** (**kwargs**)

Start a test server.

**coroutine close** ()

Stop and finish executed test server.

**make_url** *(path)*

Return an absolute URL for given path.

**coroutine request** *(method, path, *args, **kwargs)*

Routes a request to tested http server.

The interface is identical to `aiohttp.ClientSession.request()`, except the loop kwarg is overridden by the instance used by the test server.

**coroutine get** *(path, *args, **kwargs)*

Perform an HTTP GET request.

**coroutine post** *(path, *args, **kwargs)*

Perform an HTTP POST request.

**coroutine options** *(path, *args, **kwargs)*

Perform an HTTP OPTIONS request.

**coroutine head** *(path, *args, **kwargs)*

Perform an HTTP HEAD request.

**coroutine put** *(path, *args, **kwargs)*

Perform an HTTP PUT request.

**coroutine patch** *(path, *args, **kwargs)*

Perform an HTTP PATCH request.

**coroutine delete** *(path, *args, **kwargs)*

Perform an HTTP DELETE request.
coroutine ws_connect(path, *args, **kwargs)
Initiate websocket connection.

The api corresponds to aiohttp.ClientSession.ws_connect().

Utilities

aiohttp.test_utils.make_mocked_coro(return_value)
Creates a coroutine mock.

Behaves like a coroutine which returns return_value. But it is also a mock object, you might test it as usual

Mock:

```python
mocked = make_mocked_coro(1)
assert 1 == await mocked(1, 2)
mocked.assert_called_with(1, 2)
```

Parameters return_value – A value that the mock object will return when called.

Returns A mock object that behaves as a coroutine which returns return_value when called.

aiohttp.test_utils.unused_port()
Return an unused port number for IPv4 TCP protocol.

Return int ephemeral port number which could be reused by test server.

aiohttp.test_utils.loop_context(loop_factory=<function asyncio.new_event_loop>)
A contextmanager that creates an event_loop, for test purposes.

Handles the creation and cleanup of a test loop.

aiohttp.test_utils.setup_test_loop(loop_factory=<function asyncio.new_event_loop>)
Create and return an asyncio.AbstractEventLoop instance.

The caller should also call teardown_test_loop, once they are done with the loop.

Note: As side effect the function changes asyncio default loop by asyncio.set_event_loop() call.

Previous default loop is not restored.

It should not be a problem for test suite: every test expects a new test loop instance anyway.

Changed in version 3.1: The function installs a created event loop as default.

aiohttp.test_utils.teardown_test_loop(loop)
Teardown and cleanup an event_loop created by setup_test_loop.

Parameters loop (asyncio.AbstractEventLoop) – the loop to teardown

12.2.8 Server Deployment

There are several options for aiohttp server deployment:

- Standalone server
- Running a pool of backend servers behind of nginx, HAProxy or other reverse proxy server
- Using gunicorn behind of reverse proxy

Every method has own benefits and disadvantages.


Standalone

Just call `aiohttp.web.run_app()` function passing `aiohttp.web.Application` instance.
The method is very simple and could be the best solution in some trivial cases. But it does not utilize all CPU cores.
For running multiple aiohttp server instances use reverse proxies.

Nginx+supervisord

Running aiohttp servers behind nginx makes several advantages.
At first, nginx is the perfect frontend server. It may prevent many attacks based on malformed http protocol etc.
Second, running several aiohttp instances behind nginx allows to utilize all CPU cores.
Third, nginx serves static files much faster than built-in aiohttp static file support.
But this way requires more complex configuration.

Nginx configuration

Here is short extraction about writing Nginx configuration file. It does not cover all available Nginx options.
For full reference read Nginx tutorial and official Nginx documentation.
First configure HTTP server itself:

```
http {
    server {
        listen 80;
        client_max_body_size 4G;

        server_name example.com;

        location / {
            proxy_set_header Host $http_host;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
            proxy_redirect off;
            proxy_buffering off;
            proxy_pass http://aiohttp;
        }

        location /static {
            # path for static files
            root /path/to/app/static;
        }
    }
}
```

This config listens on port 80 for server named `example.com` and redirects everything to `aiohttp` backend group.
Also it serves static files from `/path/to/app/static` path as `example.com/static`.
Next we need to configure `aiohttp` upstream group:
All HTTP requests for `http://example.com` except ones for `http://example.com/static` will be redirected to `example1.sock`, `example2.sock`, `example3.sock` or `example4.sock` backend servers. By default, Nginx uses round-robin algorithm for backend selection.

**Note:** Nginx is not the only existing reverse proxy server but the most popular one. Alternatives like HAProxy may be used as well.

### Supervisord

After configuring Nginx we need to start our aiohttp backends. Better to use some tool for starting them automatically after system reboot or backend crash.

There are very many ways to do it: Supervisord, Upstart, Systemd, Gaffer, Circus, Runit etc.

Here we'll use Supervisord for example:

```ini
[program:aiohttp]
numprocs = 4
numprocs_start = 1
process_name = example_%(process_num)s

; Unix socket paths are specified by command line.
command=/path/to/aiohttp_example.py --path=/tmp/example_%(process_num)s.sock

; We can just as easily pass TCP port numbers:
; command=/path/to/aiohttp_example.py --port=808%(process_num)s

user=nobody
autostart=true
autorestart=true
```
**aiohttp server**

The last step is preparing aiohttp server for working with supervisord.

Assuming we have properly configured `aiohttp.web.Application` and port is specified by command line, the task is trivial:

```python
# aiohttp_example.py
import argparse
from aiohttp import web

parser = argparse.ArgumentParser(description="aiohttp server example")
parser.add_argument('--path')
parser.add_argument('--port')

if __name__ == '__main__':
    app = web.Application()
    # configure app
    args = parser.parse_args()
    web.run_app(app, path=args.path, port=args.port)
```

For real use cases we perhaps need to configure other things like logging etc., but it’s out of scope of the topic.

**Nginx+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.

In opposite to deployment with bare Nginx the solution does not need to manually run several aiohttp processes and use tool like supervisord for monitoring it. But nothing is for free: running aiohttp application under gunicorn is slightly slower.

**Prepare environment**

You firstly need to setup your deployment environment. This example is based on Ubuntu 16.04.

Create a directory for your application:

```
>> mkdir myapp
>> cd myapp
```

Create Python virtual environment:

```
>> python3 -m venv venv
>> source venv/bin/activate
```

Now that the virtual environment is ready, we’ll proceed to install aiohttp and gunicorn:

```
>> pip install gunicorn
>> pip install aiohttp
```
Application

Lets write a simple application, which we will save to file. We’ll name this file `my_app_module.py`:

```python
from aiohttp import web

async def index(request):
    return web.Response(text="Welcome home!")

my_web_app = web.Application()
my_web_app.router.add_get('/', index)
```

Application factory

As an option an entry point could be a coroutine that accepts no parameters and returns an application instance:

```python
from aiohttp import web

async def index(request):
    return web.Response(text="Welcome home!")

async def my_web_app():
    app = web.Application()
    app.router.add_get('/', index)
    return app
```

Start Gunicorn

When running Gunicorn, you provide the name of the module, i.e. `my_app_module`, and the name of the app or application factory, 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?)
- you may also want to use the `--accesslog` flag to enable the access log to be populated. (See logging for more information.)

The custom worker subclass is defined in `aiohttp.GunicornWebWorker`:

```bash
$ gunicorn my_app_module:my_web_app --bind localhost:8080 --worker-class aiohttp.GunicornWebWorker
[2017-03-11 18:27:21 +0000] [1249] [INFO] Starting gunicorn 19.7.1
[2017-03-11 18:27:21 +0000] [1249] [INFO] Listening at: http://127.0.0.1:8080 (1249)
[2017-03-11 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.

**Note:** If you want to use an alternative asyncio event loop uvloop, you can use the `aiohttp.GunicornUVLoopWebWorker` worker class.

---

### Proxy through NGINX

We can proxy our gunicorn workers through NGINX with a configuration like this:

```conf
worker_processes 1;
user nobody nogroup;
events {
  worker_connections 1024;
}
http {
  ## Main Server Block
  server {
    ## Open by default.
    listen 80 default_server;
    server_name main;
    client_max_body_size 200M;

    ## Main site location.
    location / {
      proxy_pass http://127.0.0.1:8080;
      proxy_set_header Host $host;
      proxy_set_header X-Forwarded-Host $server_name;
      proxy_set_header X-Real-IP $remote_addr;
    }
  }
}
```

Since gunicorn listens for requests at our localhost address on port 8080, we can use the `proxy_pass` directive to send web traffic to our workers. If everything is configured correctly, we should reach our application at the ip address of our web server.

### Proxy through NGINX + SSL

Here is an example NGINX configuration setup to accept SSL connections:

```conf
worker_processes 1;
user nobody nogroup;
events {
  worker_connections 1024;
}
http {
  ## SSL Redirect
  server {
    listen 80 default;
    return 301 https://$host$request_uri;
  }

  ## Main Server Block
```

(continues on next page)
The first server block accepts regular http connections on port 80 and redirects them to our secure SSL connection. The second block matches our previous example except we need to change our open port to https and specify where our SSL certificates are being stored with the `ssl_certificate` and `ssl_certificate_key` directives.

During development, you may want to create your own self-signed certificates for testing purposes and use another service like Let’s Encrypt when you are ready to move to production.

More information

See the official documentation for more information about suggested nginx configuration. You can also find out more about configuring for secure https connections as well.

Logging configuration

aiohttp and gunicorn use different format for specifying access log.

By default aiohttp uses own defaults:

```
'${a} ${t} "${s} ${b} "${Referrer}i" "${User-Agent}i"'
```

For more information please read Format Specification for Access Log.

Proxy through Apache at your own risk

Issues have been reported using Apache2 in front of aiohttp server: #2687 Intermittent 502 proxy errors when running behind Apache <https://github.com/aio-libs/aiohttp/issues/2687>.
12.3 Utilities

Miscellaneous API Shared between Client And Server.

12.3.1 Abstract Base Classes

Abstract routing

aiohttp has abstract classes for managing web interfaces.

The most part of `aiohttp.web` is not intended to be inherited but few of them are.

aiohttp.web is built on top of few concepts: `application`, `router`, `request` and `response`.

`router` is a `plugable` part: a library user may build a `router` from scratch, all other parts should work with new router seamlessly.

`AbstractRouter` has the only mandatory method: `AbstractRouter.resolve()` coroutine. It must return an `AbstractMatchInfo` instance.

If the requested URL handler is found `AbstractMatchInfo.handler()` is a `web-handler` for requested URL and `AbstractMatchInfo.http_exception` is `None`.

Otherwise `AbstractMatchInfo.http_exception` is an instance of `HTTPException` like `404: NotFound` or `405: Method Not Allowed`. `AbstractMatchInfo.handler()` raises `http_exception` on call.

```python
class aiohttp.abc.AbstractRouter

coroutine resolve(request)
    Performs URL resolving. It’s an abstract method, should be overridden in `router` implementation.
    Parameters request -- `aiohttp.web.Request` instance for resolving, the request has
        `aiohttp.web.Request.match_info` equals to `None` at resolving stage.

    Returns `AbstractMatchInfo` instance.
```

```python
class aiohttp.abc.AbstractMatchInfo
    Abstract match info, returned by `AbstractRouter.resolve()` call.

http_exception
    `aiohttp.web.HTTPException` if no match was found, `None` otherwise.

coroutine handler(request)
    Abstract method performing `web-handler` processing.
    Parameters request -- `aiohttp.web.Request` instance for resolving, the request has
        `aiohttp.web.Request.match_info` equals to `None` at resolving stage.

    Returns `aiohttp.web.StreamResponse` or descendants.
    Raise `aiohttp.web.HTTPException` on error

coroutine expect_handler(request)
    Abstract method for handling `100-continue` processing.
```
Abstract Class Based Views

For class based view support aiohttp has abstract `AbstractView` class which is awaitable (may be uses like `await Cls()` or `yield from Cls()`) and has a `request` as an attribute.

```python
class aiohttp.abc.AbstractView
    An abstract class, base for all class based views implementations.

    Methods __iter__ and __await__ should be overridden.

    request
        aiohttp.web.Request instance for performing the request.
```

Abstract Cookie Jar

```python
class aiohttp.abc.AbstractCookieJar
    The cookie jar instance is available as ClientSession.cookie_jar.

    The jar contains Morsel items for storing internal cookie data.

    API provides a count of saved cookies:

    ```python
    len(session.cookie_jar)
    ```

    These cookies may be iterated over:

    ```python
    for cookie in session.cookie_jar:
        print(cookie.key)
        print(cookie['domain'])
    ```

    An abstract class for cookie storage. Implements collections.abc.Iterable and collections.abc.Sized.

    ```python
    update_cookies(cookies, response_url=None)
    ```

    Update cookies returned by server in Set-Cookie header.

    Parameters

    - `cookies` – a collections.abc.Mapping (e.g. dict, SimpleCookie) or iterable of pairs with cookies returned by server’s response.

    - `response_url` (str) – URL of response. None for shared cookies. Regular cookies are coupled with server’s URL and are sent only to this server, shared ones are sent in every client request.

    ```python
    filter_cookies(request_url)
    ```

    Return jar’s cookies acceptable for URL and available in Cookie header for sending client requests for given URL.

    Parameters `response_url` (str) – request’s URL for which cookies are asked.

    Returns http.cookies.SimpleCookie with filtered cookies for given URL.
Abstract Abstract Access Logger

class aiohttp.abc.AbstractAccessLogger
   An abstract class, base for all RequestHandler access_logger implementations
   Method log should be overridden.
   log(request, response, time)
       Parameters
       • request – aiohttp.web.Request object.
       • time (float) – Time taken to serve the request.

12.3.2 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.

Reading Multipart Responses

Assume you made a request, as usual, and want to process the response multipart data:

```python
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:

```python
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:

```python
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:

```python
if part is None:
    break
```

Both `BodyPartReader` and `MultipartReader` provides access to body part headers: this allows you to filter parts by their attributes:
```python
if part.headers[aiohttp.hdrs.CONTENT_TYPE] == 'application/json':
    metadata = await part.json()
    continue
```

Nor `BodyPartReader` or `MultipartReader` instances does not 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:

```python
if part.filename != 'secret.txt':
    continue
```

If current body part does not 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 was not, 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:

```python
filedata = await part.read(decode=False)
```

Later you may decide to decode the data. It’s still simple and possible to do:

```python
filedata = part.decode(filedata)
```

Once you are done with multipart processing, just break a loop:

```python
break
```

### 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:

```python
with aiohttp.MultipartWriter('mixed') as mpwriter:
    ...
    with aiohttp.MultipartWriter('related') as subwriter:
        ...
    mpwriter.append(subwriter)
    with aiohttp.MultipartWriter('related') as subwriter:
        ...
```

(continues on next page)
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`:

```python
mpwriter.append('hello')
```

For binary data `application/octet-stream` is used:

```python
mpwriter.append(b'aiohttp')
```

You can always override these default by passing your own headers with the second argument:

```python
mpwriter.append(io.BytesIO(b'GIF89a...'),
{'CONTENT-TYPE': 'image/gif'})
```

For file objects `Content-Type` will be determined by using Python’s mod:`mimetypes` module and additionally `Content-Disposition` header will include the file’s basename:

```python
part = root.append(open(__file__, 'rb'))
```

If you want to send a file with a different name, just handle the `Payload` instance which `MultipartWriter.append()` will always return and set `Content-Disposition` explicitly by using the `Payload.set_content_disposition()` helper:

```python
part.set_content_disposition('attachment', filename='secret.txt')
```

Additionally, you may want to set other headers here:

```python
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):

```python
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:

```python
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.ClientSession.request()` data argument:

```python
await session.post('http://example.com', data=mpwriter)
```
Behind the scenes `MultipartWriter.write()` 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.write()` 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.

Example MJPEG Streaming `multipart/x-mixed-replace`. By default `MultipartWriter.write()` appends closing `--boundary--` and breaks your content. Providing `close_boundary = False` prevents this:

```python
my_boundary = b'some-boundary'
response = web.StreamResponse(
    status=200,
    reason='OK',
    headers={
        'Content-Type': f'multipart/x-mixed-replace;boundary={my_boundary}'.format(my_boundary)
    }
)

while True:
    frame = get_jpeg_frame()
    with MultipartWriter('image/jpeg', boundary=my_boundary) as mpwriter:
        mpwriter.append(frame, {
            'Content-Type': 'image/jpeg'
        })
    await mpwriter.write(response, close_boundary=False)
    await response.drain()
```

## 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 does not work.

For instance, is server used `cgi.FieldStorage` then you have to ensure that no body part contains a `Content-Length` header:

```python
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` does not 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:

```python
class Writer:
    def __init__(self):
        self.buffer = bytearray()

    async def write(self, data):
        self.buffer.extend(data)

writer = Writer()
await mpwriter.write(writer)
await aiohttp.post('http://example.com',
                   data=writer.buffer, 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:
Reading such kind of data in single stream is possible, but is not clean at all:

```python
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()])
```

12.3. Utilities
Let’s hack a reader in the way to return pairs of document and reader of the related files on each iteration:

```python
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:

```python
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 reference*
12.3.3 Multipart reference

class aiohttp.MultipartResponseWrapper (resp, stream)
Wrapper around the MultipartBodyReader to take care about underlying connection and close it when it needs in.

at_eof ()
Returns True when all response data had been read.
Return type bool

coroutine next ()
Emits next multipart reader object.

coroutine release ()
Releases the connection gracefully, reading all the content to the void.

class aiohttp.BodyPartReader (boundary, headers, content)
Multipart reader for single body part.

coroutine 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 bytarray

coroutine read_chunk (size=chunk_size)
Reads body part content chunk of the specified size.
Parameters size (int) – chunk size
Return type bytarray

coroutine readline ()
Reads body part by line by line.
Return type bytarray

coroutine release ()
Like read(), but reads all the data to the void.
Return type None

coroutine text (*, encoding=None)
Like 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

coroutine json (*, encoding=None)
Like read(), but assumes that body parts contains JSON data.
Parameters encoding (str) – Custom JSON encoding. Overrides specified in charset param of Content-Type header

coroutine form (*, encoding=None)
Like 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

at_eof ()
Returns True if the boundary was reached or False otherwise.
Return type bool
**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, binary encodings for Content-Transfer-Encoding header.

**Parameters**
- `data (bytearray)` – Data to decode.

**Raises**
- `RuntimeError` - if encoding is unknown.

**Return type**
- bytes

**get_charset** *(default=None)*
Returns charset parameter from Content-Type header or default.

**name**
A field name specified in Content-Disposition header or None if missed or header is malformed.

**filename**
A field filename specified in Content-Disposition header or None if missed or header is malformed.

**class**
```python
class aiohttp.MultipartReader(headers, content)
```
Multipart body reader.

**classmethod**
```python
from_response(cls, response)
```
Constructs reader instance from HTTP response.

**at_eof()**
- Returns True if the final boundary was reached or False otherwise.

**Return type**
- bool

**coroutine**
```python
next()
```
Emits the next multipart body part.

**coroutine**
```python
release()
```
Reads all the body parts to the void till the final boundary.

**coroutine**
```python
fetch_next_part()
```
Returns the next body part reader.

**class**
```python
class aiohttp.MultipartWriter(subtype='mixed', boundary=None, close_boundary=True)
```
Multipart body writer.

**boundary**
The string (str) representation of the boundary.

Changed in version 3.0: Property type was changed from bytes to str.

**append** *(obj, headers=None)*
- Append an object to writer.

**append_payload** *(payload)*
- Adds a new body part to multipart writer.
append_json(obj, headers=None)
Helper to append JSON part.

append_form(obj, headers=None)
Helper to append form urlencoded part.

size
Size of the payload.

coroutine write(writer, close_boundary=True)
Write body.

Parameters:
- close_boundary (bool) – The \( \text{bool} \) that will emit boundary closing.
- You may want to disable when streaming \( \text{multipart/x-mixed-replace} \)
- New in version 3.4: Support close_boundary argument.

12.3.4 Streaming API

aiohttp uses streams for retrieving \( \text{BODIES} \): aiohttp.web.Request.content and aiohttp.
ClientResponse.content are properties with stream API.

class aiohttp.StreamReader
The reader from incoming stream.

User should never instantiate streams manually but use existing aiohttp.web.Request.content and 
aiohttp.ClientResponse.content properties for accessing raw \( \text{BODY} \) data.

Reading Methods

coroutine StreamReader.read(n=-1)
Read up to \( n \) bytes. If \( n \) is not provided, or set to \( -1 \), read until EOF and return all read bytes.

- If the EOF was received and the internal buffer is empty, return an empty bytes object.
- Parameters:
- \( n (\text{int}) \) – how many bytes to read, \( -1 \) for the whole stream.
- Return bytes the given data

coroutine StreamReader.readany()
Read next data portion for the stream.

- Returns immediately if internal buffer has a data.
- Return bytes the given data

coroutine StreamReader.readexactly(n)
Read exactly \( n \) bytes.

- Raise an asyncio.IncompleteReadError if the end of the stream is reached before \( n \) can be read, the 
aiohttp.IncompleteReadError.partial attribute of the exception contains the partial read bytes.
- Parameters
- \( n (\text{int}) \) – how many bytes to read.
- Return bytes the given data

coroutine StreamReader.readline()
Read one line, where “line” is a sequence of bytes ending with \( \backslash n \).

- If EOF is received, and \( \backslash n \) was not found, the method will return the partial read bytes.
- If the EOF was received and the internal buffer is empty, return an empty bytes object.
- Return bytes the given line

coroutine StreamReader.readchunk()
Read a chunk of data as it was received by the server.
Returns a tuple of (data, end_of_HTTP_chunk).

When chunked transfer encoding is used, end_of_HTTP_chunk is a bool indicating if the end of the data corresponds to the end of a HTTP chunk, otherwise it is always False.

**Return tuple** [bytes, bool] a chunk of data and a bool that is True when the end of the returned chunk corresponds to the end of a HTTP chunk.

### Asynchronous Iteration Support

Stream reader supports asynchronous iteration over **BODY**.

By default it iterates over lines:

```python
async for line in response.content:
    print(line)
```

Also there are methods for iterating over data chunks with maximum size limit and over any available data.

**async-for** **StreamReader.iterChunked**(*n*)
Iterates over data chunks with maximum size limit:

```python
async for data in response.content.iter_chunked(1024):
    print(data)
```

**async-for** **StreamReader.iterAny**()
Iterates over data chunks in order of intaking them into the stream:

```python
async for data in response.content.iter_any():
    print(data)
```

**async-for** **StreamReader.iterChunks**()
Iterates over data chunks as received from the server:

```python
async for data, _ in response.content.iter_chunks():
    print(data)
```

If chunked transfer encoding is used, the original http chunks formatting can be retrieved by reading the second element of returned tuples:

```python
buffer = b"

async for data, end_of_http_chunk in response.content.iter_chunks():
    buffer += data
    if end_of_http_chunk:
        print(buffer)
        buffer = b"
```
Helpers

StreamReader.exception()
Get the exception occurred on data reading.

aiohttp.is_eof()
Return True if EOF was reached.
Internal buffer may be not empty at the moment.

See also:
StreamReader.at_eof()

StreamReader.at_eof()
Return True if the buffer is empty and EOF was reached.

StreamReader.read_nowait(n=None)
Returns data from internal buffer if any, empty bytes object otherwise.
Raises RuntimeError if other coroutine is waiting for stream.

Parameters
n (int) – how many bytes to read, -1 for the whole internal buffer.
Return bytes the given data

StreamReader.unread_data(data)
Rollback reading some data from stream, inserting it to buffer head.

Parameters
data (bytes) – data to push back into the stream.

Warning: The method does not wake up waiters.
E.g. read() will not be resumed.

coroutine aiohttp.wait_eof()
Wait for EOF. The given data may be accessible by upcoming read calls.

12.3.5 Signals

Signal is a list of registered asynchronous callbacks.
The signal’s life-cycle has two stages: after creation its content could be filled by using standard list operations:
sig.append() etc.
After sig.freeze() call the signal is frozen: adding, removing and dropping callbacks are forbidden.
The only available operation is calling previously registered callbacks by await sig.send(data).
For concrete usage examples see signals in aiohttp.web chapter.
Changed in version 3.0: sig.send() call is forbidden for non-frozen signal.
Support for regular (non-async) callbacks is dropped. All callbacks should be async functions.

class aiohttp.Signal
The signal, implements collections.abc.MutableSequence interface.

coroutine send(*args, **kwargs)
Call all registered callbacks one by one starting from the begin of list.

frozen
True if freeze() was called, read-only property.
freeze()
    Freeze the list. After the call any content modification is forbidden.

12.3.6 Common data structures

Common data structures used by aiohttp internally.

**FrozenList**

A list-like structure which implements `collections.abc.MutableSequence`.

The list is mutable unless `FrozenList.freeze()` is called, after that the list modification raises `RuntimeError`.

```python
class aiohttp.FrozenList(items):
    Construct a new non-frozen list from `items` iterable.

    The list implements all `collections.abc.MutableSequence` methods plus two additional APIs.

    frozen
        A read-only property, `True` is the list is frozen (modifications are forbidden).

    freeze()
        Freeze the list. There is no way to thaw it back.
```

**ChainMapProxy**

An immutable version of `collections.ChainMap`. Internally the proxy is a list of mappings (dictionaries), if the requested key is not present in the first mapping the second is looked up and so on.

The class supports `collections.abc.Mapping` interface.

```python
class aiohttp.ChainMapProxy(maps):
    Create a new chained mapping proxy from a list of mappings (`maps`).

    New in version 3.2.
```

12.3.7 WebSocket utilities

```python
class aiohttp.WSCloseCode
    An `IntEnum` for keeping close message code.

    OK
        A normal closure, meaning that the purpose for which the connection was established has been fulfilled.

    GOING_AWAY
        An endpoint is "going away", such as a server going down or a browser having navigated away from a page.

    PROTOCOL_ERROR
        An endpoint is terminating the connection due to a protocol error.

    UNSUPPORTED_DATA
        An endpoint is terminating the connection because it has received a type of data it cannot accept (e.g., an endpoint that understands only text data MAY send this if it receives a binary message).
```
An endpoint is terminating the connection because it has received data within a message that was not consistent with the type of the message (e.g., non-UTF-8 RFC 3629 data within a text message).

An endpoint is terminating the connection because it has received a message that violates its policy. This is a generic status code that can be returned when there is no other more suitable status code (e.g., unsupported_data or message_too_big) or if there is a need to hide specific details about the policy.

An endpoint is terminating the connection because it has received a message that is too big for it to process.

An endpoint (client) is terminating the connection because it has expected the server to negotiate one or more extension, but the server did not return them in the response message of the WebSocket handshake. The list of extensions that are needed should appear in the /reason/ part of the Close frame. Note that this status code is not used by the server, because it can fail the WebSocket handshake instead.

A server is terminating the connection because it encountered an unexpected condition that prevented it from fulfilling the request.

The service is restarted. A client may reconnect, and if it chooses to do, should reconnect using a randomized delay of 5-30s.

The service is experiencing overload. A client should only connect to a different IP (when there are multiple for the target) or reconnect to the same IP upon user action.

```
class aiohttp.WSMsgType
    An IntEnum for describing WSMessage type.
```

```
    CONTINUATION
        A mark for continuation frame, user will never get the message with this type.
    TEXT
        Text message, the value has str type.
    BINARY
        Binary message, the value has bytes type.
    PING
        Ping frame (sent by client peer).
    PONG
        Pong frame, answer on ping. Sent by server peer.
    CLOSE
        Close frame.
    CLOSED_FRAME
        Actually not frame but a flag indicating that websocket was closed.
    ERROR
        Actually not frame but a flag indicating that websocket was received an error.
```

```
class aiohttp.WSMessage
    Websocket message, returned by .receive() calls.
```
**type**
Message type, `WSMsgType` instance.

**data**
Message payload.
1. `str` for `WSMsgType.TEXT` messages.
2. `bytes` for `WSMsgType.BINARY` messages.
3. `WSCloseCode` for `WSMsgType.CLOSE` messages.
4. `bytes` for `WSMsgType.PING` messages.
5. `bytes` for `WSMsgType.PONG` messages.

**extra**
Additional info, `str`.
Makes sense only for `WSMsgType.CLOSE` messages, contains optional message description.

**json** (*, loads=json.loads)
Returns parsed JSON data.

Parameters `loads` – optional JSON decoder function.

## 12.4 FAQ

- Are there plans for an `@app.route` decorator like in Flask?
- Does aiohttp have a concept like Flask’s “blueprint” or Django’s “app”?
- How do I create a route that matches urls with a given prefix?
- Where do I put my database connection so handlers can access it?
- How can middleware store data for web handlers to use?
- Can a handler receive incoming events from different sources in parallel?
- How do I programatically close a WebSocket server-side?
- How do I make a request from a specific IP address?
- What is the API stability and deprecation policy?
- How do I enable gzip compression globally for my entire application?
- How do I manage a ClientSession within a web server?
- How do I access database connections from a subapplication?
- How do I perform operations in a request handler after sending the response?
- How do I make sure my custom middleware response will behave correctly?
- Why is creating a ClientSession outside of an event loop dangerous?
12.4.1 Are there plans for an @app.route decorator like in Flask?

As of aiohttp 2.3, `RouteTableDef` provides an API similar to Flask’s `@app.route`. See *Alternative ways for registering routes*.

Unlike Flask’s `@app.route`, `RouteTableDef` does not require an `app` in the module namespace (which often leads to circular imports).

Instead, a `RouteTableDef` is decoupled from an application instance:

```python
routes = web.RouteTableDef()

@routes.get('/get')
async def handle_get(request):
    ...

@routes.post('/post')
async def handle_post(request):
    ...

app.router.add_routes(routes)
```

12.4.2 Does aiohttp have a concept like Flask’s “blueprint” or Django’s “app”?  

If you’re writing a large application, you may want to consider using *nested applications*, which are similar to Flask’s “blueprints” or Django’s “apps”.

See: *Nested applications*.

12.4.3 How do I create a route that matches urls with a given prefix?

You can do something like the following:

```python
app.router.add_route('*', '/path/to/{tail:.+}', sink_handler)
```

The first argument, `*`, matches any HTTP method (`GET, POST, OPTIONS`, etc). The second argument matches URLs with the desired prefix. The third argument is the handler function.

12.4.4 Where do I put my database connection so handlers can access it?

*aiohttp.web.Application* object supports the *dict* interface and provides a place to store your database connections or any other resource you want to share between handlers.

```python
async def go(request):
    db = request.app['db']
    cursor = await db.cursor()
    await cursor.execute('SELECT 42')
    # ...
    return web.Response(status=200, text='ok')

async def init_app():
    app = Application()
```

(continues on next page)
db = await create_connection(user='user', password='123')
app['db'] = db
app.router.add_get('/', go)
return app

12.4.5 How can middleware store data for web handlers to use?

Both `aiohttp.web.Request` and `aiohttp.web.Application` support the `dict` interface. Therefore, data may be stored inside a request object.

```python
async def handler(request):
    request['unique_key'] = data
```

See https://github.com/aio-libs/aiohttp_session code for an example. The `aiohttp_session.get_session(request)` method uses `SESSION_KEY` for saving request-specific session information.

As of aiohttp 3.0, all response objects are dict-like structures as well.

12.4.6 Can a handler receive incoming events from different sources in parallel?

Yes.

As an example, we may have two event sources:

1. WebSocket for events from an end user
2. Redis PubSub for events from other parts of the application

The most native way to handle this is to create a separate task for PubSub handling.

Parallel `aiohttp.web.WebSocketResponse.receive()` calls are forbidden; a single task should perform WebSocket reading. However, other tasks may use the same WebSocket object for sending data to peers.

```python
async def handler(request):
    ws = web.WebSocketResponse()
    await ws.prepare(request)
    task = asyncio.create_task(read_subscription(ws, request.app['redis']))

    try:
        async for msg in ws:
            # handle incoming messages
            # use ws.send_str() to send data back
            ...

    finally:
        task.cancel()

async def read_subscription(ws, redis):
    channel, = await redis.subscribe('channel:1')

    try:
        async for msg in channel.iter():
            answer = process_the_message(msg)  # your function here
```

(continues on next page)
12.4.7 How do I programmatically close a WebSocket server-side?

Let's say we have an application with two endpoints:

1. /echo a WebSocket echo server that authenticates the user
2. /logout_user that, when invoked, closes all open WebSockets for that user.

One simple solution is to keep a shared registry of WebSocket responses for a user in the `aiohttp.web.Application` instance and call `aiohttp.web.WebSocketResponse.close()` on all of them in `/logout_user` handler:

```python
async def echo_handler(request):
    ws = web.WebSocketResponse()
    user_id = authenticate_user(request)
    await ws.prepare(request)
    request.app['websockets'][user_id].add(ws)
    try:
        async for msg in ws:
            ws.send_str(msg.data)
    finally:
        request.app['websockets'][user_id].remove(ws)
    return ws

async def logout_handler(request):
    user_id = authenticate_user(request)
    ws_closers = [ws.close()
                  for ws in request.app['websockets'][user_id]
                  if not ws.closed]
    # Watch out, this will keep us from returing the response
    # until all are closed
    ws_closers and await asyncio.gather(*ws_closers)
    return web.Response(text='OK')
```

```python
def main():
    loop = asyncio.get_event_loop()
    app = web.Application()
    app.router.add_route('GET', '/echo', echo_handler)
    app.router.add_route('POST', '/logout', logout_handler)
    app['websockets'] = defaultdict(set)
    web.run_app(app, host='localhost', port=8080)
```
12.4.8 How do I make a request from a specific IP address?

If your system has several IP interfaces, you may choose one which will be used to bind a socket locally:

```python
conn = aiohttp.TCPConnector(local_addr=('127.0.0.1', 0))
async with aiohttp.ClientSession(connector=conn) as session:
    ...
```

See also:

*aiohttp.TCPConnector* and *local_addr* parameter.

12.4.9 What is the API stability and deprecation policy?

*aiohttp* follows strong Semantic Versioning (SemVer).

Obsolete attributes and methods are marked as *deprecated* in the documentation and raise *DeprecationWarning* upon usage.

Assume *aiohttp X.Y.Z* where *X* is major version, *Y* is minor version and *Z* is bugfix number.

For example, if the latest released version is *aiohttp==3.0.6*:

- **3.0.7** fixes some bugs but have no new features.
- **3.1.0** introduces new features and can deprecate some API but never remove it, also all bug fixes from previous release are merged.
- **4.0.0** removes all deprecations collected from **3.Y** versions except deprecations from the **last 3.Y** release. These deprecations will be removed by **5.0.0**.

Unfortunately we may have to break these rules when a *security vulnerability* is found. If a security problem cannot be fixed without breaking backward compatibility, a bugfix release may break compatibility. This is unlikely, but possible.

All backward incompatible changes are explicitly marked in *the changelog*.

12.4.10 How do I enable gzip compression globally for my entire application?

It’s impossible. Choosing what to compress and what not to compress is a tricky matter.

If you need global compression, write a custom middleware. Or enable compression in NGINX (you are deploying aiohttp behind reverse proxy, right?).

12.4.11 How do I manage a ClientSession within a web server?

*aiohttp.ClientSession* should be created once for the lifetime of the server in order to benefit from connection pooling.

Sessions save cookies internally. If you don’t need cookie processing, use *aiohttp.DummyCookieJar*. If you need separate cookies for different http calls but process them in logical chains, use a single *aiohttp.TCPConnector* with separate client sessions and *own_connector=False*.
12.4.12 How do I access database connections from a subapplication?

Restricting access from subapplication to main (or outer) app is a deliberate choice. A subapplication is an isolated unit by design. If you need to share a database object, do it explicitly:

```python
subapp['db'] = mainapp['db']
mainapp.add_subapp('/prefix', subapp)
```

12.4.13 How do I perform operations in a request handler after sending the response?

Middlewares can be written to handle post-response operations, but they run after every request. You can explicitly send the response by calling `aiohttp.web.Response.write_eof()`, which starts sending before the handler returns, giving you a chance to execute follow-up operations:

```python
def ping_handler(request):
    """Send PONG and increase DB counter."""
    # explicitly send the response
    resp = web.json_response({'message': 'PONG'})
    await resp.prepare(request)
    await resp.write_eof()

    # increase the pong count
    APP['db'].inc_pong()
    return resp
```

A `aiohttp.web.Response` object must be returned. This is required by aiohttp web contracts, even though the response already been sent.

12.4.14 How do I make sure my custom middleware response will behave correctly?

Sometimes your middleware handlers might need to send a custom response. This is just fine as long as you always create a new `aiohttp.web.Response` object when required.

The response object is a Finite State Machine. Once it has been dispatched by the server, it will reach its final state and cannot be used again.

The following middleware will make the server hang, once it serves the second response:

```python
from aiohttp import web

def misbehaved_middleware():
    # don't do this!
    cached = web.Response(status=200, text='Hi, I am cached!')

    async def middleware(request, handler):
        # ignoring response for the sake of this example
        _res = handler(request)
        return cached

    return middleware
```
The rule of thumb is *one request, one response*.

### 12.4.15 Why is creating a ClientSession outside of an event loop dangerous?

Short answer is: life-cycle of all asyncio objects should be shorter than life-cycle of event loop.

Full explanation is longer. All asyncio object should be correctly finished/disconnected/closed before event loop shutdown. Otherwise user can get unexpected behavior. In the best case it is a warning about unclosed resource, in the worst case the program just hangs, awaiting for coroutine is never resumed etc.

Consider the following code from *mod.py*:

```python
import aiohttp

session = aiohttp.ClientSession()

async def fetch(url):
    async with session.get(url) as resp:
        return await resp.text()
```

The session grabs current event loop instance and stores it in a private variable.

The main module imports the module and installs *uvloop* (an alternative fast event loop implementation).

```python
import asyncio
import uvloop
import mod

asyncio.set_event_loop_policy(uvloop.EventLoopPolicy())
asyncio.run(main())
```

The code is broken: *session* is bound to default asyncio loop on import time but the loop is changed after the import by set_event_loop(). As result fetch() call hangs.

To avoid import dependency hell *aiohttp* encourages creation of *ClientSession* from async function. The same policy works for *web.Application* too.

Another use case is unit test writing. Very many test libraries (*aiohttp test tools* first) creates a new loop instance for every test function execution. It’s done for sake of tests isolation. Otherwise pending activity (timers, network packets etc.) from previous test may interfere with current one producing very cryptic and unstable test failure.

Note: *class variables* are hidden globals actually. The following code has the same problem as *mod.py* example, *session* variable is the hidden global object:

```python
class A:
    session = aiohttp.ClientSession()

    async def fetch(self, url):
        async with session.get(url) as resp:
            return await resp.text()
```
12.5 Miscellaneous

Helpful pages.

12.5.1 Essays

Router refactoring in 0.21

Rationale

First generation (v1) of router has mapped \((\text{method}, \text{path})\) pair to \text{web-handler}. Mapping is named \text{route}. Routes used to have unique names if any.

The main mistake with the design is coupling the \text{route} to \((\text{method}, \text{path})\) pair while really URL construction operates with \text{resources} (\text{location} is a synonym). HTTP method is not part of URI but applied on sending HTTP request only.

Having different \text{route names} for the same path is confusing. Moreover \text{named routes} constructed for the same path should have unique non overlapping names which is cumbersome in certain situations.

From other side sometimes it’s desirable to bind several HTTP methods to the same web handler. For \text{v1} router it can be solved by passing ‘*’ as HTTP method. Class based views require ‘*’ method also usually.

Implementation

The change introduces \text{resource} as first class citizen:

```
resource = router.add_resource('/path/{to}', name='name')
```

\text{Resource} has a \text{path} (dynamic or constant) and optional \text{name}.

The name is \textbf{unique} in router context.

\text{Resource} has \text{routes}.

\text{Route} corresponds to \text{HTTP method} and \text{web-handler} for the method:

```
route = resource.add_route('GET', handler)
```

User still may use wildcard for accepting all HTTP methods (maybe we will add something like \text{resource}.\text{add_wildcard}(handler) later).

Since \text{names} belongs to \text{resources} now \text{app.router['name']} returns a \text{resource} instance instead of \text{aiohttp.web.Route}.

\text{resource} has .\text{url()} method, so \text{app.router['name']}.\text{url}(\text{parts}={\text{'a': 'b'}}, \text{query}={\text{'arg': 'param'}}) still works as usual.

The change allows to rewrite static file handling and implement nested applications as well.

Decoupling of \text{HTTP location} and \text{HTTP method} makes life easier.
Backward compatibility

The refactoring is 99% compatible with previous implementation. 99% means all example and the most of current code works without modifications but we have subtle API backward incompatibles.

app.router['name'] returns a aiohttp.web.BaseResource instance instead of aiohttp.web.Route but resource has the same resource.url(...) most useful method, so end user should feel no difference.

route.match(...) is not supported anymore, use aiohttp.web.AbstractResource.resolve() instead.

```
resource = app.router.add_resource(path, name=name)
route = resource.add_route(method, handler)
return route
```

app.router.register_route(...) is still supported, it creates aiohttp.web.ResourceAdapter for every call (but it’s deprecated now).

What’s new in aiohttp 1.1

YARL and URL encoding

Since aiohttp 1.1 the library uses yarl for URL processing.

New API

yarl.URL gives handy methods for URL operations etc.

Client API still accepts str everywhere url is used, e.g. session.get('http://example.com') works as well as session.get(yarl.URL('http://example.com')).

Internal API has been switched to yarl.URL aiohttp.CookieJar accepts URL instances only.

On server side has added web.Request.url and web.Request.rel_url properties for representing relative and absolute request’s URL.

URL using is the recommended way, already existed properties for retrieving URL parts are deprecated and will be eventually removed.

Redirection web exceptions accepts yarl.URL as location parameter. str is still supported and will be supported forever.

Reverse URL processing for router has been changed.

The main API is aiohttp.web.Request.url_for(name, **kwargs) which returns a yarl.URL instance for named resource. It does not support query args but adding args is trivial: request.url_for('named_resource', param='a').with_query(arg='val').

The method returns a relative URL, absolute URL may be constructed by request.url.join(request.url_for(...) call.
**URL encoding**

YARL encodes all non-ASCII symbols on `yarl.URL` creation.

Thus `URL('https://www.python.org/')` becomes `'https://www.python.org/%D0%BF%D1%83%D1%82%D1%8C'`.

On filling route table it’s possible to use both non-ASCII and percent encoded paths:

```python
def handler(request):
    return web.json_response({'message': 'Hello from handler!'});

def main_app_routes(app):
    app.router.add_get('/', handler)
    app.router.add_get('/%D0%BF%D1%83%D1%82%D1%8C', handler)
```

are the same. Internally '/' is converted into percent-encoding representation.

Route matching also accepts both URL forms: raw and encoded by converting the route pattern to *canonical* (encoded) form on route registration.

**Sub-Applications**

Sub applications are designed for solving the problem of the big monolithic code base. Let’s assume we have a project with own business logic and tools like administration panel and debug toolbar.

Administration panel is a separate application by its own nature but all toolbar URLs are served by prefix like `/admin`.

Thus we’ll create a totally separate application named `admin` and connect it to main app with prefix:

```python
admin = web.Application()
# setup admin routes, signals and middlewares
app.add_subapp('/admin/', admin)
```

Middlewares and signals from `app` and `admin` are chained.

It means that if URL is ’/admin/something’ middlewares from `app` are applied first and `admin` middlewares are the next in the call chain.

The same is going for `on_response_prepare` signal – the signal is delivered to both top level `app` and `admin` if processing URL is routed to `admin` sub-application.

Common signals like `on_startup`, `on_shutdown` and `on_cleanup` are delivered to all registered sub-applications. The passed parameter is sub-application instance, not top-level application.

Third level sub-applications can be nested into second level ones – there are no limitation for nesting level.

**Url reversing**

Url reversing for sub-applications should generate urls with proper prefix.

But for getting URL sub-application’s router should be used:

```python
admin = web.Application()
admin.add_get('/resource', handler, name='name')
app.add_subapp('/admin/', admin)
```
The generated url from example will have a value URL('/admin/resource').

**Application freezing**

Application can be used either as main app (app.make_handler()) or as sub-application – not both cases at the same time.

After connecting application by .add_subapp() call or starting serving web-server as toplevel application the application is **frozen**.

It means that registering new routes, signals and middlewares is forbidden. Changing state (app['name'] = 'value') of frozen application is deprecated and will be eventually removed.

**Migration to 2.x**

**Client**

**chunking**

aiohttp does not support custom chunking sizes. It is up to the developer to decide how to chunk data streams. If chunking is enabled, aiohttp encodes the provided chunks in the “Transfer-encoding: chunked” format.

aiohttp does not enable chunked encoding automatically even if a transfer-encoding header is supplied: chunked has to be set explicitly. If chunked is set, then the Transfer-encoding and content-length headers are disallowed.

**compression**

Compression has to be enabled explicitly with the compress parameter. If compression is enabled, adding a content-encoding header is not allowed. Compression also enables the chunked transfer-encoding. Compression can not be combined with a Content-Length header.

**Client Connector**

1. By default a connector object manages a total number of concurrent connections. This limit was a per host rule in version 1.x. In 2.x, the limit parameter defines how many concurrent connection connector can open and a new limit_per_host parameter defines the limit per host. By default there is no per-host limit.
2. BaseConnector.close is now a normal function as opposed to coroutine in version 1.x
3. BaseConnector.conn_timeout was moved to ClientSession
**ClientResponse.release**

Internal implementation was significantly redesigned. It is not required to call `release` on the response object. When the client fully receives the payload, the underlying connection automatically returns back to pool. If the payload is not fully read, the connection is closed.

**Client exceptions**

Exception hierarchy has been significantly modified. aiohttp now defines only exceptions that covers connection handling and server response misbehaviors. For developer specific mistakes, aiohttp uses python standard exceptions like ValueError or TypeError.

Reading a response content may raise a ClientPayloadError exception. This exception indicates errors specific to the payload encoding. Such as invalid compressed data, malformed chunked-encoded chunks or not enough data that satisfy the content-length header.

All exceptions are moved from `aiohttp.errors` module to top level `aiohttp` module.

New hierarchy of exceptions:

- **ClientError** - Base class for all client specific exceptions
  - **ClientResponseError** - exceptions that could happen after we get response from server
    - **WSServerHandshakeError** - web socket server response error
    - **ClientHttpProxyError** - proxy response
  - **ClientConnectionError** - exceptions related to low-level connection problems
    - **ClientOSError** - subset of connection errors that are initiated by an OSError exception
      - **ClientConnectorError** - connector related exceptions
      - **ClientProxyConnectionError** - proxy connection initialization error
      - **ServerConnectionError** - server connection related errors
      - **ServerDisconnectedError** - server disconnected
      - **ServerTimeoutError** - server operation timeout, (read timeout, etc)
      - **ServerFingerprintMismatch** - server fingerprint mismatch
  - **ClientPayloadError** - This exception can only be raised while reading the response payload if one of these errors occurs: invalid compression, malformed chunked encoding or not enough data that satisfy content-length header.

**Client payload (form-data)**

To unify form-data/payload handling a new `Payload` system was introduced. It handles customized handling of existing types and provide implementation for user-defined types.

1. `FormData.__call__` does not take an encoding arg anymore and its return value changes from an iterator or bytes to a Payload instance. aiohttp provides payload adapters for some standard types like `str`, `byte`, `io.IOBase`, `StreamReader` or `DataQueue`.

2. a generator is not supported as data provider anymore, `streamer` can be used instead. For example, to upload data from file:
```python
@aiohttp.streamer
def file_sender(writer, file_name=None):
    with open(file_name, 'rb') as f:
        chunk = f.read(2**16)
        while chunk:
            yield from writer.write(chunk)
            chunk = f.read(2**16)

# Then you can use 'file_sender' like this:
async with session.post('http://httpbin.org/post',
                        data=file_sender(file_name='huge_file')) as resp:
    print(await resp.text())
```

Various

1. The `encoding` parameter is deprecated in `ClientSession.request()`. Payload encoding is controlled at the payload level. It is possible to specify an encoding for each payload instance.

2. The `version` parameter is removed in `ClientSession.request()` client version can be specified in the `ClientSession` constructor.

3. `aiohttp.MsgType` dropped, use `aiohttp.WSMsgType` instead.

4. `ClientResponse.url` is an instance of `yarl.URL` class (`url_obj` is deprecated)

5. `ClientResponse.raise_for_status()` raises `aiohttp.ClientResponseError` exception

6. `ClientResponse.json()` is strict about response’s content type. If content type does not match, it raises `aiohttp.ClientResponseError` exception. To disable content type check you can pass `None` as `content_type` parameter.

Server

ServerHttpProtocol and low-level details

Internal implementation was significantly redesigned to provide better performance and support HTTP pipelining. `ServerHttpProtocol` is dropped, implementation is merged with RequestHandler a lot of low-level api’s are dropped.

Application

1. Constructor parameter `loop` is deprecated. Loop is get configured by application runner, `run_app` function for any of gunicorn workers.

2. `Application.router.add_subapp` is dropped, use `Application.add_subapp` instead

3. `Application.finished` is dropped, use `Application.cleanup` instead
WebRequest and WebResponse

1. the GET and POST attributes no longer exist. Use the query attribute instead of GET
2. Custom chunking size is not supported WebResponse.chunked - developer is responsible for actual chunking.
3. Payloads are supported as body. So it is possible to use client response’s content object as body parameter for WebResponse
4. FileSender api is dropped, it is replaced with more general FileResponse class:

   ```python
   async def handle(request):
       return web.FileResponse('path-to-file.txt')
   ```

5. WebSocketResponse.protocol is renamed to WebSocketResponse.ws_protocol. WebSocketResponse.protocol is instance of RequestHandler class.

RequestPayloadError

Reading request’s payload may raise a RequestPayloadError exception. The behavior is similar to ClientPayloadError.

WSGI

WSGI support has been dropped, as well as gunicorn wsgi support. We still provide default and uvloop gunicorn workers for web.Application

What’s new in aiohttp 3.0

async/await everywhere

The main change is dropping yield from support and using async/await everywhere. Farewell, Python 3.4.

The minimal supported Python version is 3.5.3 now.

Why not 3.5.0? Because 3.5.3 has a crucial change: asyncio.get_event_loop() returns the running loop instead of default, which may be different, e.g.:

```python
loop = asyncio.new_event_loop()
loop.run_until_complete(f())
```

Note, asyncio.set_event_loop() was not called and default loop is not equal to actually executed one.

Application Runners

People constantly asked about ability to run aiohttp servers together with other asyncio code, but aiohttp.web.run_app() is blocking synchronous call.

aiohttp had support for starting the application without run_app but the API was very low-level and cumbersome.

Now application runners solve the task in a few lines of code, see Application runners for details.
Client Tracing

Other long awaited feature is tracing client request life cycle to figure out when and why client request spends a time waiting for connection establishment, getting server response headers etc.

Now it is possible by registering special signal handlers on every request processing stage. Client Tracing provides more info about the feature.

HTTPS support

Unfortunately asyncio has a bug with checking SSL certificates for non-ASCII site DNS names, e.g. https://protectbegingroup/immediate/write@unused\def
The bug has been fixed in upcoming Python 3.7 only (the change requires breaking backward compatibility in ssl API).

aiohttp installs a fix for older Python versions (3.5 and 3.6).

**Dropped obsolete API**

A switch to new major version is a great chance for dropping already deprecated features.

The release dropped a lot, see Changelog for details.

All removals was already marked as deprecated or related to very low level implementation details.

If user code did not raise DeprecationWarning it is compatible with aiohttp 3.0 most likely.
Summary

Enjoy aiohttp 3.0 release!
The full change log is here: Changelog.

12.5.2 Glossary

aiodns  DNS resolver for asyncio.
https://pypi.python.org/pypi/aiodns

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.

cchardet  eChardet is high speed universal character encoding detector - binding to charsetdetect.
https://pypi.python.org/pypi/cchardet/

chardet  The Universal Character Encoding Detector
https://pypi.python.org/pypi/chardet/

gunicorn  Gunicorn ‘Green Unicorn’ is a Python WSGI HTTP Server for UNIX.
http://gunicorn.org/

IDNA  An Internationalized Domain Name in Applications (IDNA) is an industry standard for encoding Internet Domain Names that contain in whole or in part, in a language-specific script or alphabet, such as Arabic, Chinese, Cyrillic, Tamil, Hebrew or the Latin alphabet-based characters with diacritics or ligatures, such as French. These writing systems are encoded by computers in multi-byte Unicode. Internationalized domain names are stored in the Domain Name System as ASCII strings using Punycode transcription.

keep-alive  A technique for communicating between HTTP client and server when connection is not closed after sending response but kept open for sending next request through the same socket.
It makes communication faster by getting rid of connection establishment for every request.

nginx  Nginx [engine x] is an HTTP and reverse proxy server, a mail proxy server, and a generic TCP/UDP proxy server.
https://nginx.org/en/

percent-encoding  A mechanism for encoding information in a Uniform Resource Locator (URL) if URL parts don’t fit in safe characters space.

requests  Currently the most popular synchronous library to make HTTP requests in Python.
https://requests.readthedocs.io

requoting  Applying percent-encoding to non-safe symbols and decode percent encoded safe symbols back.

According to RFC 3986 allowed path symbols are:

```
allowed = unreserved / pct-encoded / sub-delims
/ *:* / *@* / */*
```

(continues on next page)
resource  A concept reflects the HTTP path, every resource corresponds to URI.

    May have a unique name.

route  A part of resource, resource’s path coupled with HTTP method.

web-handler  An endpoint that returns HTTP response.

websocket  A protocol providing full-duplex communication channels over a single TCP connection. The WebSocket protocol was standardized by the IETF as RFC 6455

yarl  A library for operating with URL objects.

    https://pypi.python.org/pypi/yarl

12.5.3 Changelog

3.6.2 (2019-10-09)

Features

    • Made exceptions pickleable. Also changed the repr of some exceptions. #4077
    • Use Iterable type hint instead of Sequence for Application middleware parameter. #4125

Bugfixes

    • Reset the sock_read timeout each time data is received for a aiohttp.ClientResponse. #3808
    • Fix handling of expired cookies so they are not stored in CookieJar. #4063
    • Fix misleading message in the string representation of ClientConnectorError; self.ssl == None means default SSL context, not SSL disabled #4097
    • Don’t clobber HTTP status when using FileResponse. #4106

Improved Documentation

    • Added minimal required logging configuration to logging documentation. #2469
    • Update docs to reflect proxy support. #4100
    • Fix typo in code example in testing docs. #4108

12.5. Miscellaneous
3.6.1 (2019-09-19)

Features

- Compatibility with Python 3.8. #4056

Bugfixes

- correct some exception string format #4068
- Emit a warning when ssl.OP_NO_COMPRESSION is unavailable because the runtime is built against an outdated OpenSSL. #4052
- Update multidict requirement to >= 4.5 #4057

Improved Documentation

- Provide pytest-aiohttp namespace for pytest fixtures in docs. #3723

3.6.0 (2019-09-06)

Features

- Add support for Named Pipes (Site and Connector) under Windows. This feature requires Proactor event loop to work. #3629
- Removed Transfer-Encoding: chunked header from websocket responses to be compatible with more http proxy servers. #3798
- Accept non-GET request for starting websocket handshake on server side. #3980

Bugfixes

- Raise a ClientResponseError instead of an AssertionError for a blank HTTP Reason Phrase. #3532
- Fix an issue where cookies would sometimes not be set during a redirect. #3576
- Change normalize_path_middleware to use 308 redirect instead of 301. This behavior should prevent clients from being unable to use PUT/POST methods on endpoints that are redirected because of a trailing slash. #3579
- Drop the processed task from all_tasks() list early. It prevents logging about a task with unhandled exception when the server is used in conjunction with asyncio.run(). #3587
- Signal type annotation changed from `Signal[Callable[['TraceConfig'], Awaitable[None]]]` to `Signal[Callable[ClientSession, SimpleNamespace, ...]]. #3595

- Use sanitized URL as Location header in redirects #3614
- Improve typing annotations for multipart.py along with changes required by mypy in files that references multipart.py. #3621
- Close session created inside `aiohttp.request` when unhandled exception occurs #3628
- Cleanup per-chunk data in generic data read. Memory leak fixed. #3631
- Use correct type for add_view and family #3633
- Fix _keepalive field in __slots__ of RequestHandler. #3644
- Properly handle ConnectionResetError, to silence the “Cannot write to closing transport” exception when clients disconnect uncleanly. #3648
- Suppress pytest warnings due to test_utils classes #3660
- Fix overshadowing of overlapped sub-application prefixes. #3701
- Fixed return type annotation for WSMessage.json() #3720
- Properly expose TooManyRedirects publicly as documented. #3818
- Fix missing brackets for IPv6 in proxy CONNECT request #3841
- Make the signature of `aiohttp.test_utils.TestClient.request` match `asyncio.ClientSession.request` according to the docs #3852
- Use correct style for re-exported imports, makes mypy --strict mode happy. #3868
- Fixed type annotation for add_view method of UrlDispatcher to accept any subclass of View #3880
- Made cython HTTP parser set Reason-Phrase of the response to an empty string if it is missing. #3906
- Add URL to the string representation of ClientResponseError. #3959
- Accept istr keys in LooseHeaders type hints. #3976
- Fixed race conditions in _resolve_host caching and throttling when tracing is enabled. #4013
- For URLs like “unix://localhost/...” set Host HTTP header to “localhost” instead of “localhost:None”. #4039

**Improved Documentation**

- Modify documentation for Background Tasks to remove deprecated usage of event loop. #3526
- use if __name__ == '__main__': in server examples. #3775
- Update documentation reference to the default access logger. #3783
- Improve documentation for `web.BaseRequest.path` and `web.BaseRequest.raw_path`. #3791
- Removed deprecation warning in tracing example docs #3964

12.5. Miscellaneous 199
3.5.4 (2019-01-12)

Bugfixes

- Fix stream .read() / .readany() / .iter_any() which used to return a partial content only in case of compressed content #3525

3.5.3 (2019-01-10)

Bugfixes

- Fix type stubs for aiohttp.web.run_app(access_log=True) and fix edge case of access_log=True and the event loop being in debug mode. #3504
- Fix aiohttp.ClientTimeout type annotations to accept None for fields #3511
- Send custom per-request cookies even if session jar is empty #3515
- Restore Linux binary wheels publishing on PyPI

3.5.2 (2019-01-08)

Features

- FileResponse from web_fileresponse.py uses a ThreadPoolExecutor to work with files asynchronously. I/O based payloads from payload.py uses a ThreadPoolExecutor to work with I/O objects asynchronously. #3313
- Internal Server Errors in plain text if the browser does not support HTML. #3483

Bugfixes

- Preserve MultipartWriter parts headers on write. Refactor the way how Payload.headers are handled. Payload instances now always have headers and Content-Type defined. Fix Payload Content-Disposition header reset after initial creation. #3035
- Log suppressed exceptions in GunicornWebWorker. #3464
- Remove wildcard imports. #3468
- Use the same task for app initialization and web server handling in gunicorn workers. It allows to use Python3.7 context vars smoothly. #3471
- Fix handling of chunked+gzipped response when first chunk does not give uncompressed data #3477
- Replace collections.MutableMapping with collections.abc.MutableMapping to avoid a deprecation warning. #3480
- Payload.size type annotation changed from Optional[float] to Optional[int]. #3484
- Ignore done tasks when cancels pending activities on web.run_app finalization. #3497
Improved Documentation

- Add documentation for `aiohttp.web.HTTPException`. #3490

Misc

- #3487

3.5.1 (2018-12-24)

- Fix a regression about `ClientSession._requote_redirect_url` modification in debug mode.

3.5.0 (2018-12-22)

Features

- The library type annotations are checked in strict mode now.
- Add support for setting cookies for individual request (#2387)
- Application.add_domain implementation (#2809)
- The default `app` in the request returned by `test_utils.make_mocked_request` can now have objects assigned to it and retrieved using the `[ ]` operator. (#3174)
- Make `request.url` accessible when transport is closed. (#3177)
- Add `zlib_executor_size` argument to `Response` constructor to allow compression to run in a background executor to avoid blocking the main thread and potentially triggering health check failures. (#3205)
- Enable users to set `ClientTimeout` in `aiohttp.request` (#3213)
- Don’t raise a warning if `NETRC` environment variable is not set and `~/.netrc` file doesn’t exist. (#3267)
- Add default logging handler to `web.run_app` If the `Application.debug` flag is set and the default logger `aiohttp.access` is used, access logs will now be output using a `stderr` StreamHandler if no handlers are attached. Furthermore, if the default logger has no log level set, the log level will be set to `DEBUG`. (#3324)
- Add method argument to `session.ws_connect()`. Sometimes server API requires a different HTTP method for WebSocket connection establishment. For example, `Docker exec` needs POST. (#3378)
- Create a task per request handling. (#3406)

Bugfixes

- Enable passing `access_log_class` via `handler_args` (#3158)
- Return empty bytes with end-of-chunk marker in empty stream reader. (#3186)
- Accept `CIMultiDictProxy` instances for `headers` argument in `web.Response` constructor. (#3207)
- Don’t uppercase HTTP method in parser (#3233)
- Make method match regexp RFC-7230 compliant (#3235)
- Add `app.pre_frozen` state to properly handle startup signals in sub-applications. (#3237)

12.5. Miscellaneous
• Enhanced parsing and validation of helpers.BasicAuth.decode. (#3239)
• Change imports from collections module in preparation for 3.8. (#3258)
• Ensure Host header is added first to ClientRequest to better replicate browser (#3265)
• Fix forward compatibility with Python 3.8: importing ABCs directly from the collections module will not be supported anymore. (#3273)
• Keep the query string by normalize_path_middleware. (#3278)
• Fix missing parameter raise_for_status for aiohttp.request() (#3290)
• Bracket IPv6 addresses in the HOST header (#3304)
• Fix default message for server ping and pong frames. (#3308)
• Fix tests/testConnector.py typo and tests/autobahn/server.py duplicate loop def. (#3337)
• Fix false-negative indicator end_of_HTTP_chunk in StreamReader.readchunk function (#3361)
• Release HTTP response before raising status exception (#3364)
• Fix task cancellation when sendfile() syscall is used by static file handling. (#3383)
• Fix stack trace for asyncio.TimeoutError which was not logged, when it is caught in the handler. (#3414)

Improved Documentation

• Improve documentation of Application.make_handler parameters. (#3152)
• Fix BaseRequest.raw_headers doc. (#3215)
• Fix typo in TypeError exception reason in web.Application._handle (#3229)
• Make server access log format placeholder %b documentation reflect behavior and docstring. (#3307)

Deprecations and Removals

• Deprecate modification of session.requote_redirect_url (#2278)
• Deprecate stream.unread_data() (#3260)
• Deprecated use of boolean in resp.enable_compression() (#3318)
• Encourage creation of aiohttp public objects inside a coroutine (#3331)
• Drop dead Connection.detach() and Connection.writer. Both methods were broken for more than 2 years. (#3358)
• Deprecate app.loop, request.loop, client.loop and connector.loop properties. (#3374)
• Deprecate explicit debug argument. Use asyncio debug mode instead. (#3381)
• Deprecate body parameter in HTTPException (and derived classes) constructor. (#3385)
• Deprecate bare connector close, use async with connector: and await connector.close() instead. (#3417)
• Deprecate obsolete read_timeout and conn_timeout in ClientSession constructor. (#3438)
Misc

• #3341, #3351

3.4.4 (2018-09-05)

• Fix installation from sources when compiling toolkit is not available (#3241)

3.4.3 (2018-09-04)

• Add `app.pre_frozen` state to properly handle startup signals in sub-applications. (#3237)

3.4.2 (2018-09-01)

• Fix `iter_chunks` type annotation (#3230)

3.4.1 (2018-08-28)

• Fix empty header parsing regression. (#3218)
• Fix `BaseRequest.raw_headers` doc. (#3215)
• Fix documentation building on ReadTheDocs (#3221)

3.4.0 (2018-08-25)

Features

• Add type hints (#3049)
• Add `raise_for_status` request parameter (#3073)
• Add type hints to HTTP client (#3092)
• Minor server optimizations (#3095)
• Preserve the cause when `HTTPException` is raised from another exception. (#3096)
• Add `close_boundary` option in `MultipartWriter.write` method. Support streaming (#3104)
• Added a `remove_slash` option to the `normalize_path.middleware` factory. (#3173)
• The class `AbstractRouteDef` is importable from `aiohttp.web`. (#3183)
Bugfixes

• Prevent double closing when client connection is released before the last `data_received()` callback. (#3031)
• Make redirect with `normalize_path_middleware` work when using url encoded paths. (#3051)
• Postpone web task creation to connection establishment. (#3052)
• Fix `sock_read` timeout. (#3053)
• When using a server-request body as the `data=` argument of a client request, iterate over the content with `readany` instead of `readline` to avoid `Line too long` errors. (#3054)
• fix `UrlDispatcher` has no attribute `add_options`, add `web.options` (#3062)
• correct filename in content-disposition with multipart body (#3064)
• Many HTTP proxies has buggy keepalive support. Let’s not reuse connection but close it after processing every response. (#3070)
• raise 413 “Payload Too Large” rather than raising ValueError in request.post() Add helpful debug message to 413 responses (#3087)
• Fix `StreamResponse` equality, now that they are `MutableMapping` objects. (#3100)
• Fix server request objects comparison (#3116)
• Do not hang on 206 Partial Content response with `Content-Encoding: gzip` (#3123)
• Fix timeout precondition checkers (#3145)

Improved Documentation

• Add a new FAQ entry that clarifies that you should not reuse response objects in middleware functions. (#3020)
• Add FAQ section “Why is creating a ClientSession outside of an event loop dangerous?” (#3072)
• Fix link to Rambler (#3115)
• Fix TCPSite documentation on the Server Reference page. (#3146)
• Fix documentation build configuration file for Windows. (#3147)
• Remove no longer existing lingering_timeouts parameter of Application.make_handler from documentation. (#3151)
• Mention that `app.make_handler` is deprecated, recommend to use runners API instead. (#3157)

Deprecations and Removals

• Drop `loop.current_task()` from `helpers.current_task()` (#2826)
• Drop `reader` parameter from `request.multipart()` (#3090)
3.3.2 (2018-06-12)

- Many HTTP proxies have buggy keepalive support. Let’s not reuse connection but close it after processing every response. (#3070)
- Provide vendor source files in tarball (#3076)

3.3.1 (2018-06-05)

- Fix sock_read timeout. (#3053)
- When using a server-request body as the data= argument of a client request, iterate over the content with readany instead of readline to avoid Line too long errors. (#3054)

3.3.0 (2018-06-01)

Features

- Raise ConnectionResetError instead of CancelledError on trying to write to a closed stream. (#2499)
- Implement ClientTimeout class and support socket read timeout. (#2768)
- Enable logging when aiohttp.web is used as a program (#2956)
- Add canonical property to resources (#2968)
- Forbid reading response BODY after release (#2983)
- Implement base protocol class to avoid a dependency from internal asyncio.streams. FlowControlMixin (#2986)
- Cythonize @helpers.reify. 5% boost on macro benchmark (#2995)
- Optimize HTTP parser (#3015)
- Implement runner.addresses property. (#3036)
- Use bytearray instead of a list of bytes in websocket reader. It improves websocket message reading a little. (#3039)
- Remove heartbeat on closing connection on keepalive timeout. The used hack violates HTTP protocol. (#3041)
- Limit websocket message size on reading to 4 MB by default. (#3045)

Bugfixes

- Don’t reuse a connection with the same URL but different proxy/TLS settings (#2981)
- When parsing the Forwarded header, the optional port number is now preserved. (#3009)
Improved Documentation

• Make Change Log more visible in docs (#3029)
• Make style and grammar improvements on the FAQ page. (#3030)
• Document that signal handlers should be async functions since aiohttp 3.0 (#3032)

Deprecations and Removals

• Deprecate custom application’s router. (#3021)

Misc

• #3008, #3011

3.2.1 (2018-05-10)

• Don’t reuse a connection with the same URL but different proxy/TLS settings (#2981)

3.2.0 (2018-05-06)

Features

• Raise TooManyRedirects exception when client gets redirected too many times instead of returning last response. (#2631)
• Extract route definitions into separate web_routedef.py file (#2876)
• Raise an exception on request body reading after sending response. (#2895)
• ClientResponse and RequestInfo now have real_url property, which is request url without fragment part being stripped (#2925)
• Speed up connector limiting (#2937)
• Added and links property for ClientResponse object (#2948)
• Add request.config_dict for exposing nested applications data. (#2949)
• Speed up HTTP headers serialization, server micro-benchmark runs 5% faster now. (#2957)
• Apply assertions in debug mode only (#2966)

Bugfixes

• expose property app for TestClient (#2891)
• Call on_chunk_sent when write_eof takes as a param the last chunk (#2909)
• A closing bracket was added to __repr__ of resources (#2935)
• Fix compression of FileResponse (#2942)
• Fixes some bugs in the limit connection feature (#2964)
Improved Documentation

- Drop async_timeout usage from documentation for client API in favor of timeout parameter. (#2865)
- Improve Gunicorn logging documentation (#2921)
- Replace multipart writer .serialize() method with .write() in documentation. (#2965)

Deprecations and Removals

- Deprecate Application.make_handler() (#2938)

Misc

- #2958

3.1.3 (2018-04-12)

- Fix cancellation broadcast during DNS resolve (#2910)

3.1.2 (2018-04-05)

- Make LineTooLong exception more detailed about actual data size (#2863)
- Call on_chunk_sent when write_eof takes as a param the last chunk (#2909)

3.1.1 (2018-03-27)

- Support asynchronous iterators (and asynchronous generators as well) in both client and server API as request / response BODY payloads. (#2802)

3.1.0 (2018-03-21)

Welcome to aiohttp 3.1 release.
This is an incremental release, fully backward compatible with aiohttp 3.0.
But we have added several new features.
The most visible one is app.add_routes() (an alias for existing app.router.add_routes()). The addition is very important because all aiohttp docs now uses app.add_routes() call in code snippets. All your existing code still do register routes / resource without any warning but you’ve got the idea for a favorite way: noisy app.router.add_get() is replaced by app.add_routes().
The library does not make a preference between decorators:

```python
routes = web.RouteTableDef()

@routes.get('/'
async def hello(request):
    return web.Response(text="Hello, world")

app.add_routes(routes)
```

12.5. Miscellaneous
and route tables as a list:

```python
def hello(request):
    return web.Response(text="Hello, world")
```

app.add_routes([web.get('/', hello)])

Both ways are equal, user may decide basing on own code taste.
Also we have a lot of minor features, bug fixes and documentation updates, see below.

**Features**

- Relax JSON content-type checking in the `ClientResponse.json()` to allow “application/xxx+json” instead of strict “application/json”. (#2206)
- Bump C HTTP parser to version 2.8 (#2730)
- Accept a coroutine as an application factory in `web.run_app` and gunicorn worker. (#2739)
- Implement application cleanup context (app.cleanup_ctx property). (#2747)
- Make `writer.write_headers` a coroutine. (#2762)
- Add tracking signals for getting request/response bodies. (#2767)
- Deprecate ClientResponseError.code in favor of .status to keep similarity with response classes. (#2781)
- Implement `app.add_routes()` method. (#2787)
- Implement `web.static()` and `RouteTableDef.static()` API. (#2795)
- Install a test event loop as default by asyncio.set_event_loop(). The change affects aiohttp test utils but backward compatibility is not broken for 99.99% of use cases. (#2804)
- Refactor `ClientResponse` constructor: make logically required constructor arguments mandatory, drop `_post_init()` method. (#2820)
- Use `app.add_routes()` in server docs everywhere (#2830)
- Websockets refactoring, all websocket writer methods are converted into coroutines. (#2836)
- Provide `Content-Range` header for `Range` requests (#2844)

**Bugfixes**

- Fix websocket client return EofStream. (#2784)
- Fix websocket demo. (#2789)
- Property `BaseRequest.http_range` now returns a python-like slice when requesting the tail of the range. It’s now indicated by a negative value in `range.start` rather then in `range.stop` (#2805)
- Close a connection if an unexpected exception occurs while sending a request (#2827)
- Fix firing DNS tracing events. (#2841)
Improved Documentation

- Document behavior when cchardet detects encodings that are unknown to Python. (#2732)
- Add diagrams for tracing request life style. (#2748)
- Drop removed functionality for passing StreamReader as data at client side. (#2793)

3.0.9 (2018-03-14)

- Close a connection if an unexpected exception occurs while sending a request (#2827)

3.0.8 (2018-03-12)

- Use asyncio.current_task() on Python 3.7 (#2825)

3.0.7 (2018-03-08)

- Fix SSL proxy support by client. (#2810)
- Restore an imperative check in setup.py for python version. The check works in parallel to environment marker. As effect an error about unsupported Python versions is raised even on outdated systems with very old setuptools version installed. (#2813)

3.0.6 (2018-03-05)

- Add _reuse_address and _reuse_port to web_runner.TCPSite.__slots__. (#2792)

3.0.5 (2018-02-27)

- Fix InvalidStateError on processing a sequence of two RequestHandler.data_received calls on web server. (#2773)

3.0.4 (2018-02-26)

- Fix IndexError in HTTP request handling by server. (#2752)
- Fix MultipartWriter.append* no longer returning part/payload. (#2759)

3.0.3 (2018-02-25)

- Relax attrs dependency to minimal actually supported version 17.0.3 The change allows to avoid version conflicts with currently existing test tools.
3.0.2 (2018-02-23)

Security Fix

- Prevent Windows absolute URLs in static files. Paths like `/static/D:\path` and `/static/\hostname\drive\path` are forbidden.

3.0.1

- Technical release for fixing distribution problems.

3.0.0 (2018-02-12)

Features

- Speed up the `PayloadWriter.write` method for large request bodies. (#2126)
- StreamResponse and Response are now MutableMappings. (#2246)
- ClientSession publishes a set of signals to track the HTTP request execution. (#2313)
- Content-Disposition fast access in ClientResponse (#2455)
- Added support to Flask-style decorators with class-based Views. (#2472)
- Signal handlers (registered callbacks) should be coroutines. (#2480)
- Support async with `test_client.ws_connect(...)`. (#2525)
- Introduce `site` and `application runner` as underlying API for `web.run_app` implementation. (#2530)
- Only quote multipart boundary when necessary and sanitize input (#2544)
- Make the `aiohttp.ClientResponse.get_encoding` method public with the processing of invalid charset while detecting content encoding. (#2549)
- Add optional configurable per message compression for `ClientWebSocketResponse` and `WebSocketResponse`. (#2551)
- Add hysteresis to `StreamReader` to prevent flipping between paused and resumed states too often. (#2555)
- Support `.netrc` by `trust_env` (#2581)
- Avoid to create a new resource when adding a route with the same name and path of the last added resource (#2586)
- `MultipartWriter.boundary` is `str` now. (#2589)
- Allow a custom port to be used by `TestServer` (and associated pytest fixtures) (#2613)
- Add param `access_log_class` to `web.run_app` function (#2615)
- Add `ssl` parameter to client API (#2626)
- Fixes performance issue introduced by #2577. When there are no middlewares installed by the user, no additional and useless code is executed. (#2629)
- Rename PayloadWriter to StreamWriter (#2654)
- New options `reuse_port`, `reuse_address` are added to `run_app` and `TCPSite`. (#2679)
- Use custom classes to pass client signals parameters (#2686)
• Use attrs library for data classes, replace namedtuple. (#2690)
• Pytest fixtures renaming, add aiohttp_ prefix (#2578)
• Add aiohttp- prefix for pytest-aiohttp command line parameters (#2578)

Bugfixes

• Correctly process upgrade request from server to HTTP2. aiohttp does not support HTTP2 yet, the protocol is not upgraded but response is handled correctly. (#2277)
• Fix ClientConnectorSSLError and ClientProxyConnectionError for proxy connector (#2408)
• Fix connector convert OSError to ClientConnectorError (#2423)
• Fix connection attempts for multiple dns hosts (#2424)
• Fix writing to closed transport by raising asyncio.CancelledError (#2499)
• Fix warning in ClientSession.__del__ by stopping to try to close it. (#2523)
• Fixed race-condition for iterating addresses from the DNSCache. (#2620)
• Fix default value of access_log_format argument in web.run_app (#2649)
• Freeze sub-application on adding to parent app (#2656)
• Do percent encoding for .url_for() parameters (#2668)
• Correctly process request start time and multiple request/response headers in access log extra (#2641)

Improved Documentation

• Improve tutorial docs, using literalinclude to link to the actual files. (#2396)
• Small improvement docs: better example for file uploads. (#2401)
• Rename from_env to trust_env in client reference. (#2451)
• Fixed mistype in Proxy Support section where trust_env parameter was used in session.get(“http://python.org”, trust_env=True) method instead of aiohttp.ClientSession constructor as follows: aiohttp.ClientSession(trust_env=True). (#2688)
• Fix issue with unittest example not compiling in testing docs. (#2717)

Deprecations and Removals

• Simplify HTTP pipelining implementation (#2109)
• Drop StreamReaderPayload and DataQueuePayload. (#2257)
• Drop md5 and sha1 finger-prints (#2267)
• Drop WSMessage.tp (#2321)
• Drop Python 3.4 and Python 3.5.0, 3.5.1, 3.5.2. Minimal supported Python versions are 3.5.3 and 3.6.0. yield from is gone, use async/await syntax. (#2343)
• Drop aiohttp.Timeout and use async_timeout.timeout instead. (#2348)
• Drop resolve param from TCPConnector. (#2377)
• Add DeprecationWarning for returning HTTPException (#2415)
• send_str(), send_bytes(), send_json(), ping() and pong() are genuine async functions now. (#2475)
• Drop undocumented app.on_pre_signal and app.on_post_signal. Signal handlers should be coroutines, support for regular functions is dropped. (#2480)
• StreamResponse.drain() is not a part of public API anymore, just use await StreamResponse.write(). StreamResponse.write is converted to async function. (#2483)
• Drop deprecated slow_request_timeout param and **kwargs from RequestHandler. (#2500)
• Drop deprecated resource.url(). (#2501)
• Remove %u and %l format specifiers from access log format. (#2506)
• Drop deprecated request.GET property. (#2547)
• Simplify stream classes: drop ChunksQueue and FlowControlChunksQueue, merge FlowControlStreamReader functionality into StreamReader, drop FlowControlStreamReader name. (#2555)
• Do not create a new resource on router.add_get(…, allow_head=True) (#2585)
• Drop access to TCP tuning options from PayloadWriter and Response classes (#2604)
• Drop deprecated encoding parameter from client API (#2606)
• Deprecate verify_ssl, ssl_context and fingerprint parameters in client API (#2626)
• Get rid of the legacy class StreamWriter. (#2651)
• Forbid non-strings in resource.url_for() parameters. (#2668)
• Deprecate inheritance from ClientSession and web.Application and custom user attributes for ClientSession, web.Request and web.Application (#2691)
• Drop resp = await aiohttp.request(… ) syntax for sake of async with aiohttp.request(… ) as resp:. (#2540)
• Forbid synchronous context managers for ClientSession and test server/client. (#2362)

Misc

• #2552

2.3.10 (2018-02-02)

• Fix 100% CPU usage on HTTP GET and websocket connection just after it (#1955)
• Patch broken ssl.match_hostname() on Python<3.7 (#2674)

2.3.9 (2018-01-16)

• Fix colon handing in path for dynamic resources (#2670)
2.3.8 (2018-01-15)

- Do not use `yarl.unquote` internal function in aiohttp. Fix incorrectly unquoted path part in URL dispatcher (#2662)
- Fix compatibility with `yarl==1.0.0` (#2662)

2.3.7 (2017-12-27)

- Fixed race-condition for iterating addresses from the DNSCache. (#2620)
- Fix docstring for `request.host` (#2591)
- Fix docstring for `request.remote` (#2592)

2.3.6 (2017-12-04)

- Correct `request.app` context (for handlers not just middlewares). (#2577)

2.3.5 (2017-11-30)

- Fix compatibility with `pytest` 3.3+ (#2565)

2.3.4 (2017-11-29)

- Make `request.app` point to proper application instance when using nested applications (with middlewares). (#2550)
- Change base class of `ClientConnectorSSL`Error to `ClientSSL`Error from `ClientConnectorError`. (#2563)
- Return client connection back to free pool on error in `connector.connect()`. (#2567)

2.3.3 (2017-11-17)

- Having a `;` in Response content type does not assume it contains a charset anymore. (#2197)
- Use `getattr(asyncio, 'async')` for keeping compatibility with Python 3.7. (#2476)
- Ignore `NotImplementedError` raised by `set_child_watcher` from `uvloop`. (#2491)
- Fix warning in `ClientSession.__del__` by stopping to try to close it. (#2523)
- Fixed typo’s in Third-party libraries page. And added async-v20 to the list (#2510)

2.3.2 (2017-11-01)

- Fix passing client max size on cloning request obj. (#2385)
- Fix `ClientConnectorSSL`Error and `ClientProxyConnectionError` for proxy connector. (#2408)
- Drop generated `_http_parser` shared object from tarball distribution. (#2414)
- Fix connector convert `OSError` to `ClientConnectorError`. (#2423)
- Fix connection attempts for multiple dns hosts. (#2424)
• Fix ValueError for AF_INET6 sockets if a preexisting INET6 socket to the `aiohttp.web.run_app` function. (#2431)
• `_SessionRequestContextManager` closes the session properly now. (#2441)
• Rename `from_env` to `trust_env` in client reference. (#2451)

2.3.1 (2017-10-18)

• Relax attribute lookup in warning about old-styled middleware (#2340)

2.3.0 (2017-10-18)

Features

• Add SSL related params to `ClientSession.request` (#1128)
• Make `enable_compression` work on HTTP/1.0 (#1828)
• Deprecate registering synchronous web handlers (#1993)
• Switch to `multidict 3.0`. All HTTP headers preserve casing now but compared in case-insensitive way. (#1994)
• Improvement for `normalize_path_middleware`. Added possibility to handle URLs with query string. (#1995)
• Use towncrier for CHANGES.txt build (#1997)
• Implement `trust_env=True` param in `ClientSession`. (#1998)
• Added variable to customize proxy headers (#2001)
• Implement `router.add_routes` and router decorators. (#2004)
• Deprecated `BaseRequest.has_body` in favor of `BaseRequest.can_read_body` Added `BaseRequest.body_exists` attribute that stays static for the lifetime of the request (#2005)
• Provide `BaseRequest.loop` attribute (#2024)
• Make `_CoroGuard` awaitable and fix `ClientSession.close` warning message (#2026)
• Responses to redirects without Location header are returned instead of raising a `RuntimeError` (#2030)
• Added `get_client`, `get_server`, `setUpAsync` and `tearDownAsync` methods to AioHTTPTestCase (#2032)
• Add automatically a SafeChildWatcher to the test loop (#2058)
• Add ability to disable automatic response decompression (#2110)
• Add support for throttling DNS request, avoiding the requests saturation when there is a miss in the DNS cache and many requests getting into the connector at the same time. (#2111)
• Use request for getting access log information instead of message/transport pair. Add `RequestBase.remote` property for accessing to IP of client initiated HTTP request. (#2123)
• `json()` raises a `ContentTypeError` exception if the content-type does not meet the requirements instead of raising a generic `ClientResponseError`. (#2136)
• Make the HTTP client able to return HTTP chunks when chunked transfer encoding is used. (#2150)
• Add `append_version` arg into `StaticResource.url` and `StaticResource.url_for` methods for getting an url with hash (version) of the file. (#2157)
• Fix parsing the Forwarded header. * commas and semicolons are allowed inside quoted-strings; * empty forwarded-pairs (as in for=_1;;by=_2) are allowed; * non-standard parameters are allowed (although this alone could be easily done in the previous parser). (#2173)

• Don’t require ssl module to run. aiohttp does not require SSL to function. The code paths involved with SSL will only be hit upon SSL usage. Raise RuntimeError if HTTPS protocol is required but ssl module is not present. (#2221)

• Accept coroutine fixtures in pytest plugin (#2223)

• Call shutdown_asyncgens before event loop closing on Python 3.6. (#2227)

• Speed up Signals when there are no receivers (#2229)

• Raise InvalidURL instead of ValueError on fetches with invalid URL. (#2241)

• Move DummyCookieJar into cookiejar.py (#2242)

• run_app: Make print=None disable printing (#2260)

• Support brotli encoding (generic-purpose lossless compression algorithm) (#2270)

• Add server support for WebSockets Per-Message Deflate. Add client option to add deflate compress header in WebSockets request header. If calling ClientSession.ws_connect() with compress=15 the client will support deflate compress negotiation. (#2273)

• Support verify_ssl, fingerprint, ssl_context and proxy_headers by client.ws_connect. (#2292)

• Added aiohttp.ClientConnectorSSLError when connection fails due ssl.SSLError (#2294)

• aiohttp.web.Application.make_handler support access_log_class (#2315)

• Build HTTP parser extension in non-strict mode by default. (#2332)

Bugfixes

• Clear auth information on redirecting to other domain (#1699)

• Fix missing app.loop on startup hooks during tests (#2060)

• Fix issue with synchronous session closing when using ClientSession as an asynchronous context manager. (#2063)

• Fix issue with CookieJar incorrectly expiring cookies in some edge cases. (#2084)

• Force use of IPv4 during test, this will make tests run in a Docker container (#2104)

• Warnings about unawaited coroutines now correctly point to the user’s code. (#2106)

• Fix issue with IndexError being raised by the StreamReader.iter_chunks() generator. (#2112)

• Support HTTP 308 Permanent redirect in client class. (#2114)

• Fix FileResponse sending empty chunked body on 304. (#2143)

• Do not add Content-Length: 0 to GET/HEAD/TRACE/OPTIONS requests by default. (#2167)

• Fix parsing the Forwarded header according to RFC 7239. (#2170)

• Securely determining remote/scheme/host #2171 (#2171)

• Fix header name parsing, if name is split into multiple lines (#2183)

• Handle session close during connection, KeyError: <aiohttp.connector._TransportPlaceholder> (#2193)

• Fixes uncaught TypeError in helpers.guess_filename if name is not a string (#2201)
• Raise OSError on async DNS lookup if resolved domain is an alias for another one, which does not have an A or CNAME record. (#2231)
• Fix incorrect warning in StreamReader. (#2251)
• Properly clone state of web request (#2284)
• Fix C HTTP parser for cases when status line is split into different TCP packets. (#2311)
• Fix web.FileResponse overriding user supplied Content-Type (#2317)

Improved Documentation

• Add a note about possible performance degradation in await resp.text() if charset was not provided by Content-Type HTTP header. Pass explicit encoding to solve it. (#1811)
• Drop disqus widget from documentation pages. (#2018)
• Add a graceful shutdown section to the client usage documentation. (#2039)
• Document connector_owner parameter. (#2072)
• Update the doc of web.Application (#2081)
• Fix mistake about access log disabling. (#2085)
• Add example usage of on_startup and on_shutdown signals by creating and disposing an aiopg connection engine. (#2131)
• Document encoded=True for yarl.URL, it disables all yarl transformations. (#2198)
• Document that all app’s middleware factories are run for every request. (#2225)
• Reflect the fact that default resolver is threaded one starting from aiohttp 1.1 (#2228)

Deprecations and Removals

• Drop deprecated Server.finish_connections (#2006)
• Drop %O format from logging, use %b instead. Drop %e format from logging, environment variables are not supported anymore. (#2123)
• Drop deprecated secure_proxy_ssl_header support (#2171)
• Removed TimeService in favor of simple caching. TimeService also had a bug where it lost about 0.5 seconds per second. (#2176)
• Drop unused response_factory from static files API (#2290)

Misc

• #2013, #2014, #2048, #2094, #2149, #2187, #2214, #2225, #2243, #2248
2.2.5 (2017-08-03)

- Don’t raise deprecation warning on `loop.run_until_complete(client.close())` (#2065)

2.2.4 (2017-08-02)

- Fix issue with synchronous session closing when using ClientSession as an asynchronous context manager. (#2063)

2.2.3 (2017-07-04)

- Fix `_CoroGuard` for python 3.4

2.2.2 (2017-07-03)

- Allow `await session.close()` along with `yield from session.close()`

2.2.1 (2017-07-02)

- Relax `yarl` requirement to 0.11+
- Backport #2026: `session.close` is a coroutine (#2029)

2.2.0 (2017-06-20)

- Add doc for `add_head`, update doc for `add_get`. (#1944)
- Fixed consecutive calls for `Response.write_eof`.
- Retain method attributes (e.g. `__doc__`) when registering synchronous handlers for resources. (#1953)
- Added signal TERM handling in `run_app` to gracefully exit (#1932)
- Fix websocket issues caused by frame fragmentation. (#1962)
- Raise `RuntimeError` is you try to set the Content Length and enable chunked encoding at the same time (#1941)
- Small update for `unittest_run_loop`
- Use `CIMultiDict` for `ClientRequest.skip_auto_headers` (#1970)
- Fix wrong startup sequence: test server and `run_app()` are not raise `DeprecationWarning` now (#1947)
- Make sure cleanup signal is sent if startup signal has been sent (#1959)
- Fixed server keep-alive handler, could cause 100% cpu utilization (#1955)
- Connection can be destroyed before response get processed if `await aiohttp.request(..)` is used (#1981)
- `MultipartReader` does not work with `-OO` (#1969)
- Fixed `ClientPayloadError` with blank `Content-Encoding` header (#1931)
- Support `deflate` encoding implemented in `httpbin.org/deflate` (#1918)
- Fix BadStatusLine caused by extra `CRLF` after `POST` data (#1792)
- Keep a reference to `ClientSession` in response object (#1985)

12.5. Miscellaneous
• Deprecate undocumented `app.on_loop_available` signal (#1978)

2.1.0 (2017-05-26)

• Added support for experimental `async-tokio` event loop written in Rust https://github.com/PyO3/tokio
• Write to transport \r\n before closing after keepalive timeout, otherwise client can not detect socket disconnection. (#1883)
• Only call `loop.close` in `run_app` if the user did not supply a loop. Useful for allowing clients to specify their own cleanup before closing the asyncio loop if they wish to tightly control loop behavior
• Content disposition with semicolon in filename (#917)
• Added `request_info` to response object and `ClientResponseError`. (#1733)
• Added `history` to `ClientResponseError`. (#1741)
• Allow to disable redirect url re-quoting (#1474)
• Handle `RuntimeError` from transport (#1790)
• Dropped “%O” in access logger (#1673)
• Added `args` and `kwargs` to `unittest_run_loop`. Useful with other decorators, for example `@patch`. (#1803)
• Added `iter_chunks` to response.content object. (#1805)
• Avoid creating `TimerContext` when there is no timeout to allow compatibility with Tornado. (#1817) (#1180)
• Add `proxy_from_env` to `ClientRequest` to read from environment variables. (#1791)
• Add `DummyCookieJar` helper. (#1830)
• Fix assertion errors in Python 3.4 from `noop` helper. (#1847)
• Do not unquote + in `match_info` values (#1816)
• Use `Forwarded`, `X-Forwarded-Scheme` and `X-Forwarded-Host` for better scheme and host resolution. (#1134)
• Fix sub-application middlewares resolution order (#1853)
• Fix applications comparison (#1866)
• Fix static location in index when prefix is used (#1662)
• Make test server more reliable (#1896)
• Extend list of web exceptions, add `HTTPUnprocessableEntity`, `HTTPFailedDependency`, `HTTPInsufficientStorage` status codes (#1920)

2.0.7 (2017-04-12)

• Fix `pypi` distribution
• Fix exception description (#1807)
• Handle socket error in `FileResponse` (#1773)
• Cancel websocket heartbeat on close (#1793)
2.0.6 (2017-04-04)

- Keeping blank values for request.post() and multipart.form() (#1765)
- TypeError in data_received of ResponseHandler (#1770)
- Fix web.run_app not to bind to default host-port pair if only socket is passed (#1786)

2.0.5 (2017-03-29)

- Memory leak with aiohttp.request (#1756)
  - Disable cleanup closed ssl transports by default.
  - Exception in request handling if the server responds before the body is sent (#1761)

2.0.4 (2017-03-27)

- Memory leak with aiohttp.request (#1756)
- Encoding is always UTF-8 in POST data (#1750)
- Do not add “Content-Disposition” header by default (#1755)

2.0.3 (2017-03-24)

- Call https website through proxy will cause error (#1745)
- Fix exception on multipart/form-data post if content-type is not set (#1743)

2.0.2 (2017-03-21)

- Fixed Application.on_loop_available signal (#1739)
- Remove debug code

2.0.1 (2017-03-21)

- Fix allow-head to include name on route (#1737)
- Fixed AttributeError in WebSocketResponse.can_prepare (#1736)

2.0.0 (2017-03-20)

- Added json to ClientSession.request() method (#1726)
- Added session’s raise_for_status parameter, automatically calls raise_for_status() on any request. (#1724)
- response.json() raises ClientResponseError exception if response’s content type does not match (#1723)
  - Cleanup timer and loop handle on any client exception.
- Deprecate loop parameter for Application’s constructor

12.5. Miscellaneous
2.0.0rc1 (2017-03-15)

- Properly handle payload errors (#1710)
- Added `ClientWebSocketResponse.get_extra_info()` (#1717)
- It is not possible to combine Transfer-Encoding and chunked parameter, same for `compress` and `Content-Encoding` (#1655)
- Connector’s `limit` parameter indicates total concurrent connections. New `limit_per_host` added, indicates total connections per endpoint. (#1601)
- Use url’s `raw_host` for name resolution (#1685)
- Change `ClientResponse.url` to `yarl.URL` instance (#1654)
- Add max_size parameter to web.Request reading methods (#1133)
- Web Request.post() stores data in temp files (#1469)
- Add the `allow_head=True` keyword argument for `add_get` (#1618)
- `run_app` and the Command Line Interface now support serving over Unix domain sockets for faster inter-process communication.
- `run_app` now supports passing a preexisting socket object. This can be useful e.g. for socket-based activated applications, when binding of a socket is done by the parent process.
- Implementation for Trailer headers parser is broken (#1619)
- Fix FileResponse to not fall on bad request (range out of file size)
- Fix FileResponse to correct stream video to Chromes
- Deprecate public low-level api (#1657)
- Deprecate `encoding` parameter for ClientSession.request() method
- Dropped aiohttp.wsgi (#1108)
- Dropped `version` from ClientSession.request() method
- Dropped websocket version 76 support (#1160)
- Dropped: `aiohttp.protocol.HttpPrefixParser` (#1590)
- Dropped: Servers response’s `.started`, `.start()` and `.can_start()` method (#1591)
- Dropped: Adding `sub app` via `app.router.add_subapp()` is deprecated use `app.add_subapp()` instead (#1592)
- Dropped: `Application.finish()` and `Application.register_on_finish()` (#1602)
- Dropped: `web.Request.GET` and `web.Request.POST`
- Dropped: `aiohttp.get()`, `aiohttp.options()`, `aiohttp.head()`, `aiohttp.post()`, `aiohttp.put()`, `aiohttp.patch()`, `aiohttp.delete()`, and `aiohttp.ws_connect()` (#1593)
- Dropped: `aiohttp.web.WebSocketResponse.receive_msg()` (#1605)
- Dropped: `ServerHttpProtocol.keep_alive_timeout` attribute and `keep-alive, keep_alive_on, timeout, log` constructor parameters (#1606)
- Dropped: `TCPConnector’s .resolve`, `.resolved_hosts`, `.clear_resolved_hosts()` attributes and `resolve` constructor parameter (#1607)
- Dropped `ProxyConnector` (#1609)
1.3.5 (2017-03-16)

- Fixed None timeout support (#1720)

1.3.4 (2017-03-14)

- Revert timeout handling in client request
- Fix StreamResponse representation after eof
- Fix file_sender to not fall on bad request (range out of file size)
- Fix file_sender to correct stream video to Chromes
- Fix NotImplemented server exception (#1703)
- Clearer error message for URL without a host name. (#1691)
- Silence deprecation warning in __repr__ (#1690)
- IDN + HTTPS = ssl.CertificateError (#1685)

1.3.3 (2017-02-19)

- Fixed memory leak in time service (#1656)

1.3.2 (2017-02-16)

- awaiting on WebSocketResponse.send_* does not work (#1645)
- Fix multiple calls to client ws_connect when using a shared header dict (#1643)
- Make CookieJar.filter_cookies() accept plain string parameter. (#1636)

1.3.1 (2017-02-09)

- Handle CLOSING in WebSocketResponse.__anext__
- Fixed AttributeError ‘drain’ for server websocket handler (#1613)

1.3.0 (2017-02-08)

- Multipart writer validates the data on append instead of on a request send (#920)
- Multipart reader accepts multipart messages with or without their epilogue to consistently handle valid and legacy behaviors (#1526) (#1581)
- Separate read + connect + request timeouts # 1523
- Do not swallow Upgrade header (#1587)
- Fix polls demo run application (#1487)
- Ignore unknown 1XX status codes in client (#1353)
- Fix sub-Multipart messages missing their headers on serialization (#1525)
- Do not use readline when reading the content of a part in the multipart reader (#1535)

12.5. Miscellaneous
• Add optional flag for quoting *FormData* fields (#916)
• 416 Range Not Satisfiable if requested range end > file size (#1588)
• Having a : or @ in a route does not work (#1552)
• Added `receive_timeout` timeout for websocket to receive complete message. (#1325)
• Added `heartbeat` parameter for websocket to automatically send `ping` message. (#1024) (#777)
• Remove `web.Application` dependency from `web.UrlDispatcher` (#1510)
• Accepting back-pressure from slow websocket clients (#1367)
• Do not pause transport during `set_parser` stage (#1211)
• Lingering close does not terminate before timeout (#1559)
• `setsockopt` may raise `OSError` exception if socket is closed already (#1595)
• Lots of `CancelledError` when requests are interrupted (#1565)
• Allow users to specify what should happen to decoding errors when calling a `responses.text()` method (#1542)
• Back port std module `http.cookies` for python3.4.2 (#1566)
• Maintain url’s fragment in client response (#1314)
• Allow concurrently close WebSockets connection (#754)
• Gzipped responses with empty body raises `ContentEncodingError` (#609)
• Return 504 if request handle raises `TimeoutError`.
• Refactor how we use keep-alive and close lingering timeouts.
• Close response connection if we can not consume whole http message during client response release
• Abort closed ssl client transports, broken servers can keep socket open un-limit time (#1568)
• Log warning instead of `RuntimeError` is websocket connection is closed.
• Deprecated: `aiohttp.protocol.HttpPrefixParser` will be removed in 1.4 (#1590)
• Deprecated: Servers response’s `.started`, `.start()` and `.can_start()` method will be removed in 1.4 (#1591)
• Deprecated: Adding `sub app` via `app.router.add_subapp()` is deprecated use `app.add_subapp()` instead, will be removed in 1.4 (#1592)
• Deprecated: `aiohttp.get()`, `aiohttp.options()`, `aiohttp.head()`, `aiohttp.post()`, `aiohttp.put()`, `aiohttp.patch()`, `aiohttp.delete()`, and `aiohttp.ws_connect()` will be removed in 1.4 (#1593)
• Deprecated: `Application.finish()` and `Application.register_on_finish()` will be removed in 1.4 (#1602)

1.2.0 (2016-12-17)

• Extract `BaseRequest` from `web.Request`, introduce `web.Server` (former `RequestHandlerFactory`), introduce new low-level web server which is not coupled with `web.Application` and routing (#1362)
• Make `TestServer.make_url` compatible with `yarl.URL` (#1389)
• Implement range requests for static files (#1382)
• Support task attribute for StreamResponse (#1410)
• Drop `TestClient.app` property, use `TestClient.server.app` instead (BACKWARD INCOMPATIBLE)
• Drop `TestClient.handler` property, use `TestClient.server.handler` instead (BACKWARD INCOMPATIBLE)
• *TestClient.server* property returns a test server instance, was *asyncio.AbstractServer* (BACKWARD INCOMPATIBLE)

• Follow gunicorn’s signal semantics in *Gunicorn[UVLoop]WebWorker* (#1201)

• Call worker_int and worker_abort callbacks in *Gunicorn[UVLoop]WebWorker* (#1202)

• Has functional tests for client proxy (#1218)

• Fix bugs with client proxy target path and proxy host with port (#1413)

• Fix bugs related to the use of unicode hostnames (#1444)

• Preserve cookie quoting/escaping (#1453)

• FileSender will send gzipped response if gzip version available (#1426)

• Don’t override *Content-Length* header in *web.Response* if no body was set (#1400)

• Introduce *router.post_init()* for solving (#1373)

• Fix raise error in case of multiple calls of *TimeServive.stop()*

• Allow to raise web exceptions on router resolving stage (#1460)

• Add a warning for session creation outside of coroutine (#1468)

• Avoid a race when application might start accepting incoming requests but startup signals are not processed yet e98e8c6

• Raise a *RuntimeError* when trying to change the status of the HTTP response after the headers have been sent (#1480)

• Fix bug with https proxy acquired cleanup (#1340)

• Use UTF-8 as the default encoding for multipart text parts (#1484)

1.1.6 (2016-11-28)

• Fix *BodyPartReader.read_chunk* bug about returns zero bytes before EOF (#1428)

1.1.5 (2016-11-16)

• Fix static file serving in fallback mode (#1401)

1.1.4 (2016-11-14)

• Make *TestServer.make_url* compatible with *yarl.URL* (#1389)

• Generate informative exception on redirects from server which does not provide redirection headers (#1396)
1.1.3 (2016-11-10)

- Support root resources for sub-applications (#1379)

1.1.2 (2016-11-08)

- Allow starting variables with an underscore (#1379)
- Properly process UNIX sockets by gunicorn worker (#1375)
- Fix ordering for FrozenList
- Don’t propagate pre and post signals to sub-application (#1377)

1.1.1 (2016-11-04)

- Fix documentation generation (#1120)

1.1.0 (2016-11-03)

- Drop deprecated WSClientDisconnectedError (BACKWARD INCOMPATIBLE)
- Use yarl.URL in client API. The change is 99% backward compatible but ClientResponse.url is an yarl.URL instance now. (#1217)
- Close idle keep-alive connections on shutdown (#1222)
- Modify regex in AccessLogger to accept underscore and numbers (#1225)
- Use yarl.URL in web server API. web.Request.rel_url and web.Request.url are added. URLs and templates are percent-encoded now. (#1224)
- Accept yarl.URL by server redirections (#1278)
- Return yarl.URL by .make_url() testing utility (#1279)
- Properly format IPv6 addresses by aiohttp.web.run_app (#1139)
- Use yarl.URL by server API (#1288)
  - Introduce resource.url_for(), deprecate resource.url().
  - Implement StaticResource.
  - Inherit SystemRoute from AbstractRoute
  - Drop old-style routes: Route, PlainRoute, DynamicRoute, StaticRoute, ResourceAdapter.
- Revert resp.url back to str, introduce resp.url_obj (#1292)
- Raise ValueError if BasicAuth login has a “:” character (#1307)
- Fix bug when ClientRequest send payload file with opened as open(‘filename’, ‘r+b’) (#1306)
- Enhancement to AccessLogger (pass extra dict) (#1303)
- Show more verbose message on import errors (#1319)
- Added save and load functionality for CookieJar (#1219)
- Added option on StaticRoute to follow symlinks (#1299)
- Force encoding of application/json content type to utf-8 (#1339)
• Fix invalid invocations of `errors.LineTooLong` (#1335)
• Websockets: Stop `async for` iteration when connection is closed (#1144)
• Ensure TestClient HTTP methods return a context manager (#1318)
• Raise `ClientDisconnectedError` to `FlowControlStreamReader` read function if `ClientSession` object is closed by client when reading data. (#1323)
• Document deployment without `Gunicorn` (#1120)
• Add deprecation warning for MD5 and SHA1 digests when used for fingerprint of site certs in TCPConnector. (#1186)
• Implement sub-applications (#1301)
• Don’t inherit `web.Request` from `dict` but implement `MutableMapping` protocol.
• Implement frozen signals
• Don’t inherit `web.Application` from `dict` but implement `MutableMapping` protocol.
• Support freezing for web applications
• Accept access_log parameter in `web.run_app`, use `None` to disable logging
• Don’t flap `tcp_cork` and `tcp_nodelay` in regular request handling. `tcp_nodelay` is still enabled by default.
• Improve performance of web server by removing premature computing of Content-Type if the value was set by `web.Response` constructor.

While the patch boosts speed of trivial `web.Response(text='OK', content_type='text/plain)` very well please don’t expect significant boost of your application – a couple DB requests and business logic is still the main bottleneck.

• Boost performance by adding a custom time service (#1350)
• Extend `ClientResponse` with `content_type` and `charset` properties like in `web.Request`. (#1349)
• Disable aiodns by default (#559)
• Don’t flap `tcp_cork` in client code, use TCP_NODELAY mode by default.
• Implement `web.Request.clone()` (#1361)

1.0.5 (2016-10-11)

• Fix StreamReader._read_nowait to return all available data up to the requested amount (#1297)

1.0.4 (2016-09-22)

• Fix FlowControlStreamReader.read_nowait so that it checks whether the transport is paused (#1206)
1.0.2 (2016-09-22)

- Make CookieJar compatible with 32-bit systems (#1188)
- Add missing WSMsgType to web.ws.__all__, see (#1200)
- Fix CookieJar ctor when called with loop=None (#1203)
- Fix broken upper-casing in wsgi support (#1197)

1.0.1 (2016-09-16)

- Restore aiohttp.web.MsgType alias for aiohttp.WSMsgType for sake of backward compatibility (#1178)
- Tune alabaster schema.
- Use text/html content type for displaying index pages by static file handler.
- Fix AssertionError in static file handling (#1177)
- Fix access log formats %O and %b for static file handling
- Remove debug setting of GunicornWorker, use app.debug to control its debug-mode instead

1.0.0 (2016-09-16)

- Change default size for client session’s connection pool from unlimited to 20 (#977)
- Add IE support for cookie deletion. (#994)
- Remove deprecated WebSocketResponse.wait_closed method (BACKWARD INCOMPATIBLE)
- Remove deprecated force parameter for ClientResponse.close method (BACKWARD INCOMPATIBLE)
- Avoid using of mutable CIMultiDict kw param in make_mocked_request (#997)
- Make WebSocketResponse.close a little bit faster by avoiding new task creating just for timeout measurement
- Add proxy and proxy_auth params to client.get() and family, deprecate ProxyConnector (#998)
- Add support for websocket send_json and receive_json, synchronize server and client API for websockets (#984)
- Implement router shortucts for most useful HTTP methods, use app.router.add_get(), app.router.add_post() etc. instead of app.router.add_route() (#986)
- Support SSL connections for gunicorn worker (#1003)
- Move obsolete examples to legacy folder
- Switch to multidict 2.0 and title-cased strings (#1015)
- {FOO}e logger format is case-sensitive now
- Fix logger report for unix socket 8e8469b
- Rename aiohttp.websocket to aiohttp._ws_impl
- Rename aiohttp.MsgType tp aiohttp.WSMsgType
- Introduce aiohttp.WSMessage officially
- Rename Message -> WSMessage
- Remove deprecated decode param from resp.read(decode=True)
- Use 5min default client timeout (#1028)
• Relax HTTP method validation in UrlDispatcher (#1037)
• Pin minimal supported asyncio version to 3.4.2+ (loop.is_closed() should be present)
• Remove aiohttp.websocket module (BACKWARD INCOMPATIBLE) Please use high-level client and server approaches
• Link header for 451 status code is mandatory
• Fix test_client fixture to allow multiple clients per test (#1072)
• make_mocked_request now accepts dict as headers (#1073)
• Add Python 3.5.2/3.6+ compatibility patch for async generator protocol change (#1082)
• Improvement test_client can accept instance object (#1083)
• Simplify ServerHttpProtocol implementation (#1060)
• Add a flag for optional showing directory index for static file handling (#921)
• Define web.Application.on_startup() signal handler (#1103)
• Drop ChunkedParser and LinesParser (#1111)
• Call Application.startup in GunicornWebWorker (#1105)
• Fix client handling hostnames with 63 bytes when a port is given in the url (#1044)
• Implement proxy support for ClientSession.ws_connect (#1025)
• Return named tuple from WebSocketResponse.can_prepare (#1016)
• Fix access_log_format in GunicornWebWorker (#1117)
• Setup Content-Type to application/octet-stream by default (#1124)
• Deprecate debug parameter from app.make_handler(), use Application(debug=True) instead (#1121)
• Remove fragment string in request path (#846)
• Use aiodns.DNSResolver.gethostbyname() if available (#1136)
• Fix static file sending on uvloop when sendfile is available (#1093)
• Make prettier urls if query is empty dict (#1143)
• Fix redirects for HEAD requests (#1147)
• Default value for StreamReader.read_nowait is -1 from now (#1150)
• aiohttp.StreamReader is not inherited from asyncio.StreamReader from now (BACKWARD INCOMPATIBLE) (#1150)
• Streams documentation added (#1150)
• Add multipart coroutine method for web Request object (#1067)
• Publish ClientSession.loop property (#1149)
• Fix static file with spaces (#1140)
• Fix piling up asyncio loop by cookie expiration callbacks (#1061)
• Drop Timeout class for sake of async_timeout external library. aiohttp.Timeout is an alias for async_timeout.timeout
• use_dns_cache parameter of aiohttp.TCPConnector is True by default (BACKWARD INCOMPATIBLE) (#1152)
• `aiohttp.TCPConnector` uses asynchronous DNS resolver if available by default (BACKWARD INCOMPATIBLE) (#1152)

• Conform to RFC3986 - do not include url fragments in client requests (#1174)

• Drop `ClientSession.cookies` (BACKWARD INCOMPATIBLE) (#1173)

• Refactor `AbstractCookieJar` public API (BACKWARD INCOMPATIBLE) (#1173)

• Fix clashing cookies with have the same name but belong to different domains (BACKWARD INCOMPATIBLE) (#1125)

• Support binary Content-Transfer-Encoding (#1169)

0.22.5 (08-02-2016)

• Pin multidict version to >=1.2.2

0.22.3 (07-26-2016)

• Do not filter cookies if unsafe flag provided (#1005)

0.22.2 (07-23-2016)

• Suppress CancelledError when Timeout raises TimeoutError (#970)

• Don’t expose `aiohttp.__version__`

• Add unsafe parameter to CookieJar (#968)

• Use unsafe cookie jar in test client tools

• Expose aiohttp.CookieJar name

0.22.1 (07-16-2016)

• Large cookie expiration/max-age does not break an event loop from now (fixes (#967))

0.22.0 (07-15-2016)

• Fix bug in serving static directory (#803)

• Fix command line arg parsing (#797)

• Fix a documentation chapter about cookie usage (#790)

• Handle empty body with gzipped encoding (#758)

• Support 451 Unavailable For Legal Reasons http status (#697)

• Fix Cookie share example and few small typos in docs (#817)

• `UrlDispatcher.add_route` with partial coroutine handler (#814)

• Optional support for aiodns (#728)

• Add ServiceRestart and TryAgainLater websocket close codes (#828)

• Fix prompt message for `web.run_app` (#832)
• Allow to pass None as a timeout value to disable timeout logic (#834)
• Fix leak of connection slot during connection error (#835)
• Gunicorn worker with uvloop support `aiohttp_worker.GunicornUVLoopWebWorker` (#878)
• Don’t send body in response to HEAD request (#838)
• Skip the preamble in MultipartReader (#881)
• Implement BasicAuth decode classmethod. (#744)
• Don’t crash logger when transport is None (#889)
• Use a create_future compatibility wrapper instead of creating Futures directly (#896)
• Add test utilities to aiohttp (#902)
• Improve Request.__repr__ (#875)
• Skip DNS resolving if provided host is already an ip address (#874)
• Add headers to ClientSession.ws_connect (#785)
• Document that server can send pre-compressed data (#906)
• Don’t add Content-Encoding and Transfer-Encoding if no body (#891)
• Add json() convenience methods to websocket message objects (#897)
• Add client_resp.raise_for_status() (#908)
• Implement cookie filter (#799)
• Include an example of middleware to handle error pages (#909)
• Fix error handling in StaticFileMixin (#856)
• Add mocked request helper (#900)
• Fix empty ALLOW Response header for cls based View (#929)
• Respect CONNECT method to implement a proxy server (#847)
• Add pytest_plugin (#914)
• Add tutorial
• Add backlog option to support more than 128 (default value in “create_server” function) concurrent connections (#892)
• Allow configuration of header size limits (#912)
• Separate sending file logic from StaticRoute dispatcher (#901)
• Drop deprecated share_cookies connector option (BACKWARD INCOMPATIBLE)
• Drop deprecated support for tuple as auth parameter. Use aiohttp.BasicAuth instead (BACKWARD INCOMPATIBLE)
• Remove deprecated request.payload property, use content instead. (BACKWARD INCOMPATIBLE)
• Drop all mentions about api changes in documentation for versions older than 0.16
• Allow to override default cookie jar (#963)
• Add manylinux wheel builds
• Dup a socket for sendfile usage (#964)
0.21.6 (05-05-2016)

- Drop initial query parameters on redirects (#853)

0.21.5 (03-22-2016)

- Fix command line arg parsing (#797)

0.21.4 (03-12-2016)

- Fix ResourceAdapter: don’t add method to allowed if resource is not match (#826)
- Fix Resource: append found method to returned allowed methods

0.21.2 (02-16-2016)

- Fix a regression: support for handling ~/path in static file routes was broken (#782)

0.21.1 (02-10-2016)

- Make new resources classes public (#767)
- Add `router.resources()` view
- Fix cmd-line parameter names in doc

0.21.0 (02-04-2016)

- Introduce on_shutdown signal (#722)
- Implement raw input headers (#726)
- Implement web.run_app utility function (#734)
- Introduce on_cleanup signal
- Deprecate Application.finish() / Application.register_on_finish() in favor of on_cleanup.
- Get rid of bare aiohttp.request(), aiohttp.get() and family in docs (#729)
- Deprecate bare aiohttp.request(), aiohttp.get() and family (#729)
- Refactor keep-alive support (#737):
  - Enable keepalive for HTTP 1.0 by default
  - Disable it for HTTP 0.9 (who cares about 0.9, BTW?)
  - For keepalived connections
    * Send `Connection: keep-alive` for HTTP 1.0 only
    * don’t send `Connection` header for HTTP 1.1
  - For non-keepalived connections
    * Send `Connection: close` for HTTP 1.1 only
    * don’t send `Connection` header for HTTP 1.0
• Add version parameter to ClientSession constructor, deprecate it for session.request() and family (#736)
• Enable access log by default (#735)
• Deprecate app.router.register_route() (the method was not documented intentionally BTW).
• Deprecate app.router.named_routes() in favor of app.router.named_resources()
• route.add_static accepts pathlib.Path now (#743)
• Add command line support: $ python -m aiohttp.web package.main (#740)
• FAQ section was added to docs. Enjoy and fill free to contribute new topics
• Add async context manager support to ClientSession
• Document ClientResponse’s host, method, url properties
• Use CORK/NODELAY in client API (#748)
• ClientSession.close and Connector.close are coroutines now
• Close client connection on exception in ClientResponse.release()
• Allow to read multipart parts without content-length specified (#750)
• Add support for unix domain sockets to gunicorn worker (#470)
• Add test for default Expect handler (#601)
• Add the first demo project
• Rename loader keyword argument in web.Request.json method. (#646)
• Add local socket binding for TCPConnector (#678)

0.20.2 (01-07-2016)

• Enable use of await for a class based view (#717)
• Check address family to fill wsgi env properly (#718)
• Fix memory leak in headers processing (thanks to Marco Paolini) (#723)

0.20.1 (12-30-2015)

• Raise RuntimeError is Timeout context manager was used outside of task context.
• Add number of bytes to stream.read_nowait (#700)
• Use X-FORWARDED-PROTO for wsgi.url_scheme when available

0.20.0 (12-28-2015)

• Extend list of web exceptions, add HTTPMisdirectedRequest, HTTPUpgradeRequired, HTTPPreconditionRe-
  quired, HTTPTooManyRequests, HTTPRequestHeaderFieldsTooLarge, HTTPVariantAlsoNegotiates, HTTP-
  NotExtended, HTTPNetworkAuthenticationRequired status codes (#644)
• Do not remove AUTHORIZATION header by WSGI handler (#649)
• Fix broken support for https proxies with authentication (#617)
• Get REMOTE_* and SEVER_* http vars from headers when listening on unix socket (#654)
• Add HTTP 308 support (#663)
• Add Tf format (time to serve request in seconds, %06f format) to access log (#669)
• Remove one and a half years long deprecated ClientResponse.read_and_close() method
• Optimize chunked encoding: use a single syscall instead of 3 calls on sending chunked encoded data
• Use TCP_CORK and TCP_NODELAY to optimize network latency and throughput (#680)
• Websocket XOR performance improved (#687)
• Avoid sending cookie attributes in Cookie header (#613)
• Round server timeouts to seconds for grouping pending calls. That leads to less amount of poller syscalls e.g. epoll.poll(). (#702)
• Close connection on websocket handshake error (#703)
• Implement class based views (#684)
• Add headers parameter to ws_connect() (#709)
• Drop unused function parse_remote_addr() (#708)
• Close session on exception (#707)
• Store http code and headers in WSServerHandshakeError (#706)
• Make some low-level message properties readonly (#710)

0.19.0 (11-25-2015)

• Memory leak in ParserBuffer (#579)
• Support 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)

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

0.18.3 (25-10-2015)

• Fix formatting for _RequestContextManager helper (#590)

0.18.2 (22-10-2015)

• Fix regression for OpenSSL < 1.0.0 (#583)

0.18.1 (20-10-2015)

• Relax rule for router names: they may contain dots and columns starting from now

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 RuntimeWarning 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()` does not 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 a year-old deprecated `files` parameter from client API.
• Added async for support for aiohttp stream (#542)

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

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)

0.17.2 (08-11-2015)

• Don’t forget to pass `data` argument forward (#462)
• Fix multipart read bytes count (#463)
0.17.1 (08-10-2015)

- Fix multidict comparison to arbitrary abc.Mapping

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
- Don’t send 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)

12.5. Miscellaneous 235
0.16.6 (07-15-2015)

- Skip compilation on Windows if vcvarsall.bat cannot be found (#438)

0.16.5 (06-13-2015)

- Get rid of all comprehensions and yielding in _multidict (#410)

0.16.4 (06-13-2015)

- Don’t clear current exception in multidict’s __repr__ (cythonized versions) (#410)

0.16.3 (05-30-2015)

- Fix StaticRoute vulnerability to directory traversal attacks (#380)

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)

0.16.1 (05-27-2015)

- Fix regression in static file handling (#377)

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* does not 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()`

0.15.3 (04-22-2015)

• Fix graceful shutdown handling
• Fix *Expect* header handling for not found and not allowed routes (#340)

0.15.2 (04-19-2015)

• Flow control subsystem refactoring
• HTTP server performance 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.

0.15.1 (03-31-2015)

• Pass Autobahn Testsuite tests
• Fixed websocket fragmentation
• Fixed websocket close procedure
• Fixed parser buffer limits
• Added `timeout` parameter to `WebSocketResponse` ctor
• Added `WebSocketResponse.close_code` attribute

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

0.14.4 (01-29-2015)

• Fix issue with error during constructing of url with regex parts (#264)

0.14.3 (01-28-2015)

• Use path='/' by default for cookies (#261)

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)

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 incompatible 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 does not accept **kwargs anymore ((#243)).
• Request is inherited from dict now for making per-request storage to middlewares ((#242)).

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))

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)

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() (#186)
• Added POST attribute
• Response processing refactoring: constructor does not 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
0.11.0 (11-29-2014)

- Support named routes in aiohttp.web.UrlDispatcher (#179)
- Make websocket subprotocols conform to spec (#181)

0.10.2 (11-19-2014)

- Don’t unquote environ['PATH_INFO'] in wsgi.py (#177)

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.

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)

0.9.3 (10-30-2014)

- Fix compatibility with asyncio 3.4.1+ (#170)

0.9.2 (10-16-2014)

- Improve redirect handling (#157)
- Send raw files as is (#153)
- Better websocket support (#150)

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).
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).

0.8.4 (07-04-2014)

- Change ProxyConnector authorization parameters.

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.

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.

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.

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

12.5. Miscellaneous
• Exceptions refactoring

0.7.3 (05-20-2014)
• Simple HTTP proxy support.

0.7.2 (05-14-2014)
• Get rid of __del__ methods
• Use ResourceWarning instead of logging warning record.

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.

0.7.0 (04-16-2014)
• Connection flow control.
• HTTP client session/connection pool refactoring.
• Better handling for bad server requests.

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.

0.6.4 (02-27-2014)
• Log content-length missing warning only for put and post requests.
0.6.3 (02-27-2014)

• Better support for server exit.
• Read response body until EOF if content-length is not defined (#14)

0.6.2 (02-18-2014)

• Fix trailing char in allowed_methods.
• Start slow request timer for first request.

0.6.1 (02-17-2014)

• Added utility method HttpResponse.read_and_close()
• Added slow request timeout.
• Enable socket SO_KEEPALIVE if available.

0.6.0 (02-12-2014)

• Better handling for process exit.

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

0.4.4 (11-15-2013)

• Resolve only AF_INET family, because it is not clear how to pass extra info to asyncio.

0.4.3 (11-15-2013)

• Allow to wait completion of request with HttpResponse.wait_for_close()

0.4.2 (11-14-2013)

• Handle exception in client request stream.
• Prevent host resolving for each client request.
0.4.1 (11-12-2013)

- Added client support for `expect: 100-continue` header.

0.4 (11-06-2013)

- Added custom wsgi application close procedure
- Fixed concurrent host failure in HttpClient

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

0.2

- Fix packaging

12.5.4 Indices and tables

- genindex
- modindex
- search

12.6 Who uses aiohttp?

The list of aiohttp users: both libraries, big projects and web sites.

Please don’t hesitate to add your awesome project to the list by making a Pull Request on GitHub.

If you like the project – please go to GitHub and press Star button!

12.6.1 Third-Party libraries

aiohttp is not the library for making HTTP requests and creating WEB server only.

It is the grand basement for libraries built on top of aiohttp.

This page is a list of these tools.

Please feel free to add your open sourced library if it’s not enlisted yet by making Pull Request to https://github.com/aio-libs/aiohttp/

- Why do you might want to include your awesome library into the list?
- Just because the list increases your library visibility. People will have an easy way to find it.
Officially supported

This list contains libraries which are supported by *aio-libs* team and located on https://github.com/aio-libs

**aiohttp extensions**

- `aiohttp-session` provides sessions for `aiohttp.web`.
- `aiohttp-debugtoolbar` is a library for *debug toolbar* support for `aiohttp.web`.
- `aiohttp-security` auth and permissions for `aiohttp.web`.
- `aiohttp-devtools` provides development tools for `aiohttp.web` applications.
- `aiohttp-cors` CORS support for aiohttp.
- `aiohttp-sse` Server-sent events support for aiohttp.
- `pytest-aiohttp` pytest plugin for aiohttp support.
- `aiohttp-mako` Mako template renderer for aiohttp.web.
- `aiohttp-jinja2` Jinja2 template renderer for aiohttp.web.
- `aiozipkin` distributed tracing instrumentation for `aiohttp` client and server.

**Database drivers**

- `aiopg` PostgreSQL async driver.
- `aiomysql` MySql async driver.
- `aioredis` Redis async driver.

**Other tools**

- `aiodocker` Python Docker API client based on asyncio and aiohttp.
- `aiobotocore` asyncio support for botocore library using aiohttp.

**Approved third-party libraries**

The libraries are not part of *aio-libs* but they are proven to be very well written and highly recommended for usage.

- `uvloop` Ultra fast implementation of asyncio event loop on top of *libuv*.
  
  We are highly recommending to use it instead of standard `asyncio`.
Database drivers

- asyncpg Another PostgreSQL async driver. It’s much faster than aiopg but it is not drop-in replacement – the API is different. Anyway please take a look on it – the driver is really incredible fast.

Others

The list of libraries which are exists but not enlisted in former categories.
They may be perfect or not – we don’t know.
Please add your library reference here first and after some time period ask to raise the status.

- octomachinery A framework for developing GitHub Apps and GitHub Actions. Python 3.7+ is required.
- aiomixcloud Mixcloud API wrapper for Python and Async IO.
- aiohttp-cache A cache system for aiohttp server.
- aiocache Caching for asyncio with multiple backends (framework agnostic)
- gain Web crawling framework based on asyncio for everyone.
- aiohttp-swagger Swagger API Documentation builder for aiohttp server.
- aiohttp-swagger3 Library for Swagger documentation builder and validating aiohttp requests using swagger specification 3.0.
- aiohttp-swaggerify Library to automatically generate swagger2.0 definition for aiohttp endpoints.
- aiohttp-validate Simple library that helps you validate your API endpoints requests/responses with json schema.
- raven-aiohttp An aiohttp transport for raven-python (Sentry client).
- webargs A friendly library for parsing HTTP request arguments, with built-in support for popular web frameworks, including Flask, Django, Bottle, Tornado, Pyramid, webapp2, Falcon, and aiohttp.
- aioauth-client OAuth client for aiohttp.
- aiohttpretty A simple asyncio compatible httpretty mock using aiohttp.
- aioresponses a helper for mock/fake web requests in python aiohttp package.
- aiohttp-transmute A transmute implementation for aiohttp.
- aiohttp_apiset Package to build routes using swagger specification.
- aiohttp-login Registration and authorization (including social) for aiohttp applications.
- aiohttp_utils Handy utilities for building aiohttp.web applications.
- aiohttpproxy Simple aiohttp HTTP proxy.
- aiohttp_traversal Traversal based router for aiohttp.web.
- aiohttp_autoreload Makes aiohttp server auto-reload on source code change.
- gidgethub An async GitHub API library for Python.
- aiohttp_rpc aiohttp JSON-RPC service.
- fbemissary A bot framework for the Facebook Messenger platform, built on asyncio and aiohttp.
- aioslacker slacker wrapper for asyncio.
- aioreloader Port of tornado reloader to asyncio.
- aiohttp_babel Babel localization support for aiohttp.
- python-mock a socket mock framework - for all kinds of socket animals, web-clients included.
- aioraft asyncio RAFT algorithm based on aiohttp.
- home-assistant Open-source home automation platform running on Python 3.
- discord.py Discord client library.
- aiogram A fully asynchronous library for Telegram Bot API written with asyncio and aiohttp.
- vk.py Extremely-fast Python 3.6+ toolkit for create applications work’s with VKAPI.
- aiohttp-graphql GraphQL and GraphIQL interface for aiohttp.
- aiohttp-sentry An aiohttp middleware for reporting errors to Sentry. Python 3.5+ is required.
- aiohttp-datadog An aiohttp middleware for reporting metrics to DataDog. Python 3.5+ is required.
- async-v20 Asynchronous FOREX client for OANDA’s v20 API. Python 3.6+
- aiohttp-jwt An aiohttp middleware for JWT(JSON Web Token) support. Python 3.5+ is required.
- AWS Xray Python SDK Native tracing support for Aiohttp applications.
- GINO An asyncio ORM on top of SQLAlchemy core, delivered with an aiohttp extension.
- aiohttp-apispec Build and document REST APIs with aiohttp and apispec.
- eider-py Python implementation of the Eider RPC protocol.
- asynapplicationinsights A client for Azure Application Insights implemented using aiohttp client, including a middleware for aiohttp servers to collect web apps telemetry.
- DBGR Terminal based tool to test and debug HTTP APIs with aiohttp.

12.6.2 Built with aiohttp

aiohttp is used to build useful libraries built on top of it, and there’s a page dedicated to list them: Third-Party libraries. There are also projects that leverage the power of aiohttp to provide end-user tools, like command lines or software with full user interfaces.

This page aims to list those projects. If you are using aiohttp in your software and if it’s playing a central role, you can add it here in this list.

You can also add a Built with aiohttp link somewhere in your project, pointing to https://github.com/aio-libs/aiohttp.

- Molotov Load testing tool.
- Arsenic Async WebDriver.
- Home Assistant Home Automation Platform.
- Backend.AI Code execution API service.
- doh-proxy DNS Over HTTPS Proxy.
- Mariner Command-line torrent searcher.
- DEEPaaS API REST API for Machine learning, Deep learning and artificial intelligence applications.
12.6.3 Powered by aiohttp

Web sites powered by aiohttp.
Feel free to fork documentation on github, add a link to your site and make a Pull Request!

- Farmer Business Network
- Home Assistant
- KeepSafe
- Skyscanner Hotels
- Ocean S.A.
- GNS3
- TutorCruncher socket
- Morpheus messaging microservice
- Eyepea - Custom telephony solutions
- ALLOcloud - Telephony in the cloud
- helpmanual - comprehensive help and man page database
- bedevere - CPython’s GitHub bot, helps maintain and identify issues with a CPython pull request.
- miss-islington - CPython’s GitHub bot, backports and merge CPython’s pull requests
- noa technologies - Bike-sharing management platform - SSE endpoint, pushes real time updates of bikes location.
- Wargaming: World of Tanks
- Yandex
- Rambler
- Escargot - Chat server
- Prom.ua - Online trading platform
- globo.com - (some parts) Brazilian largest media portal
- Glose - Social reader for E-Books
- Emoji Generator - Text icon generator

12.7 Contributing

12.7.1 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 using the --recurse-submodules argument
2. Setup your machine with the required dev environment
3. Make a change
4. Make sure all tests passed
5. Add a file into CHANGES folder (see Changelog update for how).
6. Commit changes to your own aiohttp clone
7. Make a pull request from the github page of your clone against the master branch
8. Optionally make backport Pull Request(s) for landing a bug fix into released aiohttp versions.

**Note:** The project uses Squash-and-Merge strategy for GitHub Merge button.

Basically it means that there is **no need to rebase** a Pull Request against master branch. Just `git merge master` into your working copy (a fork) if needed. The Pull Request is automatically squashed into the single commit once the PR is accepted.

**Note:** GitHub issue and pull request threads are automatically locked when there has not been any recent activity for one year. Please open a new issue for related bugs.

If you feel like there are important points in the locked discussions, please include those excerpts into that new issue.

### 12.7.2 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:

```bash
$ cd aiohttp
$ virtualenv --python=`which python3` venv
$ . venv/bin/activate
```

For standard python `venv`:

```bash
$ cd aiohttp
$ python3 -m venv venv
$ . venv/bin/activate
```

For `virtualenvwrapper`:

```bash
$ 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:

```bash
$ make install-dev
```
Note: For now, the development tooling depends on make and assumes an Unix OS. If you wish to contribute to aiohttp from a Windows machine, the easiest way is probably to configure the WSL so you can use the same instructions. If it’s not possible for you or if it doesn’t work, please contact us so we can find a solution together.

Warning: If you plan to use temporary print(), pdb or ipdb within the test suite, execute it with `-s`:

```bash
$ py.test tests -s
```

in order to run the tests without output capturing.

Congratulations, you are ready to run the test suite!

**12.7.3 Run aiohttp test suite**

After all the preconditions are met you can run tests typing the next command:

```bash
$ 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.

**12.7.4 Tests coverage**

We are trying hard to have good test coverage; please don’t make it worse.

Use:

```bash
$ 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/htmlcov/index.html`

Please go to the link and make sure that your code change is covered.

The project uses codecov.io for storing coverage results. Visit https://codecov.io/gh/aio-libs/aiohttp for looking on coverage of master branch, history, pull requests etc.

The browser extension https://docs.codecov.io/docs/browser-extension is highly recommended for analyzing the coverage just in Files Changed tab on GitHub Pull Request review page.
12.7.5 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.

12.7.6 Spell checking

We use pyenchant and sphinxcontrib-spelling for running spell checker for documentation:

```
$ make doc-spelling
```

Unfortunately there are problems with running spell checker on MacOS X.
To run spell checker on Linux box you should install it first:

```
$ sudo apt-get install enchant
$ pip install sphinxcontrib-spelling
```

12.7.7 Changelog update

The CHANGES.rst file is managed using towncrier tool and all non trivial changes must be accompanied by a news entry.

To add an entry to the news file, first you need to have created an issue describing the change you want to make. A Pull Request itself may function as such, but it is preferred to have a dedicated issue (for example, in case the PR ends up rejected due to code quality reasons).

Once you have an issue or pull request, you take the number and you create a file inside of the CHANGES/ directory named after that issue number with an extension of .removal,.feature,.bugfix,.or,.doc. Thus if your issue or PR number is 1234 and this change is fixing a bug, then you would create a file CHANGES/1234.bugfix. PRs can span multiple categories by creating multiple files (for instance, if you added a feature and deprecated/removed the old feature at the same time, you would create CHANGES/NNNN.feature and CHANGES/NNNN.removal). Likewise if a PR touches multiple issues/PRs you may create a file for each of them with the exact same contents and Towncrier will deduplicate them.

The contents of this file are reStructuredText formatted text that will be used as the content of the news file entry. You do not need to reference the issue or PR numbers here as towncrier will automatically add a reference to all of the affected issues when rendering the news file.
12.7.8 Making a Pull Request

After finishing all steps make a GitHub Pull Request with master base branch.

12.7.9 Backporting

All Pull Requests are created against master git branch.

If the Pull Request is not a new functionality but bug fixing backport to maintenance branch would be desirable.

*aiohttp* project committer may ask for making a backport of the PR into maintained branch(es), in this case he or she adds a github label like needs backport to 3.1.

Backporting is performed after main PR merging into master. Please do the following steps:

1. Find Pull Request's commit for cherry-picking.

   *aiohttp* does squashing PRs on merging, so open your PR page on github and scroll down to message like asvetlov merged commit f7b8921 into master 9 days ago. f7b8921 is the required commit number.

2. Run cherry_picker tool for making backport PR (the tool is already pre-installed from ./requirements/dev.txt), e.g. cherry_picker f7b8921 3.1.

3. In case of conflicts fix them and continue cherry-picking by cherry_picker --continue.

   cherry_picker --abort stops the process.

   cherry_picker --status shows current cherry-picking status (like git status)

4. After all conflicts are done the tool opens a New Pull Request page in a browser with pre-filed information.

   Create a backport Pull Request and wait for review/merging.

5. *aiohttp committer* should remove backport Git label after merging the backport.

12.7.10 How to become an aiohttp committer

Contribute!

The easiest way is providing Pull Requests for issues in our bug tracker. But if you have a great idea for the library improvement – please make an issue and Pull Request.

The rules for committers are simple:

1. No wild commits! Everything should go through PRs.

2. Take a part in reviews. It’s very important part of maintainer’s activity.

3. Pickup issues created by others, especially if they are simple.

4. Keep test suite comprehensive. In practice it means leveling up coverage. 97% is not bad but we wish to have 100% someday. Well, 99% is good target too.

5. Don’t hesitate to improve our docs. Documentation is very important thing, it’s the key for project success. The documentation should not only cover our public API but help newbies to start using the project and shed a light on non-obvious gotchas.

After positive answer aiohttp committer creates an issue on github with the proposal for nomination. If the proposal will collect only positive votes and no strong objection – you’ll be a new member in our team.
a
aiohttp.abc, 165
aiohttp.test_utils, 148
aiohttp.web, 78
Symbols

_create_connection() (aiohttp.BaseConnector method), 52

A

AbstractResource (class in aiohttp.web), 126
AbstractRoute (class in aiohttp.web), 129
AbstractRouteDef (class in aiohttp.web), 130
AbstractView (class in aiohttp.abc), 166
add_delete() (aiohttp.web.UrlDispatcher method), 124
add_domain() (aiohttp.web.Application method), 121
add_field() (aiohttp.FormData method), 63
add_fields() (aiohttp.FormData method), 64
add_get() (aiohttp.web.UrlDispatcher method), 124
add_head() (aiohttp.web.UrlDispatcher method), 124
add_patch() (aiohttp.web.UrlDispatcher method), 124
add_post() (aiohttp.web.UrlDispatcher method), 124
add_resource() (aiohttp.web.UrlDispatcher method), 123
add_route() (aiohttp.web.Application method), 127
add_resource() (aiohttp.web.UrlDispatcher method), 124
add_routes() (aiohttp.web.Application method), 122
add_routes() (aiohttp.web.UrlDispatcher method), 123
add_static() (aiohttp.web.UrlDispatcher method), 124
add_subapp() (aiohttp.web.Application method), 121
add_view() (aiohttp.web.UrlDispatcher method), 124
addresses (aiohttp.web.BaseRunner attribute), 134
aiodns, 196
aiohttp.abc (module), 165
aiohttp.abc(AbstractAccessLogger (class in aiohttp.abc), 167
aiohttp.abc(AbstractCookieJar (class in aiohttp.abc), 166
aiohttp.abc(AbstractMatchInfo (class in aiohttp.abc), 165
aiohttp.abc(AbstractRouter (class in aiohttp.abc), 165
aiohttp.test_utils (module), 148
aiohttp.web (module), 78
aiohttp_client (in module pytest_aiohttp), 150
aiohttp_raw_server (in module pytest_aiohttp), 150
aiohttp_server (in module pytest_aiohttp), 150
aiohttp_unused_port (in module pytest_aiohttp), 151
AioHTTPTestCase (class in aiohttp.test_utils), 152
allowed_methods (aiohttp.web.HTTPMethodNotAllowed attribute), 143
app (aiohttp.test_utils.AioHTTPTestCase attribute), 152
app (aiohttp.test_utils.TestClient attribute), 157
app (aiohttp.test_utils.TestServer attribute), 156
app (aiohttp.web.AppRunner attribute), 135
app (aiohttp.web.Request attribute), 109
append() (aiohttp.MultipartWriter method), 174
append_form() (aiohttp.MultipartWriter method), 175
append_json() (aiohttp.MultipartWriter method), 174
append_payload() (aiohttp.MultipartWriter method), 174
Application (class in aiohttp.web), 120
AppRunner (class in aiohttp.web), 134
asyncio, 196
at_eof() (aiohttp.BodyPartReader method), 173
at_eof() (aiohttp.MultipartReader method), 174
at_eof() (aiohttp.MultipartResponseWrapper method), 173
at_eof() (aiohttp.StreamReader method), 177
auth (in module aiohttp), 44
auto_decompress (in module aiohttp), 44

B

BaseConnector (class in aiohttp), 51
BaseRequest (class in aiohttp.web), 104
BaseRunner (class in aiohttp.web), 134
BaseSite (class in aiohttp.web), 136
BaseTestServer (class in aiohttp.test_utils), 155
BasicAuth (class in aiohttp), 61
BINARY (aiohttp.WSMsgType attribute), 179
body (aiohttp.web.Response attribute), 114
body_exists (aiohttp.web.BaseRequest attribute), 106
BodyPartReader (class in aiohttp), 173
boundary (aiohttp.MultiPartWriter attribute), 174
C
cached_hosts (aiohttp.TCPConnector attribute), 53
callable, 196
can_prepare() (aiohttp.web.WebSocketResponse method), 116
can_read_body (aiohttp.web.BaseRequest attribute), 106
canonical (aiohttp.web.AbstractResource attribute), 127
canonical (aiohttp.web.DynamicResource attribute), 128
canonical (aiohttp.web.PlainResource attribute), 127
canonical (aiohttp.web.PrefixedSubAppResource attribute), 128
canonical (aiohttp.web.StaticResource attribute), 128
cchardet, 196
ChainMapProxy (class in aiohttp), 178
chardet, 196
charset (aiohttp.ClientResponse attribute), 56
charset (aiohttp.web.BaseRequest attribute), 107
chunk (aiohttp.TraceRequestChunkSentParams attribute), 73
chunk (aiohttp.TraceResponseChunkSentParams attribute), 73
chunked (aiohttp.web.StreamResponse attribute), 111
cleanup() (aiohttp.web.Application method), 122
cleanup() (aiohttp.web.AppRunner method), 135
cleanup() (aiohttp.web.BaseRunner method), 134
cleanup_ctx (aiohttp.web.Application attribute), 121
clear_dns_cache () (aiohttp.TCPConnector method), 53
client (aiohttp.test_utils.AioHTTPTestCase attribute), 152
ClientConnectionError (class in aiohttp), 66
ClientConnectorCertificateError (class in aiohttp), 66
ClientConnectorError (class in aiohttp), 66
ClientConnectorSSLLError (class in aiohttp), 66
ClientError, 64
ClientOSSError (class in aiohttp), 66
ClientPayloadError (class in aiohttp), 64
ClientProxyConnectionError (class in aiohttp), 66
ClientResponse (class in aiohttp), 55
ClientResponseError, 65
ClientSSLError (class in aiohttp), 66
ClientTimeout (class in aiohttp), 60
ClientWebSocketResponse (class in aiohttp), 58
ClientWSTimeout (class in aiohttp), 60
close() (aiohttp.web.BaseRequest method), 107
CLOSE (aiohttp.WSMsgType attribute), 179
close() (aiohttp.BaseConnector method), 52
close() (aiohttp.ClientResponse method), 56
close() (aiohttp.ClientWebSocketResponse method), 59
close() (aiohttp.Connection method), 54
close() (aiohttp.test_utils.BaseTestServer method), 156
close() (aiohttp.test_utils.TestClient method), 157
close() (aiohttp.web.WebSocketResponse method), 117
close() (in module aiohttp), 49
close_code (aiohttp.web.WebSocketResponse attribute), 116
closed (aiohttp.BaseConnector attribute), 51
closed (aiohttp.ClientWebSocketResponse attribute), 58
closed (aiohttp.Connection attribute), 54
closed (aiohttp.web.WebSocketResponse attribute), 116
closed (in module aiohttp), 43
code (aiohttp.ClientResponseError attribute), 65
compression (aiohttp.web.StreamResponse attribute), 111
cnfign_dict (aiohttp.web.Request attribute), 110
connect (aiohttp.ClientTimeout attribute), 60
connect() (aiohttp.BaseConnector method), 52
connection (aiohttp.ClientResponse attribute), 55
Connection (class in aiohttp), 54
connections (aiohttp.web.Server attribute), 122
connector (in module aiohttp), 43
connector_owner (in module aiohttp), 44
content (aiohttp.ClientResponse attribute), 55
content (aiohttp.web.BaseRequest attribute), 106
content_disposition (aiohttp.ClientResponse attribute), 56
content_length (aiohttp.web.BaseRequest attribute), 107
content_length (aiohttp.web.StreamResponse attribute), 113
content_type (aiohttp.ClientResponse attribute), 55
content_type (aiohttp.web.BaseRequest attribute), 107
content_type (aiohttp.web.FileField attribute), 137
content_type (aiohttp.web.StreamResponse attribute), 113
ContentCoding (class in aiohttp.web), 139
ContentDisposition (class in aiohttp), 65
ContentTypeError (class in aiohttp), 65
handler (aiohttp.web.RouteDef attribute), 130
handler (aiohttp.web.UrlMappingMatchInfo attribute), 133
handler () (aiohttp.abc.aiohttp.abc.AbstractMatchInfo method), 165
head () (aiohttp.test_utils.TestClient method), 157
head() (aiohttp.web.RouteTableDef method), 132
head() (in module aiohttp), 48
head() (in module aiohttp.web), 131
headers (aiohttp.ClientResponse attribute), 55
headers (aiohttp.ClientResponseError attribute), 65
headers (aiohttp.RequestInfo attribute), 61
headers (aiohttp.TraceRequestEndParams attribute), 73
headers (aiohttp.TraceRequestExceptionParams attribute), 73
headers (aiohttp.TraceRequestRedirectParams attribute), 74
headers (aiohttp.TraceRequestStartParams attribute), 72
headers (aiohttp.web.BaseRequest attribute), 106
headers (aiohttp.web.HTTPException attribute), 142
headers (in module aiohttp), 44
history (aiohttp.ClientResponse attribute), 56
history (aiohttp.ClientResponseError attribute), 65
host (aiohttp.test_utils.BaseTestServer attribute), 155
host (aiohttp.test_utils.TestClient attribute), 157
Host (aiohttp.TraceDnsCacheHitParams attribute), 75
Host (aiohttp.TraceDnsCacheMissParams attribute), 75
Host (aiohttp.TraceDnsResolveHostEndParams attribute), 75
Host (aiohttp.TraceDnsResolveHostStartParams attribute), 75
host (aiohttp.web.BaseRequest attribute), 105
http_exception (aiohttp.abc.aiohttp.abc.AbstractMatchInfo attribute), 165
http_range (aiohttp.web.BaseRequest attribute), 107
HTTPAccepted, 142
HTTPBadGateway, 145
HTTPBadRequest, 143
HTTPClientError, 143
HTTPConflict, 144
HTTPCreated, 142
HTTPExpection, 141
HTTPExpectationFailed, 144
HTTPFailedDependency, 144
HTTPForbidden, 143
HTTPFound, 143
HTTPGatewayTimeout, 145
HTTPGone, 144
HTTPInsufficientStorage, 145
HTTPInternalServerError, 145
HTTPLengthRequired, 144
HTTPMethodNotAllowed, 143
HTTPMissedRedirectRequest, 144
HTTPMove, 142
HTTPMovedPermanently, 143
HTTPMultipleChoices, 142
HTTPNetworkAuthenticationRequired, 145
HTTPNoContent, 142
HTTPNonAuthoritativeInformation, 142
HTTPNotAcceptable, 144
HTTPNotExtended, 145
HTTPNotFound, 143
HTTPNotImplemented, 145
HTTPNotModified, 143
HTTPOK, 142
HTTPPartialContent, 142
HTTPPaymentRequired, 143
HTTPPermanentRedirect, 143
HTTPPreconditionFailed, 144
HTTPPreconditionRequired, 144
HTTPProxyAuthenticationRequired, 144
HTTPRedirection, 142
HTTPRequestEntityTooLarge, 144
HTTPRequestHeaderFieldsTooLarge, 144
HTTPRequestRangeNotSatisfiable, 144
HTTPRequestTimeout, 144
HTTPRequestURITooLong, 144
HTTPResetContent, 142
HTTPSeeOther, 143
HTTPServerError, 145
HTTPServiceUnavailable, 145
HTTPSuccessful, 142
HTTPTemporaryRedirect, 143
HTTPTooManyRequests, 144
HTTPUnauthorized, 143
HTTPUnprocessableEntity, 144
HTTPUnsupportedMediaType, 144
HTTPUpgradeRequired, 144
HTTPUseProxy, 143
HTTPVariantAlsoNegotiates, 145
HTTPVersionNotSupported, 145

identity (aiohttp.web.ContentCoding attribute), 139
IDNA, 196
if_modified_since (aiohttp.web.BaseRequest attribute), 107
if_range (aiohttp.web.BaseRequest attribute), 107
if_unmodified_since (aiohttp.web.BaseRequest attribute), 107
INTERNAL_ERROR (aiohttp.WSCloseCode attribute), 179
INVALID_TEXT (aiohttp.WSCloseCode attribute), 178
on_dns_cache_hit (aiohttp.StatusCode attribute), 72
on_dns_cache_miss (aiohttp.StatusCode attribute), 72
on_dns_resolvehost_end (aiohttp.StatusCode attribute), 72
on_dns_resolvehost_start (aiohttp.StatusCode attribute), 72
on_request_chunk_sent (aiohttp.StatusCode attribute), 71
on_request_end (aiohttp.StatusCode attribute), 71
on_request_exception (aiohttp.StatusCode attribute), 71
on_request_redirect (aiohttp.StatusCode attribute), 71
on_request_start (aiohttp.StatusCode attribute), 71
on_response_chunk_received (aiohttp.StatusCode attribute), 71
on_response_prepare (aiohttp.web.Application attribute), 120
on_shutdown (aiohttp.web.Application attribute), 120
on_startup (aiohttp.web.Application attribute), 120
options() (aiohttp.test_utils.TestClient method), 157
options() (in module aiohttp), 48

parameters (aiohttp.ContentDisposition attribute), 65
patch() (aiohttp.test_utils.TestClient method), 157
patch() (aiohttp.web.RouteTableDef method), 132
patch() (in module aiohttp), 48
patch() (in module aiohttp.web), 131
path (aiohttp.UnixConnector attribute), 54
path (aiohttp.web.BaseRequest method), 105
path (aiohttp.web.RouteDef method), 130
path (aiohttp.web.StaticDef method), 130
path_qs (aiohttp.web.BaseRequest attribute), 105
percent_encoding (in module aiohttp), 196
PING (aiohttp.WSMsgType attribute), 179
ping() (aiohttp.ClientWebSocketResponse method), 58
ping() (aiohttp.web.WebSocketResponse method), 116
PlainResource (class in aiohttp.web), 127
POLICY_VIOLATION (aiohttp.WSCloseCode attribute), 179
PONG (aiohttp.WSMsgType attribute), 179
pong() (aiohttp.ClientWebSocketResponse method), 58
pong() (aiohttp.web.WebSocketResponse method), 116
port (aiohttp.test_utils.BaseTestServer attribute), 156
port (aiohttp.test_utils.TestClient attribute), 157
post() (aiohttp.test_utils.TestClient method), 157
post() (aiohttp.web.BaseRequest method), 109
post() (aiohttp.web.RouteTableDef method), 132
post() (in module aiohttp), 47
post() (in module aiohttp.web), 131
prefix (aiohttp.web.StaticDef attribute), 130
PrefixedSubAppResource (class in aiohttp.web), 128
prepare() (aiohttp.web.StreamResponse method), 113
prepare() (aiohttp.web.WebSocketResponse method), 116
prepared (aiohttp.web.StreamResponse attribute), 111
protocol (aiohttp.ClientWebSocketResponse attribute), 58
protocol (aiohttp.web.WebSocketReady attribute), 119
PROTOCOL_ERROR (aiohttp.WSCloseCode attribute), 178
put() (aiohttp.test_utils.TestClient method), 157
put() (aiohttp.web.RouteTableDef method), 132
put() (in module aiohttp), 47
put() (in module aiohttp.web), 131
Python Enhancement Proposals
PEP 3156, 196
Q
query (aiohttp.web.BaseRequest attribute), 106
query_string (aiohttp.web.BaseRequest attribute), 106

R
raise_for_status (in module aiohttp), 44
raise_for_status() (aiohttp.ClientResponse method), 56
raw_headers (aiohttp.ClientResponse attribute), 55
raw_headers (aiohttp.web.BaseRequest attribute), 106
raw_path (aiohttp.web.BaseRequest attribute), 106
RawTestServer (class in aiohttp.test_utils), 156
read() (aiohttp.BodyPartReader method), 173
read() (aiohttp.ClientResponse method), 56
read() (aiohttp.StreamReader method), 175
read() (aiohttp.web.BaseRequest method), 108
read_chunk() (aiohttp.BodyPartReader method), 173
read_nowait() (aiohttp.StreamReader method), 177
readany() (aiohttp.StreamReader method), 175
readexactly() (aiohttp.StreamReader method), 175
readline() (aiohttp.BodyPartReader method), 173
readline() (aiohttp.StreamReader method), 175
real_url (aiohttp.ClientResponse attribute), 55
real_url (aiohttp.RequestInfo attribute), 61
reason (aiohttp.ClientResponse attribute), 55
reason (aiohttp.web.HTTPException attribute), 141
reason (aiohttp.web.StreamResponse attribute), 111
reason (aiohttp.web.SystemRoute attribute), 129
receive() (aiohttp.ClientWebSocketResponse method), 59
receive() (aiohttp.web.WebSocketResponse method), 118
RFC 2068, 140
RFC 2109, 34
RFC 2616, 56, 107
RFC 3629, 179
RFC 3986, 196
RFC 6455, 197
RFC 7230, 33
RFC 7239, 105
RFC 7239#section-4, 105
RFC 7239#section-6, 105
route, 197
route (aiohttp.web.UrlMappingMatchInfo attribute), 133
route() (aiohttp.web.RouteTableDef method), 133
route() (in module aiohttp.web), 131
RouteDef (class in aiohttp.web), 130
router (aiohttp.web.Application attribute), 120
routes() (aiohttp.web.UrlDispatcher method), 125
RouteTableDef (class in aiohttp.web), 132
run_app() (in module aiohttp.web), 137

S
save() (aiohttp.CookieJar method), 62
scheme (aiohttp.test_utils.BaseTestServer attribute), 155
scheme (aiohttp.test_utils.TestClient attribute), 157
scheme (aiohttp.web.BaseRequest attribute), 104
secure (aiohttp.web.BaseRequest attribute), 104
send() (aiohttp.Signal method), 177
send_bytes() (aiohttp.ClientWebSocketResponse method), 58
send_bytes() (aiohttp.web.WebSocketResponse method), 117
send_json() (aiohttp.ClientWebSocketResponse method), 58
send_json() (aiohttp.web.WebSocketResponse method), 117
send_str() (aiohttp.ClientWebSocketResponse method), 58
send_str() (aiohttp.web.WebSocketResponse method), 117
server (aiohttp.test_utils.AioHTTPTestCase attribute), 152
server (aiohttp.test_utils.BaseTestServer attribute), 156
server (aiohttp.test_utils.TestClient attribute), 157
server (aiohttp.web.BaseRunner attribute), 134
Server (class in aiohttp.web), 122
ServerConnectionError (class in aiohttp), 66
ServerDisconnectedError (class in aiohttp), 66
ServerFingerprintMismatch (class in aiohttp), 66
ServerRunner (class in aiohttp), 135
ServerTimeoutError (class in aiohttp), 66
SERVICE_RESTART (aiohttp.WSCloseCode attribute), 179
session (aiohttp.test_utils.TestClient attribute), 157
set_cookie() (aiohttp.web.StreamResponse method), 112
set_status() (aiohttp.web.StreamResponse method), 111
setUp() (aiohttp.test_utils.AioHTTPTestCase method), 153
setUp() (aiohttp.web.AppRunner method), 135
setUp() (aiohttp.web.BaseRunner method), 134
setUp_test_loop() (in module aiohttp.test_utils), 158
setUpAsync() (aiohttp.test_utils.AioHTTPTestCase method), 152
shutdown() (aiohttp.web.Application method), 122
shutdown() (aiohttp.web.Server method), 122
Signal (class in aiohttp), 177
sites (aiohttp.web.BaseRunner attribute), 134
size (aiohttp.MultiPartWriter attribute), 175
skip_auto_headers (in module aiohttp), 44
sock_connect (aiohttp.ClientTimeout attribute), 60
sock_read (aiohttp.ClientTimeout attribute), 60
SockSite (in module aiohttp), 136
start() (aiohttp.web.BaseSite method), 136
start_server() (aiohttp.test_utils.BaseTestServer method), 156
start_server() (aiohttp.test_utils.TestClient method), 157
startup() (aiohttp.web.Application method), 122
static() (aiohttp.web.RouteTableDef method), 133
static() (in module aiohttp.web), 131
StaticDef (class in aiohttp.web), 130
StaticResource (class in aiohttp.web), 128
status (aiohttp.ClientResponse attribute), 55
status (aiohttp.ClientResponseError attribute), 65
status (aiohttp.web.HTTPException attribute), 141
status (aiohttp.web.StreamResponse attribute), 111
status (aiohttp.web.SystemRoute attribute), 129
stop() (aiohttp.web.BaseSite method), 136
StreamReader (class in aiohttp), 175
StreamResponse (class in aiohttp.web), 110
SystemRoute (class in aiohttp.web), 129

T

Task (aiohttp.web.StreamResponse attribute), 111
TCPConnector (class in aiohttp), 52
TCPSite (class in aiohttp.web), 136
tearDown() (aiohttp.test_utils.AioHTTPTestCase method), 153
tearDown_test_loop() (in module aiohttp.test_utils), 158
tearDownAsync() (aiohttp.test_utils.AioHTTPTestCase method), 152
TestClient (class in aiohttp.test_utils), 156
TestServer (class in aiohttp.test_utils), 156
text (aiohttp.web.HTTPException attribute), 142
text (aiohttp.web.Response attribute), 114
TEXT (aiohttp.WSMsgType attribute), 179
text() (aiohttp.BodyPartReader method), 173
text() (aiohttp.ClientResponse method), 56
text() (aiohttp.web.BaseRequest method), 108
timeout (in module aiohttp), 44
TooManyRedirects (class in aiohttp), 65
total (aiohttp.ClientTimeout attribute), 60
trace_config (in module aiohttp), 44
trace_config_ctx() (aiohttp.TraceConfig method), 71
TraceConfig (class in aiohttp), 71
TraceConnectionCreateEndParams (class in aiohttp), 74
TraceConnectionCreateStartParams (class in aiohttp), 74
TraceConnectionQueuedEndParams (class in aiohttp), 74
TraceConnectionQueuedStartParams (class in aiohttp), 74
TraceConnectionReuseConnParams (class in aiohttp), 75
TraceDnsCacheHitParams (class in aiohttp), 75
TraceDnsCacheMissParams (class in aiohttp), 75
TraceDnsResolveHostEndParams (class in aiohttp), 75
TraceDnsResolveHostStartParams (class in aiohttp), 75
TraceRequestChunkSentParams (class in aiohttp), 75
TraceRequestEndParams (class in aiohttp), 73
TraceRequestExceptionParams (class in aiohttp), 73
TraceRequestRedirectParams (class in aiohttp), 74
TraceRequestStartParams (class in aiohttp), 72
TraceResponseChunkSentParams (class in aiohttp), 73
transport (aiohttp.Connection attribute), 54
transport (aiohttp.web.BaseRequest attribute), 106
trust_env (in module aiohttp), 44
TRY_AGAIN_LATER (aiohttp.WSCloseCode attribute), 179
type (aiohttp.WSMessage attribute), 179

U

UnixConnector (class in aiohttp), 54
UnixSite (class in aiohttp.web), 136
unread_data() (aiohttp.StreamReader method), 177
UNSupported_DatA  (aiohttp.WSCloseCode attribute), 178  
unused_port () (in module aiohttp.test_utils), 158  
update_cookies ()  (aiohttp.abc.aiohttp.abc.AbstractCookieJar method), 166  
update_cookies () (aiohttp.CookieJar method), 62  
url (aiohttp.ClientResponse attribute), 55  
url (aiohttp.InvalidURL attribute), 64  
url (aiohttp.RequestInfo attribute), 61  
url (aiohttp.TraceRequestEndParams attribute), 73  
url (aiohttp.TraceRequestExceptionParams attribute), 73  
url (aiohttp.TraceRequestRedirectParams attribute), 74  
url (aiohttp.TraceRequestStartParams attribute), 72  
url_for ()  (aiohttp.web.BaseRequest attribute), 104  
url_for ()  (aiohttp.web.AbstractResource method), 127  
url_for () (aiohttp.web.AbstractRoute method), 129  
url_for () (aiohttp.web.DynamicResource method), 128  
url_for () (aiohttp.web.PlainResource method), 127  
url_for () (aiohttp.web.PrefixedSubAppResource method), 128  
url_for () (aiohttp.web.StaticResource method), 128  
UrlDispatcher (class in aiohttp.web), 123  
UrlMappingMatchInfo (class in aiohttp.web), 133  

V  
value (aiohttp.ContentDisposition attribute), 65  
version (aiohttp.ClientResponse attribute), 55  
version (aiohttp.web.BaseRequest attribute), 104  
View (class in aiohttp.web), 133  
view ()  (aiohttp.web.RouteTableDef method), 133  
view () (in module aiohttp.web), 131  

W  
wait_eof () (in module aiohttp), 177  
wait_for_disconnection () (aiohttp.web.BaseRequest method), 109  
web-handler, 197  
websocket, 197  
WebSocketReady (class in aiohttp.web), 119  
WebSocketResponse (class in aiohttp.web), 115  
write () (aiohttp.MultipartWriter method), 175  
write () (aiohttp.web.StreamResponse method), 113  
write_eof () (aiohttp.web.StreamResponse method), 113  
ws_close (aiohttp.ClientWSTimeout attribute), 60  
ws_connect () (aiohttp.test_utils.TestClient method), 157  
ws_connect () (in module aiohttp.test_utils.TestClient), 157  
ws_protocol (aiohttp.web.WebSocketResponse attribute), 116  
ws_receive (aiohttp.ClientWSTimeout attribute), 60  
WCloseCode (class in aiohttp), 178  
WSMessage (class in aiohttp), 179  
WMsgType (class in aiohttp), 179  
WSServerHandshakeError (class in aiohttp), 65  

Y  
yarl, 197  

Index 263