
aiohttp Documentation

Release 2.2.0-

aiohttp contributors

Jun 20, 2017

Contents

1	Features	3
2	Library Installation	5
3	Getting Started	7
4	Tutorial	9
5	Source code	11
6	Dependencies	13
7	Discussion list	15
8	Contributing	17
9	Authors and License	19
10	Policy for Backward Incompatible Changes	21
11	Contents	23
11.1	Migration to 2.x	23
11.2	Client	26
11.3	Client Reference	37
11.4	Server Tutorial	54
11.5	Server Usage	61
11.6	Server Reference	79
11.7	Low Level Server	106
11.8	Abstract Base Classes	107
11.9	Working with Multipart	109
11.10	Streaming API	114
11.11	Helpers API	116
11.12	Logging	121
11.13	Testing	123
11.14	aiohhttp server deployment	132
11.15	Frequently Asked Questions	137
11.16	Third-Party libraries	142
11.17	Essays	144

11.18 Contributing	147
11.19 Changes	149
11.20 Glossary	176
12 Indices and tables	177
Python Module Index	179

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

CHAPTER 1

Features

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

CHAPTER 2

Library Installation

```
$ pip install aiohttp
```

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

```
$ pip install cchardet
```

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

```
$ pip install aiодns
```


CHAPTER 3

Getting Started

Client example:

```
import aiohttp
import asyncio
import asyncio_timeout

async def fetch(session, url):
    with asyncio_timeout.timeout(10):
        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())
```

Server example:

```
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.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

Note: Throughout this documentation, examples utilize the *async/await* syntax introduced by [PEP 492](#) that is only valid for Python 3.5+.

If you are using Python 3.4, please replace `await` with `yield from` and `async def` with a `@coroutine` decorator. For example, this:

```
async def coro(...):  
    ret = await f()
```

should be replaced by:

```
@asyncio.coroutine  
def coro(...):  
    ret = yield from f()
```

CHAPTER 4

Tutorial

Polls tutorial

CHAPTER 5

Source code

The project is hosted on [GitHub](#)

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

The library uses [Travis](#) for Continuous Integration.

- Python 3.4.2+
- *chardet*
- *multidict*
- *async_timeout*
- *yaml*
- *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
```


CHAPTER 7

Discussion list

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

Feel free to post your questions and ideas here.

CHAPTER 8

Contributing

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

CHAPTER 9

Authors and License

The `aihttp` 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 guaranties 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 the own rule for 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.

11.1 Migration to 2.x

11.1.1 Client

chunking

aiohhttp 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.

aiohhttp 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:

```

@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 *yaml.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.
7. *ClientSession.close()* is a regular function returning *None*, not a coroutine.

11.1.2 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 *unicorn* 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 support *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:

```
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.
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*

11.2 Client

11.2.1 Make a Request

Begin by importing the aiohttp module:

```
import aiohttp
```

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

```
async with aiohttp.ClientSession() as session:
    async with session.get('https://api.github.com/events') 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.

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

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

Other HTTP methods are available as well:

```
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.

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

11.2.2 JSON Request

Any of session's request methods like *request*, *get*, *post* etc accept *json* parameter:

```
async with aiohttp.ClientSession() as session:
    async with session.post(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:

```
import ujson

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

11.2.3 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:

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

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:

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

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

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

11.2.4 Response Content

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

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

will printout something like:

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

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

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

11.2.5 Binary Response Content

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

```
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.

11.2.6 JSON Response Content

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

```
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.

11.2.7 Streaming Response Content

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

```
async with 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:


```

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`.

11.2.8 RequestInfo

`ClientResponse` object contains `request_info` property, which contains request fields: `url` and `headers`. On `raise_for_status` structure is copied to `ClientResponseError` instance.

11.2.9 Custom Headers

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

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

```

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

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

```

11.2.10 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"}}

```

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

11.2.11 More complicated POST requests

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

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

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

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

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

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

async with session.post(url, data=json.dumps(payload)) as resp:
    ...
```

11.2.12 POST a Multipart-Encoded File

To upload Multipart-encoded files:

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

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

You can set the filename, content_type explicitly:

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

await 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

11.2.13 Streaming uploads

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

into memory.

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

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

Or you can use `aiohttp.streamer` object:

```
@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` 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())
```

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

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

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

    return h.hexdigest()

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

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

Because the response content attribute is a `StreamReader`, you can chain get and post requests together:

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

11.2.14 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 `zlib`) as the value of the `Content-Encoding` header:

```
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
```

11.2.15 Keep-Alive, connection pooling and cookie sharing

`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('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'
```

You also can set default headers for all session requests:

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

`ClientSession` supports keep-alive requests and connection pooling out-of-the-box.

11.2.16 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:

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

11.2.17 Connectors

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

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

Note: You can not re-use custom `connector`, `session` object takes ownership of the `connector`.

See also:

Connectors section for more information about different connector types and configuration options.

11.2.18 Limiting connection pool size

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

```
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).

11.2.19 Resolving using custom nameservers

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

```
from aiohttp.resolver import AsyncResolver

resolver = AsyncResolver(nameservers=["8.8.8.8", "8.8.4.4"])
conn = aiohttp.TCPConnector(resolver=resolver)
```

11.2.20 SSL control for TCP sockets

TCPConnector constructor accepts mutually exclusive *verify_ssl* and *ssl_context* params.

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

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

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

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

If you need to verify **client-side** certificates, you can do the same thing as the previous example, but add another call to `load_cret_chain` with the key pair:

```
sslcontext = ssl.create_default_context(
    cafile='/path/to/client-side-ca-bundle.crt')
sslcontext.load_cert_chain('/path/to/client/public/key.pem', '/path/to/client/private/
↪key.pem')
conn = aiohttp.TCPConnector(ssl_context=sslcontext)
session = aiohttp.ClientSession(connector=conn)
r = await session.get('https://server-with-client-side-certificates-validation.com')
```

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

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

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

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

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

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

11.2.21 Unix domain sockets

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

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

11.2.22 Proxy support

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

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

Contrary to the `requests` library, it won't read environment variables by default. But you can do so by setting `proxy_from_env` to `True`. It will use the `getproxies()` method from `urllib` and thus read the value of the `$url-scheme_proxy` variable:

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

It also supports proxy authorization:

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

Authentication credentials can be passed in proxy URL:

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

11.2.23 Response Status Codes

We can check the response status code:

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

11.2.24 Response Headers

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

```
>>> resp.headers
{'ACCESS-CONTROL-ALLOW-ORIGIN': '*',
 'CONTENT-TYPE': 'application/json',
 'DATE': 'Tue, 15 Jul 2014 16:49:51 GMT',
 'SERVER': 'unicorn/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:

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

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

All headers 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:

```
>>> 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'))
```

11.2.25 Response Cookies

If a response contains some 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.

11.2.26 Response History

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

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

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

11.2.27 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:

```
session = aiohttp.ClientSession()
async with session.ws_connect('http://example.org/websocket') 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.CLOSED:
    break
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 `ws.send_str('data')` for example).

11.2.28 Timeouts

By default all IO operations have 5min timeout. The timeout may be overridden by passing `timeout` parameter into `ClientSession.get()` and family:

```

async with session.get('https://github.com', timeout=60) as r:
    ...

```

`None` or `0` disables timeout check.

The example wraps a client call in `async_timeout.timeout()` context manager, adding timeout for both connecting and response body reading procedures:

```

import async_timeout

with async_timeout.timeout(0.001, loop=session.loop):
    async with session.get('https://github.com') as r:
        await r.text()

```

Note: Timeout is cumulative time, it includes all operations like sending request, redirects, response parsing, consuming response, etc.

11.3 Client Reference

11.3.1 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:

```

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(loop):

```

```

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

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

```

New in version 0.17.

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

```

class aiohttp.ClientSession(*, connector=None, loop=None, cookies=None, headers=None,
                             skip_auto_headers=None, auth=None,
                             json_serialize=func:'json.dumps', version=aiohttp.HttpVersion11,
                             cookie_jar=None, read_timeout=None, conn_timeout=None,
                             raise_for_status=False)

```

The class for creating client sessions and making requests.

Parameters

- **connector** (*aiohttp.connector.BaseConnector*) – BaseConnector subclass instance to support connection pooling.
- **loop** – event loop used for processing HTTP requests.
If *loop* is None the constructor borrows it from *connector* if specified.
`asyncio.get_event_loop()` is used for getting default event loop otherwise.
- **cookies** (*dict*) – Cookies to send with the request (optional)
- **headers** – HTTP Headers to send with 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.
New in version 0.21.
- **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.helpers.DummyCookieJar* instance can be provided.
New in version 0.22.
- **json_serialize** (*callable*) – *Json serializer* function. (`json.dumps()` by default)

- **raise_for_status** (*bool*) – Automatically call *raise_for_status()* for each response. (default is `False`)

New in version 2.0.

- **read_timeout** (*float*) – Request operations timeout. *read_timeout* is cumulative for all request operations (request, redirects, responses, data consuming). By default, the read timeout is 5*60 seconds. Use `None` or `0` to disable timeout checks.
- **conn_timeout** (*float*) – timeout for connection establishing (optional). Values `0` or `None` mean no timeout.

Changed in version 1.0.

.cookies attribute was dropped. Use `cookie_jar` instead.

closed

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

A read-only property.

connector

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

A read-only property.

cookie_jar

The session cookies, *AbstractCookieJar* instance.

Gives access to cookie jar's content and modifiers.

A read-only property.

New in version 1.0.

requote_redirect_url

aiohttp re quote's redirect urls by default, but some servers require exact url from location header. to disable *re-quote* system set *requote_redirect_url* to `False`.

Note: this parameter affects all subsequent requests.

loop

A loop instance used for session creation.

A read-only property.

coroutine async-with request (*method*, *url*, *, *params=None*, *data=None*, *json=None*, *headers=None*, *skip_auto_headers=None*, *auth=None*, *allow_redirects=True*, *max_redirects=10*, *compress=None*, *chunked=None*, *expect100=False*, *read_until_eof=True*, *proxy=None*, *proxy_auth=None*, *timeout=5*60*)

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** – Dictionary, bytes, or file-like object to send in the body of the request (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)
 - **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).
 - **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** (*int*) – 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).
 - **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 (`read_timeout`) for IO operations.

Return ClientResponse a *client response* object.

New in version 1.0: Added `proxy` and `proxy_auth` parameters.

Added `timeout` parameter.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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** – Dictionary, bytes, or file-like object to send in the body of the request (optional)

Return ClientResponse a *client response* object.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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** – Dictionary, bytes, or file-like object to send in the body of the request (optional)

Return ClientResponse a *client response* object.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with 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** – Dictionary, bytes, or file-like object to send in the body of the request (optional)

Return ClientResponse a *client response* object.

Changed in version 1.1: URLs may be either `str` or `URL`

coroutine async-with ws_connect (*url*, *, *protocols=()*, *timeout=10.0*, *receive_timeout=None*, *auth=None*, *autoclose=True*, *autoping=True*, *heartbeat=None*, *origin=None*, *proxy=None*, *proxy_auth=None*)

Create a websocket connection. Returns a *ClientWebSocketResponse* object.

Parameters

- **url** – Websocket server url, `str` or `URL`
- **protocols** (*tuple*) – Websocket protocols
- **timeout** (*float*) – Timeout for websocket to close. 10 seconds by default
- **receive_timeout** (*float*) – Timeout for websocket to receive complete message. `None`(unlimited) seconds by default
- **auth** (`aiohttp.BasicAuth`) – an object that represents HTTP Basic Authorization (optional)
- **autoclose** (*bool*) – Automatically close websocket connection on close message from server. If *autoclose* is `False` then close procedure has to be handled manually
- **autoping** (*bool*) – automatically send *pong* on *ping* message from server
- **heartbeat** (*float*) – Send *ping* message every *heartbeat* seconds and wait *pong* response, if *pong* response is not received then close connection.
- **origin** (*str*) – Origin header to send to server

- **proxy** (*str*) – Proxy URL, *str* or *URL* (optional)
- **proxy_auth** (`aiohttp.BasicAuth`) – an object that represents proxy HTTP Basic Authorization (optional)

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

New in version 0.18: Add `auth` parameter.

New in version 0.19: Add `origin` parameter.

New in version 1.0: Added `proxy` and `proxy_auth` parameters.

Changed in version 1.1: URLs may be either *str* or *URL*

close()

Close underlying connector.

Release all acquired resources.

Changed in version 2.0.

detach()

Detach connector from session without closing the former.

Session is switched to closed state anyway.

11.3.2 Basic API

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

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

coroutine `aiohttp.request` (*method*, *url*, *, *params=None*, *data=None*, *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*, *connector=None*, *loop=None*, *read_until_eof=True*)

Perform an asynchronous HTTP request. Return a response object (`ClientResponse` or derived from).

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** – Dictionary, bytes, or file-like object to send in the body of the request (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).
- **connector** (*aiohttp.connector.BaseConnector*) – `BaseConnector` subclass 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).
- **loop** – event loop used for processing HTTP requests. If param is `None`, `asyncio.get_event_loop()` is used for getting default event loop, but we strongly recommend to use explicit loops everywhere. (optional)

Return ClientResponse a *client response* object.

Usage:

```
import aiohttp

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

Changed in version 1.1: URLs may be either `str` or `URL`

11.3.3 Connectors

Connectors are transports for aiohttp client API.

There are standard connectors:

1. *TCPConnector* for regular *TCP sockets* (both *HTTP* and *HTTPS* schemes supported).
2. *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

```
class aiohttp.BaseConnector(*, keepalive_timeout=30, limit=100, limit_per_host=None,
                           force_close=False, loop=None)
```

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 for simultaneous connections to the same endpoint. Endpoints are the same if they have equal (*host*, *port*, *is_ssl*) triple. If *limit* is *None* the connector has no limit (default: *None*).
- **force_close** (*bool*) – do close underlying sockets after connection releasing (optional).
- **loop** – *event loop* used for handling connections. If param is *None*, `asyncio.get_event_loop()` is used for getting default event loop, but we strongly recommend to use explicit loops everywhere. (optional)

closed

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

force_close

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

New in version 0.16.

limit

The 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 have equal (*host*, *port*, *is_ssl*) triple.

If *limit_per_host* is *None* the connector has no limit per host.

Read-only property.

close()

Close all opened connections.

New in version 2.0.

coroutine connect(request)

Get a free connection from pool or create new one if connection is absent in the pool.

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

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

Returns `Connection` object.

coroutine _create_connection(req)

Abstract method for actual connection establishing, should be overridden in subclasses.

TCPConnector

```
class aiohttp.TCPConnector(*, verify_ssl=True, fingerprint=None, use_dns_cache=True,
                           ttl_dns_cache=10, family=0, ssl_context=None, conn_timeout=None,
                           keepalive_timeout=30, limit=None, force_close=False, loop=None,
                           local_addr=None, disable_cleanup_closed=True)
```

Connector for working with *HTTP* and *HTTPS* via *TCP* sockets.

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

`TCPConnector` inherits from `BaseConnector`.

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

Parameters

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

Note: use of MD5 or SHA1 digests is insecure and deprecated.

New in version 0.16.

- **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.

New in version 0.17.

Changed in version 1.0: The default is changed to `True`

- **ttl_dns_cache** (*int*) – expire after some seconds the DNS entries, `None` means cached forever. By default 10 seconds.

By default DNS entries are cached forever, 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.

New in version 2.0.8.

- **resolver** (*aiohttp.abc.AbstractResolver*) – Custom resolver instance to use. `aiohttp.DefaultResolver` by default (asynchronous if `aiodns` >= 1.1 is installed).

Custom resolvers allow to resolve hostnames differently than the way the host is configured.

New in version 0.22.

Changed in version 1.0: The resolver is `aiohttp.AsyncResolver` now if *aiodns* is installed.

- **family** (*int*) –

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

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

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

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

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

New in version 0.21.

- **enable_cleanup_closed** (*tuple*) – 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.

verify_ssl

Check *ssl certifications* if True.

Read-only *bool* property.

ssl_context

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

family

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

Read-only property.

dns_cache

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

Read-only *bool* property.

New in version 0.17.

cached_hosts

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

Read-only `types.MappingProxyType` property.

New in version 0.17.

fingerprint

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

Read-only *bytes* property.

New in version 0.16.

clear_dns_cache (*self*, *host=None*, *port=None*)

Clear internal *DNS* cache.

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

New in version 0.17.

UnixConnector

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

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:

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

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

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

path

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

Connection**class aiohttp.Connection**

Encapsulates single connection in connector object.

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

closed

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

loop

Event loop used for connection

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.

detach()

Detach underlying socket from connection.

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

11.3.4 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.:

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

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

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

version

Response's version, *HttpVersion* instance.

status

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

reason

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

method

Request's method (`str`).

url

URL of request (`URL`).

connection

`Connection` used for handling response.

content

Payload stream, contains response's BODY (`StreamReader`).

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 mis-formed 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.

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.

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.

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*)

Read response's body and return decoded `str` using specified *encoding* parameter.

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

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

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

Return str decoded *BODY*

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

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

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

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

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

param callable loads `callable()` used for loading *JSON* data, `json.loads()` by default.

param str content_type 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*).

return 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.

11.3.5 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.

ping (*message=b''*)

Send *PING* to peer.

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

coroutine send_str (*data*)

Send *data* to peer as *TEXT* message.

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

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

coroutine send_bytes (*data*)

Send *data* to peer as *BINARY* message.

Parameters *data* – data to send.

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

coroutine send_json (*data*, *, *dumps=json.loads*)

Send *data* to peer as JSON string.

Parameters

- **data** – data to send.
- **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*

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 *pong* message, *str* (converted to *UTF-8* encoded bytes) or *bytes*.

coroutine receive ()

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

The *coroutine* implicitly handles *PING*, *PONG* and *CLOSE* without returning the message.

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

Returns *WSMessage*, *tp* is a type from *WSMsgType* enumeration.

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`.
- `ValueError` – if message is not valid JSON.

11.3.6 Utilities

BasicAuth

class `aiohhttp.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`.

encode()

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

Returns encoded authentication data, `str`.

CookieJar

class `aiohhttp.CookieJar(unsafe=False, loop=None)`

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.
- **loop** (*bool*) – an `event loop` instance. See `aiohttp.abc.AbstractCookieJar`

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.

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 attributes in *aiohhttp* module.

Hierarchy of exceptions:

- *aiohhttp.ClientError* - Base class for all client specific exceptions
 - *aiohhttp.ClientResponseError* - exceptions that could happen after we get response from server.
 - request_info* - Instance of *RequestInfo* object, contains information about request.
 - history* - History from *ClientResponse* object, if available, else empty tuple.
 - * *aiohhttp.WSServerHandshakeError* - web socket server response error
 - * *aiohhttp.ClientHttpProxyError* - proxy response
 - *aiohhttp.ClientConnectionError* - exceptions related to low-level connection problems
 - * *aiohhttp.ClientOSError* - subset of connection errors that are initiated by an *OSError* exception
 - *aiohhttp.ClientConnectorError* - connector related exceptions
 - *aiohhttp.ClientProxyConnectionError* - proxy connection initialization error
 - * *aiohhttp.ServerConnectionError* - server connection related errors
 - * *aiohhttp.ServerDisconnectedError* - server disconnected
 - message* - Partially parsed http message (optional)
 - * *aiohhttp.ServerTimeoutError* - server operation timeout, (read timeout, etc)
 - * *aiohhttp.ServerFingerprintMismatch* - server fingerprint mismatch
 - *aiohhttp.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.

11.4 Server Tutorial

Are you going to learn *aiohhttp* but don't know where to start? We have example for you. Polls application is a great example for getting started with *aiohhttp*.

If you want the full source code in advance or for comparison, check out the [demo source](#).

11.4.1 Setup your environment

First of all check you python version:

```
$ python -V
Python 3.5.0
```

Tutorial requires Python 3.5.0 or newer.

We'll assume that you have already installed *aiohhttp* library. You can check *aiohhttp* is installed and which version by running the following command:

```
$ python3 -c 'import aiohttp; print(aiohttp.__version__)'
2.0.5
```

Project structure looks very similar to other python based web projects:

```

.
- README.rst
- polls
  - Makefile
  - README.rst
  - aiohttpdemo_polls
    | - __init__.py
    | - __main__.py
    | - db.py
    | - main.py
    | - routes.py
    | - templates
    | - utils.py
    | - views.py
  - config
    | - polls.yaml
  - images
    | - example.png
  - setup.py
  - sql
    | - create_tables.sql
    | - install.sh
    | - sample_data.sql
  - static
    - style.css

```

11.4.2 Getting started with aiohttp first app

This tutorial based on Django polls tutorial.

11.4.3 Application

All aiohttp server is built around `aiohttp.web.Application` instance. It is used for registering *startup/cleanup* signals, connecting routes etc.

The following code creates an application:

```

from aiohttp import web

app = web.Application()
web.run_app(app, host='127.0.0.1', port=8080)

```

Save it under `aiohttpdemo_polls/main.py` and start the server:

```
$ python3 main.py
```

You'll see the following output on the command line:

```

===== Running on http://127.0.0.1:8080 =====
(Press CTRL+C to quit)

```

Open `http://127.0.0.1:8080` in browser or do

```
$ curl -X GET localhost:8080
```

Alas, for now both return only 404: Not Found. To show something more meaningful let's create a route and a view.

11.4.4 Views

Let's start from first views. Create the file `aiohhttpdemo_polls/views.py` with the following:

```
from aiohttp import web

async def index(request):
    return web.Response(text='Hello Aiohttp!')
```

This is the simplest view possible in Aiohttp. Now we should create a route for this `index` view. Put this into `aiohhttpdemo_polls/routes.py` (it is a good practice to separate views, routes, models etc. You'll have more of each, and it is nice to have them in different places):

```
from views import index

def setup_routes(app):
    app.router.add_get('/', index)
```

Also, we should call `setup_routes` function somewhere, and the best place is in the `main.py`

```
from aiohttp import web
from routes import setup_routes

app = web.Application()
setup_routes(app)
web.run_app(app, host='127.0.0.1', port=8080)
```

Start server again. Now if we open browser we can see:

```
$ curl -X GET localhost:8080
Hello Aiohttp!
```

Success! For now your working directory should look like this:

```
.
- ..
- polls
  - aiohttpdemo_polls
    | - main.py
    | - routes.py
    | - views.py
```

11.4.5 Configuration files

`aiohttp` is configuration agnostic. It means the library does not require any configuration approach and does not have builtin support for any config schema.

But please take into account these facts:

1. 99% of servers have configuration files.
2. Every product (except Python-based solutions like Django and Flask) does not store config files as part as source code.

For example Nginx has own configuration files stored by default under `/etc/nginx` folder.

Mongo pushes config as `/etc/mongodb.conf`.

3. Config files validation is good idea, strong checks may prevent silly errors during product deployment.

Thus we **suggest** to use the following approach:

1. Pushing configs as `yaml` files (`json` or `ini` is also good but `yaml` is the best).
2. Loading `yaml` config from a list of predefined locations, e.g. `./config/app_cfg.yaml`, `/etc/app_cfg.yaml`.
3. Keeping ability to override config file by command line parameter, e.g. `./run_app --config=/opt/config/app_cfg.yaml`.
4. Applying strict validation checks to loaded dict. `trafaret`, `colander` or `JSON schema` are good candidates for such job.

Load config and push into application:

```
# load config from yaml file in current dir
conf = load_config(str(pathlib.Path('.') / 'config' / 'polls.yaml'))
app['config'] = conf
```

11.4.6 Database

Setup

In this tutorial we will use the latest PostgreSQL database. You can install PostgreSQL using this instruction <http://www.postgresql.org/download/>

Database schema

We use SQLAlchemy to describe database schemas. For this tutorial we can use two simple models `question` and `choice`:

```
import sqlalchemy as sa

meta = sa.MetaData()

question = sa.Table(
    'question', meta,
    sa.Column('id', sa.Integer, nullable=False),
    sa.Column('question_text', sa.String(200), nullable=False),
    sa.Column('pub_date', sa.Date, nullable=False),

    # Indexes #
    sa.PrimaryKeyConstraint('id', name='question_id_pkey'))

choice = sa.Table(
    'choice', meta,
```

```

sa.Column('id', sa.Integer, nullable=False),
sa.Column('question_id', sa.Integer, nullable=False),
sa.Column('choice_text', sa.String(200), nullable=False),
sa.Column('votes', sa.Integer, server_default="0", nullable=False),

# Indexes #
sa.PrimaryKeyConstraint('id', name='choice_id_pkey'),
sa.ForeignKeyConstraint(['question_id'], [question.c.id],
                        name='choice_question_id_fkey',
                        ondelete='CASCADE'),
)

```

You can find below description of tables in database:

First table is question:

question
id
question_text
pub_date

and second table is choice table:

choice
id
choice_text
votes
question_id

Creating connection engine

For making DB queries we need an engine instance. Assuming `conf` is a `dict` with configuration info Postgres connection could be done by the following coroutine:

```

async def init_pg(app):
    conf = app['config']
    engine = await aiopg.sa.create_engine(
        database=conf['database'],
        user=conf['user'],
        password=conf['password'],
        host=conf['host'],
        port=conf['port'],
        minsize=conf['minsize'],
        maxsize=conf['maxsize'])
    app['db'] = engine

```

The best place for connecting to DB is `on_startup` signal:

```

app.on_startup.append(init_pg)

```

Graceful shutdown

There is a good practice to close all resources on program exit.

Let's close DB connection in `on_cleanup` signal:

```
async def close_pg(app):
    app['db'].close()
    await app['db'].wait_closed()

app.on_cleanup.append(close_pg)
```

11.4.7 Templates

Let's add more useful views:

```
@aiohttp_jinja2.template('detail.html')
async def poll(request):
    async with request['db'].acquire() as conn:
        question_id = request.match_info['question_id']
        try:
            question, choices = await db.get_question(conn,
                                                    question_id)

        except db.RecordNotFound as e:
            raise web.HTTPNotFound(text=str(e))
        return {
            'question': question,
            'choices': choices
        }
```

Templates are very convenient way for web page writing. We return a dict with page content, `aiohttp_jinja2`. `template` decorator processes it by `jinja2` template renderer.

For setting up template engine we need to install `aiohttp_jinja2` library first:

```
$ pip install aiohttp_jinja2
```

After installing we need to setup the library:

```
import aiohttp_jinja2
import jinja2

aiohttp_jinja2.setup(
    app, loader=jinja2.PackageLoader('aiohttpdemo_polls', 'templates'))
```

In the tutorial we push template files under `polls/aiohttpdemo_polls/templates` folder.

11.4.8 Static files

Any web site has static files: images, JavaScript sources, CSS files etc.

The best way to handle static in production is setting up reverse proxy like NGINX or using CDN services.

But for development handling static files by `aiohttp` server is very convenient.

Fortunately it can be done easy by single call:

```
app.router.add_static('/static/',
                    path=str(project_root / 'static'),
                    name='static')
```

where `project_root` is the path to root folder.

11.4.9 Middlewares

Middlewares are stacked around every web-handler. They are called *before* handler for pre-processing request and *after* getting response back for post-processing given response.

Here we'll add a simple middleware for displaying pretty looking pages for *404 Not Found* and *500 Internal Error*.

Middlewares could be registered in `app` by adding new middleware to `app.middlewares` list:

```
def setup_middlewares(app):
    error_middleware = error_pages({404: handle_404,
                                    500: handle_500})
    app.middlewares.append(error_middleware)
```

Middleware itself is a factory which accepts *application* and *next handler* (the following middleware or *web-handler* in case of the latest middleware in the list).

The factory returns *middleware handler* which has the same signature as regular *web-handler* – it accepts *request* and returns *response*.

Middleware for processing HTTP exceptions:

```
def error_pages(overrides):
    async def middleware(app, handler):
        async def middleware_handler(request):
            try:
                response = await handler(request)
                override = overrides.get(response.status)
                if override is None:
                    return response
                else:
                    return await override(request, response)
            except web.HTTPException as ex:
                override = overrides.get(ex.status)
                if override is None:
                    raise
                else:
                    return await override(request, ex)
        return middleware_handler
    return middleware
```

Registered overrides are trivial Jinja2 template renderers:

```
async def handle_404(request, response):
    response = aiohttp_jinja2.render_template('404.html',
                                             request,
                                             {})
    return response

async def handle_500(request, response):
    response = aiohttp_jinja2.render_template('500.html',
                                             request,
                                             {})
    return response
```


See also:*Middlewares*

11.5 Server Usage

11.5.1 Run a Simple Web Server

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

A request handler is a *coroutine* or regular function that accepts a *Request* instance as its only parameter and returns a *Response* instance:

```

from aiohttp import web

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

```

Next, create an *Application* instance and register the request handler with the application's *router* on a particular *HTTP method* and *path*:

```

app = web.Application()
app.router.add_get('/', hello)

```

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

```

web.run_app(app)

```

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

See also:

Graceful shutdown section explains what *run_app()* does and how to implement complex server initialization/finalization from scratch.

11.5.2 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:

```

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

```

11.5.3 Handler

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

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

A handler **may** also be a [coroutine](#), in which case `aihttp.web` will await the handler:

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

Handlers are setup to handle requests by registering them with the `Application.router` on a particular route (*HTTP method* and *path* pair) using methods like `UrlDispatcher.add_get` and `UrlDispatcher.add_post`:

```
app.router.add_get('/', handler)
app.router.add_post('/post', post_handler)
app.router.add_put('/put', put_handler)
```

`add_route()` also supports the wildcard *HTTP method*, allowing a handler to serve incoming requests on a *path* having **any** *HTTP method*:

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

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

By default endpoints added with `add_get()` 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:

```
app.router.add_get('/', handler, allow_head=False)
```

Here handler won't be called and the server will response with 405.

Note: This is a change as of **aihttp v2.0** to act in accordance with [RFC 7231](#).

Previous version always returned 405 for HEAD requests to routes added with `add_get()`.

If you have handlers which perform lots of processing to write the response body you may wish to improve performance by skipping that processing in the case of HEAD requests while still taking care to respond with the same headers as with GET requests.

11.5.4 Resources and Routes

Internally `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*.

`UrlDispatcher.add_get()` / `UrlDispatcher.add_post()` and family are plain shortcuts for `UrlDispatcher.add_route()`.

`UrlDispatcher.add_route()` in turn is just a shortcut for pair of `UrlDispatcher.add_resource()` and `Resource.add_route()`:

```
resource = app.router.add_resource(path, name=name)
route = resource.add_route(method, handler)
return route
```

See also:

Router refactoring in 0.21 for more details

New in version 0.21.0: Introduce resources.

11.5.5 Custom resource implementation

To register custom resource use `UrlDispatcher.register_resource()`. Resource instance must implement `AbstractResource` interface.

New in version 1.2.1.

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:

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

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

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

You can also specify a custom regex in the form `{identifier:regex}`:

```
resource = app.router.add_resource(r'/{name:\d+}')
```

Note: Regex should match against *percent encoded* URL (`request.rel_url_raw_path`). E.g. *space character* is encoded as `%20`.

According to [RFC 3986](#) allowed in path symbols are:

```
allowed      = unreserved / pct-encoded / sub-delims
              / ":" / "@" / "/"

pct-encoded  = "%" HEXDIG HEXDIG

unreserved  = ALPHA / DIGIT / "-" / "." / "_" / "~"

sub-delims  = "!" / "$" / "&" / "'" / "(" / ")"
              / "*" / "+" / "," / ";" / "="
```

Reverse URL Constructing using Named Resources

Routes can also be given a *name*:

```
resource = app.router.add_resource('/root', name='root')
```

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

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

A more interesting example is building *URLs* for *variable resources*:

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

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

```
>>> request.app.router['user-info'].url_for(user='john_doe') \
...                                     .with_query("a=b")
'/john_doe/info?a=b'
```

Organizing Handlers in Classes

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

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

```
class Handler:

    def __init__(self):
        pass

    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()
app.router.add_get('/intro', handler.handle_intro)
app.router.add_get('/greet/{name}', handler.handle_greeting)
```

Class Based Views

aiohhttp.web has support for django-style class based views.

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

```
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:

```
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:

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

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

```
for name, resource in app.router.named_resources().items():
    print(name, resource)
```

New in version 0.18: `UrlDispatcher.routes()`

New in version 0.19: `UrlDispatcher.named_routes()`

Deprecated since version 0.21: Use `UrlDispatcher.named_resources()` / `UrlDispatcher.resources()` instead of `UrlDispatcher.named_routes()` / `UrlDispatcher.routes()`.

11.5.6 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:

```
class AcceptChooser:

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

    async def do_route(self, request):
        for accept in request.headers.getall('ACCEPT', []):
            acceptor = self._accepts.get(accept)
            if acceptor is not None:
                return (await acceptor(request))
        raise HTTPNotAcceptable()
```

```

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

async def handle_json(request):
    # do json handling

async def handle_xml(request):
    # do xml handling

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

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

```

11.5.7 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 `UrlDispatcher.add_static()` call:

```
app.router.add_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`:

```
app.router.add_static('/prefix', path_to_static_folder, show_index=True)
```

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

```
app.router.add_static('/prefix', path_to_static_folder, follow_symlinks=True)
```

11.5.8 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()`:

```
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:

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

If you prefer the [Mako](#) template engine, please take a look at the [aiohttp_mako](#) library.

11.5.9 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()`:

```
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.

11.5.10 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:

```
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)

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.router.add_route('GET', '/', handler)
    return app

web.run_app(make_app())
```

11.5.11 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:

```
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.router.add_get('/', hello, expect_handler=check_auth)
```

11.5.12 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:

```
<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:

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

11.5.13 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:

```
<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:

```
async def store_mp3_handler(request):

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

    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})
```

You might be noticed a big warning in example above. The general issue is that `Request.post()` reads whole payload in memory. That's may hurt with OOM (Out Of Memory) error. To avoid this, for multipart uploads, you should use `Request.multipart()` which returns *multipart reader* back:

```
async def store_mp3_handler(request):

    reader = await request.multipart()

    # /\ Don't forget to validate your inputs /\

    mp3 = await reader.next()
```

```

filename = mp3.filename

# You cannot rely on Content-Length if transfer is chunked.
size = 0
with open(os.path.join('/spool/yarrr-media/mp3/', filename), 'wb') as f:
    while True:
        chunk = await mp3.read_chunk() # 8192 bytes by default.
        if not chunk:
            break
        size += len(chunk)
        f.write(chunk)

return web.Response(text='{} sized of {} successfully stored'
                    '.format(filename, size))

```

11.5.14 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:

```

async def websocket_handler(request):

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

    async for msg in ws:
        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

```

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 `aiohttp.web.WebSocketResponse.receive()` from two tasks.

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

11.5.15 Exceptions

`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.

The exceptions are also a subclass of `Response`, allowing you to either raise or return them in a *request handler* for the same effect.

The following snippets are the same:

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

and:

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

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

HTTP Exception hierarchy chart:

```
Exception
  HTTPException
    HTTPSuccessful
      * 200 - HTTPOk
      * 201 - HTTPCreated
      * 202 - HTTPAccepted
      * 203 - HTTPNonAuthoritativeInformation
      * 204 - HTTPNoContent
      * 205 - HTTPResetContent
      * 206 - HTTPPartialContent
    HTTPRedirection
      * 300 - HTTPMultipleChoices
      * 301 - HTTPMovedPermanently
      * 302 - HTTPFound
      * 303 - HTTPSeeOther
      * 304 - HTTPNotModified
      * 305 - HTTPUseProxy
      * 307 - HTTPTemporaryRedirect
      * 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 - HTTPRequestRangeNotSatisfiable
* 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:

```
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:

```
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:

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

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

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

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

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

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

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

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

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

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

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

11.5.17 Middlewares

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

Middlewares are setup by providing a sequence of *middleware factories* to the keyword-only *middlewares* parameter when creating an *Application*:

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

A *middleware factory* is simply a coroutine that implements the logic of a *middleware*. For example, here's a trivial *middleware factory*:

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

Every *middleware factory* should accept two parameters, an *app* instance and a *handler*, and return a new handler.

The *handler* passed in to a *middleware factory* is the handler returned by the **next** *middleware factory*. The last *middleware factory* always receives the *request handler* selected by the router itself (by *UrlDispatcher.resolve()*).

Middleware factories should return a new handler that has the same signature as a *request handler*. That is, it should accept a single *Request* instance and return a *Response*, or raise an exception.

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 *middleware factories* are themselves coroutines, they may perform extra `await` calls when creating a new handler, e.g. call database etc.

Middlewares usually call the inner handler, but they may choose to ignore it, e.g. displaying *403 Forbidden page* or raising `HTTPForbidden` exception if user has no 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.

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:

```
import json
from aiohttp import web

def json_error(message):
    return web.Response(
        body=json.dumps({'error': message}).encode('utf-8'),
        content_type='application/json')

async def error_middleware(app, handler):
    async def middleware_handler(request):
        try:
            response = await handler(request)
            if response.status == 404:
                return json_error(response.message)
            return response
        except web.HTTPException as ex:
            if ex.status == 404:
                return json_error(ex.reason)
            raise
    return middleware_handler

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

11.5.18 Signals

New in version 0.18.

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 *prepare()*), but sometimes we need a hook for changing HTTP headers for streamed responses and WebSockets. This can be accomplished by subscribing to the *on_response_prepare* signal:

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

app.on_response_prepare.append(on_prepare)
```

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

Warning: Signals API has provisional status, meaning it may be changed in future releases.

Signal subscription and sending will most likely be the same, but signal object creation is subject to change. As long as you are not creating new signals, but simply reusing existing ones, you will not be affected.

11.5.19 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 `add_subapp()`:

```
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 for sub-applications should generate urls with proper prefix.

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

```
admin = web.Application()
admin.router.add_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:

```
admin = web.Application()
admin.router.add_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()
```

11.5.20 Flow control

`aiohttp.web` has sophisticated flow control for underlying TCP sockets write buffer.

The problem is: by default TCP sockets use [Nagle's algorithm](#) for output buffer which is not optimal for streaming data protocols like HTTP.

Web server response may have one of the following states:

1. **CORK** (`tcp_cork` is `True`). Don't send out partial TCP/IP frames. All queued partial frames are sent when the option is cleared again. Optimal for sending big portion of data since data will be sent using minimum frames count.
If OS does not support **CORK** mode (neither `socket.TCP_CORK` nor `socket.TCP_NOPUSH` exists) the mode is equal to *Nagle's enabled* one. The most widespread OS without **CORK** support is *Windows*.
2. **NODELAY** (`tcp_nodelay` is `True`). Disable the Nagle algorithm. This means that small data pieces are always sent as soon as possible, even if there is only a small amount of data. Optimal for transmitting short messages.
3. Nagle's algorithm enabled (both `tcp_cork` and `tcp_nodelay` are `False`). Data is buffered until there is a sufficient amount to send out. Avoid using this mode for sending HTTP data until you have no doubts.

By default streaming data (`StreamResponse`), regular responses (`Response` and http exceptions derived from it) and websockets (`WebSocketResponse`) use **NODELAY** mode, static file handlers work in **CORK** mode.

To manual mode switch `set_tcp_cork()` and `set_tcp_nodelay()` methods can be used. It may be helpful for better streaming control for example.

11.5.21 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:

```
app = web.Application()
app['websockets'] = []

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

    request.app['websockets'].append(ws)
    try:
        async for msg in ws:
            ...
    finally:
        request.app['websockets'].remove(ws)

    return ws
```

Signal handler may look like:

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

app.on_shutdown.append(on_shutdown)
```


Proper finalization procedure has three steps:

1. Stop accepting new client connections by `asyncio.Server.close()` and `asyncio.Server.wait_closed()` calls.
2. Fire `Application.shutdown()` event.
3. Close accepted connections from clients by `Server.shutdown()` call with reasonable small delay.
4. Call registered application finalizers by `Application.cleanup()`.

The following code snippet performs proper application start, run and finalizing. It's pretty close to `run_app()` utility function:

```

loop = asyncio.get_event_loop()
handler = app.make_handler()
f = loop.create_server(handler, '0.0.0.0', 8080)
srv = loop.run_until_complete(f)
print('serving on', srv.sockets[0].getsockname())
try:
    loop.run_forever()
except KeyboardInterrupt:
    pass
finally:
    srv.close()
    loop.run_until_complete(srv.wait_closed())
    loop.run_until_complete(app.shutdown())
    loop.run_until_complete(handler.shutdown(60.0))
    loop.run_until_complete(app.cleanup())
loop.close()

```

11.5.22 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:

```

async def listen_to_redis(app):
    try:
        sub = await aioredis.create_redis(('localhost', 6379), loop=app.loop)
        ch, *_ = await sub.subscribe('news')
        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'] = app.loop.create_task(listen_to_redis(app))

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

app = web.Application()
app.on_startup.append(start_background_tasks)
app.on_cleanup.append(cleanup_background_tasks)
web.run_app(app)
```

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.

11.5.23 Handling error pages

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

11.5.24 Swagger support

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

11.5.25 CORS support

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

11.5.26 Debug Toolbar

`aiohttp_debugtoolbar` is a very useful library that provides a debugging toolbar while you're developing an `aiohttp.web` application.

Install it via pip:

```
$ pip install aiohttp_debugtoolbar
```

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

```
import aiohttp_debugtoolbar
from aiohttp_debugtoolbar import toolbar_middleware_factory
```

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

The toolbar is ready to use. Enjoy!!!

11.5.27 Dev Tools

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

Install via pip:

```
$ 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](#).

11.6 Server Reference

11.6.1 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) and `Request` has an *application* and *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.

Returns `aiohttp.protocol.HttpVersion` instance.

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*, if `secure_proxy_ssl_header` is matching (deprecated), if the `proto` portion of a `Forward` header is present and contains `https`, or if an `X-Forwarded-Proto` header is present and contains `https`.

'http' otherwise.

Read-only `str` property.

See also:

[Application.make_handler\(\)](#)

Deprecated since version 1.1: Use `url(request.url.scheme)` instead.

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.

Host name is resolved through the following headers, in this order:

- `Forwarded`
- `X-Forwarded-Host`
- `Host`

Returns `str`, or `None` if no host name is found in the headers.

path_qs

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

Read-only `str` property.

path

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

Read-only `str` property.

raw_path

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

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

Read-only `str` property.

query

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:

```
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.

has_body

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

Read-only `bool` property.

New in version 0.16.

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):

```
rng = request.http_range
with open(filename, 'rb') as f:
    f.seek(rng.start)
    return f.read(rng.stop-rng.start)
```

2. Slice-style:

```
return buffer[request.http_range]
```

New in version 1.2.

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.

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.

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)

Read request body decoded as *json*.

The method is just a boilerplate *coroutine* implemented as:

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

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

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

coroutine multipart (*, reader=aiohttp.multipart.MultipartReader)

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

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 `aiohhttp.web` internal machinery.

class `aiohhttp.web.Request`

An request used for receiving request's information by *web handler*.

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

The class is 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.

Note: You should never create the `Request` instance manually – `aiohhttp.web` does it for you. But `clone ()` may be used for cloning *modified* request copy with changed *path*, *method* etc.

11.6.2 Response classes

For now, `aiohhttp.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:

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


StreamResponse

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

The base class for the *HTTP response* handling.

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

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

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

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

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

Parameters

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

prepared

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

New in version 0.18.

task

A task that serves HTTP request handling.

May be useful for graceful shutdown of long-running requests (streaming, long polling or web-socket).

New in version 1.2.

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 encourage using of *cookies* and *set_cookie()*, *del_cookie()* for cookie manipulations.

set_cookie (*name*, *value*, *, *path='/'*, *expires=None*, *domain=None*, *max_age=None*, *secure=None*, *httponly=None*, *version=None*)

Convenient way for setting *cookies*, allows to specify some additional properties like *max_age* in a single call.

Parameters

- **name** (*str*) – cookie name
- **value** (*str*) – cookie value (will be converted to *str* if value has another type).
- **expires** – expiration date (optional)
- **domain** (*str*) – cookie domain (optional)
- **max_age** (*int*) – defines the lifetime of the cookie, in seconds. The delta-seconds value is a decimal non-negative integer. After delta-seconds seconds

elapse, the client should discard the cookie. A value of zero means the cookie should be discarded immediately. (optional)

- **path** (*str*) – specifies the subset of URLs to which this cookie applies. (optional, '/' by default)
- **secure** (*bool*) – attribute (with no value) directs the user agent to use only (unspecified) secure means to contact the origin server whenever it sends back this cookie. The user agent (possibly under the user’s control) may determine what level of security it considers appropriate for “secure” cookies. The *secure* should be considered security advice from the server to the user agent, indicating that it is in the session’s interest to protect the cookie contents. (optional)
- **httponly** (*bool*) – True if the cookie HTTP only (optional)
- **version** (*int*) – a decimal integer, identifies to which version of the state management specification the cookie conforms. (Optional, *version=1* by default)

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

Changed in version 1.0: Fixed cookie expiration support for Internet Explorer (version less than 11).

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.

tcp_cork

TCP_CORK (linux) or TCP_NOPUSH (FreeBSD and MacOSX) is applied to underlying transport if the property is `True`.

Use `set_tcp_cork()` to assign new value to the property.

Default value is `False`.

set_tcp_cork (*value*)

Set *tcp_cork* property to *value*.

Clear *tcp_nodelay* if *value* is `True`.

tcp_nodelay

TCP_NODELAY is applied to underlying transport if the property is `True`.

Use `set_tcp_nodelay()` to assign new value to the property.

Default value is `True`.

set_tcp_nodelay (*value*)

Set `tcp_nodelay` property to *value*.

Clear `tcp_cork` if *value* is `True`.

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.

New in version 0.18.

write (*data*)

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

`prepare()` must be called before.

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

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

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

coroutine drain ()

A *coroutine* to let the write buffer of the underlying transport a chance to be flushed.

The intended use is to write:

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

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

coroutine write_eof ()

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

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

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

Response

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

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

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

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

Parameters

- **body** (*bytes*) – response’s BODY

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

body

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

Setting *body* also recalculates *content_length* value.

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

text

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

WebSocketResponse

```
class aiohttp.web.WebSocketResponse (*, timeout=10.0, receive_timeout=None, autoclose=True,
                                     autoping=True, heartbeat=None, protocols=())
```

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.

New in version 1.3.0.

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.

New in version 1.3.0.

Parameters

- **heartbeat** (*float*) – Send *ping* message every *heartbeat* seconds and wait *pong* response, close connection if *pong* response is not received.
- **receive_timeout** (*float*) – Timeout value for *receive* operations. Default value is None (no timeout for receive operation)

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

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

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

coroutine prepare (*request*)

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

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

Raises `HTTPException` – if websocket handshake has failed.

New in version 0.18.

can_prepare (*request*)

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

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

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

Returns

`WebSocketReady` instance.

`WebSocketReady.ok` is `True` on success, `WebSocketReady.protocol` is websocket subprotocol which is passed by client and accepted by server (one of `protocols` 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.

protocol

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

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

exception()

Returns last occurred exception or `None`.

ping (*message=b''*)

Send `PING` to peer.

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

Raises `RuntimeError` – if connections is not started or closing.

pong (*message=b''*)

Send *unsolicited PONG* to peer.

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

Raises `RuntimeError` – if connections is not started or closing.

coroutine send_str (*data*)

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

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

Raises

- `RuntimeError` – if connection is not started or closing

- **TypeError** – if data is not `str`

coroutine `send_bytes` (*data*)

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

Parameters `data` – data to send.

Raises

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

coroutine `send_json` (*data*, *, *dumps*=`json.loads`)

Send *data* to peer as JSON string.

Parameters

- `data` – data to send.
- `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`

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

A *coroutine* that initiates closing handshake by sending `CLOSE` message.

It is save to call `close()` from different task.

Parameters

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

Raises **RuntimeError** – if connection is not started

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.

Return `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.

Return `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).

Parameters `timeout` –

timeout for receive operation. `timeout` value overrides response's `receive_timeout` attribute.

return `dict` loaded JSON content

raise `TypeError` if message is `BINARY`.

raise `ValueError` if message is not valid JSON.

New in version 0.22.

See also:

WebSockets handling

WebSocketReady

class `aiohttp.web.WebSocketReady`

A named tuple for returning result from `WebSocketResponse.can_prepare()`.

Has `bool` check implemented, e.g.:

```
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()`

11.6.3 json_response

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

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

11.6.4 Application and Router

Application

Application is a synonym for web-server.

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

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

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

```

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

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

```

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

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

The class inherits `dict`.

Parameters

- **router** – `aiohttp.abc.AbstractRouter` instance, the system creates `UrlDispatcher` by default if `router` is `None`.
- **logger** – `logging.Logger` instance for storing application logs.
By default the value is `logging.getLogger("aiohttp.web")`
- **middlewares** – list of middleware factories, see *Middlewares* for details.
- **debug** – Switches debug mode.
- **loop** – loop parameter is deprecated. loop is get set during freeze stage.

router

Read-only property that returns `router instance`.

logger

`logging.Logger` instance for storing application logs.

loop

`event loop` used for processing HTTP requests.

debug

Boolean value indicating whether the debug mode is turned on or off.

on_response_prepare

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

Signal handlers should have the following signature:

```
async def 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:

```
async def on_startup(app):  
    pass
```

See also:

Background tasks.

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:

```
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:

```
async def on_cleanup(app):  
    pass
```

See also:

Graceful shutdown and *on_shutdown*.

make_handler (*loop=None, **kwargs*)

Creates HTTP protocol factory for handling requests.

Parameters

- **loop** – event loop used for processing HTTP requests.
If param is None `asyncio.get_event_loop()` used for getting default event loop.
- **secure_proxy_ssl_header** (*tuple*) – Default: None.
Deprecated since version 2.1: See `request.url.scheme` for built-in resolution of the current scheme using the standard and de-facto standard headers.
- **tcp_keepalive** (*bool*) – Enable TCP Keep-Alive. Default: True.
- **keepalive_timeout** (*int*) – Number of seconds before closing Keep-Alive connection. Default: 75 seconds (NGINX’s default value).
- **slow_request_timeout** – Slow request timeout. Default: 0.
- **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`.
- **debug** (*bool*) – Switches debug mode. Default: False.
Deprecated since version 1.0: The usage of `debug` parameter in `Application.make_handler()` is deprecated in favor of `Application.debug`. The `Application`’s debug mode setting should be used as a single point to setup a debug mode.
- **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 ignore additional data coming from the client when lingering close is on. Use 0 for disabling lingering on server channel closing.
- **lingering_timeout** (*float*) – maximum waiting time for more client data to arrive when lingering close is in effect

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

```
loop = asyncio.get_event_loop()

app = Application(loop=loop)

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

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

coroutine startup()

A *coroutine* that will be called along with the application’s request handler.

The purpose of the method is calling *on_startup* signal handlers.

coroutine `shutdown()`

A *coroutine* that should be called on server stopping but before *cleanup()*.

The purpose of the method is calling *on_shutdown* signal handlers.

coroutine `cleanup()`

A *coroutine* that should be called on server stopping but after *shutdown()*.

The purpose of the method is calling *on_cleanup* signal handlers.

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

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

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

Server

A protocol factory compatible with *create_server()*.

class `aiohhttp.web.Server`

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

`Server.connections`

List of all currently opened connections.

`aiohhttp.web.requests_count`

Amount of processed requests.

New in version 1.0.

coroutine `Server.shutdown(timeout)`

A *coroutine* that should be called to close all opened connections.

coroutine `Server.finish_connections(timeout)`

Deprecated since version 1.2: A deprecated alias for *shutdown()*.

Changed in version 1.2: *Server* was called *RequestHandlerFactory* before *aiohhttp==1.2*.

The rename has no deprecation period but it's safe: no user should instantiate the class by hands.

Router

For dispatching URLs to *handlers* *aiohhttp.web* uses *routers*.

Router is any object that implements *AbstractRouter* interface.

aiohhttp.web provides an implementation called *UrlDispatcher*.

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

class `aiohhttp.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* is converted to coroutine internally when it is a regular function.

Parameters

- **method** (*str*) – HTTP method for route. Should be one of `'GET'`, `'POST'`, `'PUT'`, `'DELETE'`, `'PATCH'`, `'HEAD'`, `'OPTIONS'` or `'*'` for any method.
The parameter is case-insensitive, e.g. you can push `'get'` as well as `'GET'`.
- **path** (*str*) – route path. Should be started with slash (`'/'`).
- **handler** (*callable*) – route handler.
- **name** (*str*) – optional route name.
- **expect_handler** (*coroutine*) – optional *expect* header handler.

Returns new `PlainRoute` or `DynamicRoute` instance.

add_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'`.

New in version 1.0.

Changed in version 2.0: *allow_head* parameter added.

add_post (*path*, *handler*, ***kwargs*)

Shortcut for adding a POST handler. Calls the `add_route ()` with

method equals to `'POST'`.

New in version 1.0.

add_head (*path, handler, **kwargs*)

Shortcut for adding a HEAD handler. Calls the `add_route()` with `method` equals to 'HEAD'.

New in version 1.0.

add_put (*path, handler, **kwargs*)

Shortcut for adding a PUT handler. Calls the `add_route()` with `method` equals to 'PUT'.

New in version 1.0.

add_patch (*path, handler, **kwargs*)

Shortcut for adding a PATCH handler. Calls the `add_route()` with `method` equals to 'PATCH'.

New in version 1.0.

add_delete (*path, handler, **kwargs*)

Shortcut for adding a DELETE handler. Calls the `add_route()` with `method` equals to 'DELETE'.

New in version 1.0.

add_static (*prefix, path, *, name=None, expect_handler=None, chunk_size=256*1024, response_factory=StreamResponse, show_index=False, follow_symlinks=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`.

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

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

Changed in version 1.2.0: Send gzip version if file path + `.gz` exists.

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.

New in version 0.16.

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

New in version 0.17.

- **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.

Returns new *StaticRoute* instance.

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.

New in version 1.1.

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:

```
len(app.router.resources())
```

2. Iterate over registered resources:

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

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

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

New in version 0.21.1.

routes ()

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

New in version 0.18.

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**):

```
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"]
```

New in version 0.21.

named_routes ()

An alias for `named_resources ()` starting from aiohttp 0.21.

New in version 0.19.

Changed in version 0.21: The method is an alias for `named_resources ()`, so it iterates over resources instead of routes.

Deprecated since version 0.21: Please use named **resources** instead of named **routes**.

Several routes which belongs to the same resource shares the resource 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:

```
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:


```
url = app.router['resource_name'].url_for().with_query({'a': 1, 'b': 2})
```

Resource classes hierarchy:

```
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`.

coroutine `resolve` (*method*, *path*)

Resolve resource by finding appropriate *web-handler* for (*method*, *path*) combination.

Parameters

- **method** (*str*) – requested HTTP method.
- **path** (*str*) – *path* part of request.

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.

New in version 1.1.

url (***kwargs*)

Construct an URL for route with additional params.

kwargs depends on a list accepted by inherited resource class parameters.

Returns `str` – resulting URL string.

Deprecated since version 1.1: Use `url_for()` instead.

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.

url_for ()

Returns a URL for the resource.

New in version 1.1.

class aiohttp.web.**DynamicResource**

A resource, inherited from *Resource*.

The class corresponds to resources with *variable* matching, e.g. '/path/{to}/{param}' etc.

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')

New in version 1.1.

class aiohttp.web.**StaticResource**

A resource, inherited from *Resource*.

The class corresponds to resources for *static file serving*.

url_for (*filename*)

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')

New in version 1.1.

class aiohttp.web.**PrefixedSubAppResource**

A resource for serving nested applications. The class instance is returned by *add_subapp* call.

New in version 1.1.

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 (*args, **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

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

    async def post(self):
        resp = await post_response(self.request)
        return resp

app.router.add_route('*', '/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](#)

11.6.5 Utilities

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](#)

```
aiohttp.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_format=None, access_log=aiohttp.log.access_logger, handle_signals=True, loop=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 function uses `app.loop` as event loop to run.

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 *aiohttp server deployment* for ways of distributing work for increased performance.

Parameters

- **app** – *Application* instance to run
- **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.
- **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.
- **loop** – an *event loop* used for running the application (`None` by default).

If the loop is not explicitly specified the function closes it by `close()` call but **does nothing** for **non-default** loop.

11.6.6 Constants

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

deflate
DEFLATE compression

gzip
GZIP compression

`aiohttp.web.identity`
no compression

11.6.7 Middlewares

Normalize path middleware

`aiohttp.web.normalize_path_middleware` (*, *append_slash=True, merge_slashes=True*)

Middleware that normalizes the path of a request. By normalizing it means:

- Add 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 all enabled is 1) `merge_slashes`, 2) `append_slash` and 3) both `merge_slashes` and `append_slash`. If the path resolves with at least one of those conditions, it will redirect to the new path.

If `append_slash` is `True` append slash when needed. If a resource is defined with trailing slash and the request comes without it, it will append it automatically.

If `merge_slashes` is `True`, merge multiple consecutive slashes in the path into one.

11.7 Low Level Server

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

11.7.1 Abstract

Sometimes user don't need high-level concepts introduced in *Server Usage*: 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()` / `StreamResponse.drain()`, return finished response.

3. Raise `HTTPException` derived exception (see *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*).

11.7.2 Run a Basic Low-Level Server

The following code demonstrates very trivial usage example:

```
import asyncio
from aiohttp import web

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

async def main(loop):
    server = web.Server(handler)
    await loop.create_server(server, "127.0.0.1", 8080)
    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(loop))
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 `loop.create_server` call to serve `http://127.0.0.1:8080/`.

The handler should process every request: GET, POST, Web-Socket for every *path*.

The example is very basic: it always return 200 OK response, real life code should be much more complex.

11.8 Abstract Base Classes

11.8.1 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 *pluggable* 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.

class `aiohttp.AbstractRouter`

Abstract router, `aiohttp.web.Application` accepts it as `router` parameter and returns as `aiohttp.web.Application.router`.

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.

class `aiohttp.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.

11.8.2 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.

class `aiohttp.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.

11.8.3 Abstract Cookie Jar

class `aiohttp.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:

```
len(session.cookie_jar)
```

These cookies may be iterated over:


```
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`.

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.

11.9 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.

11.9.1 Reading Multipart Responses

Assume you made a request, as usual, and want to process the response multipart data:

```
async with aiohttp.request(...) as resp:
    pass
```

First, you need to wrap the response with a `MultipartReader.from_response()`. This needs to keep the implementation of `MultipartReader` separated from the response and the connection routines which makes it more portable:

```
reader = aiohttp.MultipartReader.from_response(resp)
```

Let’s assume with this response you’d received some JSON document and multiple files for it, but you don’t need all of them, just a specific one.

So first you need to enter into a loop where the multipart body will be processed:

```
metadata = None
filedata = None
while True:
    part = await reader.next()
```

The returned type depends on what the next part is: if it's a simple body part then you'll get `BodyPartReader` instance here, otherwise, it will be another `MultipartReader` instance for the nested multipart. Remember, that multipart format is recursive and supports multiple levels of nested body parts. When there are no more parts left to fetch, `None` value will be returned - that's the signal to break the loop:

```
if part is None:
    break
```

Both `BodyPartReader` and `MultipartReader` provides access to body part headers: this allows you to filter parts by their attributes:

```
if part.headers[aiohhttp.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:

```
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:

```
filedata = await part.read(decode=False)
```

Later you may decide to decode the data. It's still simple and possible to do:

```
filedata = part.decode(filedata)
```

Once you are done with multipart processing, just break a loop:

```
break
```

11.9.2 Sending Multipart Requests

`MultipartWriter` provides an interface to build multipart payload from the Python data and serialize it into chunked binary stream. Since multipart format is recursive and supports deeply nesting, you can use `with` statement

to design your multipart data closer to how it will be:

```
with aiohttp.MultipartWriter('mixed') as mpwriter:
    ...
    with aiohttp.MultipartWriter('related') as subwriter:
        ...
        mpwriter.append(subwriter)

    with aiohttp.MultipartWriter('related') as subwriter:
        ...
        with aiohttp.MultipartWriter('related') as subsubwriter:
            ...
            subwriter.append(subsubwriter)
        mpwriter.append(subwriter)

    with aiohttp.MultipartWriter('related') as subwriter:
        ...
        mpwriter.append(subwriter)
```

The `MultipartWriter.append()` is used to join new body parts into a single stream. It accepts various inputs and determines what default headers should be used for.

For text data default *Content-Type* is `text/plain; charset=utf-8`:

```
mpwriter.append('hello')
```

For binary data `application/octet-stream` is used:

```
mpwriter.append(b'aiohttp')
```

You can always override these default by passing your own headers with the second argument:

```
mpwriter.append(io.BytesIO(b'GIF89a...'),
                {'CONTENT-TYPE': 'image/gif'})
```

For file objects *Content-Type* will be determined by using Python's `mimetypes` module and additionally *Content-Disposition* header will include the file's basename:

```
part = root.append(open(__file__, 'rb'))
```

If you want to send a file with a different name, just handle the `BodyPartWriter` instance which `MultipartWriter.append()` will always return and set *Content-Disposition* explicitly by using the `BodyPartWriter.set_content_disposition()` helper:

```
part.set_content_disposition('attachment', filename='secret.txt')
```

Additionally, you may want to set other headers here:

```
part.headers[aiohttp.hdrs.CONTENT_ID] = 'X-12345'
```

If you'd set *Content-Encoding*, it will be automatically applied to the data on serialization (see below):

```
part.headers[aiohttp.hdrs.CONTENT_ENCODING] = 'gzip'
```

There are also `MultipartWriter.append_json()` and `MultipartWriter.append_form()` helpers which are useful to work with JSON and form urlencoded data, so you don't have to encode it every time manually:

```
mpwriter.append_json({'test': 'passed'})
mpwriter.append_form([('key', 'value')])
```

When it's done, to make a request just pass a root `MultipartWriter` instance as `aiohttp.client.request()` `data` argument:

```
await aiohttp.post('http://example.com', data=mpwriter)
```

Behind the scenes `MultipartWriter.serialize()` will yield chunks of every part and if body part has `Content-Encoding` or `Content-Transfer-Encoding` they will be applied on streaming content.

Please note, that on `MultipartWriter.serialize()` all the file objects will be read until the end and there is no way to repeat a request without rewinding their pointers to the start.

11.9.3 Hacking Multipart

The Internet is full of terror and sometimes you may find a server which implements multipart support in strange ways when an oblivious solution does not work.

For instance, is server used `cgi.FieldStorage` then you have to ensure that no body part contains a `Content-Length` header:

```
for part in mpwriter:
    part.headers.pop(aiohttp.hdrs.CONTENT_LENGTH, None)
```

On the other hand, some server may require to specify `Content-Length` for the whole multipart request. `aiohttp` 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:

```
body = b''.join(mpwriter.serialize())
await aiohttp.post('http://example.com',
                  data=body, headers=mpwriter.headers)
```

Sometimes the server response may not be well formed: it may or may not contains nested parts. For instance, we request a resource which returns JSON documents with the files attached to it. If the document has any attachments, they are returned as a nested multipart. If it has not it responds as plain body parts:

```
CONTENT-TYPE: multipart/mixed; boundary=--:
--:
CONTENT-TYPE: application/json

{"_id": "foo"}
--:
CONTENT-TYPE: multipart/related; boundary=----:
----:
CONTENT-TYPE: application/json

{"_id": "bar"}
----:
CONTENT-TYPE: text/plain
CONTENT-DISPOSITION: attachment; filename=bar.txt

bar! bar! bar!
----:--
```

```

--:
CONTENT-TYPE: application/json

{"_id": "boo"}
--:
CONTENT-TYPE: multipart/related; boundary=----:

----:
CONTENT-TYPE: application/json

{"_id": "baz"}
----:
CONTENT-TYPE: text/plain
CONTENT-DISPOSITION: attachment; filename=baz.txt

baz! baz! baz!
----:--
--:--

```

Reading such kind of data in single stream is possible, but is not clean at all:

```

result = []
while True:
    part = await reader.next()

    if part is None:
        break

    if isinstance(part, aiohttp.MultipartReader):
        # Fetching files
        while True:
            filepart = await part.next()
            if filepart is None:
                break
            result[-1].append((await filepart.read()))

    else:
        # Fetching document
        result.append([await part.json()])

```

Let's hack a reader in the way to return pairs of document and reader of the related files on each iteration:

```

class PairsMultipartReader(aiohttp.MultipartReader):

    # keep reference on the original reader
    multipart_reader_cls = aiohttp.MultipartReader

    async def next(self):
        """Emits a tuple of document object (:class:`dict`) and multipart
        reader of the followed attachments (if any).

        :rtype: tuple
        """
        reader = await super().next()

        if self._at_eof:
            return None, None

```

```

if isinstance(reader, self.multipart_reader_cls):
    part = await reader.next()
    doc = await part.json()
else:
    doc = await reader.json()

return doc, reader

```

And this gives us a more cleaner solution:

```

reader = PairsMultipartReader.from_response(resp)
result = []
while True:
    doc, files_reader = await reader.next()

    if doc is None:
        break

    files = []
    while True:
        filepart = await files_reader.next()
        if filepart is None:
            break
        files.append((await filepart.read()))

    result.append((doc, files))

```

See also:

Multipart API in *Helpers API* section.

11.10 Streaming API

aiohttp uses streams for retrieving *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 BODY data.

11.10.1 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* (*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 `asyncio.IncompleteReadError.partial` attribute of the exception contains the partial read bytes.

Parameters *n* (*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 `\n`.

If EOF is received, and `\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

11.10.2 Asynchronous Iteration Support

Stream reader supports asynchronous iteration over BODY.

By default it iterates over lines:

```
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.iter_chunked(n)`

Iterates over data chunks with maximum size limit:

```
async for data in response.content.iter_chunked(1024):
    print(data)
```

async-for `StreamReader.iter_any()`

Iterates over data chunks in order of intaking them into the stream:

```
async for data in response.content.iter_any():
    print(data)
```

async-for `StreamReader.iter_chunks()`

Iterates over data chunks as received from the server:

```
async for data in response.content.iter_chunks():
    print(data)
```

11.10.3 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 `aiohhttp.wait_eof()`

Wait for EOF. The given data may be accessible by upcoming read calls.

11.11 Helpers API

All public names from submodules `client`, `multipart`, `protocol` and `utils` are exported into `aiohhttp` namespace.

11.11.1 WebSocket utilities

class `aiohhttp.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).

INVALID_TEXT

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).

POLICY_VIOLATION

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.

MESSAGE_TOO_BIG

An endpoint is terminating the connection because it has received a message that is too big for it to process.

MANDATORY_EXTENSION

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.

INTERNAL_ERROR

A server is terminating the connection because it encountered an unexpected condition that prevented it from fulfilling the request.

SERVICE_RESTART

The service is restarted. a client may reconnect, and if it chooses to do, should reconnect using a randomized delay of 5-30s.

TRY_AGAIN_LATER

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.

New in version 0.22.

Parameters **loads** – optional JSON decoder function.

tp

Deprecated alias for *type*.

Deprecated since version 1.0.

11.11.2 aiohttp.helpers module

Various helper functions

`aiohttp.helpers.create_future(loop)`

`aiohttp.helpers.parse_mimetype(mimetype)`

Parses a MIME type into its components.

Parameters **mimetype** (*str*) – MIME type

Returns 4 element tuple for MIME type, subtype, suffix and parameters

Return type *tuple*

Example:

```
>>> parse_mimetype('text/html; charset=utf-8')
('text', 'html', '', {'charset': 'utf-8'})
```

`aiohttp.helpers.Timeout`

alias of `timeout`

`aiohttp.helpers.ensure_future(coro_or_future, *, loop=None)`

Wrap a coroutine or an awaitable in a future.

If the argument is a Future, it is returned directly.

coroutine `aiohttp.helpers.noop(*args, **kwargs)`

class `aiohttp.helpers.DummyCookieJar(*, loop=None)`

Bases: `aiohttp.abc.AbstractCookieJar`

Implements a dummy cookie storage.

It can be used with the `ClientSession` when no cookie processing is needed.

clear ()

filter_cookies (*request_url*)

update_cookies (*cookies, response_url=None*)

11.11.3 aiohttp.multipart module

class `aiohttp.multipart.MultipartReader(headers, content)`

Bases: `object`

Multipart body reader.

at_eof()
Returns True if the final boundary was reached or False otherwise.
Return type bool

coroutine fetch_next_part()
Returns the next body part reader.

classmethod from_response(response)
Constructs reader instance from HTTP response.
Parameters response – ClientResponse instance

multipart_reader_cls = None
Multipart reader class, used to handle multipart/* body parts. None points to type(self)

coroutine next()
Emits the next multipart body part.

part_reader_cls
Body part reader class for non multipart/* content types.
alias of *BodyPartReader*

coroutine release()
Reads all the body parts to the void till the final boundary.

response_wrapper_cls
Response wrapper, used when multipart readers constructs from response.
alias of *MultipartResponseWrapper*

class aiohttp.multipart.MultipartWriter (*subtype='mixed', boundary=None*)
Bases: *aiohttp.payload.Payload*
Multipart body writer.
append(obj, headers=None)
append_form(obj, headers=None)
Helper to append form urlencoded part.
append_json(obj, headers=None)
Helper to append JSON part.
append_payload(payload)
Adds a new body part to multipart writer.
boundary
size
Size of the payload.
coroutine write(writer)
Write body.

class aiohttp.multipart.BodyPartReader (*boundary, headers, content*)
Bases: *object*
Multipart reader for single body part.
at_eof()
Returns True if the boundary was reached or False otherwise.
Return type bool
chunk_size = 8192

decode (*data*)

Decodes data according the specified *Content-Encoding* or *Content-Transfer-Encoding* headers value.

Supports `gzip`, `deflate` and `identity` encodings for *Content-Encoding* header.

Supports `base64`, `quoted-printable`, `binary` encodings for *Content-Transfer-Encoding* header.

Parameters `data` (*bytearray*) – Data to decode.

Raises `RuntimeError` - if encoding is unknown.

Return type `bytes`

filename

Returns filename specified in *Content-Disposition* header or `None` if missed or header is malformed.

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

get_charset (*default=None*)

Returns `charset` parameter from *Content-Type* header or default.

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

name

Returns filename specified in *Content-Disposition* header or `None` if missed or header is malformed.

coroutine next ()**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 `bytearray`

coroutine read_chunk (*size=8192*)

Reads body part content chunk of the specified size.

Parameters `size` (*int*) – chunk size

Return type `bytearray`

coroutine readline ()

Reads body part by line by line.

Return type `bytearray`

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`

exception aiohttp.multipart.BadContentDispositionHeader

Bases: `RuntimeWarning`

exception `aiohttp.multipart.BadContentDispositionParam`

Bases: `RuntimeWarning`

`aiohttp.multipart.parse_content_disposition` (*header*)

`aiohttp.multipart.content_disposition_filename` (*params, name='filename'*)

11.11.4 aiohttp.signals module

class `aiohttp.signals.BaseSignal`

Bases: `aiohttp._frozenlist.FrozenList`

class `aiohttp.signals.DebugSignal`

Bases: `aiohttp.signals.BaseSignal`

coroutine `send` (*ordinal, name, *args, **kwargs*)

class `aiohttp.signals.FuncSignal`

Bases: `aiohttp.signals.BaseSignal`

Callback-based signal implementation.

To connect a callback to a signal, use any list method.

Signals are fired using the `send()` method, which takes named arguments.

send (**args, **kwargs*)

Sends data to all registered receivers.

class `aiohttp.signals.PostSignal`

Bases: `aiohttp.signals.DebugSignal`

class `aiohttp.signals.PreSignal`

Bases: `aiohttp.signals.DebugSignal`

ordinal ()

class `aiohttp.signals.Signal` (*app*)

Bases: `aiohttp.signals.BaseSignal`

Coroutine-based signal implementation.

To connect a callback to a signal, use any list method.

Signals are fired using the `send()` coroutine, which takes named arguments.

coroutine `send` (**args, **kwargs*)

Sends data to all registered receivers.

11.12 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'`

- 'aiohhttp.web'
- 'aiohhttp.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.

11.12.1 Access logs

Access log by default is switched on and uses 'aiohhttp.access' logger name.

The log may be controlled by `aiohhttp.web.Application.make_handler()` call.

Pass `access_log` parameter with value of `logging.Logger` instance to override default logger.

Note: Use `app.make_handler(access_log=None)` for disabling access logs.

Other parameter called `access_log_format` may be used for specifying log format (see below).

Format specification

The library provides custom micro-language to specifying info about request and response:

Option	Meaning
%%	The percent sign
%a	Remote IP-address (IP-address of proxy if using reverse proxy)
%t	Time when the request was started to process
%P	The process ID of the child that serviced the request
%r	First line of request
%s	Response status code
%b	Size of response in bytes, excluding HTTP headers
%O	Bytes sent, including headers
%T	The time taken to serve the request, in seconds
%Tf	The time taken to serve the request, in seconds with fraction in <code>%.06f</code> format
%D	The time taken to serve the request, in microseconds
{FOO}i	<code>request.headers['FOO']</code>
{FOO}o	<code>response.headers['FOO']</code>
{FOO}e	<code>os.environ['FOO']</code>

Default access log format is:

```
'%a %l %u %t "%r" %s %b "%{Referrer}i" "%{User-Agent}i"'
```

Note: 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.

11.12.2 Error logs

`aiohttp.web` uses logger named `'aiohttp.server'` to store errors given on web requests handling.

The log is enabled by default.

To use different logger name please specify `logger` parameter (`logging.Logger` instance) on performing `aiohttp.web.Application.make_handler()` call.

11.13 Testing

11.13.1 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 are 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 encapsulates `aiohttp.ClientSession` by providing proxy methods to the client for common operations such as `ws_connect`, `get`, `post`, etc.

Pytest

The `test_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(test_client, loop):
    app = web.Application()
    app.router.add_get('/', hello)
    client = await test_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')
    return web.Response(
        body='value: {}'.format(request.app['value']).encode('utf-8'))

@pytest.fixture
def cli(loop, test_client):
    app = web.Application()
    app.router.add_get('/', previous)
    app.router.add_post('/', previous)
    return loop.run_until_complete(test_client(app))

async def test_set_value(cli):
    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:

`aiohttp.test_utils.test_server` (*app*, ***kwargs*)

A fixture factory that creates *TestServer*:

```

async def test_f(test_server):
    app = web.Application()
    # fill route table

    server = await test_server(app)

```


The server will be destroyed on exit from test function.

app is the `aiohttp.web.Application` used to start server.

kwargs are parameters passed to `aiohttp.web.Application.make_handler()`

```
aiohttp.test_utils.test_client(app, **kwargs)
```

```
aiohttp.test_utils.test_client(server, **kwargs)
```

```
aiohttp.test_utils.test_client(raw_server, **kwargs)
```

A fixture factory that creates `TestClient` for access to tested server:

```
async def test_f(test_client):
    app = web.Application()
    # fill route table

    client = await test_client(app)
    resp = await client.get('/')
```

`client` and responses are cleaned up after test function finishing.

The fixture accepts `aiohttp.web.Application`, `aiohttp.test_utils.TestServer` or `aiohttp.test_utils.RawTestServer` instance.

kwargs are parameters passed to `aiohttp.test_utils.TestClient` constructor.

```
aiohttp.test_utils.raw_test_server(handler, **kwargs)
```

A fixture factory that creates `RawTestServer` instance from given web handler.

`handler` should be a coroutine which accepts a request and returns response, e.g.:

```
async def test_f(raw_test_server, test_client):

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

    raw_server = await raw_test_server(handler)
    client = await test_client(raw_server)
    resp = await client.get('/')
```

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.
        """
        return web.Application()

    # 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):
```

```

request = await self.client.request("GET", "/")
assert request.status == 200
text = await request.text()
assert "Hello, world" in text

# a vanilla example
def test_example(self):
    async def test_get_route():
        url = root + "/"
        resp = await self.client.request("GET", url, loop=loop)
        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.

loop

The event loop in which the application and server are running.

app

The application returned by `get_app()` (*aiohttp.web.Application* instance).

coroutine get_application()

This async method should be overridden to return the *aiohttp.web.Application* object to test.

Returns *aiohttp.web.Application* instance.

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:

```

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*.

11.13.2 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:

```
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.

`aiohttp.test_utils.make_mocked_request` (*method*, *path*, *headers=None*, *, *version=HttpVersion(1, 1)*, *closing=False*, *app=None*, *reader=sentinel*, *writer=sentinel*, *transport=sentinel*, *payload=sentinel*, *sslcontext=None*, *secure_proxy_ssl_header=None*)

Creates 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.
- **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** – object for managing outgoing data
- **transport** (*asyncio.transports.Transport*) – asyncio transport instance
- **payload** (*aiohttp.streams.FlowControlStreamReader*) – raw payload reader object
- **sslcontext** (*ssl.SSLContext*) – ssl.SSLContext object, for HTTPS connection

- `secure_proxy_ssl_header` (*tuple*) – A tuple representing a HTTP header/value combination that signifies a request is secure.

Returns `aiohttp.web.Request` object.

Framework Agnostic Utilities

High level test creation:

```
from aiohttp.test_utils import TestClient, loop_context
from aiohttp import request

# loop_context is provided as a utility. You can use any
# asyncio.BaseEventLoop class in it's place.
with loop_context() as loop:
    app = _create_example_app()
    with TestClient(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

loop.run_until_complete(test_get_route())
```

If it's preferred to handle the creation / teardown on a more granular basis, the `TestClient` object can be used directly:

```
from aiohttp.test_utils import TestClient

with loop_context() as loop:
    app = _create_example_app()
    client = TestClient(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](#)

11.13.3 Writing testable services

Some libraries like `motor`, `aios` and others depend on the `asyncio` loop for executing the code. When running your normal program, these libraries pick the main event loop by doing `asyncio.get_event_loop`. The problem during testing is that there is no main loop assigned because an independent loop for each test is created without assigning it as the main one.

This raises a problem when those libraries try to find it. Luckily, the ones that are well written, allow passing the loop explicitly. Let's have a look at the `aios` client signature:

```
def __init__(self, endpoints, *, loop=None, **kwargs)
```

As you can see, there is an optional `loop` kwarg. Of course, we are not going to test directly the aioes client but our service that depends on it will. So, if we want our `AioESService` to be easily testable, we should define it as follows:

```
import asyncio

from aioes import Elasticsearch

class AioESService:

    def __init__(self, loop=None):
        self.es = Elasticsearch(["127.0.0.1:9200"], loop=loop)

    async def get_info(self):
        cluster_info = await self.es.info()
        print(cluster_info)

if __name__ == "__main__":
    client = AioESService()
    loop = asyncio.get_event_loop()
    loop.run_until_complete(client.get_info())
```

Note that it is accepting an optional `loop` kwarg. For the normal flow of execution it won't affect because we can still call the service without passing the loop explicitly having a main loop available. The problem comes when you try to do a test like:

```
import pytest

from main import AioESService

class TestAioESService:

    async def test_get_info(self):
        cluster_info = await AioESService().get_info()
        assert isinstance(cluster_info, dict)
```

If you try to run the test, it will fail with a similar error:

```
...
RuntimeError: There is no current event loop in thread 'MainThread'.
```

If you check the stack trace, you will see aioes is complaining that there is no current event loop in the main thread. Pass explicit loop to solve it.

If you rely on code which works with *implicit* loops only you may try to use hackish approach from [FAQ](#).

11.13.4 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 `aiohhttp.test_utils.TestClient` which provides handy client methods for accessing to the server.

class `aiohhttp.test_utils.BaseTestServer` (*, *scheme*='http', *host*='127.0.0.1')

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.

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

A random *port* used to start a server.

handler

`aiohhttp.web.WebServer` used for HTTP requests serving.

server

`asyncio.AbstractServer` used for managing accepted connections.

coroutine `start_server` (*loop*=None, ***kwargs*)

Parameters `loop` (`asyncio.AbstractEventLoop`) – the *event_loop* to use

Start a test server.

coroutine `close` ()

Stop and finish executed test server.

make_url (*path*)

Return an *absolute URL* for given *path*.

class `aiohhttp.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 `aiohhttp.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.

class `aiohhttp.test_utils.TestServer` (*app*, *, *scheme*="http", *host*='127.0.0.1')

Test server (derived from `BaseTestServer`) for starting `Application`.

Parameters

- **app** – `aiohhttp.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.

app

`aiohhttp.web.Application` instance to run.

Test Client

```
class aiohttp.test_utils.TestClient (app_or_server, *, loop=None, 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.
If the parameter is *aiohttp.web.Application* the tool creates *TestServer* implicitly for serving the application.
- **cookie_jar** – an optional *aiohttp.CookieJar* instance, may be useful with *CookieJar(unsafe=True)* option.
- **scheme** (*str*) – HTTP scheme, non-protected "http" by default.
- **loop** (*asyncio.AbstractEventLoop*) – the event_loop to use
- **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

A random *port* used to start a server.

server

BaseTestServer test server instance used in conjunction with client.

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 *asyncio.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:

```
mocked = make_mocked_coro(1)
assert 1 == await mocked(1, 2)
mocked.assert_called_with(1, 2)
```

Parameters *return_value* – A value that the 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.

`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

11.14 aiohttp 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.

11.14.1 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*.

11.14.2 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.

Ngix 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*:

```
http {
    upstream aiohttp {
        # fail_timeout=0 means we always retry an upstream even if it failed
        # to return a good HTTP response

        # Unix domain servers
        server unix:/tmp/example_1.sock fail_timeout=0;
        server unix:/tmp/example_2.sock fail_timeout=0;
        server unix:/tmp/example_3.sock fail_timeout=0;
        server unix:/tmp/example_4.sock fail_timeout=0;

        # Unix domain sockets are used in this example due to their high performance,
        # but TCP/IP sockets could be used instead:
        # server 127.0.0.1:8081 fail_timeout=0;
        # server 127.0.0.1:8082 fail_timeout=0;
        # server 127.0.0.1:8083 fail_timeout=0;
        # server 127.0.0.1:8084 fail_timeout=0;
    }
}
```

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:

```
[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:

```
# 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.

11.14.3 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 supervisor 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 14.04*.

Create a directory for your application:

```
>> mkdir myapp
>> cd myapp
```

Ubuntu has a bug in pyenv, so to create virtualenv you need to do some extra manipulation:

```
>> pyenv-3.4 --without-pip venv
>> source venv/bin/activate
>> curl https://bootstrap.pypa.io/get-pip.py | python
>> deactivate
>> source venv/bin/activate
```

Now that the virtual environment is ready, we'll proceed to install aiohttp and gunicorn:

```
>> pip install gunicorn
>> pip install -e git+https://github.com/aio-libs/aiohttp.git#egg=aiohttp
```

Application

Lets write a simple application, which we will save to file. We'll name this file `my_app_module.py`:

```
from aiohttp import web

def index(request):
    return web.Response(text="Welcome home!")

my_web_app = web.Application()
my_web_app.router.add_get('/', index)
```

Start Gunicorn

When [Running Gunicorn](#), you provide the name of the module, i.e. *my_app_module*, and the name of the app, i.e. *my_web_app*, along with other [Gunicorn Settings](#) provided as command line flags or in your config file.

In this case, we will use:

- the `–bind` flag to set the server’s socket address;
- the `–worker-class` flag to tell Gunicorn that we want to use a custom worker subclass instead of one of the Gunicorn default worker types;
- you may also want to use the `–workers` flag to tell Gunicorn how many worker processes to use for handling requests. (See the documentation for recommendations on [How Many Workers?](#))

The custom worker subclass is defined in *aiohttp.GunicornWebWorker* and should be used instead of the *gunicorn* worker provided by Gunicorn, which supports only aiohttp.wsgi applications:

```
>> gunicorn my_app_module:my_web_app --bind localhost:8080 --worker-class aiohttp.
↪GunicornWebWorker
[2015-03-11 18:27:21 +0000] [1249] [INFO] Starting gunicorn 19.3.0
[2015-03-11 18:27:21 +0000] [1249] [INFO] Listening at: http://127.0.0.1:8080 (1249)
[2015-03-11 18:27:21 +0000] [1249] [INFO] Using worker: aiohttp.worker.
↪GunicornWebWorker
[2015-03-11 18:27:21 +0000] [1253] [INFO] Booting worker with pid: 1253
```

Gunicorn is now running and ready to serve requests to your app’s worker processes.

Note: If you want to use an alternative asyncio event loop *uvloop*, you can use the *aiohttp.GunicornUVLoopWebWorker* worker class.

More information

The Gunicorn documentation recommends deploying Gunicorn behind an Nginx proxy server. See the [official documentation](#) for more information about suggested nginx configuration.

Logging configuration

aiohttp and *gunicorn* use different format for specifying access log.

By default *aiohttp* uses own defaults:

```
'%a %l %u %t "%r" %s %b "%{Referrer}i" "%{User-Agent}i"'
```

For more information please read *Format Specification for Access Log*.

11.15 Frequently Asked Questions

- *Are there any plans for @app.route decorator like in Flask?*
- *Has aiohttp the Flask Blueprint or Django App concept?*
- *How to create route that catches urls with given prefix?*
- *Where to put my database connection so handlers can access it?*
- *Why the minimal supported version is Python 3.4.2*
- *How a middleware may store a data for using by web-handler later?*
- *How to receive an incoming events from different sources in parallel?*
- *How to programmatically close websocket server-side?*
- *How to make request from a specific IP address?*
- *How to use aiohttp test features with code which works with implicit loop?*
- *API stability and deprecation policy*
- *How to enable gzip compression globally for the whole application?*

11.15.1 Are there any plans for @app.route decorator like in Flask?

There are couple issues here:

- This adds huge problem name “configuration as side effect of importing”.
- Route matching is order specific, it is very hard to maintain import order.
- In semi large application better to have routes table defined in one place.

For this reason feature will not be implemented. But if you really want to use decorators just derive from `web.Application` and add desired method.

11.15.2 Has aiohttp the Flask Blueprint or Django App concept?

If you’re planing to write big applications, maybe you must consider use nested applications. They acts as a Flask Blueprint or like the Django application concept.

Using nested application you can add sub-applications to the main application.

see: *Nested applications*.

11.15.3 How to create route that catches urls with given prefix?

Try something like:

```
app.router.add_route('*', '/path/to/{tail:.+}', sink_handler)
```

Where first argument, star, means catch any possible method (*GET, POST, OPTIONS*, etc), second matching url with desired prefix, third – handler.

11.15.4 Where to put my database connection so handlers can access it?

`aiohttp.web.Application` object supports `dict` interface, and right place to store your database connections or any other resource you want to share between handlers. Take a look on following example:

```

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(loop):
    app = Application(loop=loop)
    db = await create_connection(user='user', password='123')
    app['db'] = db
    app.router.add_get('/', go)
    return app

```

11.15.5 Why the minimal supported version is Python 3.4.2

As of aiohttp **v0.18.0** we dropped support for Python 3.3 up to 3.4.1. The main reason for that is the `object.__del__()` method, which is fully working since Python 3.4.1 and we need it for proper resource closing.

The last Python 3.3, 3.4.0 compatible version of aiohttp is **v0.17.4**.

This should not be an issue for most aiohttp users (for example *Ubuntu* 14.04.3 LTS provides python upgraded to 3.4.3), however libraries depending on aiohttp should consider this and either freeze aiohttp version or drop Python 3.3 support as well.

As of aiohttp **v1.0.0** we dropped support for Python 3.4.1 up to 3.4.2+ also. The reason is: `loop.is_closed` appears in 3.4.2+

Again, it should be not an issue at 2016 Summer because all major distributions are switched to Python 3.5 now.

11.15.6 How a middleware may store a data for using by web-handler later?

`aiohttp.web.Request` supports `dict` interface as well as `aiohttp.web.Application`.

Just put data inside `request`:

```

async def handler(request):
    request['unique_key'] = data

```

See https://github.com/aio-libs/aiohttp_session code for inspiration, `aiohttp_session.get_session(request)` method uses `SESSION_KEY` for saving request specific session info.

11.15.7 How to receive an incoming events from different sources in parallel?

For example we have two event sources:

1. WebSocket for event from end user
2. Redis PubSub from receiving events from other parts of app for sending them to user via websocket.

The most native way to perform it is creation of separate task for pubsub handling.

Parallel `aiohttp.web.WebSocketResponse.receive()` calls are forbidden, only the single task should perform websocket reading.

But other tasks may use the same websocket object for sending data to peer:

```

async def handler(request):

    ws = web.WebSocketResponse()
    await ws.prepare(request)
    task = request.app.loop.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_message(msg)
            ws.send_str(answer)
    finally:
        await redis.unsubscribe('channel:1')

```

11.15.8 How to programmatically close websocket server-side?

For example we have an application with two endpoints:

1. `/echo` a websocket echo server that authenticates the user somehow
2. `/logout_user` that when invoked needs to close all open websockets for that user.

Keep in mind that you can only `.close()` a websocket from inside the handler task, and since the handler task is busy reading from the websocket, it can't react to other events.

One simple solution is keeping a shared registry of websocket handler tasks for a user in the `aiohttp.web.Application` instance and `cancel()` them in `/logout_user` handler:

```

async def echo_handler(request):

    ws = web.WebSocketResponse()
    user_id = authenticate_user(request)
    await ws.prepare(request)
    request.app['websockets'][user_id].add(asyncio.Task.current_task())

    try:
        async for msg in ws:
            # handle incoming messages
            ...

```

```

except asyncio.CancelledError:
    print('websocket cancelled')
finally:
    request.app['websockets'][user_id].remove(asyncio.Task.current_task())
await ws.close()
return ws

async def logout_handler(request):

    user_id = authenticate_user(request)

    for task in request.app['websockets'][user_id]:
        task.cancel()

    # return response
    ...

def main():
    loop = asyncio.get_event_loop()
    app = aiohttp.web.Application(loop=loop)
    app.router.add_route('GET', '/echo', echo_handler)
    app.router.add_route('POST', '/logout', logout_handler)
    app['websockets'] = defaultdict(set)
    aiohttp.web.run_app(app, host='localhost', port=8080)

```

11.15.9 How to make request from a specific IP address?

If your system has several IP interfaces you may choose one which will be used to bind socket locally:

```

conn = aiohttp.TCPConnector(local_addr=('127.0.0.1', 0), loop=loop)
async with aiohttp.ClientSession(connector=conn) as session:
    ...

```

See also:

aiohttp.TCPConnector and *local_addr* parameter.

11.15.10 How to use aiohttp test features with code which works with implicit loop?

Passing explicit loop everywhere is the recommended way. But sometimes, in case you have many nested non well-written services, this is impossible.

There is a technique based on monkey-patching your low level service that depends on aioes, to inject the loop at that level. This way, you just need your *AioESService* with the loop in its signature. An example would be the following:

```

import pytest

from unittest.mock import patch, MagicMock

from main import AioESService, create_app

class TestAcceptance:

    async def test_get(self, test_client, loop):

```



```

with patch("main.AioESService", MagicMock(
    side_effect=lambda *args, **kwargs: AioESService(*args,
                                                    **kwargs,
                                                    loop=loop)):

    client = await test_client(create_app)
    resp = await client.get("/")
    assert resp.status == 200

```

Note how we are patching the `AioESService` with an instance of itself but adding the explicit loop as an extra (you need to load the loop fixture in your test signature).

The final code to test all this (you will need a local instance of elasticsearch running):

```

import asyncio

from aioes import Elasticsearch
from aiohttp import web

class AioESService:

    def __init__(self, loop=None):
        self.es = Elasticsearch(["127.0.0.1:9200"], loop=loop)

    async def get_info(self):
        return await self.es.info()

class MyService:

    def __init__(self):
        self.aioes_service = AioESService()

    async def get_es_info(self):
        return await self.aioes_service.get_info()

async def hello_aioes(request):
    my_service = MyService()
    cluster_info = await my_service.get_es_info()
    return web.Response(text="{}".format(cluster_info))

def create_app(loop=None):

    app = web.Application(loop=loop)
    app.router.add_route('GET', '/', hello_aioes)
    return app

if __name__ == "__main__":
    web.run_app(create_app())

```

And the full tests file:

```

from unittest.mock import patch, MagicMock

from main import AioESService, create_app

```

```

class TestAioESService:

    async def test_get_info(self, loop):
        cluster_info = await AioESService("random_arg", loop=loop).get_info()
        assert isinstance(cluster_info, dict)

class TestAcceptance:

    async def test_get(self, test_client, loop):
        with patch("main.AioESService", MagicMock(
            side_effect=lambda *args, **kwargs: AioESService(*args,
                                                                **kwargs,
                                                                loop=loop))):

            client = await test_client(create_app)
            resp = await client.get("/")
            assert resp.status == 200

```

Note how we are using the `side_effect` feature for injecting the loop to the `AioESService.__init__` call. The use of `**args, **kwargs` is mandatory in order to propagate the arguments being used by the caller.

11.15.11 API stability and deprecation policy

aiohttp tries to not break existing users code.

Obsolete attributes and methods are marked as *deprecated* in documentation and raises `DeprecationWarning` on usage.

Deprecation period is usually a year and half.

After the period is passed out deprecated code is be removed.

Unfortunately we should break own rules if new functionality or bug fixing forces us to do it (for example proper cookies support on client side forced us to break backward compatibility twice).

All *backward incompatible* changes are explicitly marked in *CHANGES* chapter.

11.15.12 How to enable gzip compression globally for the whole application?

It's impossible. Choosing what to compress and where don't apply such time consuming operation is very tricky matter.

If you need global compression – write own custom middleware. Or enable compression in NGINX (you are deploying aiohttp behind reverse proxy, is not it).

11.16 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/>

- 17. Why do you might want to include your awesome library into the list?
- A. Just because the list increases your library visibility. People will have an easy way to find it.

11.16.1 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*.

Database drivers

- **aiopg** PostgreSQL async driver.
- **aiomysql** MySQL async driver.
- **aioredis** Redis async driver.

11.16.2 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.

11.16.3 Others

The list of libs which are exists but not enlisted in former categories.

They are may be perfect or not – we don't know.

Please add your library reference here first and after some time period ask to raise he status.

- **aiohttp-cache** A cache system for *aiohttp* server.
- **aiocache** Caching for *asyncio* with multiple backends (framework agnostic)
- **aiohttp-devtools** provides development tools for *aiohttp.web* applications

11.17 Essays

11.17.1 Router refactoring in 0.21

Rationale

First generation (v1) of router has mapped (method, path) pair to *web-handler*. Mapping is named **route**. Routes used to have unique names if any.

The main mistake with the design is coupling the **route** to (method, path) pair while really URL construction operates with **resources** (**location** is a synonym). HTTP method is not part of URI but applied on sending HTTP request only.

Having different **route names** for the same path is confusing. Moreover **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 v1 router it can be solved by passing '*' as HTTP method. Class based views require '*' method also usually.

Implementation

The change introduces **resource** as first class citizen:

```
resource = router.add_resource('/path/{to}', name='name')
```

Resource has a **path** (dynamic or constant) and optional **name**.

The name is **unique** in router context.

Resource has **routes**.

Route corresponds to *HTTP method* and *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 `resource.add_wildcard(handler)` later).

Since **names** belongs to **resources** now `app.router['name']` returns a **resource** instance instead of `aiohttp.web.Route`.

resource has `.url()` method, so `app.router['name'].url(parts={'a': 'b'}, query={'arg': 'param'})` still works as usual.

The change allows to rewrite static file handling and implement nested applications as well.

Decoupling of *HTTP location* and *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.

`app.router.add_route(method, path, handler, name='name')` now is just shortcut for:

```
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).

11.17.2 What's new in aiohttp 1.1

YARL and URL encoding

Since aiohttp 1.1 the library uses *yaml* for URL processing.

New API

`yaml.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(yaml.URL('http://example.com'))`.

Internal API has been switched to `yaml.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 `yaml.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 `yaml.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 `yaml.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:

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

and:

```
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:

```
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:

```
admin = web.Application()
admin.add_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')`.

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.

11.18 Contributing

11.18.1 Instructions for contributors

In order to make a clone of the [GitHub](#) repo: open the link and press the “Fork” button on the upper-right menu of the web page.

I hope everybody knows how to work with git and github nowadays :)

Workflow is pretty straightforward:

1. Clone the [GitHub](#) repo
2. Make a change
3. Make sure all tests passed
4. Commit changes to own aiohttp clone
5. Make pull request from github page for your clone against master branch

Note: If your PR has long history or many commits please rebase it from main repo before creating PR.

11.18.2 Preconditions for running aiohttp test suite

We expect you to use a python virtual environment to run our tests.

There are several ways to make a virtual environment.

If you like to use *virtualenv* please run:

```
$ cd aiohttp
$ virtualenv --python=`which python3` venv
$ . venv/bin/activate
```

For standard python *venv*:

```
$ cd aiohttp
$ python3 -m venv venv
$ . venv/bin/activate
```

For *virtualenvwrapper* (my choice):

```
$ cd aiohttp
$ mkvirtualenv --python=`which python3` aiohttp
```

There are other tools like *pyvenv* but you know the rule of thumb now: create a python3 virtual environment and activate it.

After that please install libraries required for development:

```
$ pip install -r requirements-dev.txt
```

We also recommend to install `ipdb` but it's on your own:

```
$ pip install ipdb
```

Note: If you plan to use `ipdb` within the test suite, execute:

```
$ py.test tests -s -p no:timeout  
command to run the tests with disabled timeout guard and output  
capturing.
```

Congratulations, you are ready to run the test suite

11.18.3 Run aiohttp test suite

After all the preconditions are met you can run tests typing the next command:

```
$ make test
```

The command at first will run the `flake8` tool (sorry, we don't accept pull requests with pep8 or pyflakes errors).

On `flake8` success the tests will be run.

Please take a look on the produced output.

Any extra texts (print statements and so on) should be removed.

11.18.4 Tests coverage

We are trying hard to have good test coverage; please don't make it worse.

Use:

```
$ make cov
```

to run test suite and collect coverage information. Once the command has finished check your coverage at the file that appears in the last line of the output: `open file:///.../aiohttp/coverage/index.html`

Please go to the link and make sure that your code change is covered.

11.18.5 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.

11.18.6 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
```

11.18.7 The End

After finishing all steps make a [GitHub Pull Request](#), thanks.

11.19 Changes

11.19.1 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
- Deprecate undocumented `app.on_loop_available` signal #1978

11.19.2 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

11.19.3 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

11.19.4 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

11.19.5 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

11.19.6 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

11.19.7 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

11.19.8 2.0.2 (2017-03-21)

- Fixed `Application.on_loop_available` signal #1739
- Remove debug code

11.19.9 2.0.1 (2017-03-21)

- Fix allow-head to include name on route #1737
- Fixed `AttributeError` in `WebSocketResponse.can_prepare` #1736

11.19.10 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 `ClientReponseError` 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

11.19.11 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 `yaml.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

11.19.12 1.3.5 (2017-03-16)

- Fixed None timeout support #1720

11.19.13 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

11.19.14 1.3.3 (2017-02-19)

- Fixed memory leak in time service #1656

11.19.15 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

11.19.16 1.3.1 (2017-02-09)

- Handle CLOSING in WebSocketResponse.__anext__
- Fixed AttributeError 'drain' for server websocket handler #1613

11.19.17 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

- 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 *WebSocket* 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

11.19.18 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 *yaml.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 *TimeServe.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

11.19.19 1.1.6 (2016-11-28)

- Fix *BodyPartReader.read_chunk* bug about returns zero bytes before *EOF* #1428

11.19.20 1.1.5 (2016-11-16)

- Fix static file serving in fallback mode #1401

11.19.21 1.1.4 (2016-11-14)

- Make *TestServer.make_url* compatible with *yaml.URL* #1389
- Generate informative exception on redirects from server which does not provide redirection headers #1396

11.19.22 1.1.3 (2016-11-10)

- Support *root* resources for sub-applications #1379

11.19.23 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

11.19.24 1.1.1 (2016-11-04)

- Fix documentation generation #1120

11.19.25 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

11.19.26 1.0.5 (2016-10-11)

- Fix *StreamReader.read_nowait* to return all available data up to the requested amount #1297

11.19.27 1.0.4 (2016-09-22)

- Fix *FlowControlStreamReader.read_nowait* so that it checks whether the transport is paused #1206

11.19.28 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

11.19.29 1.0.1 (2016-09-16)

- Restore *aiohhttp.web.MsgType* alias for *aiohhttp.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

11.19.30 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 shortcuts 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 *aiohhttp.websocket* to *aiohhttp._ws_impl*
- Rename *aiohhttp.MsgType* to *aiohhttp.WSMsgType*
- Introduce *aiohhttp.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_close()* should be present)
- Remove *aiohhttp.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

11.19.31 0.22.5 (08-02-2016)

- Pin multidict version to >=1.2.2

11.19.32 0.22.3 (07-26-2016)

- Do not filter cookies if unsafe flag provided #1005

11.19.33 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

11.19.34 0.22.1 (07-16-2016)

- Large cookie expiration/max-age does not break an event loop from now (fixes #967)

11.19.35 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
- Unicorn 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

11.19.36 0.21.6 (05-05-2016)

- Drop initial query parameters on redirects #853

11.19.37 0.21.5 (03-22-2016)

- Fix command line arg parsing #797

11.19.38 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

11.19.39 0.21.2 (02-16-2016)

- Fix a regression: support for handling ~/path in static file routes was broken #782

11.19.40 0.21.1 (02-10-2016)

- Make new resources classes public #767
- Add `router.resources()` view
- Fix cmd-line parameter names in doc

11.19.41 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 `aiohhttp.request()`, `aiohhttp.get()` and family in docs #729
- Deprecate bare `aiohhttp.request()`, `aiohhttp.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

11.19.42 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

11.19.43 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

11.19.44 0.20.0 (12-28-2015)

- Extend list of web exceptions, add HTTPMisdirectedRequest, HTTPUpgradeRequired, HTTPPreconditionRequired, HTTPTooManyRequests, HTTPRequestHeaderFieldsTooLarge, HTTPVariantAlsoNegotiates, HTTPNotExtended, 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

11.19.45 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

11.19.46 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

11.19.47 0.18.3 (25-10-2015)

- Fix formatting for `_RequestContextManager` helper #590

11.19.48 0.18.2 (22-10-2015)

- Fix regression for OpenSSL < 1.0.0 #583

11.19.49 0.18.1 (20-10-2015)

- Relax rule for router names: they may contain dots and columns starting from now

11.19.50 0.18.0 (19-10-2015)

- Use `errors.HttpProcessingError.message` as HTTP error reason and message #459
- Optimize cythonized multidict a bit
- Change repr's of multidicts and multidict views
- default headers in `ClientSession` are now case-insensitive
- Make '=' char and 'wss://' schema safe in urls #477
- `ClientResponse.close()` forces connection closing by default from now #479
N.B. Backward incompatible change: was `.close(force=False)` Using `force` parameter for the method is deprecated: use `.release()` instead.
- Properly requote URL's path #480
- add `skip_auto_headers` parameter for client API #486
- Properly parse URL path in `aiohttp.web.Request` #489
- Raise `RuntimeError` when chunked enabled and HTTP is 1.0 #488
- Fix a bug with processing `io.BytesIO` as data parameter for client API #500
- Skip auto-generation of Content-Type header #507
- Use `sendfile` facility for static file handling #503
- Default `response_factory` in `app.router.add_static` now is `StreamResponse`, not `None`. The functionality is not changed if default is not specified.
- Drop `ClientResponse.message` attribute, it was always implementation detail.
- Streams are optimized for speed and mostly memory in case of a big HTTP message sizes #496
- Fix a bug for server-side cookies for dropping cookie and setting it again without `Max-Age` parameter.
- Don't trim redirect URL in client API #499
- Extend precision of access log "D" to milliseconds #527

- Deprecate *StreamResponse.start()* method in favor of *StreamResponse.prepare()* coroutine #525
.start() is still supported but responses begun with .start() 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

11.19.51 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

11.19.52 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

11.19.53 0.17.2 (08-11-2015)

- Don't forget to pass *data* argument forward #462
- Fix multipart read bytes count #463

11.19.54 0.17.1 (08-10-2015)

- Fix multidict comparison to arbitrary *abc.Mapping*

11.19.55 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

11.19.56 0.16.6 (07-15-2015)

- Skip compilation on Windows if vcvarsall.bat cannot be found #438

11.19.57 0.16.5 (06-13-2015)

- Get rid of all comprehensions and yielding in _multidict #410

11.19.58 0.16.4 (06-13-2015)

- Don't clear current exception in multidict's `__repr__` (cythonized versions) #410

11.19.59 0.16.3 (05-30-2015)

- Fix StaticRoute vulnerability to directory traversal attacks #380

11.19.60 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

11.19.61 0.16.1 (05-27-2015)

- Fix regression in static file handling #377

11.19.62 0.16.0 (05-26-2015)

- Unset waiter future after cancellation #363
- Update request url with query parameters #372
- Support new *fingerprint* param of `TCPCConnector` 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()`

11.19.63 0.15.3 (04-22-2015)

- Fix graceful shutdown handling
- Fix *Expect* header handling for not found and not allowed routes #340

11.19.64 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.

11.19.65 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

11.19.66 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
- Unicorn 19.2.x compatibility

11.19.67 0.14.4 (01-29-2015)

- Fix issue with error during constructing of url with regex parts #264

11.19.68 0.14.3 (01-28-2015)

- Use path='/' by default for cookies #261

11.19.69 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

11.19.70 0.14.1 (01-15-2015)

- `HttpMessage._add_default_headers` does not overwrite existing headers #216
- Expose multidict classes at package level
- add `aiohhttp.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 `aiohhttp.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).

11.19.71 0.13.1 (12-31-2014)

- Add `aiohhttp.web.StreamResponse.started` property #213
- HTML escape traceback text in `ServerHttpProtocol.handle_error`
- Mention handler and middlewares in `aiohhttp.web.RequestHandler.handle_request` on error (#218)

11.19.72 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 *TCPCConnector* #206

11.19.73 0.12.0 (12-12-2014)

- Deep refactoring of *aiohttp.web* in backward-incompatible manner. Sorry, we have to do this.
- Automatically force *aiohttp.web* handlers to coroutines in *UrlDispatcher.add_route()* #186
- Rename *Request.POST()* function to *Request.post()*
- Added POST attribute
- Response processing refactoring: constructor 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*

11.19.74 0.11.0 (11-29-2014)

- Support named routes in *aiohttp.web.UrlDispatcher* #179
- Make websocket subprotocols conform to spec #181

11.19.75 0.10.2 (11-19-2014)

- Don't unquote *environ['PATH_INFO']* in *wsgi.py* #177

11.19.76 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.

11.19.77 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

11.19.78 0.9.3 (10-30-2014)

- Fix compatibility with asyncio 3.4.1+ #170

11.19.79 0.9.2 (10-16-2014)

- Improve redirect handling #157
- Send raw files as is #153
- Better websocket support #150

11.19.80 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.

11.19.81 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.

11.19.82 0.8.4 (07-04-2014)

- Change ProxyConnector authorization parameters.

11.19.83 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.

11.19.84 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`.

11.19.85 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.

11.19.86 0.8.0 (06-06-2014)

- Add support for utf-8 values in HTTP headers
- Allow to use custom response class instead of `HttpResponse`
- Use `MultiDict` for client request headers
- Use `MultiDict` for server request/response headers
- Store response headers in `ClientResponse.headers` attribute
- Get rid of timeout parameter in `aiohttp.client` API
- Exceptions refactoring

11.19.87 0.7.3 (05-20-2014)

- Simple HTTP proxy support.

11.19.88 0.7.2 (05-14-2014)

- Get rid of `__del__` methods
- Use `ResourceWarning` instead of logging warning record.

11.19.89 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.

11.19.90 0.7.0 (04-16-2014)

- Connection flow control.
- HTTP client session/connection pool refactoring.
- Better handling for bad server requests.

11.19.91 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.

11.19.92 0.6.4 (02-27-2014)

- Log content-length missing warning only for put and post requests.

11.19.93 0.6.3 (02-27-2014)

- Better support for server exit.
- Read response body until EOF if content-length is not defined #14

11.19.94 0.6.2 (02-18-2014)

- Fix trailing char in allowed_methods.
- Start slow request timer for first request.

11.19.95 0.6.1 (02-17-2014)

- Added utility method HttpResponse.read_and_close()
- Added slow request timeout.
- Enable socket SO_KEEPALIVE if available.

11.19.96 0.6.0 (02-12-2014)

- Better handling for process exit.

11.19.97 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

11.19.98 0.4.4 (11-15-2013)

- Resolve only `AF_INET` family, because it is not clear how to pass extra info to `asyncio`.

11.19.99 0.4.3 (11-15-2013)

- Allow to wait completion of request with `HttpResponse.wait_for_close()`

11.19.100 0.4.2 (11-14-2013)

- Handle exception in client request stream.
- Prevent host resolving for each client request.

11.19.101 0.4.1 (11-12-2013)

- Added client support for *expect: 100-continue* header.

11.19.102 0.4 (11-06-2013)

- Added custom `wsgi` application close procedure
- Fixed concurrent host failure in `HttpClient`

11.19.103 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

11.19.104 0.2

- Fix packaging

11.20 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 cChardet 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/>

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/>

resource A concept reflects the HTTP **path**, every resource corresponds to *URI*.

May have a unique name.

Contains *route*’s for different HTTP methods.

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**

yaml A library for operating with URL objects.

<https://pypi.python.org/pypi/yarl>

CHAPTER 12

Indices and tables

- `genindex`
- `modindex`
- `search`

a

`aiohttp`, 114
`aiohttp.helpers`, 118
`aiohttp.multipart`, 109
`aiohttp.signals`, 121
`aiohttp.streams`, 114
`aiohttp.web`, 79

Symbols

`_create_connection()` (aiohttp.BaseConnector method), 45

A

AbstractCookieJar (class in aiohttp), 108
 AbstractMatchInfo (class in aiohttp), 108
 AbstractResource (class in aiohttp.web), 101
 AbstractRoute (class in aiohttp.web), 103
 AbstractRouter (class in aiohttp), 108
 AbstractView (class in aiohttp), 108
`add_delete()` (aiohttp.web.UrlDispatcher method), 98
`add_get()` (aiohttp.web.UrlDispatcher method), 97
`add_head()` (aiohttp.web.UrlDispatcher method), 97
`add_patch()` (aiohttp.web.UrlDispatcher method), 98
`add_post()` (aiohttp.web.UrlDispatcher method), 97
`add_put()` (aiohttp.web.UrlDispatcher method), 98
`add_resource()` (aiohttp.web.UrlDispatcher method), 97
`add_route()` (aiohttp.web.Resource method), 101
`add_route()` (aiohttp.web.UrlDispatcher method), 97
`add_static()` (aiohttp.web.UrlDispatcher method), 98
`add_subapp()` (aiohttp.web.UrlDispatcher method), 99
 aiodns, 176
 aiohttp (module), 26, 37, 107, 114, 116
 aiohttp.helpers (module), 118
 aiohttp.multipart (module), 109, 118
 aiohttp.signals (module), 121
 aiohttp.streams (module), 114
 aiohttp.web (module), 79
 AioHTTPTestCase (class in aiohttp.test_utils), 126
 app (aiohttp.test_utils.AioHTTPTestCase attribute), 126
 app (aiohttp.test_utils.TestServer attribute), 130
 app (aiohttp.web.Request attribute), 84
`append()` (aiohttp.multipart.MultipartWriter method), 119
`append_form()` (aiohttp.multipart.MultipartWriter method), 119
`append_json()` (aiohttp.multipart.MultipartWriter method), 119

`append_payload()` (aiohttp.multipart.MultipartWriter method), 119

Application (class in aiohttp.web), 93

asyncio, 176

`at_eof()` (aiohttp.multipart.BodyPartReader method), 119

`at_eof()` (aiohttp.multipart.MultipartReader method), 118

`at_eof()` (aiohttp.StreamReader method), 115

B

BadContentDispositionHeader, 120

BadContentDispositionParam, 120

BaseConnector (class in aiohttp), 44

BaseRequest (class in aiohttp.web), 79

BaseSignal (class in aiohttp.signals), 121

BaseTestServer (class in aiohttp.test_utils), 130

BasicAuth (class in aiohttp), 52

BINARY (aiohttp.WSMsgType attribute), 117

body (aiohttp.web.Response attribute), 89

BodyPartReader (class in aiohttp.multipart), 119

boundary (aiohttp.multipart.MultipartWriter attribute), 119

C

cached_hosts (aiohttp.TCPConnector attribute), 47

callable, 176

`can_prepare()` (aiohttp.web.WebSocketResponse method), 90

cchardet, 176

chardet, 176

charset (aiohttp.ClientResponse attribute), 49

charset (aiohttp.web.BaseRequest attribute), 82

charset (aiohttp.web.StreamResponse attribute), 87

chunk_size (aiohttp.multipart.BodyPartReader attribute), 119

chunked (aiohttp.web.StreamResponse attribute), 86

`cleanup()` (aiohttp.web.Application method), 96

`clear()` (aiohttp.helpers.DummyCookieJar method), 118

`clear_dns_cache()` (aiohttp.TCPConnector method), 47

client (aiohttp.test_utils.AioHTTPTestCase attribute), 126

- ClientResponse (class in aiohttp), 48
 - ClientSession (class in aiohttp), 38
 - ClientWebSocketResponse (class in aiohttp), 50
 - clone() (aiohttp.web.BaseRequest method), 82
 - CLOSE (aiohttp.WSMsgType attribute), 117
 - close() (aiohttp.BaseConnector method), 45
 - close() (aiohttp.ClientResponse method), 49
 - close() (aiohttp.ClientSession method), 43
 - close() (aiohttp.ClientWebSocketResponse method), 51
 - close() (aiohttp.Connection method), 48
 - close() (aiohttp.test_utils.BaseTestServer method), 130
 - close() (aiohttp.test_utils.TestClient method), 131
 - close() (aiohttp.web.WebSocketResponse method), 91
 - close_code (aiohttp.web.WebSocketResponse attribute), 90
 - closed (aiohttp.BaseConnector attribute), 45
 - closed (aiohttp.ClientSession attribute), 39
 - closed (aiohttp.ClientWebSocketResponse attribute), 50
 - closed (aiohttp.Connection attribute), 48
 - closed (aiohttp.web.WebSocketResponse attribute), 90
 - compression (aiohttp.web.StreamResponse attribute), 85
 - connect() (aiohttp.BaseConnector method), 45
 - connection (aiohttp.ClientResponse attribute), 49
 - Connection (class in aiohttp), 48
 - connections (aiohttp.web.Server attribute), 96
 - connector (aiohttp.ClientSession attribute), 39
 - content (aiohttp.ClientResponse attribute), 49
 - content (aiohttp.web.BaseRequest attribute), 81
 - content_disposition_filename() (in module aiohttp.multipart), 121
 - content_length (aiohttp.web.BaseRequest attribute), 82
 - content_length (aiohttp.web.StreamResponse attribute), 87
 - content_type (aiohttp.ClientResponse attribute), 49
 - content_type (aiohttp.web.BaseRequest attribute), 82
 - content_type (aiohttp.web.FileField attribute), 104
 - content_type (aiohttp.web.StreamResponse attribute), 87
 - ContentCoding (class in aiohttp.web), 106
 - CONTINUATION (aiohttp.WSMsgType attribute), 117
 - cookie_jar (aiohttp.ClientSession attribute), 39
 - CookieJar (class in aiohttp), 52
 - cookies (aiohttp.ClientResponse attribute), 49
 - cookies (aiohttp.web.BaseRequest attribute), 81
 - cookies (aiohttp.web.StreamResponse attribute), 86
 - create_future() (in module aiohttp.helpers), 118
- ## D
- data (aiohttp.WSMessage attribute), 117
 - debug (aiohttp.web.Application attribute), 93
 - DebugSignal (class in aiohttp.signals), 121
 - decode() (aiohttp.BasicAuth class method), 52
 - decode() (aiohttp.multipart.BodyPartReader method), 119
 - deflate (aiohttp.web.ContentCoding attribute), 106
 - del_cookie() (aiohttp.web.StreamResponse method), 87
 - delete() (aiohttp.ClientSession method), 41
 - delete() (aiohttp.test_utils.TestClient method), 132
 - detach() (aiohttp.ClientSession method), 43
 - detach() (aiohttp.Connection method), 48
 - dns_cache (aiohttp.TCPConnector attribute), 47
 - drain() (aiohttp.web.StreamResponse method), 88
 - DummyCookieJar (class in aiohttp.helpers), 118
 - DynamicResource (class in aiohttp.web), 102
- ## E
- enable_chunked_encoding() (aiohttp.web.StreamResponse method), 86
 - enable_compression() (aiohttp.web.StreamResponse method), 86
 - encode() (aiohttp.BasicAuth method), 52
 - ensure_future() (in module aiohttp.helpers), 118
 - ERROR (aiohttp.WSMsgType attribute), 117
 - exception() (aiohttp.ClientWebSocketResponse method), 51
 - exception() (aiohttp.StreamReader method), 115
 - exception() (aiohttp.web.WebSocketResponse method), 90
 - expect_handler (aiohttp.web.UrlMappingMatchInfo attribute), 103
 - expect_handler() (aiohttp.AbstractMatchInfo method), 108
 - extra (aiohttp.WSMessage attribute), 117
- ## F
- family (aiohttp.TCPConnector attribute), 47
 - fetch_next_part() (aiohttp.multipart.MultipartReader method), 119
 - file (aiohttp.web.FileField attribute), 104
 - FileField (class in aiohttp.web), 104
 - filename (aiohttp.multipart.BodyPartReader attribute), 120
 - filename (aiohttp.web.FileField attribute), 104
 - filter_cookies() (aiohttp.AbstractCookieJar method), 109
 - filter_cookies() (aiohttp.CookieJar method), 53
 - filter_cookies() (aiohttp.helpers.DummyCookieJar method), 118
 - fingerprint (aiohttp.TCPConnector attribute), 47
 - finish_connections() (aiohttp.web.Server method), 96
 - force_close (aiohttp.BaseConnector attribute), 45
 - force_close() (aiohttp.web.StreamResponse method), 85
 - form() (aiohttp.multipart.BodyPartReader method), 120
 - forwarded (aiohttp.web.BaseRequest attribute), 80
 - from_response() (aiohttp.multipart.MultipartReader class method), 119
 - FuncSignal (class in aiohttp.signals), 121
- ## G
- get() (aiohttp.ClientSession method), 40

get() (aiohttp.test_utils.TestClient method), 131
 get_application() (aiohttp.test_utils.AioHTTPTestCase method), 126
 get_charset() (aiohttp.multipart.BodyPartReader method), 120
 get_extra_info() (aiohttp.ClientWebSocketResponse method), 50
 get_info() (aiohttp.web.AbstractResource method), 101
 GOING_AWAY (aiohttp.WSCloseCode attribute), 116
 gunicorn, 176
 gzip (aiohttp.web.ContentCoding attribute), 106

H

handle_expect_header() (aiohttp.web.AbstractRoute method), 103
 handler (aiohttp.test_utils.BaseTestServer attribute), 130
 handler (aiohttp.web.AbstractRoute attribute), 103
 handler (aiohttp.web.UrlMappingMatchInfo attribute), 103
 handler() (aiohttp.AbstractMatchInfo method), 108
 has_body (aiohttp.web.BaseRequest attribute), 81
 head() (aiohttp.ClientSession method), 41
 head() (aiohttp.test_utils.TestClient method), 131
 headers (aiohttp.ClientResponse attribute), 49
 headers (aiohttp.web.BaseRequest attribute), 81
 headers (aiohttp.web.StreamResponse attribute), 86
 history (aiohttp.ClientResponse attribute), 49
 host (aiohttp.test_utils.BaseTestServer attribute), 130
 host (aiohttp.test_utils.TestClient attribute), 131
 host (aiohttp.web.BaseRequest attribute), 80
 http_exception (aiohttp.AbstractMatchInfo attribute), 108
 http_range (aiohttp.web.BaseRequest attribute), 82

I

identity (in module aiohttp.web), 106
 if_modified_since (aiohttp.web.BaseRequest attribute), 82
 INTERNAL_ERROR (aiohttp.WSCloseCode attribute), 117
 INVALID_TEXT (aiohttp.WSCloseCode attribute), 116
 is_eof() (in module aiohttp), 115
 iter_any() (aiohttp.StreamReader method), 115
 iter_chunked() (aiohttp.StreamReader method), 115
 iter_chunks() (aiohttp.StreamReader method), 115

J

json() (aiohttp.ClientResponse method), 50
 json() (aiohttp.multipart.BodyPartReader method), 120
 json() (aiohttp.web.BaseRequest method), 83
 json() (aiohttp.WSMMessage method), 118
 json_response() (in module aiohttp.web), 93

K

keep-alive, 176

keep_alive (aiohttp.web.BaseRequest attribute), 81
 keep_alive (aiohttp.web.StreamResponse attribute), 85

L

last_modified (aiohttp.web.StreamResponse attribute), 87
 limit (aiohttp.BaseConnector attribute), 45
 limit_per_host (aiohttp.BaseConnector attribute), 45
 load() (aiohttp.CookieJar method), 53
 logger (aiohttp.web.Application attribute), 93
 loop (aiohttp.ClientSession attribute), 39
 loop (aiohttp.Connection attribute), 48
 loop (aiohttp.test_utils.AioHTTPTestCase attribute), 126
 loop (aiohttp.web.Application attribute), 93
 loop_context() (in module aiohttp.test_utils), 132

M

make_handler() (aiohttp.web.Application method), 94
 make_mocked_coro() (in module aiohttp.test_utils), 132
 make_mocked_request() (in module aiohttp.test_utils), 127
 make_url() (aiohttp.test_utils.BaseTestServer method), 130
 make_url() (aiohttp.test_utils.TestClient method), 131
 MANDATORY_EXTENSION (aiohttp.WSCloseCode attribute), 117
 match_info (aiohttp.web.Request attribute), 84
 MESSAGE_TOO_BIG (aiohttp.WSCloseCode attribute), 116
 method (aiohttp.ClientResponse attribute), 48
 method (aiohttp.web.AbstractRoute attribute), 103
 method (aiohttp.web.BaseRequest attribute), 79
 multipart() (aiohttp.web.BaseRequest method), 83
 multipart_reader_cls (aiohttp.multipart.MultipartReader attribute), 119
 MultipartReader (class in aiohttp.multipart), 118
 MultipartWriter (class in aiohttp.multipart), 119

N

name (aiohttp.multipart.BodyPartReader attribute), 120
 name (aiohttp.web.AbstractResource attribute), 101
 name (aiohttp.web.AbstractRoute attribute), 103
 name (aiohttp.web.FileField attribute), 104
 named_resources() (aiohttp.web.UrlDispatcher method), 99
 named_routes() (aiohttp.web.UrlDispatcher method), 100
 next() (aiohttp.multipart.BodyPartReader method), 120
 next() (aiohttp.multipart.MultipartReader method), 119
 nginx, 176
 noop() (in module aiohttp.helpers), 118
 normalize_path_middleware() (in module aiohttp.web), 106

O

ok (aiohttp.web.WebSocketReady attribute), 92

OK (aiohttp.WSCloseCode attribute), 116
 on_cleanup (aiohttp.web.Application attribute), 94
 on_response_prepare (aiohttp.web.Application attribute), 94
 on_shutdown (aiohttp.web.Application attribute), 94
 on_startup (aiohttp.web.Application attribute), 94
 options() (aiohttp.ClientSession method), 42
 options() (aiohttp.test_utils.TestClient method), 131
 ordinal() (aiohttp.signals.PreSignal method), 121

P

parse_content_disposition() (in module aiohttp.multipart), 121
 parse_mimetype() (in module aiohttp.helpers), 118
 part_reader_cls (aiohttp.multipart.MultipartReader attribute), 119
 patch() (aiohttp.ClientSession method), 42
 patch() (aiohttp.test_utils.TestClient method), 132
 path (aiohttp.UnixConnector attribute), 48
 path (aiohttp.web.BaseRequest attribute), 81
 path_qs (aiohttp.web.BaseRequest attribute), 80
 PING (aiohttp.WSMsgType attribute), 117
 ping() (aiohttp.ClientWebSocketResponse method), 51
 ping() (aiohttp.web.WebSocketResponse method), 90
 PlainResource (class in aiohttp.web), 102
 POLICY_VIOLATION (aiohttp.WSCloseCode attribute), 116
 PONG (aiohttp.WSMsgType attribute), 117
 pong() (aiohttp.web.WebSocketResponse method), 90
 port (aiohttp.test_utils.BaseTestServer attribute), 130
 port (aiohttp.test_utils.TestClient attribute), 131
 post() (aiohttp.ClientSession method), 41
 post() (aiohttp.test_utils.TestClient method), 131
 post() (aiohttp.web.BaseRequest method), 83
 PostSignal (class in aiohttp.signals), 121
 PrefixedSubAppResource (class in aiohttp.web), 102
 prepare() (aiohttp.web.StreamResponse method), 88
 prepare() (aiohttp.web.WebSocketResponse method), 89
 prepared (aiohttp.web.StreamResponse attribute), 85
 PreSignal (class in aiohttp.signals), 121
 protocol (aiohttp.ClientWebSocketResponse attribute), 50
 protocol (aiohttp.web.WebSocketReady attribute), 92
 protocol (aiohttp.web.WebSocketResponse attribute), 90
 PROTOCOL_ERROR (aiohttp.WSCloseCode attribute), 116
 put() (aiohttp.ClientSession method), 41
 put() (aiohttp.test_utils.TestClient method), 131
 Python Enhancement Proposals
 PEP 3156, 1, 176
 PEP 492, 8

Q

query (aiohttp.web.BaseRequest attribute), 81
 query_string (aiohttp.web.BaseRequest attribute), 81

R

raise_for_status() (aiohttp.ClientResponse method), 50
 raw_headers (aiohttp.ClientResponse attribute), 49
 raw_headers (aiohttp.web.BaseRequest attribute), 81
 raw_path (aiohttp.web.BaseRequest attribute), 81
 raw_test_server (in module aiohttp.test_utils), 125
 RawTestServer (class in aiohttp.test_utils), 130
 read() (aiohttp.ClientResponse method), 49
 read() (aiohttp.multipart.BodyPartReader method), 120
 read() (aiohttp.StreamReader method), 114
 read() (aiohttp.web.BaseRequest method), 83
 read_chunk() (aiohttp.multipart.BodyPartReader method), 120
 read_nowait() (aiohttp.StreamReader method), 116
 readany() (aiohttp.StreamReader method), 114
 readexactly() (aiohttp.StreamReader method), 114
 readline() (aiohttp.multipart.BodyPartReader method), 120
 readline() (aiohttp.StreamReader method), 115
 reason (aiohttp.ClientResponse attribute), 48
 reason (aiohttp.web.StreamResponse attribute), 85
 reason (aiohttp.web.SystemRoute attribute), 103
 receive() (aiohttp.ClientWebSocketResponse method), 51
 receive() (aiohttp.web.WebSocketResponse method), 91
 receive_bytes() (aiohttp.ClientWebSocketResponse method), 52
 receive_bytes() (aiohttp.web.WebSocketResponse method), 92
 receive_json() (aiohttp.ClientWebSocketResponse method), 52
 receive_json() (aiohttp.web.WebSocketResponse method), 92
 receive_str() (aiohttp.ClientWebSocketResponse method), 51
 receive_str() (aiohttp.web.WebSocketResponse method), 91
 rel_url (aiohttp.web.BaseRequest attribute), 79
 release() (aiohttp.ClientResponse method), 49
 release() (aiohttp.Connection method), 48
 release() (aiohttp.multipart.BodyPartReader method), 120
 release() (aiohttp.multipart.MultipartReader method), 119
 release() (aiohttp.web.BaseRequest method), 84
 request (aiohttp.AbstractView attribute), 108
 request (aiohttp.web.View attribute), 104
 Request (class in aiohttp.web), 84
 request() (aiohttp.ClientSession method), 39
 request() (aiohttp.test_utils.TestClient method), 131
 request() (in module aiohttp), 43
 request_info (aiohttp.ClientResponse attribute), 50
 requests_count (in module aiohttp.web), 96
 requote_redirect_url (aiohttp.ClientSession attribute), 39
 resolve() (aiohttp.AbstractRouter method), 108
 resolve() (aiohttp.web.AbstractResource method), 101

- resolve() (aiohttp.web.UrlDispatcher method), 99
 resource, 176
 resource (aiohttp.web.AbstractRoute attribute), 103
 Resource (class in aiohttp.web), 101
 ResourceRoute (class in aiohttp.web), 103
 resources() (aiohttp.web.UrlDispatcher method), 99
 Response (class in aiohttp.web), 88
 response_wrapper_cls (aiohttp.multipart.MultipartReader attribute), 119
 RFC
 RFC 2068, 71
 RFC 2109, 32
 RFC 2616, 49, 82
 RFC 3629, 116
 RFC 6455, 176
 RFC 7230, 36
 RFC 7239, 80
 RFC 7239#section-4, 80
 RFC 7239#section-6, 80
 route, 176
 route (aiohttp.web.UrlMappingMatchInfo attribute), 103
 router (aiohttp.web.Application attribute), 93
 routes() (aiohttp.web.UrlDispatcher method), 99
 run_app() (in module aiohttp.web), 104
 S
 save() (aiohttp.CookieJar method), 53
 scheme (aiohttp.test_utils.BaseTestServer attribute), 130
 scheme (aiohttp.test_utils.TestClient attribute), 131
 scheme (aiohttp.web.BaseRequest attribute), 80
 secure (aiohttp.web.BaseRequest attribute), 80
 send() (aiohttp.signals.DebugSignal method), 121
 send() (aiohttp.signals.FuncSignal method), 121
 send() (aiohttp.signals.Signal method), 121
 send_bytes() (aiohttp.ClientWebSocketResponse method), 51
 send_bytes() (aiohttp.web.WebSocketResponse method), 91
 send_json() (aiohttp.ClientWebSocketResponse method), 51
 send_json() (aiohttp.web.WebSocketResponse method), 91
 send_str() (aiohttp.ClientWebSocketResponse method), 51
 send_str() (aiohttp.web.WebSocketResponse method), 90
 server (aiohttp.test_utils.BaseTestServer attribute), 130
 server (aiohttp.test_utils.TestClient attribute), 131
 Server (class in aiohttp.web), 96
 SERVICE_RESTART (aiohttp.WSCloseCode attribute), 117
 session (aiohttp.test_utils.TestClient attribute), 131
 set_cookie() (aiohttp.web.StreamResponse method), 86
 set_status() (aiohttp.web.StreamResponse method), 85
 set_tcp_cork() (aiohttp.web.StreamResponse method), 87
 set_tcp_nodelay() (aiohttp.web.StreamResponse method), 88
 setUp() (aiohttp.test_utils.AioHTTPTestCase method), 126
 setup_test_loop() (in module aiohttp.test_utils), 132
 shutdown() (aiohttp.web.Application method), 95
 shutdown() (aiohttp.web.Server method), 96
 Signal (class in aiohttp.signals), 121
 size (aiohttp.multipart.MultipartWriter attribute), 119
 ssl_context (aiohttp.TCPConnector attribute), 47
 start_server() (aiohttp.test_utils.BaseTestServer method), 130
 start_server() (aiohttp.test_utils.TestClient method), 131
 startup() (aiohttp.web.Application method), 95
 StaticResource (class in aiohttp.web), 102
 status (aiohttp.ClientResponse attribute), 48
 status (aiohttp.web.StreamResponse attribute), 85
 status (aiohttp.web.SystemRoute attribute), 103
 StreamReader (class in aiohttp), 114
 StreamResponse (class in aiohttp.web), 85
 SystemRoute (class in aiohttp.web), 103
 T
 task (aiohttp.web.StreamResponse attribute), 85
 tcp_cork (aiohttp.web.StreamResponse attribute), 87
 tcp_nodelay (aiohttp.web.StreamResponse attribute), 87
 TCPConnector (class in aiohttp), 45
 tearDown() (aiohttp.test_utils.AioHTTPTestCase method), 126
 teardown_test_loop() (in module aiohttp.test_utils), 132
 test_client (in module aiohttp.test_utils), 125
 test_server (in module aiohttp.test_utils), 124
 TestClient (class in aiohttp.test_utils), 131
 TestServer (class in aiohttp.test_utils), 130
 text (aiohttp.web.Response attribute), 89
 TEXT (aiohttp.WSMsgType attribute), 117
 text() (aiohttp.ClientResponse method), 50
 text() (aiohttp.multipart.BodyPartReader method), 120
 text() (aiohttp.web.BaseRequest method), 83
 Timeout (in module aiohttp.helpers), 118
 tp (aiohttp.WSMessage attribute), 118
 transport (aiohttp.Connection attribute), 48
 transport (aiohttp.web.BaseRequest attribute), 81
 TRY_AGAIN_LATER (aiohttp.WSCloseCode attribute), 117
 type (aiohttp.WSMessage attribute), 117
 U
 UnixConnector (class in aiohttp), 47
 unread_data() (aiohttp.StreamReader method), 116
 UNSUPPORTED_DATA (aiohttp.WSCloseCode attribute), 116
 unused_port() (in module aiohttp.test_utils), 132

update_cookies() (aiohttp.AbstractCookieJar method),
109
update_cookies() (aiohttp.CookieJar method), 53
update_cookies() (aiohttp.helpers.DummyCookieJar
method), 118
url (aiohttp.ClientResponse attribute), 49
url (aiohttp.web.BaseRequest attribute), 79
url() (aiohttp.web.AbstractResource method), 101
url_for() (aiohttp.web.AbstractResource method), 101
url_for() (aiohttp.web.AbstractRoute method), 103
url_for() (aiohttp.web.DynamicResource method), 102
url_for() (aiohttp.web.PlainResource method), 102
url_for() (aiohttp.web.PrefixedSubAppResource method),
102
url_for() (aiohttp.web.StaticResource method), 102
UrlDispatcher (class in aiohttp.web), 96
UrlMappingMatchInfo (class in aiohttp.web), 103

V

verify_ssl (aiohttp.TCPConnector attribute), 47
version (aiohttp.ClientResponse attribute), 48
version (aiohttp.web.BaseRequest attribute), 79
View (class in aiohttp.web), 104

W

wait_eof() (in module aiohttp), 116
web-handler, **176**
websocket, **176**
WebSocketReady (class in aiohttp.web), 92
WebSocketResponse (class in aiohttp.web), 89
write() (aiohttp.multipart.MultipartWriter method), 119
write() (aiohttp.web.StreamResponse method), 88
write_eof() (aiohttp.web.StreamResponse method), 88
ws_connect() (aiohttp.ClientSession method), 42
ws_connect() (aiohttp.test_utils.TestClient method), 132
WSCloseCode (class in aiohttp), 116
WSMessage (class in aiohttp), 117
WSMsgType (class in aiohttp), 117

Y

yaml, **176**