Advanced Client Usage¶

Client Session¶

`ClientSession` is the heart and the main entry point for all client API operations.

Create the session first, use the instance for performing HTTP requests and initiating WebSocket connections.

The session contains a cookie storage and connection pool, thus cookies and connections are shared between HTTP requests sent by the same session.

Custom Request Headers¶

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

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

```
url = 'http://example.com/image'
payload = b'GIF89a\x01\x00\x01\x00\x00\xff\x00,\x00\x00'
          b'\x00\x00\x01\x00\x01\x00\x00\x02\x00;'
headers = {'content-type': 'image/gif'}

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

You also can set default headers for all session requests:

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

Typical use case is sending JSON body. You can specify content type directly as shown above, but it is more convenient to use special keyword `json`:

```
await session.post(url, json={'example': 'text'})
```

For *text/plain*

```
await session.post(url, data='Привет, Мир!')
```

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`.

`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'
```

Response Headers and Cookies¶

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

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

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

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

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

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

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

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

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

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

Note

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

Redirection History¶

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

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

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

Cookie Jar¶

Cookie Safety¶

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

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

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

Dummy Cookie Jar¶

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

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

Uploading pre-compressed data¶

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

```
async def my_coroutine(session, headers, my_data):
    data = zlib.compress(my_data)
    headers = {'Content-Encoding': 'deflate'}
    async with session.post('http://httpbin.org/post',
                            data=data,
                            headers=headers)
        pass
```

Disabling content type validation for JSON responses¶

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

This can be worked around in two ways:

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

   ```
   await resp.json(content_type='custom/type').
   ```

2. Disable the check entirely:

`await resp.json(content_type=None)`.

Client Tracing¶

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

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

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

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

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

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

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

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

```
async def on_request_start(
        session, trace_config_ctx, params):
    trace_config_ctx.start = session.loop.time()

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

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

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

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


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

See also

Tracing Reference section for more information about the different signals supported.

Connectors¶

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

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

Note

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

See also

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

Limiting connection pool size¶

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

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

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

The default is *100*.

If you explicitly want not to have limits, pass *0*. For example:

```
conn = aiohttp.TCPConnector(limit=0)
```

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

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

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

The default is *0* (no limit on per host bases).

Tuning the DNS cache¶

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

```
conn = aiohttp.TCPConnector(ttl_dns_cache=300)
```

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

```
conn = aiohttp.TCPConnector(use_dns_cache=False)
```

Resolving using custom nameservers¶

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

```
from aiohttp.resolver import AsyncResolver

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

Unix domain sockets¶

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

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

SSL control for TCP sockets¶

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

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

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

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

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

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

There is explicit errors when ssl verification fails

`aiohttp.ClientConnectorSSLError`:

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

`aiohttp.ClientConnectorCertificateError`:

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

If you need to skip both ssl related errors

`aiohttp.ClientSSLError`:

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

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

You may also verify certificates via *SHA256* fingerprint:

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

# www.python.org cert's actual fingerprint
assert exc.got == b'...'
```

Note that this is the fingerprint of the DER-encoded certificate. If you have the certificate in PEM format, you can convert it to DER with e.g:

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

Note

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

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

Proxy support¶

aiohttp supports HTTP/HTTPS proxies. You have to use *proxy* parameter:

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

It also supports proxy authorization:

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

Authentication credentials can be passed in proxy URL:

```
session.get("http://python.org",
            proxy="http://user:[email protected]")
```

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

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

Proxy credentials are given from `~/.netrc` file if present (see `aiohttp.ClientSession` for more details).

Graceful Shutdown¶

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

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

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

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

loop = asyncio.get_event_loop()
loop.run_until_complete(read_website())
# Zero-sleep to allow underlying connections to close
loop.run_until_complete(asyncio.sleep(0))
loop.close()
```

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

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

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

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

Logo

Async HTTP client/server for asyncio and Python

Star 6,587

- build passing
- codecov 97%
- pypi package 3.4.4
- gitter join chat

Navigation

- Client
  - Quickstart
  - Advanced Usage
  - Reference
  - Tracing Reference
- Server
- Utilities

Quick search

Go