aiosonic

Release 0.16.1

Johanderson Mogollon

Mar 27, 2023

CONTENTS

1 Features **3**

2 Requirements **5**

3 Install **7**

4 Getting Started **9**

5 Benchmarks **11**

6 Contributing **13**

7 Indices and tables **15**

7.1 Examples 15

7.2 Reference 19

Index **25**

Really Fast Python asyncio HTTP 1.1 and 2.0 client.

Current version is 0.16.1.

Repo is hosted at Github.

CHAPTER

ONE

FEATURES

- Keepalive and Smart Pool of Connections
- Multipart File Uploads
- Chunked responses handling
- Chunked requests
- Fully type annotated.
- Connection Timeouts
- Automatic Decompression
- Follow Redirects
- 100% test coverage (Sometimes not).

CHAPTER

TWO

REQUIREMENTS

- Python>=3.7
- PyPy >=3.7

CHAPTER

THREE

INSTALL

```
$ pip install aiosonic
```

CHAPTER

FOUR

GETTING STARTED

```
import asyncio
import aiosonic
import json


async def run():
    client = aiosonic.HTTPClient()

    # ###################
    # Sample get request
    # ###################
    response = await client.get('https://www.google.com/')
    assert response.status_code == 200
    assert 'Google' in (await response.text())

    # ###################
    # Post data as multipart form
    # ###################
    url = "https://postman-echo.com/post"
    posted_data = {'foo': 'bar'}
    response = await client.post(url, data=posted_data)

    assert response.status_code == 200
    data = json.loads(await response.content())
    assert data['form'] == posted_data

    # ###################
    # Posted as json
    # ###################
    response = await client.post(url, json=posted_data)

    assert response.status_code == 200
    data = json.loads(await response.content())
    assert data['json'] == posted_data

    # ###################
    # Sample request + timeout
    # ###################
    from aiosonic.timeout import Timeouts
    timeouts = Timeouts(
```

(continues on next page)

(continued from previous page)

```
        sock_read=10,
        sock_connect=3
    )
    response = await client.get('https://www.google.com/', timeouts=timeouts)
    assert response.status_code == 200
    assert 'Google' in (await response.text())
    await client.shutdown()

    print('success')


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run())
```

CHAPTER

FIVE

BENCHMARKS

Some benchmarking

```
» python tests/performance.py
doing tests...
{
 "aiosonic": "1000 requests in 182.03 ms",
 "aiosonic cyclic": "1000 requests in 370.55 ms",
 "aiohttp": "1000 requests in 367.66 ms",
 "requests": "1000 requests in 4613.77 ms",
 "httpx": "1000 requests in 812.41 ms"
}
aiosonic is 101.97% faster than aiohttp
aiosonic is 2434.55% faster than requests
aiosonic is 103.56% faster than aiosonic cyclic
aiosonic is 346.29% faster than httpx
```

This is a *very basic, dummy test*, machine dependant. If you look for performance, test and compare your code with this and other packages like aiohttp.

You can perform this test by installing all test dependencies with *pip install -e ".[test]"* and doing *python tests/performance.py* in your own machine

CHAPTER

SIX

CONTRIBUTING

1. Fork
2. create a branch *feature/your_feature*
3. commit - push - pull request

Thanks :)

CHAPTER

SEVEN

INDICES AND TABLES

- genindex
- modindex
- search

7.1 Examples

TODO: More examples

7.1.1 Download file

```
import asyncio
import aiosonic
import json


async def run():
    url = 'https://images.dog.ceo/breeds/leonberg/n02111129_2301.jpg'
    async with aiosonic.HTTPClient() as client:

       res = await client.get(url)
       assert res.status_code == 200

    if res.chunked:
        # write in chunks
        with open('dog_image.jpg', 'wb') as _file:
            async for chunk in res.read_chunks():
                _file.write(chunk)
    else:
        # or write all bytes, for chunked this also works
        with open('dog_image.jpg', 'wb') as _file:
            _file.write(await res.content())


if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(run())
```

7.1.2 Concurrent Requests

```
import aiosonic
import asyncio


async def main():
    urls = [
        'https://www.facebook.com/',
        'https://www.google.com/',
        'https://twitch.tv/',
        'https://linkedin.com/',
    ]
    async with aiosonic.HTTPClient() as client:
        # asyncio.gather is the key for concurrent requests.
        responses = await asyncio.gather(*[client.get(url) for url in urls])

        # stream/chunked responses doesn't release the connection acquired
        # from the pool until the response has been read, so better to read
        # it.
        for response in responses:
            if response.chunked:
                await response.text()

        assert all([res.status_code in [200, 301] for res in responses])

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

7.1.3 Chunked Requests

Specifying an iterator as the request body, it will make the request transfer made by chunks

```
import aiosonic
import asyncio
import json


async def main():
    async def data():
        yield b'foo'
        yield b'bar'

    async with aiosonic.HTTPClient() as client:
        url = 'https://postman-echo.com/post'
        response = await client.post(url, data=data())
        print(json.dumps(await response.json(), indent=10))


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

7.1.4 Cookies handling

Adding *handle_cookies=True* to the client, it will save response cookies and send it again for new requests. This is useful to have same cookies workflow as in browsers, also for web scraping.

```
import aiosonic
import asyncio


async def main():
    async with aiosonic.HTTPClient(handle_cookies=True) as client:
        cookies = {'foo1': 'bar1', 'foo2': 'bar2'}
        url = 'https://postman-echo.com/cookies/set'
        # server will respond those cookies
        response = await client.get(url, params=cookies, follow=True)
        # client keep cookies in "cookies_map"
        print(client.cookies_map['postman-echo.com'])
        print(await response.text())


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

7.1.5 Use custom DNS

Install *aiodns* in your dependencies and use AsyncResolver

```
import aiosonic
import asyncio
from aiosonic.resolver import AsyncResolver


async def main():
    resolver = AsyncResolver(nameservers=["8.8.8.8", "8.8.4.4"])
    connector = aiosonic.TCPConnector(resolver=resolver)

    async with aiosonic.HTTPClient(connector=connector) as client:
        data = {'foo1': 'bar1', 'foo2': 'bar2'}
        url = 'https://postman-echo.com/post'
        # server will respond those cookies
        response = await client.post(url, json=data)
        # client keep cookies in "cookies_map"
        print(await response.text())

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

7.1.6 Use a Proxy Server

Just use Proxy class.

You can install proxy.py and use it as a proxy demo.

```
import asyncio

from aiosonic import HTTPClient, Proxy


async def main():
    # Proxy class accepts `auth` argument in the format `user:password`
    client = HTTPClient(proxy=Proxy("http://localhost:8899"))

    res = await client.get("https://www.google.com/")
    print(res)
    print(await res.text())
    assert res.status_code == 200


asyncio.run(main())
```

7.1.7 Debug log

Configure aiosonic logger at debug level to see some logging

```
import asyncio
import aiosonic
import json
import logging


async def run():
    # setup debug level at log
    logger = logging.getLogger('aiosonic')
    logger.setLevel(logging.DEBUG)

    async with aiosonic.HTTPClient() as client:
      response = await client.get('https://www.google.com/')
      assert response.status_code == 200
      assert 'Google' in (await response.text())

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

7.2 Reference

TODO: get better this page

7.2.1 Connector and Client Client

class aiosonic.connectors.**TCPConnector**(*pool_size: int = 25*, *timeouts: Optional[aiosonic.timeout.Timeouts] = None*, *connection_cls=None*, *pool_cls=None*, *resolver=None*, *ttl_dns_cache=10000*, *use_dns_cache=True*, *conn_max_requests=100*)

TCPConnector.

Holds the main logic for making connections to destination hosts.

Params:

- **pool_size**: size for pool of connections
- **timeouts**: global timeouts to use for connections with this connector. default: *aiosonic.timeout.Timeouts* instance with default args.
- **connection_cls**: connection class to be used. default: `aiosonic.connection.Connection`
- **pool_cls**: pool class to be used. default: *aiosonic.pools.SmartPool*
- **resolver**: resolver to be used. default: `aiosonic.resolver.DefaultResolver`
- **ttl_dns_cache**: ttl in milliseconds for dns cache. default: *10000* 10 seconds
- **use_dns_cache**: Flag to indicate usage of dns cache. default: *True*
- **conn_max_requests**: Max requests allowed for a connection. default: *100*

class aiosonic.**HTTPClient**(*connector: Optional[aiosonic.connectors.TCPConnector] = None*, *handle_cookies=False*, *verify_ssl=True*, *proxy: Optional[aiosonic.proxy.Proxy] = None*)

aiosonic.HTTPClient class.

This class holds the client creation that will be used for requests.

Params:

- **connector**: TCPConnector to be used if provided
- **handle_cookies: Flag to indicate if keep response cookies in** client and send them in next requests.
- **verify_ssl**: Flag to indicate if verify ssl certificates.

async aiosonic.HTTPClient.**request**(*self*, *url: str*, *method: str = 'GET'*, *headers: Optional[Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders]] = None*, *params: Optional[Union[Dict[str, str], Sequence[Tuple[str, str]]]] = None*, *data: Optional[Union[str, bytes, dict, tuple, AsyncIterator[bytes], Iterator[bytes]]] = None*, *multipart: bool = False*, *verify: bool = True*, *ssl: Optional[ssl.SSLContext] = None*, *timeouts: Optional[aiosonic.timeout.Timeouts] = None*, *follow: bool = False*, *http2: bool = False*) → *aiosonic.HttpResponse*

Do http request.

Params:

- **url**: url of request
- **method**: Http method of request
- **headers**: headers to add in request
- **params**: query params to add in request if not manually added
- **data**: Data to be sent, this param is ignored for get requests.
- **multipart**: Tell aiosonic if request is multipart
- **verify**: parameter to indicate whether to verify ssl
- **ssl**: this parameter allows to specify a custom ssl context
- **timeouts**: parameter to indicate timeouts for request
- **follow**: parameter to indicate whether to follow redirects
- **http2**: flag to indicate whether to use http2 (experimental)

async aiosonic.HTTPClient.**get**(*self*, *url: str*, *headers: Optional[Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders]] = None*, *params: Optional[Union[Dict[str, str], Sequence[Tuple[str, str]]]] = None*, *verify: bool = True*, *ssl: Optional[ssl.SSLContext] = None*, *timeouts: Optional[aiosonic.timeout.Timeouts] = None*, *follow: bool = False*, *http2: bool = False*) → *aiosonic.HttpResponse*

Do get http request.

async aiosonic.HTTPClient.**post**(*self, url: str, data: Optional[Union[str, bytes, dict, tuple, AsyncIterator[bytes], Iterator[bytes]]] = None, headers: Optional[Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders]] = None, json: Optional[Union[dict, list]] = None, params: Optional[Union[Dict[str, str], Sequence[Tuple[str, str]]]] = None, json_serializer=<function dumps>, multipart: bool = False, verify: bool = True, ssl: Optional[ssl.SSLContext] = None, timeouts: Optional[aiosonic.timeout.Timeouts] = None, follow: bool = False, http2: bool = False*) → *aiosonic.HttpResponse*

Do post http request.

async aiosonic.HTTPClient.**put**(*self, url: str, data: Optional[Union[str, bytes, dict, tuple, AsyncIterator[bytes], Iterator[bytes]]] = None, headers: Optional[Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders]] = None, json: Optional[Union[dict, list]] = None, params: Optional[Union[Dict[str, str], Sequence[Tuple[str, str]]]] = None, json_serializer=<function dumps>, multipart: bool = False, verify: bool = True, ssl: Optional[ssl.SSLContext] = None, timeouts: Optional[aiosonic.timeout.Timeouts] = None, follow: bool = False, http2: bool = False*) → *aiosonic.HttpResponse*

Do put http request.

async aiosonic.HTTPClient.**patch**(*self, url: str, data: Optional[Union[str, bytes, dict, tuple, AsyncIterator[bytes], Iterator[bytes]]] = None, headers: Optional[Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders]] = None, json: Optional[Union[dict, list]] = None, params: Optional[Union[Dict[str, str], Sequence[Tuple[str, str]]]] = None, json_serializer=<function dumps>, multipart: bool = False, verify: bool = True, ssl: Optional[ssl.SSLContext] = None, timeouts: Optional[aiosonic.timeout.Timeouts] = None, follow: bool = False, http2: bool = False*) → *aiosonic.HttpResponse*

Do patch http request.

async aiosonic.HTTPClient.**delete**(*self, url: str, data: Union[str, bytes, dict, tuple, AsyncIterator[bytes], Iterator[bytes]] = b'', headers: Optional[Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders]] = None, json: Optional[Union[dict, list]] = None, params: Optional[Union[Dict[str, str], Sequence[Tuple[str, str]]]] = None, json_serializer=<function dumps>, multipart: bool = False, verify: bool = True, ssl: Optional[ssl.SSLContext] = None, timeouts: Optional[aiosonic.timeout.Timeouts] = None, follow: bool = False, http2: bool = False*) → *aiosonic.HttpResponse*

Do delete http request.

async aiosonic.HTTPClient.**wait_requests**(*self, timeout: int = 30*)

Wait until all pending requests are done.

If timeout, returns false.

This is useful when doing safe shutdown of a process.

7.2.2 Classes

class aiosonic.**HttpHeaders**(*data=None*, ***kwargs*)
Http headers dict.

class aiosonic.**HttpResponse**
Custom HttpResponse class for handling responses.

Properties:

- **status_code** (int): response status code
- **headers** (*aiosonic.HttpHeaders*): headers in case insensitive dict
- **cookies** (http.cookies.SimpleCookie): instance of SimpleCookies if cookies present in respone.
- **raw_headers** (List[Tuple[bytes, bytes]]): headers as raw format

async **content**() → bytes
Read response body.

async **json**(*json_decoder=<function loads>*) → dict
Read response body.

read_chunks() → AsyncIterator[bytes]
Read chunks from chunked response.

property **status_code: int**
Get status code.

async **text**() → str
Read response body.

7.2.3 Tiemout Class

class aiosonic.timeout.**Timeouts**(*sock_connect: Optional[float] = 5*, *sock_read: Optional[float] = 30*, *pool_acquire: Optional[float] = None*, *request_timeout: Optional[float] = 60*)
Timeouts class wrapper.

Arguments:

- sock_connect(float): time for establish connection to server
- sock_read(float): time until get first read
- pool_acquire(float): time until get connection from connection's pool
- request_timeout(float): time until complete request.

7.2.4 Pool Classes

class aiosonic.pools.**SmartPool**(*connector*, *pool_size*, *connection_cls*)
Pool which utilizes alive connections.

async acquire(*urlparsed: Optional[urllib.parse.ParseResult] = None*)
Acquire connection.

async cleanup() → None
Get all conn and close them, this method let this pool unusable.

is_all_free()
Indicates if all pool is free.

release(*conn*) → None
Release connection.

class aiosonic.pools.**CyclicQueuePool**(*connector*, *pool_size*, *connection_cls*)
Cyclic queue pool of connections.

async acquire(*_urlparsed: Optional[urllib.parse.ParseResult] = None*)
Acquire connection.

async cleanup()
Get all conn and close them, this method let this pool unusable.

is_all_free()
Indicates if all pool is free.

async release(*conn*)
Release connection.

7.2.5 DNS Resolver

For custom dns servers, you sould install *aiodns* package and use Async resolver as follow

```
from aiosonic.resolver import AsyncResolver

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

Then, pass connector to aiosonic HTTPClient instance.

class aiosonic.resolver.**AsyncResolver**(**args: Any*, ***kwargs: Any*)
Use the *aiodns* package to make asynchronous DNS lookups

async close() → None
Release resolver

async resolve(*host: str*, *port: int = 0*, *family: int = AddressFamily.AF_INET*) → List[Dict[str, Any]]
Return IP address for given hostname

class aiosonic.resolver.**ThreadedResolver**
Use Executor for synchronous getaddrinfo() calls, which defaults to concurrent.futures.ThreadPoolExecutor.

async close() → None
Release resolver

async resolve(*hostname: str*, *port: int = 0*, *family: int = AddressFamily.AF_INET*) → List[Dict[str, Any]]
Return IP address for given hostname

7.2.6 Proxy Support

class aiosonic.proxy.**Proxy**(*host: str*, *auth: Optional[str] = None*)
Proxy class.

Args:

- host (str): proxy server where to connect
- auth (str): auth data in the format of *user:password*

INDEX

A

acquire() (*aiosonic.pools.CyclicQueuePool method*), 23
acquire() (*aiosonic.pools.SmartPool method*), 23
AsyncResolver (*class in aiosonic.resolver*), 23

C

cleanup() (*aiosonic.pools.CyclicQueuePool method*), 23
cleanup() (*aiosonic.pools.SmartPool method*), 23
close() (*aiosonic.resolver.AsyncResolver method*), 23
close() (*aiosonic.resolver.ThreadedResolver method*), 24
content() (*aiosonic.HttpResponse method*), 22
CyclicQueuePool (*class in aiosonic.pools*), 23

D

delete() (*in module aiosonic.HTTPClient*), 21

G

get() (*in module aiosonic.HTTPClient*), 20

H

HTTPClient (*class in aiosonic*), 19
HttpHeaders (*class in aiosonic*), 22
HttpResponse (*class in aiosonic*), 22

I

is_all_free() (*aiosonic.pools.CyclicQueuePool method*), 23
is_all_free() (*aiosonic.pools.SmartPool method*), 23

J

json() (*aiosonic.HttpResponse method*), 22

P

patch() (*in module aiosonic.HTTPClient*), 21
post() (*in module aiosonic.HTTPClient*), 20
Proxy (*class in aiosonic.proxy*), 24
put() (*in module aiosonic.HTTPClient*), 21

R

read_chunks() (*aiosonic.HttpResponse method*), 22
release() (*aiosonic.pools.CyclicQueuePool method*), 23
release() (*aiosonic.pools.SmartPool method*), 23
request() (*in module aiosonic.HTTPClient*), 19
resolve() (*aiosonic.resolver.AsyncResolver method*), 23
resolve() (*aiosonic.resolver.ThreadedResolver method*), 24

S

SmartPool (*class in aiosonic.pools*), 23
status_code (*aiosonic.HttpResponse property*), 22

T

TCPConnector (*class in aiosonic.connectors*), 19
text() (*aiosonic.HttpResponse method*), 22
ThreadedResolver (*class in aiosonic.resolver*), 24
Timeouts (*class in aiosonic.timeout*), 22

W

wait_requests() (*in module aiosonic.HTTPClient*), 21