aiosonic

Release 0.9.6

Johanderson Mogollon

Mar 20, 2021

CONTENTS

Really Fast Python asyncio HTTP 1.1 client, Support for http 2.0 is planned.

Current version is 0.9.6.

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.

CHAPTER

TWO

REQUIREMENTS

- Python>=3.6
- PyPy >=3.6

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(
        sock_read=10,
        sock_connect=3
    )
    response = await client.get('https://www.google.com/', timeouts=timeouts)
```

(continues on next page)

(continued from previous page)

```
    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 110.03 ms",
 "aiosonic cyclic": "1000 requests in 332.10 ms",
 "aiohttp": "1000 requests in 427.31 ms",
 "requests": "1000 requests in 4915.04 ms",
 "httpx": "1000 requests in 638.04 ms"
}
aiosonic is 288.36% faster than aiohttp
aiosonic is 4367.04% faster than requests
aiosonic is 201.83% faster than aiosonic cyclic
aiosonic is 479.89% faster than httpx
```

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])

asyncio.run(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))


 asyncio.run(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
from urllib.parse import urlencode


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())


asyncio.run(main())
```

7.2 Reference

TODO: get better this page

7.2.1 Connector and Client Client

class aiosonic.connectors.**TCPConnector**(*pool_size: int = 25*, *timeouts:* aiosonic.timeout.Timeouts *= None*, *connection_cls=None*, *pool_cls=None*)

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*

class aiosonic.**HTTPClient**(*connector:* aiosonic.connectors.TCPConnector *= None*, *handle_cookies=False*, *verify_ssl=True*)

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: Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders] = None*, *params: Union[Dict[str, str], Sequence[Tuple[str, str]]] = None*, *data: Union[str, bytes, dict, tuple, AsyncIterator[bytes], Iterator[bytes]] = None*, *multipart: bool = False*, *verify: bool = True*, *ssl: ssl.SSLContext = None*, *timeouts: 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: Union[Dict[str, str], List[Tuple[str, str]], aiosonic.HttpHeaders] = None*, *params: Union[Dict[str, str], Sequence[Tuple[str, str]]] = None*, *verify: bool = True*, *ssl: ssl.SSLContext = None*, *timeouts: aiosonic.timeout.Timeouts = None*, *follow: bool = False*, *http2: bool = False*) → *aiosonic.HttpResponse*

Do get http request.

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

Do post http request.

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

Do put http request.

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

INDEX