

Julien Danjou – Python and fast HTTP clients

PYTHON

Python and fast HTTP clients



JULIEN DANJOU

7 OCT 2019 • 6 MIN READ



Nowadays, it is more than likely that you will have to write an HTTP client for your application that will have to talk to another HTTP server. The ubiquity of REST API makes HTTP a first class citizen. That's why knowing optimization patterns are a prerequisite.

Julien Danjou – Python and fast HTTP clients

work with is [requests](#). It is the de-factor standard nowadays.

Persistent Connections

The first optimization to take into account is the use of a persistent connection to the Web server. Persistent connections are a standard since HTTP 1.1 though many applications do not leverage them. This lack of optimization is simple to explain if you know that when using *requests* in its simple mode (e.g. with the `get` function) the connection is closed on return. To avoid that, an application needs to use a `Session` object that allows reusing an already opened connection.

```
import requests

session = requests.Session()
session.get("http://example.com")
# Connection is re-used
session.get("http://example.com")
```

Using Session with requests

Each connection is stored in a pool of connections (10 by default), the size of which is also configurable:

```
import requests

session = requests.Session()
adapter = requests.adapters.HTTPAdapter(
    pool_connections=100,
    pool_maxsize=100)
```

Reusing the TCP connection to send out several HTTP requests offers a number of performance advantages:

- Lower CPU and memory usage (fewer connections opened simultaneously).
- Reduced latency in subsequent requests (no TCP handshaking).
- Exceptions can be raised without the penalty of closing the TCP connection.

The HTTP protocol also provides pipelining, which allows sending several requests on the same connection without waiting for the replies to come (think batch). Unfortunately, this is not supported by the *requests* library. However, pipelining requests may not be as fast as sending them in parallel. Indeed, the HTTP 1.1 protocol forces the replies to be sent in the same order as the requests were sent – first-in first-out.

Parallelism

requests also has one major drawback: it is synchronous. Calling `requests.get("http://example.org")` blocks the program until the HTTP server replies completely. Having the application waiting and doing nothing can be a drawback here. It is possible that the program could do something else rather than sitting idle.

A smart application can mitigate this problem by using a pool of threads like the ones provided by `concurrent.futures`. It allows parallelizing the HTTP requests in a very rapid way.

```
from concurrent import futures

import requests
```

Julien Danjou – Python and fast HTTP clients

```
with futures.ThreadPoolExecutor(max_workers=4) as executor:
    futures = [
        executor.submit(
            lambda: requests.get("http://example.org")
        ) for _ in range(8)
    ]

results = [
    f.result().status_code
    for f in futures
]

print("Results: %s" % results)
```

Using futures with requests

This pattern being quite useful, it has been packaged into a library named [requests-futures](#). The usage of `Session` objects is made transparent to the developer:

```
from requests_futures import sessions

session = sessions.FuturesSession()

futures = [
    session.get("http://example.org")
    for _ in range(8)
]

results = [
    f.result().status_code
    for f in futures
]
```

Using futures with requests

By default a worker with two threads is created, but a program can easily customize this value by passing the `max_workers` argument or even its own executor to the `FuturesSession` object – for example like this: `FuturesSession(executor=ThreadPoolExecutor(max_workers=10))` .

Asynchronicity

As explained earlier, *requests* is entirely synchronous. That blocks the application while waiting for the server to reply, slowing down the program. Making HTTP requests in threads is one solution, but threads do have their own overhead and this implies parallelism, which is not something everyone is always glad to see in a program.

Starting with version 3.5, Python offers asynchronicity as its core using *asyncio*. The `aiohttp` library provides an asynchronous HTTP client built on top of *asyncio*. This library allows sending requests in series but without waiting for the first reply to come back before sending the new one. In contrast to HTTP pipelining, *aiohttp* sends the requests over multiple connections in parallel, avoiding the ordering issue explained earlier.

```
import aiohttp
import asyncio

async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            return response

loop = asyncio.get_event_loop()
```

Julien Danjou – Python and fast HTTP clients

```
results = loop.run_until_complete(asyncio.gather(*c
print("Results: %s" % results)
```

Using aiohttp

All those solutions (using `Session`, *threads*, *futures* or *asyncio*) offer different approaches to making HTTP clients faster.

Performances

The snippet below is an HTTP client sending requests to `httpbin.org`, an HTTP API that provides (among other things) an endpoint simulating a long request (a second here). This example implements all the techniques listed above and times them.

```
import contextlib
import time

import aiohttp
import asyncio
import requests
from requests_futures import sessions

URL = "http://httpbin.org/delay/1"
TRIES = 10

@contextlib.contextmanager
def report_time(test):
    t0 = time.time()
    yield
    print("Time needed for '%s' called: %.2fs"
```


Julien Danjou – Python and fast HTTP clients

```
with report_time("serialized"):
    for i in range(TRIES):
        requests.get(URL)

session = requests.Session()
with report_time("Session"):
    for i in range(TRIES):
        session.get(URL)

session = sessions.FuturesSession(max_workers=2)
with report_time("FuturesSession w/ 2 workers"):
    futures = [session.get(URL)
                for i in range(TRIES)]
    for f in futures:
        f.result()

session = sessions.FuturesSession(max_workers=TRIES)
with report_time("FuturesSession w/ max workers"):
    futures = [session.get(URL)
                for i in range(TRIES)]
    for f in futures:
        f.result()

async def get(url):
    async with aiohttp.ClientSession() as session:
        async with session.get(url) as response:
            await response.read()

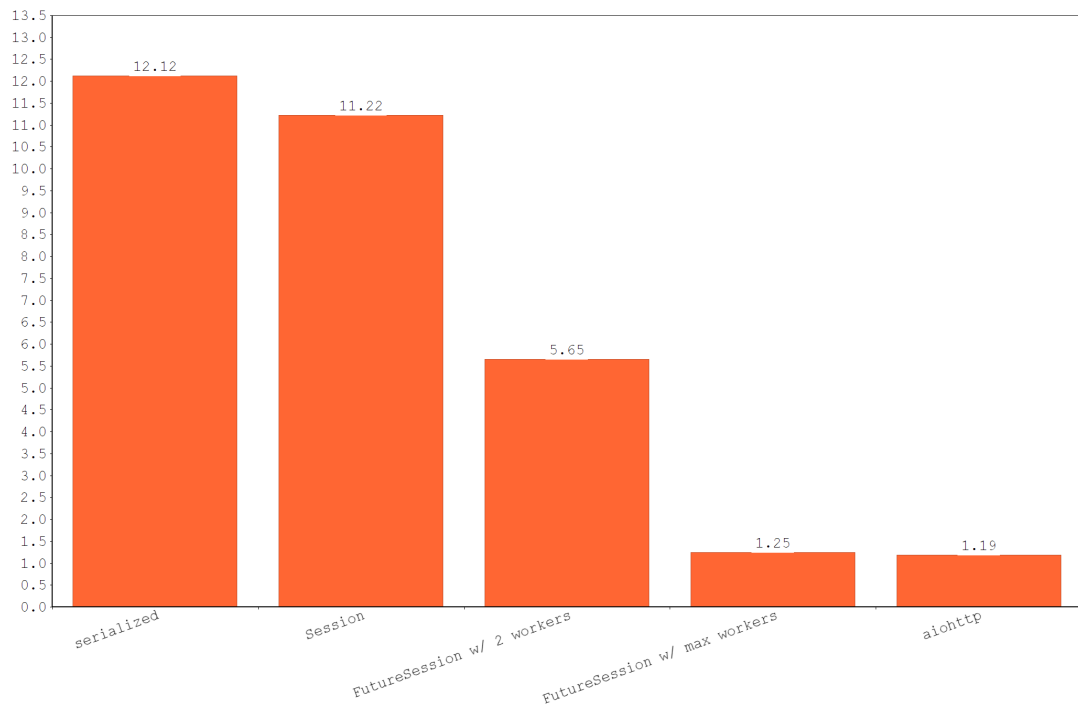
loop = asyncio.get_event_loop()
with report_time("aiohttp"):
    loop.run_until_complete(
```

Julien Danjou – Python and fast HTTP clients

Program to compare the performances of different requests usage

Running this program gives the following output:

```
Time needed for `serialized` called: 12.12s  
Time needed for `Session` called: 11.22s  
Time needed for `FuturesSession w/ 2 workers` called: 5.65s  
Time needed for `FuturesSession w/ max workers` called: 1.25s  
Time needed for `aiohttp` called: 1.19s
```



Without any surprise, the slower result comes with the dumb serialized version, since all the requests are made one after another without reusing the connection — 12 seconds to make 10 requests.

Using a `Session` object and therefore reusing the connection means saving 8% in terms of time, which is already a big and easy win.

Julien Danjou – Python and fast HTTP clients

... your system and program with a large number of threads, it is a good idea to use them to parallelize the requests. However threads have some overhead, and they are not weight-less. They need to be created, started and then joined.

Unless you are still using old versions of Python, without a doubt using *aiohttp* should be the way to go nowadays if you want to write a fast and asynchronous HTTP client. It is the fastest and the most scalable solution as it can handle hundreds of parallel requests. The alternative, managing hundreds of threads in parallel is not a great option.

Streaming

Another speed optimization that can be efficient is streaming the requests. When making a request, by default the body of the response is downloaded immediately. The `stream` parameter provided by the *requests* library or the `content` attribute for *aiohttp* both provide a way to not load the full content in memory as soon as the request is executed.

```
import requests

# Use `with` to make sure the response stream is closed and
# be returned back to the pool.
with requests.get('http://example.org', stream=True) as r:
    print(list(r.iter_content()))
```

Streaming with requests

```
import aiohttp
import asyncio
```

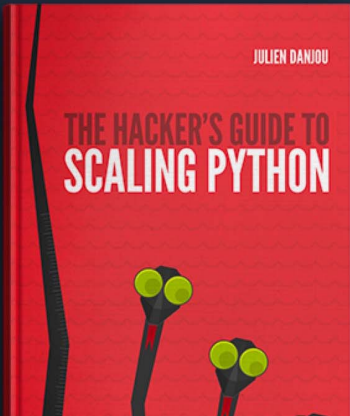
Julien Danjou – Python and fast HTTP clients

```
async def get(url):  
    async with aiohttp.ClientSession() as session:  
        async with session.get(url) as response:  
            return await response.content.read()  
  
loop = asyncio.get_event_loop()  
tasks = [asyncio.ensure_future(get("http://example.com/"))]  
loop.run_until_complete(asyncio.wait(tasks))  
print("Results: %s" % [task.result() for task in tasks])
```

Streaming with aiohttp

Not loading the full content is extremely important in order to avoid allocating potentially hundred of megabytes of memory for nothing. If your program does not need to access the entire content as a whole but can work on chunks, it is probably better to just use those methods. For example, if you're going to save and write the content to a file, reading only a chunk and writing it at the same time is going to be much more memory efficient than reading the whole HTTP body, allocating a giant pile of memory, and then writing it to disk.

I hope that'll make it easier for you to write proper HTTP clients and requests. If you know any other useful technic or method, feel free to write it down in the comment section below!



Free Chapter!

A book about building scalable apps

Get my free chapter

What do you think?

23 Responses

👤 Upvote 🤔 Funny ❤️ Love 😮 Surprised

Julien Danjou – Python and fast HTTP clients



Angry



Sad

[Comments](#)[Community](#)[Privacy Policy](#)[1 Login](#)[Recommend 2](#)[Tweet](#)[Share](#)[Sort by Best](#)

LOG IN WITH

OR SIGN UP WITH DISQUS

**Just Dale** • 7 months ago

Will aiohttp help me in a Flask application where I typically make one back end REST API request per request that I receive (and immediately process the response)?

4 | • [Reply](#) • [Share](#)**Julien Danjou** Mod Just Dale • 7 months ago

Not really. If there's nothing else for your code to do while it waits for the request to be received, you're stuck.

The best thing you could do, is to improve your Web server throughput by using a Web framework that would be able to yield the execution time to another user request while you wait for your backend request to finish. You'd need an asyncio compatible framework — Flask is not, unfortunately.

2 | • [Reply](#) • [Share](#)**Ci Rok** Just Dale • 7 months ago

Try integrating Responder of Fastapi in your architecture they are fast Asgi frameworks that both allow mounting any Asgi or Wsgi app for better performance and easy migration from the wsgi standard to faster asgi..... cheers

1 | • [Reply](#) • [Share](#)**Alexei Martchenko** • 7 months ago

Hmmmmm really good post, thanks

3 | • [Reply](#) • [Share](#)

Julien Danjou – Python and fast HTTP clients

[Home](#) | [About](#) | [Contact](#) | [Reply](#) | [Share](#)

MORE IN PYTHON

Interview: The Performance of Python

11 May 2020 – 1 min read

Attending FOSDEM 2020

6 Feb 2020 – 1 min read

Python Logging with Datadog

3 Feb 2020 – 4 min read

[See all 65 posts →](#)



PYTHON

Sending Emails in Python — Tutorial with Code Examples

What do you need to send an email with Python? Some basic programming and web knowledge along with the elementary Python skills. I assume you've already had a web

15 OCT 2019 • 11 MIN READ

Julien Danjou – Python and fast HTTP clients



PYTHON

Dependencies Handling in Python

Dependencies are a nightmare for many people. Some even argue they are technical debt. Managing the list of the libraries of your software is a horrible experience. Updating them — automatically?

2 SEP 2019 • 5 MIN READ

Julien Danjou © 2020

[Latest Posts](#) [Facebook](#) [Twitter](#) [Ghost](#)