aiohttp - Asynchronous HTTP requests

William Gómez - Web developer at Hourly

Who am I?

William Gómez

- Web Developer at Hourly
- Python Medellin meetups co-organizer
- Data Science FEM - Mentor
- Volleyball and soccer lover
- Learning about fitness

Contents

Introduction - What is aiohttp?

aiohttp is a Python module built on top of asyncIO.

It allow us to create a client to make HTTP request or a server for listen HTTP request.

aiohttp avoids blocking steps where CPU is doing nothing, giving back the control to the CPU.

Async HTTP client/server for asyncio and Python

Star 11,556

Introduction - Non-blocking code

Blocking IO code will block the thread until data from IO has returned.

So asynchronous code is code that can hang while waiting for a result, in order to let other code run in the meantime.

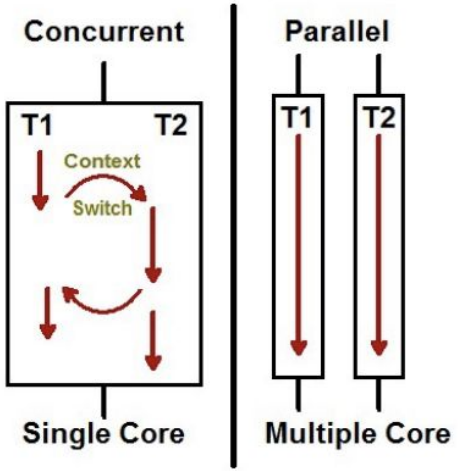
It doesn't "block" other code from running so we can call it "non-blocking" code.

Introduction - asyncIO

asyncIO is a library to write concurrent code using the async/await syntax.

asyncIO provides a set of high-level APIs to:

- Run Python coroutines concurrently
- Distribute tasks via queues
- Synchronize concurrent code



https://breadcrumbscollector.tech/asyncio-choosing-the-right-executor/

Introduction - Coroutines

A coroutine is a specialized version of a “Python generator function”.

A coroutine is a function that can suspend its execution before reaching return, and it can indirectly pass control to another coroutine for some time.

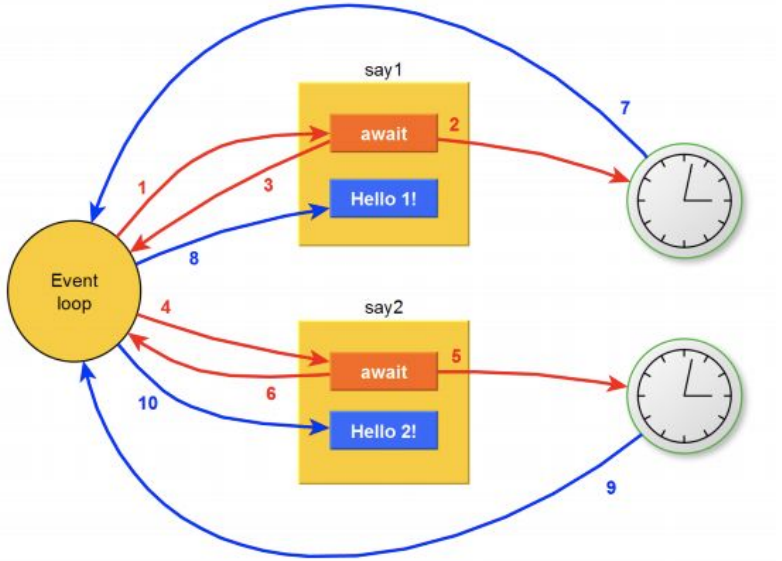
Python

```python
#!/usr/bin/env python3
# countasync.py

import asyncio

async def count():
    print("One")
    await asyncio.sleep(1)
    print("Two")

async def main():
    await asyncio.gather(count(), count(), count())

if __name__ == "__main__":
    import time
    s = time.perf_counter()
    asyncio.run(main())
    elapsed = time.perf_counter() - s
    print(f"{__file__} executed in {elapsed:0.2f} seconds.")
```

Introduction - Event Loop

You can think of an event loop as something like a while True loop that:

- Monitors coroutines
- Taking feedback on what's idle
- Looking around for things that can be executed in the meantime.



Introduction - Pokemon Example

Comparison of retrieving all the first 151st pokemons.

Using requests vs aiohttp client.

Even without using concurrency aiohttp presents a better performance, this is due to its creators were focused on performance.

https://www.twilio.com/blog/asynchronous-http-requests-in-python-with-aiohttp

Introduction - aiohttp Key Features

- Supports both Client and HTTP Server.
- Supports both Server WebSockets and Client WebSockets out-of-the-box without the Callback Hell.
- Web-server has Middlewares, Signals and plugable routing.

aiohttp Client

Most front-end applications need to communicate with a server over the HTTP protocol, in order to download or upload data and access other back-end services.

An HTTP Client can be used to send requests and retrieve their responses.

aiohttp Client - Client Session

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

The idea is to create the session first, then use the instance for performing HTTP requests or initiating WebSocket connections.

aiohttp Client - Client Session

Features:

- Cookies and Connections are shared by the same session.
- Custom Request Headers
- Custom Cookies
- Redirection History
- Proxy support (HTTP) with Auth
- Graceful Shutdown

aiohttp Client - Request Lifecycle

The first time you use aiohttp, you'll notice that a simple HTTP request is performed not with one, but with up to three steps.

```
async with aiohttp.ClientSession() as session:
    async with session.get('http://python.org') as response:
        print(await response.text())
```

The aiohttp Request Lifecycle - Verbose

```
response = requests.get('http://python.org')
print(response.text)
```

```
async with aiohttp.ClientSession() as session:
    async with session.get('http://python.org') as response:
        print(await response.text())
```

aiohttp API is designed to make the most out of non-blocking network operations.

aiohttp gives the event loop three opportunities to switch context.

The aiohttp Request Lifecycle - GET

```
async with session.get('http://python.org') as response:
```

When doing the .get(), both libraries send a GET request to the remote server.

For aiohttp, this means asynchronous I/O, which is marked here with an **async with** that gives you the guarantee that not only it doesn't block, **but that it's cleanly finalized.**

The aiohttp Request Lifecycle - Body loading

```
print(await response.text())
```

aiohttp loads **only the headers** when .get() is executed, letting you decide to pay the cost of loading the body afterward, in a second asynchronous operation.

Hence the **await** response.text().

The aiohttp Request Lifecycle - Close

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

async with aiohttp.ClientSession() does not perform I/O when entering the block, but at the **end of it**, it will ensure all remaining resources are closed correctly.

This is done asynchronously and must be marked as such.

aiohttp Server - Simple Web Server

In order to implement a web server, first create a request handler. Next, create an Application instance and register the request handler on a particular HTTP method and path.

After that, run the application by run_app() call.

```
from aiohttp import web

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

app = web.Application()
app.add_routes([web.get('/', hello)])
web.run_app(app)
```

aiohttp Server - Handler - Coroutines

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

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

aiohttp Server - Handler - Coroutines

Handlers are setup to handle requests by registering them with the **Application.add_routes()** on a particular route (HTTP method and path pair) using helpers like get() and post():

```
app.add_routes([web.get('/', handler),
                web.post('/post', post_handler),
                web.put('/put', put_handler)])
```

aiohttp Server - Additional Features

- Reverse URL Constructing
- User Sessions aiohttp_session
- HTTP Forms
- File Uploads
- WebSockets
- Redirects
- Nested applications
- Background tasks

aiohttp Server - Dev Tools

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

aiohttp-devtools

runserver provides a development server with auto-reload, live-reload, static file serving and aiohttp-debugtoolbar integration.

aiohttp Server - Data Sharing

For storing **global-like variables**, feel free to save them in an **Application instance** and get it back in the web-handler:

```
app['my_private_key'] = data



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

aiohttp Server - Data Sharing

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"
```

aiohttp Server - Middlewares

A middleware is a coroutine that can modify either the request or response.

Every middleware should accept two parameters, a request instance and a handler, and return the response or raise an exception.

aiohttp Server - Middlewares

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

Questions?