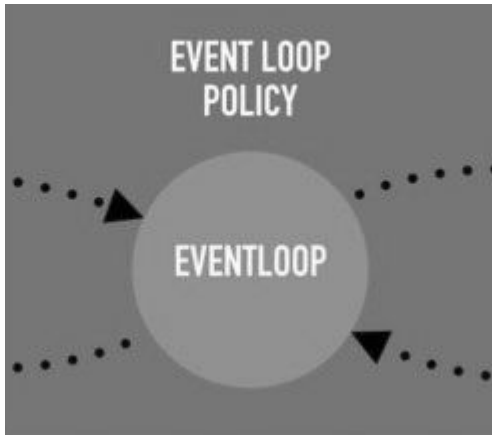
Asynchronous HTTP requests



Leveraging Asynchronous HTTP Requests in Python for Speed and Scalability.

WG

About me

William Gomez

Web Developer & Data Engineer at **Huge**

Full stack wannabe

Python Medellin meetups co-organizer

Volleyball and soccer lover

Learning about **fitness & cooking**

Introduction

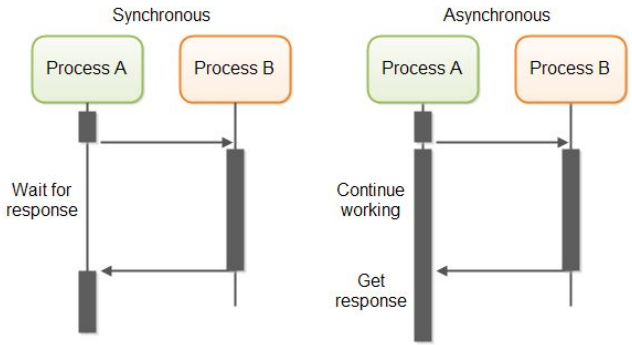
Introduction

Synchronous vs. Asynchronous Requests

Before diving into asynchronous HTTP requests, it's essential to understand the difference between synchronous and asynchronous requests.

Synchronous requests are made one at a time, blocking the program's execution until each request is complete.

On the other hand, asynchronous requests allow multiple requests to be executed concurrently, eliminating blocking and enabling better resource utilization.



Introduction

asyncIO

Asynchronous programming in Python relies on event loops and coroutines. **asyncio** library is at the heart of this paradigm, providing the necessary tools to write asynchronous code.

asyncIO provides a set of high-level APIs to:

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

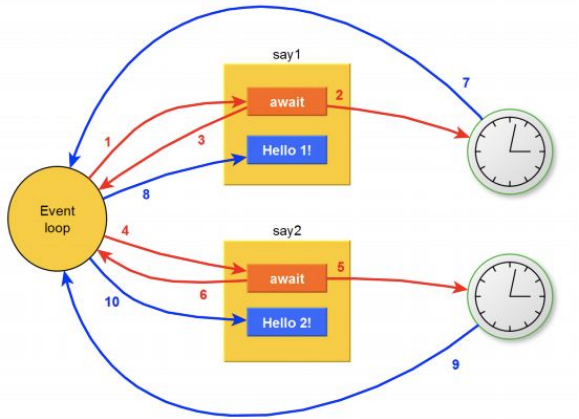
```python
import asyncio

async def delayedPrint(delay, text):
    i = 0
    while(i < 5):
        await asyncio.sleep(delay)
        print(text)
        i += 1


async def main():
    task1 = asyncio.create_task(delayedPrint(2, "First"))
    task2 = asyncio.create_task(delayedPrint(2, "Second"))

    await task1
    await task2

asyncio.run(main())
```



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.

```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

Coroutines

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

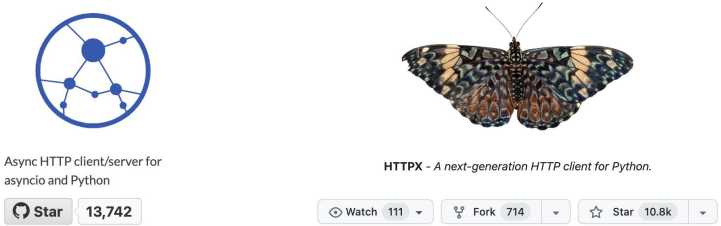
A **coroutine** is a function that can suspend its execution before reaching return, and it can indirectly pass control to another coroutine for some time.

Introduction

Python Libraries for Asynchronous HTTP Requests

Python provides several libraries to make asynchronous HTTP requests. Two popular choices are **aiohttp** and **httpx**.

aiohttp is a powerful library known for its flexibility, while **httpx** is praised for its user-friendly interface and support for HTTP/1.1 and HTTP/2.



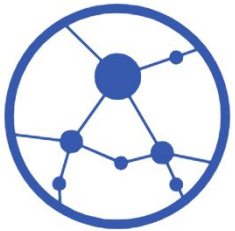
What is aiohttp?

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.

Supports both **Client** and HTTP **Server**.



Async HTTP client/server for asyncio and Python

Star 13,742

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

```
aerodactyl
snorlax
articuno
zapdos
moltres
dratini
dragonair
dragonite
mewtwo
--- 8.101269960403442 seconds ---
```

aiohttp

```
aerodactyl
snorlax
articuno
zapdos
moltres
dratini
dragonair
dragonite
mewtwo
--- 28.87265682220459 seconds ---
```

requests

```
aerodactyl
snorlax
articuno
zapdos
moltres
dratini
dragonair
dragonite
mewtwo
--- 1.5340027809143066 seconds ---
```

aiohttp and concurrency

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.

Features:

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

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())
```

Request Lifecycle

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

aiohttp gives the event loop three opportunities to **switch context**.

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

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.

```
async with aiohttp.ClientSession()
```

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().

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

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.

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

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)
```

Features:

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

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()
```

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)])
```

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.

Demos

https://github.com/aio-libs/aiohttp-demos

Q & A

Quieres dar una charla?