**Модуль 5: Асинхронне програмування в Python**

**Заняття 2: WebSockets та Aiohttp клієнт**

**Aiohttp клієнт для асинхронних HTTP запитів**

[**AIOHTTP**](https://docs.aiohttp.org/en/stable/)— це асинхронний фреймворк, в якому реалізовано web-стек на основі asyncio в Python. В AIOHTTP реалізовані клієнтська частина та серверна частина. Реалізація HTTP і сам веб-сервер не використовують додаткові зовнішні бібліотеки. Вбудований веб-сервер реалізований на рівні, достатньому для використання в реальних навантажених проектах.

Фреймворк доступний у репозиторії пакетів **pip** та встановлюється:

pip install aiohttp

**Виконання запиту**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/intro#%D0%B2%D0%B8%D0%BA%D0%BE%D0%BD%D0%B0%D0%BD%D0%BD%D1%8F-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D1%83)Клієнтська частина AIOHTTP зосереджена на підвищенні продуктивності та використанні asyncio. Приклад асинхронного запиту на адресу https://python.org:

import platform

import aiohttp

import asyncio

async def main():

async with aiohttp.ClientSession() as session:

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

print("Status:", response.status)

print("Content-type:", response.headers['content-type'])

html = await response.text()

print("Body:", html[:15], "...")

if \_\_name\_\_ == "\_\_main\_\_":

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

asyncio.run(main())

Виведення:

Status: 200

Content-type: text/html; charset=utf-8

Body: <!doctype html> ...

Відмінною рисою AIOHTTP є обов'язкове використання механізму сесій з'єднань.

**INFO**

Насправді всі пакети, які реалізують веб-клієнтів, дають можливість використовувати та налаштовувати сесії, але часто цей механізм приховується для простоти та кожен запит на сервер супроводжується створенням нового з'єднання. Такий підхід спрощує роботу з пакетом, але сильно зменшує продуктивність.

AIOHTTP сфокусовано на підвищенні продуктивності, тому обійти етап створення сесій не можна. Безпосередньо запит до сервера відбувається в контексті сесії, для цього об'єкт класу ClientSession (session у нашому прикладі) є методи get, post, put та інші.

Ці методи отримують обов'язковий аргумент, адресу сервера та ряд іменованих аргументів, які залежать від запиту:

* data — byte об'єкт із тілом запиту;
* params — словник із набором параметрів запиту;
* json — Python об'єкт, який буде перетворений на JSON і надісланий серверу в тілі запиту;
* headers — словник, значення якого буде додано у заголовок запиту;
* cookies — словник з полями для кукі.

Наступний фрагмент коду у прикладі:

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

ми використовуємо, щоб уникнути помилки RuntimeError: Event loop is closed в системі Windows

**Отримання відповіді сервера**

Відповідь сервера завжди гарантовано містить заголовок і він буде доступним, щойно отримає відповідь. Тіло запиту може бути досить великим (файл великого розміру, потік даних) та AIOHTTP дає можливість обробити тільки заголовок, щоб вирішити завантажувати далі тіло запиту чи ні.

Як приклад, давайте зробимо запит на публічне [API Приватбанку](https://api.privatbank.ua/#p24/exchange), щодо поточного курсу валют.

import platform

import aiohttp

import asyncio

async def main():

async with aiohttp.ClientSession() as session:

async with session.get('https://api.privatbank.ua/p24api/pubinfo?json&exchange&coursid=5') as response:

print("Status:", response.status)

print("Content-type:", response.headers['content-type'])

print('Cookies: ', response.cookies)

print(response.ok)

result = await response.json()

return result

if \_\_name\_\_ == "\_\_main\_\_":

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

r = asyncio.run(main())

print(r)

Виведення прикладу:

Status: 200

Content-type: application/json; charset=UTF-8

Cookies:

True

[{'ccy': 'USD', 'base\_ccy': 'UAH', 'buy': '38.90000', 'sale': '39.40000'}, {'ccy': 'EUR', 'base\_ccy': 'UAH', 'buy': '38.00000', 'sale': '39.00000'}, {'ccy': 'BTC', 'base\_ccy': 'USD', 'buy': '19662.0514', 'sale': '21731.7410'}]

Отримавши відповідь (response у прикладі), відразу доступні атрибути заголовка відповіді:

* status — HTTP [**статус код**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Status)відповіді сервера;
* headers — словник із полями [**заголовка**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers);
* cookies — набір [**http cookies**](https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies), якщо вони були передані;
* ok — спеціальне поле типу bool, яке має значення True, якщо запит був успішним, тобто його статус менше 400;

Обробивши заголовок відповіді, можна отримати тіло запиту, використовуючи один із асинхронних методів:

* read — повертає тіло відповіді у bytes;
* text — рядок із тіла запиту, якщо заголовок Content-type дорівнює text/plain;
* json — автоматично декодує **JSON** об'єкт з тіла запиту в Python об'єкт;

У прикладі ми обробили response методом json та значення result – це список словників:

[

{

'ccy': 'USD',

'base\_ccy': 'UAH',

'buy': '38.90000',

'sale': '39.40000'

},

{

'ccy': 'EUR',

'base\_ccy': 'UAH',

'buy': '38.00000',

'sale': '39.00000'

},

{

'ccy': 'BTC',

'base\_ccy': 'USD',

'buy': '19662.0514',

'sale': '21731.7410'

}

]

**Робота із сесіями Aiohttp клієнта**

**Створення сесій**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/context#%D1%81%D1%82%D0%B2%D0%BE%D1%80%D0%B5%D0%BD%D0%BD%D1%8F-%D1%81%D0%B5%D1%81%D1%96%D0%B9)У розглянутих раніше прикладах використовуються менеджери контексту, щоб у будь-якому випадку коректно завершити всі з'єднання та повернути системні ресурси. Це рекомендований підхід і варто використовувати його скрізь, де це можливо.

Однак, ви також можете створювати сесію та закривати її так, як вам зручно:

import platform

import aiohttp

import asyncio

async def main():

session = aiohttp.ClientSession()

response = await session.get('https://python.org')

print("Status:", response.status)

print("Content-type:", response.headers['content-type'])

html = await response.text()

response.close()

await session.close()

return f"Body: {html[:15]}..."

if \_\_name\_\_ == "\_\_main\_\_":

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

r = asyncio.run(main())

print(r)

Виведення:

Status: 200

Content-type: text/html; charset=utf-8

Body: <!doctype html>...

Цей приклад робить те саме, але без використання менеджерів контексту. Але тут ми самостійно закриваємо об'єкт response: response.close(). І об'єкт сесії await session.close()

**Використання однієї сесії при запитах**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/context#%D0%B2%D0%B8%D0%BA%D0%BE%D1%80%D0%B8%D1%81%D1%82%D0%B0%D0%BD%D0%BD%D1%8F-%D0%BE%D0%B4%D0%BD%D1%96%D1%94%D1%97-%D1%81%D0%B5%D1%81%D1%96%D1%97-%D0%BF%D1%80%D0%B8-%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D0%B0%D1%85)Типовий підхід використовувати одну сесію для з'єднання з одним сервісом — це значно прискорює виконання кількох запитів на один і той самий сервіс. У у такому разі ви можете передавати створену сесію як аргумент у функцію:

import platform

import aiohttp

import asyncio

async def index(session):

url = 'https://python.org'

async with session.get(url) as response:

print("Status:", response.status)

print("Content-type:", response.headers['content-type'])

html = await response.text()

return f"Body: {html[:15]}..."

async def doc(session):

url = "https://www.python.org/doc/"

async with session.get(url) as response:

print("Status:", response.status)

print("Content-type:", response.headers['content-type'])

html = await response.text()

return f"Body: {html[:15]}..."

async def main():

async with aiohttp.ClientSession() as session:

result = await asyncio.gather(index(session), doc(session))

return result

if \_\_name\_\_ == "\_\_main\_\_":

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

r = asyncio.run(main())

print(r)

Виведення:

Status: 200

Content-type: text/html; charset=utf-8

Status: 200

Content-type: text/html; charset=utf-8

['Body: <!doctype html>...', 'Body: <!doctype html>...']

У цьому прикладі ми робимо два запити на один і той самий сервіс, але на різні ресурси. Запити виконуються "одночасно" та використовують те саме з'єднання (сесію). Таким чином, час виконання двох запитів буде швидшим і трохи більшим, ніж час виконання найтривалішого із запитів.

**Набір параметрів сесії**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/context#%D0%BD%D0%B0%D0%B1%D1%96%D1%80-%D0%BF%D0%B0%D1%80%D0%B0%D0%BC%D0%B5%D1%82%D1%80%D1%96%D0%B2-%D1%81%D0%B5%D1%81%D1%96%D1%97)Кожен об'єкт ClientSession може бути індивідуально налаштований, щоб усі запити у цій сесії використовували загальний набір параметрів. Для цього ви можете передати в сесію набір параметрів запиту і всі вони будуть автоматично додані у кожен запит цієї сесії:

import platform

import aiohttp

import asyncio

from uuid import uuid4

async def main():

timeout = aiohttp.ClientTimeout(total=1)

async with aiohttp.ClientSession(

headers={"Request-Id": str(uuid4())},

timeout=timeout,

) as session:

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

print("Status:", response.status)

print("Content-type:", response.headers['content-type'])

html = await response.text()

return f"Body: {html[:15]}..."

if \_\_name\_\_ == "\_\_main\_\_":

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

r = asyncio.run(main())

print(r)

У цьому прикладі всім запитам у цій сесії встановили в заголовок поле Request-Id та таймаут на читання 1 секунди.

**Запити іншими методами HTTP**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/context#%D0%B7%D0%B0%D0%BF%D0%B8%D1%82%D0%B8-%D1%96%D0%BD%D1%88%D0%B8%D0%BC%D0%B8-%D0%BC%D0%B5%D1%82%D0%BE%D0%B4%D0%B0%D0%BC%D0%B8-http)Поки що ми виконували тільки GET запити, але документація наводить приклади відправлення для інших методів HTTP:

session.post('http://httpbin.org/post', data=b'data')

session.put('http://httpbin.org/put', data=b'data')

session.delete('http://httpbin.org/delete')

session.head('http://httpbin.org/get')

session.options('http://httpbin.org/get')

session.patch('http://httpbin.org/patch', data=b'data')

Розглянемо приклад відправлення POST запиту. Спочатку створимо найпростіший Web-сервер, який прийматиме і надсилатиме назад дані POST запиту:

from http.server import HTTPServer, BaseHTTPRequestHandler

class HttpHandler(BaseHTTPRequestHandler):

def do\_POST(self):

data = self.rfile.read(int(self.headers['Content-Length']))

print(data)

self.send\_response(201)

self.end\_headers()

self.wfile.write(b'Done request!' + data)

def do\_GET(self):

self.send\_response(200)

self.end\_headers()

self.wfile.write(b'Hello, world!')

def run(server\_class=HTTPServer, handler\_class=HttpHandler):

server\_address = ('', 5000)

http = server\_class(server\_address, handler\_class)

try:

http.serve\_forever()

except KeyboardInterrupt:

http.server\_close()

if \_\_name\_\_ == '\_\_main\_\_':

run()

Він працює на порту 5000 і просто відправить рядок 'Done request!' і додасть набір даних, які ми йому відправимо в POST запиті.

Реалізуємо клієнт:

import platform

import aiohttp

import asyncio

from uuid import uuid4

async def main():

async with aiohttp.ClientSession() as session:

async with session.post('http://localhost:5000', data={"message": "Hello world!"}, ssl=False) as response:

print("Status:", response.status)

html = await response.text()

return f"Body: {html}"

if \_\_name\_\_ == "\_\_main\_\_":

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

r = asyncio.run(main())

print(r)

При запуску клієнта ми відправимо серверу змінну 'message' з рядковим значенням "Hello world!". В результаті отримаємо наступні виведення для сервера:

127.0.0.1 - - [31/Oct/2022 16:29:15] "POST / HTTP/1.1" 201 -

b'message=Hello+world%21'

та для клієнта

Status: 201

Body: Done request!message=Hello+world%21

Аналогічно працює відправлення і для інших методів HTTP.

**Обробка помилок під час запитів**

На жаль, у мережевих запитах трапляються помилки. Це може бути неіснуючий шлях http://www.python.org/asdf або так звані "биті" посилання http://test. Їх потрібно не забувати опрацьовувати.

Розглянемо приклад:

import aiohttp

import asyncio

import platform

urls = ['https://www.google.com', 'https://www.python.org/asdf', 'https://duckduckgo.com', 'http://test']

async def main():

async with aiohttp.ClientSession() as session:

for url in urls:

print(f'Starting {url}')

try:

async with session.get(url) as resp:

if resp.status == 200:

html = await resp.text()

print(url, html[:150])

else:

print(f"Error status: {resp.status} for {url}")

except aiohttp.ClientConnectorError as err:

print(f'Connection error: {url}', str(err))

if \_\_name\_\_ == '\_\_main\_\_':

if platform.system() == 'Windows':

asyncio.set\_event\_loop\_policy(asyncio.WindowsSelectorEventLoopPolicy())

asyncio.run(main())

Виведення:

Starting https://www.google.com

https://www.google.com <!doctype html><html itemscope="" itemtype="http://schema.org/WebPage" lang="uk"><head><meta content="text/html; charset=UTF-8" http-equiv="Content-Ty

Starting https://www.python.org/asdf

Error status: 404 for https://www.python.org/asdf

Starting https://duckduckgo.com

https://duckduckgo.com <!DOCTYPE html>

<!--[if IEMobile 7 ]> <html lang="en-US" class="no-js iem7"> <![endif]-->

<!--[if lt IE 7]> <html class="ie6 lt-ie10 lt-ie9 lt-ie8 lt-

Starting http://test

Connection error: http://test Cannot connect to host test:80 ssl:default [getaddrinfo failed]

Помилки зі статусом вище 200 ми обробляємо простою умовою перевірки:

if resp.status == 200:

html = await resp.text()

print(url, html[:150])

else:

print(f"Error status: {resp.status} for {url}")

Виведення помилки 404:

Starting https://www.python.org/asdf

Error status: 404 for https://www.python.org/asdf

Помилки підключення – це помилки класу aiohttp.ClientConnectorError, які обробляємо конструкцією try/except

try:

async with session.get(url) as resp:

if resp.status == 200:

html = await resp.text()

print(url, html[:150])

else:

print(f"Error status: {resp.status} for {url}")

except aiohttp.ClientConnectorError as err:

print(f'Connection error: {url}', str(err))

Виведення помилки підключення:

Starting http://test

Connection error: http://test Cannot connect to host test:80 ssl:default [getaddrinfo failed]

**WebSockets**

Технологія, де найпотужніше використовується інструмент async/await, – це WebSocket. [WebSocket](https://developer.mozilla.org/en-US/docs/Web/API/WebSocket) - це протокол двоспрямованого обміну даними, що характеризує повністю дуплексний характер взаємодії. На практиці це означає наступне. WebSockets встановлює одне, причому єдине, з'єднання клієнта з сервером. Після необхідних перевірок, що підтверджують, що сервер може працювати з WebSocket, сервер та клієнт можуть надсилати через нього текстові повідомлення, причому передача відбувається відразу ж, при надсиланні WebSockets створює двоспрямовані канали зв'язку. З'єднання постійно тримається відкритим, що дозволяє не передавати зайвих НТTР-заголовків. При цьому у веб-сокетах немає обмежень ні на кількість з'єднань, ні на черговість запитів.

Основні переваги WebSocket у порівнянні з [моделлю довгого опитування HTTP](https://ably.com/topic/long-polling):

* Інформація може передаватися в обидві сторони у будь-який час протягом терміну дії веб-сокету з'єднання.
* Клієнт та сервер постійно з'єднані – дані можуть надсилатися клієнтам увесь час без необхідності отримувати запит.
* Простота роботи з WebSocket у Python. Виконати ефективно те саме завдання з довгим опитуванням HTTP було б значно складніше.

Порівняльний аналіз [WebSockets vs Long Polling](https://ably.com/blog/websockets-vs-long-polling)

**Пакет websockets**[**​**](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/websocket#%D0%BF%D0%B0%D0%BA%D0%B5%D1%82-websockets)

Для того щоб почати працювати з веб-сокетами, потрібні лише дві речі - браузер, що підтримує WebSocket, і сервер, який реалізує цю технологію. Хоча клієнт для WebSocket ми можемо писати і на Python. На стороні браузера все просто - WebSocket підтримується більшістю сучасних версій браузерів.

Для роботи з веб-сокетами потрібен Python версії 3.6.1 і вище.

Пакет [**websockets**](https://websockets.readthedocs.io/)для роботи з веб-сокетами в Python легко встановити наступною командою:

pip install websockets

Розглянемо базовий приклад із документації. У нас є серверна частина:

**server.py**

import asyncio

import websockets

async def hello(websocket):

name = await websocket.recv()

print(f"<<< {name}")

greeting = f"Hello {name}!"

await websocket.send(greeting)

print(f">>> {greeting}")

async def main():

async with websockets.serve(hello, "localhost", 8765):

await asyncio.Future() # run forever

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(main())

Запуск сервера виконується командою asyncio.run(main()). Сама функція main за допомогою асинхронного контексту створює сервер WebSocket командою websockets.serve(hello, "localhost", 8765).

Аргумент hello – це функція обробник повідомлень між сервером та клієнтом, аргумент `localhost' визначає хост для сервера, а 8765 – порт, на якому буде встановлено з'єднання.

У функції hello ми отримуємо параметр websocket, що має тип WebSocketServerProtocol. Отримуємо повідомлення від клієнта за допомогою name = await websocket.recv(), а відправляємо повідомлення виразом await websocket.send(greeting). Як бачимо, по суті, це ехо-сервер.

**INFO**

Документація каже, що serve() виконує співпрограму обробки з'єднання hello() один раз для кожного з'єднання WebSocket. Він закриває з'єднання WebSocket, коли обробник повертається.

Тепер розглянемо клієнтську частину:

**client.py**

import asyncio

import websockets

async def hello():

uri = "ws://localhost:8765"

async with websockets.connect(uri) as websocket:

name = input("What's your name? ")

await websocket.send(name)

print(f">>> {name}")

greeting = await websocket.recv()

print(f"<<< {greeting}")

if \_\_name\_\_ == "\_\_main\_\_":

asyncio.run(hello())

Тут лише одна функція hello, яка надсилає ім'я на сервер, отримує привітання та закриває з'єднання.

URL ресурсу веб-сокету використовує власну схему, що починається з ws (або wss для безпечного підключення). Далі йде ім'я хосту та номер порту ws://localhost:8765.

async with websockets.connect('ws://localhost:8765') as websocket:

Наступний рядок відкриває з'єднання із веб-сокетом, використовуючи websockets.connect. Очікування з'єднання викликає WebSocketClientProtocol, який потім використовується для надсилання та отримання повідомлень. Цей рядок використовує async with, який працює з асинхронним контекстним менеджером. З'єднання закривається під час виходу з контексту. Далі йде вже знайоме нам по серверу відправлення та отримання повідомлень за допомогою await websocket.send(name) і greeting = await websocket.recv().

Запустимо сервер, а далі виконаємо код клієнта у консолі.

Виведення у терміналі клієнта:

What's your name? Krabat

>>> Krabat

<<< Hello Krabat!

Виведення у терміналі сервера:

<<< Krabat

>>> Hello Krabat!

Клієнт після відправлення повідомлення завершує виконання, а сервер продовжує працювати і чекати на нові підключення, хоча поточне з'єднання клієнта і закрилося.

Ось так просто надсилати повідомлення по протоколу WebSocket. Наступним кроком ми розглянемо роботу WebSocket з браузером.

**WebSockets робота з браузером**

Спочатку створимо сервер за аналогією до попереднього прикладу.

import asyncio

import websockets

async def handler(websocket, path):

data = await websocket.recv()

reply = f"Data recieved as: {data}!"

print(reply)

await websocket.send(reply)

async def main():

async with websockets.serve(handler, "localhost", 8000):

await asyncio.Future() # run forever

if \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(main())

Тут нам уже все знайоме, ми отримуємо дані, далі додаємо перед ними рядок Data recieved as: і відправляємо назад клієнту. Різниця в тому, що сервер веб-сокет працює на 8000 порту.

Клієнт цього разу у нас буде на javascript вбудований всередину html файлу.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<title>WebSocker Client</title>

</head>

<body>

<button id="btnSend">Click Here</button>

<div id="result"></div>

</body>

<script>

const connect = (message) => {

let socket = new WebSocket('ws://localhost:8000');

socket.addEventListener('open', function (event) {

if (message) {

socket.send(message)

} else {

socket.send('Connection Established');

}

});

socket.addEventListener('message', function (event) {

console.log(event.data);

result.textContent = event.data

});

socket.addEventListener('close', (e) => {

console.log('Socket is closed.', e.reason);

result.textContent = result.textContent + ' Socket is closed.'

})

}

connect()

btnSend.addEventListener('click', (e) => {

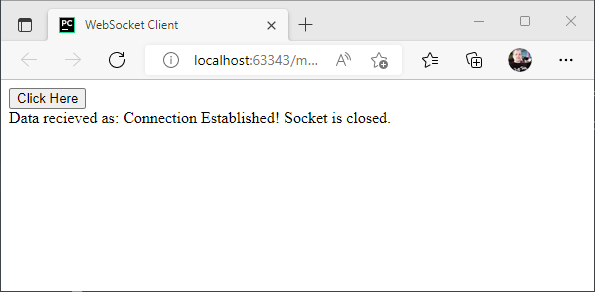
connect("Click button");

})

</script>

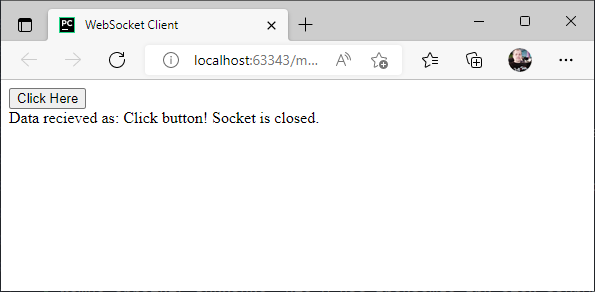
</html>

Якщо ми відкриємо нашу сторінку, то у нас повинна бути наступна картина:



**Перше з'єднання з веб-сокет сервером**

Можна звернути увагу, що у нас відбулося при цьому закриття з'єднання WebSocket, тому що ми бачимо фразу 'Socket is closed.' Сталася подія close. Якщо ми виконаємо клік по кнопці Click Here, то javascript піде на хитрощі — відкриє з'єднання веб-сокет заново і надішле нове повідомлення.



**Повторне з'єднання з веб-сокет сервером по кліку**

Це робочий приклад, але наш сервер зараз не зберігає з'єднання. Простими словами сервер просто забуває клієнта, щойно надішле йому повідомлення.

Запустіть для демонстрації живий приклад. Спочатку запустіть сервер, якщо він ще не запущений.

Після цього ви можете оновити сторінку клієнта, натиснувши кнопку Rerun справа внизу фрейму.

NOTE

Цілком можливо, що веб-сокет вже запущений іншим учнем. Тоді ви побачите повідомлення

"Data recieved as: Connection Established! Socket is closed." Це свідчить про те, що з'єднання встановлено і веб-сокет активний. Для незапущеного веб-сокету повідомлення буде "Socket is closed."

На наступному кроці ми створимо веб-сокет із постійним з'єднанням.

**WebSockets — з постійним з'єднанням**

Ми створимо три різні скрипти.

* consumer.py — який буде слухати та отримувати повідомлення від сервера Websocket
* producer.py — який надсилає повідомлення на сервер
* server.py — власне сам Websocket сервер, який пам'ятатиме про свої підключення

**Consumer**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/connect#consumer)Спочатку створимо простого споживача повідомлень від Websocket сервера.

**consumer.py**

import asyncio

import logging

import websockets

logging.basicConfig(level=logging.INFO)

async def consumer(hostname: str, port: int):

ws\_resource\_url = f"ws://{hostname}:{port}"

async with websockets.connect(ws\_resource\_url) as ws:

async for message in ws:

logging.info(f"Message: {message}")

if \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(consumer('localhost', 4000))

Тут за допомогою функції consumer ми підключаємося до віддаленого Websocket сервера на порту 4000. Як і раніше, через асинхронний контекст підключення, отримуємо екземпляр ws класу WebSocketClientProtocol:

async with websockets.connect(ws\_resource\_url) as ws:

Далі за допомогою асинхронного циклу async for ми виконуватимемо перебір асинхронного ітератора ws і логуємо отримані повідомлення.

async for message in ws:

logging.info(f"Message: {message}")

**Producer**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/connect#producer)Відправлятимемо повідомлення за допомогою producer. Ми підключаємося до веб-сокету так само, як робили це раніше з consumer. А далі відправляємо асинхронне повідомлення на сервер await ws.send(message) та відключаємося.

**producer.py**

import asyncio

import websockets

import sys

async def producer(message: str, host: str, port: int):

async with websockets.connect(f"ws://{host}:{port}") as ws:

await ws.send(message)

if \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(producer(message=sys.argv[1], host='localhost', port=4000))

**Server**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/connect#server)Інтерес для нас представляє сервер. Він є класом Server, який поєднує весь функціонал сервера. Цей сервер розсилає повідомлення, надіслані producer, всім слухачам consumer.

Екземпляр класу Server має асинхронну функцію ws\_handler, яка і визначає співпрограму оброблювача веб-сокета. При підключенні клієнта, функція ws\_handler приймає з'єднання, створює екземпляр ws класу WebSocketServerProtocol і здійснює "рукостискання" (handshake). Далі ми запам'ятовуємо екземпляр клієнта за допомогою функції register і поміщаємо його у змінну класу clients, яка є множиною. Як тільки обробник завершує роботу, нормально або за винятком WebSocketProtocolError, сервер виконує закриваюче "рукостискання" та закриває з'єднання, і видаляє дані про клієнта за допомогою функції unregister.

import asyncio

import logging

import websockets

from websockets import WebSocketServerProtocol

from websockets.exceptions import ConnectionClosedOK

logging.basicConfig(level=logging.INFO)

class Server:

clients = set()

async def register(self, ws: WebSocketServerProtocol):

self.clients.add(ws)

logging.info(f'{ws.remote\_address} connects')

async def unregister(self, ws: WebSocketServerProtocol):

self.clients.remove(ws)

logging.info(f'{ws.remote\_address} disconnects')

async def send\_to\_clients(self, message: str):

if self.clients:

[await client.send(message) for client in self.clients]

async def ws\_handler(self, ws: WebSocketServerProtocol):

await self.register(ws)

try:

await self.distrubute(ws)

except ConnectionClosedOK:

pass

finally:

await self.unregister(ws)

async def distrubute(self, ws: WebSocketServerProtocol):

async for message in ws:

await self.send\_to\_clients(message)

async def main():

server = Server()

async with websockets.serve(server.ws\_handler, 'localhost', 4000):

await asyncio.Future() # run forever

if \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(main())

Запуск сервера призведе до виконання співпрограми main, яка запустить веб-сокет server.ws\_handler. Наш метод ws\_handler реєструє з'єднання await self.register(ws), відправляє повідомлення підключеним клієнтам await self.distrubute(ws) і, нарешті, закриває з'єднання await self.unregister(ws).

Нагадаємо, що consumer залишається підключеним до сервера, у той час як producer скасовує власну реєстрацію.

**INFO**

Відключення від сервера призведе до помилки ConnectionClosedOK. Сервер чекає наступного повідомлення з recv(), коли клієнт вимкнувся. Коли це відбувається, websockets підіймає виняток ConnectionClosedOK, щоб ви знали, що ви не отримаєте інше повідомлення по цьому з'єднанні. Тому обробляємо його як рекомендує документація.

Метод distribute надсилатиме кожне повідомлення у веб-сокеті всім клієнтам у списку підключених клієнтів clients.

**Запуск**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/connect#%D0%B7%D0%B0%D0%BF%D1%83%D1%81%D0%BA)Запустимо сервер і ми повинні побачити наступне виведення:

INFO:websockets.server:server listening on 127.0.0.1:4000

INFO:websockets.server:server listening on [::1]:4000

Запускаємо 'consumer.py' у нього поки немає жодного виведення в термінал, а у сервера бачимо нове виведення:

INFO:websockets.server:connection open

INFO:root:('::1', 53648, 0, 0) connects

Це означає, що consumer успішно з'єднався із сервером і чекає на повідомлення. Сервер зберіг з'єднання consumer у властивість clients.

Запускаємо producer командою:

py .\produce.py 'Hi all!'

У терміналі consumer ми побачимо отримане повідомлення:

INFO:root:Message: Hi all!

У сервера з'явиться запис у логах, що producer приєднався, відправив повідомлення та успішно закрив з'єднання:

INFO:websockets.server:connection open

INFO:root:('::1', 53657, 0, 0) connects

INFO:root:('::1', 53657, 0, 0) disconnects

INFO:websockets.server:connection closed

Все працює, як ми і задумали спочатку.

**Створюємо найпростіший чат на WebSockets**

**Код сервера**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/chat#%D0%BA%D0%BE%D0%B4-%D1%81%D0%B5%D1%80%D0%B2%D0%B5%D1%80%D0%B0)Давайте тепер побудуємо найпростіший веб-чат. За основу візьмемо сервер із попереднього розділу.

import asyncio

import logging

import websockets

import names

from websockets import WebSocketServerProtocol

from websockets.exceptions import ConnectionClosedOK

logging.basicConfig(level=logging.INFO)

class Server:

clients = set()

async def register(self, ws: WebSocketServerProtocol):

ws.name = names.get\_full\_name()

self.clients.add(ws)

logging.info(f'{ws.remote\_address} connects')

async def unregister(self, ws: WebSocketServerProtocol):

self.clients.remove(ws)

logging.info(f'{ws.remote\_address} disconnects')

async def send\_to\_clients(self, message: str):

if self.clients:

[await client.send(message) for client in self.clients]

async def ws\_handler(self, ws: WebSocketServerProtocol):

await self.register(ws)

try:

await self.distrubute(ws)

except ConnectionClosedOK:

pass

finally:

await self.unregister(ws)

async def distrubute(self, ws: WebSocketServerProtocol):

async for message in ws:

await self.send\_to\_clients(f"{ws.name}: {message}")

async def main():

server = Server()

async with websockets.serve(server.ws\_handler, 'localhost', 8080):

await asyncio.Future() # run forever

if \_\_name\_\_ == '\_\_main\_\_':

asyncio.run(main())

Тільки тут ми додали імпорт пакету names, який генеруватиме випадкове ім'я користувачеві в чаті. При реєстрації веб-сокет з'єднання, в методі register, ми його додаємо в екземпляр ws: ws.name = names.get\_full\_name(). Потім ми використовуємо його, коли надсилаємо повідомлення всім клієнтам: await self.send\_to\_clients(f"{ws.name}: {message}").

**Код клієнта**

[​](https://textbook.edu.goit.global/python-web-textbook/uk/docs/module-05/module-05-02/chat#%D0%BA%D0%BE%D0%B4-%D0%BA%D0%BB%D1%96%D1%94%D0%BD%D1%82%D0%B0)**index.html**

<!DOCTYPE html>

<html>

<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />

<link rel="shortcut icon" href="favicon.ico" type="image/x-icon" />

<link rel="stylesheet" href="main.css" />

</head>

<body>

<form id="formChat">

<input type="text" id="textField" />

<button type="submit">Send message</button>

</form>

<div id="subscribe"></div>

<script src="main.js"></script>

</body>

</html>

Тут є форма відправлення повідомлення:

<form id="formChat">

<input type="text" id="textField"/>

<button type="submit">Send message</button>

</form>

І блоковий елемент div, в який ми поміщаємо повідомлення від веб-сокету <div id="subscribe"></div>

**main.js**

console.log('Hello world!')

const ws = new WebSocket('ws://localhost:8080')

formChat.addEventListener('submit', (e) => {

e.preventDefault()

ws.send(textField.value)

textField.value = null

})

ws.onopen = (e) => {

console.log('Hello WebSocket!')

}

ws.onmessage = (e) => {

console.log(e.data)

text = e.data

const elMsg = document.createElement('div')

elMsg.textContent = text

subscribe.appendChild(elMsg)

}

Давайте коротко пояснимо цей код і що він робить.

Виконуємо з'єднання до веб-сокету:

const ws = new WebSocket('ws://localhost:8080')

Обробляємо подію submit форми при натисканні кнопки <button type="submit">Send message</button>. Зупиняємо стандартну обробку форми, щоб браузер не надіслав повідомлення самостійно командою e.preventDefault().

Надсилаємо повідомлення самостійно на сервер командою ws.send(textField.value), де textField.value – значення інпута <input type="text" id="textField"/>. Обнулюємо поле введення командою textField.value = null.

formChat.addEventListener('submit', (e) => {

e.preventDefault()

ws.send(textField.value)

textField.value = null

})

Виводимо вітальне повідомлення у консоль браузера при з'єднанні з веб-сокетом.

ws.onopen = (e) => {

console.log('Hello WebSocket!')

}

Цей код спрацьовує, коли сервер надсилає повідомлення клієнту методом send\_to\_clients. Він отримує повідомлення від веб-сокету та додає його в DOM дерево веб-сторінки.

ws.onmessage = (e) => {

console.log(e.data)

text = e.data

const elMsg = document.createElement('div')

elMsg.textContent = text

subscribe.appendChild(elMsg)

}

І трохи стилізуємо сторінку стилями:

**main.css**

input[type=text] {

width: 300px;

border: 1px solid lightseagreen;

padding: 5px 10px;

color: darkred;

font-size: 16px;

font-family: 'Cambria', 'Cochin', 'Georgia', 'Times', 'Times New Roman', serif;

}

button {

border: 1px solid #ccc;

padding: 5px 10px;

font-size: 16px;

}

#subscribe {

color: darkblue;

font-size: 14px;

font-family: 'Courier New', 'Courier', monospace;

margin-top: 20px;

}

Якщо ми запустимо сервер і відкриємо веб-сторінку index.html, то матимемо найпростіший чат.

Запустіть для демонстрації живий приклад. Спочатку запустіть сервер, якщо він ще не запущений.

CAUTION

Якщо сервер не запущений, то клієнт, при ініціалізації фрейму, зможе встановити Websocket з'єднання. Оновіть фрейм клієнта, натиснувши кнопку Rerun справа внизу вікна фрейму, після запуску сервера. Все має працювати.

Ці та нові клієнти можуть спілкуватися між собою у нашому веб-чаті.