HUGE

# Hello

**Creating a chat service using WebSockets**

April 16, 2020

## Who am I?

# William Gómez

Engineer at Huge

(Full stack wannabe web engineer at Huge)

Python Medellin meetups co-organizer

Google Cloud Certified Data Engineer

Freelancer

Machine learning lover

Data science FEM - Mentor

# Agenda.

1. Real-time problem
2. Strategies
3. Why WebSockets?
4. Go Async - AsyncIO
5. Python Chat
6. Quart and WebSockets API
7. Auth for WebSockets

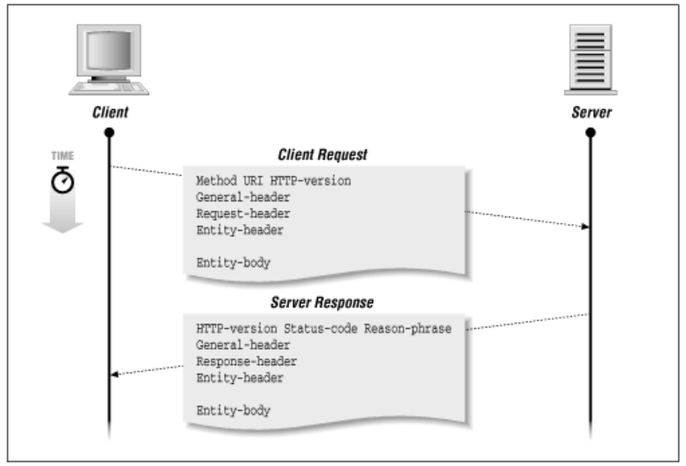
1

# Real-Time Problem

## Real Time Information

When a browser visits a web page:

1. An HTTP request is sent.
2. The server acknowledges this request and sends back the response.




**What if we want to fetch continuously real-time data?** (for example, for stock prices, news reports, ticket sales, traffic patterns, medical device readings)

You could constantly refresh that page manually, but that's obviously not a great solution.

https://www.websocket.org/quantum.html

2

# Strategies

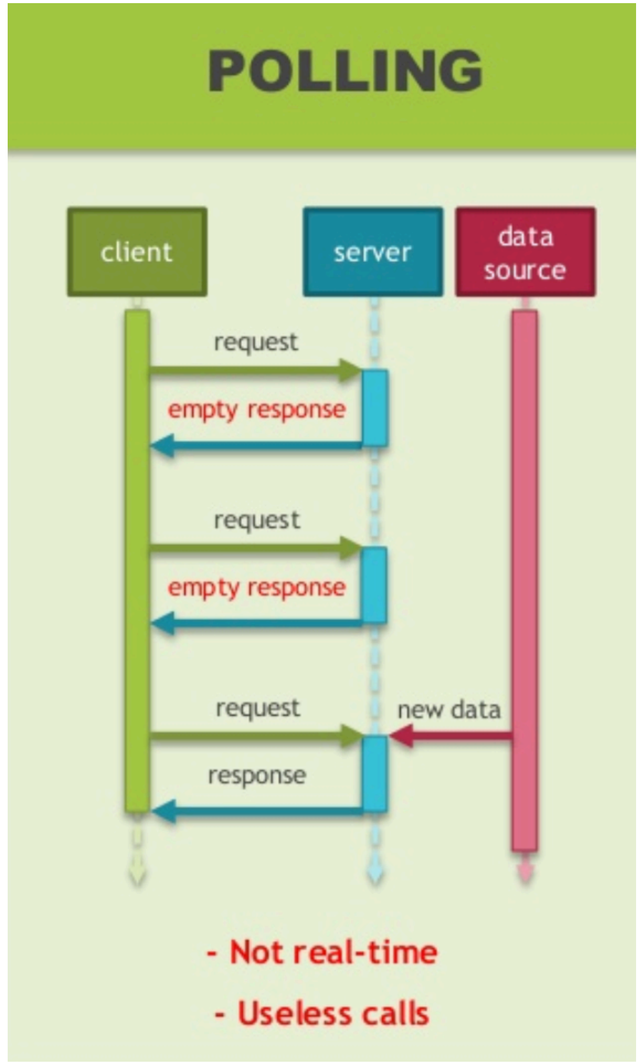
## Polling, Long-Polling, and Streaming

Polling:

1. The browser sends HTTP requests at regular intervals and immediately receives a response.

Good solution if the exact interval of message delivery is known

Real-time data is often not that predictable, making unnecessary requests inevitable.



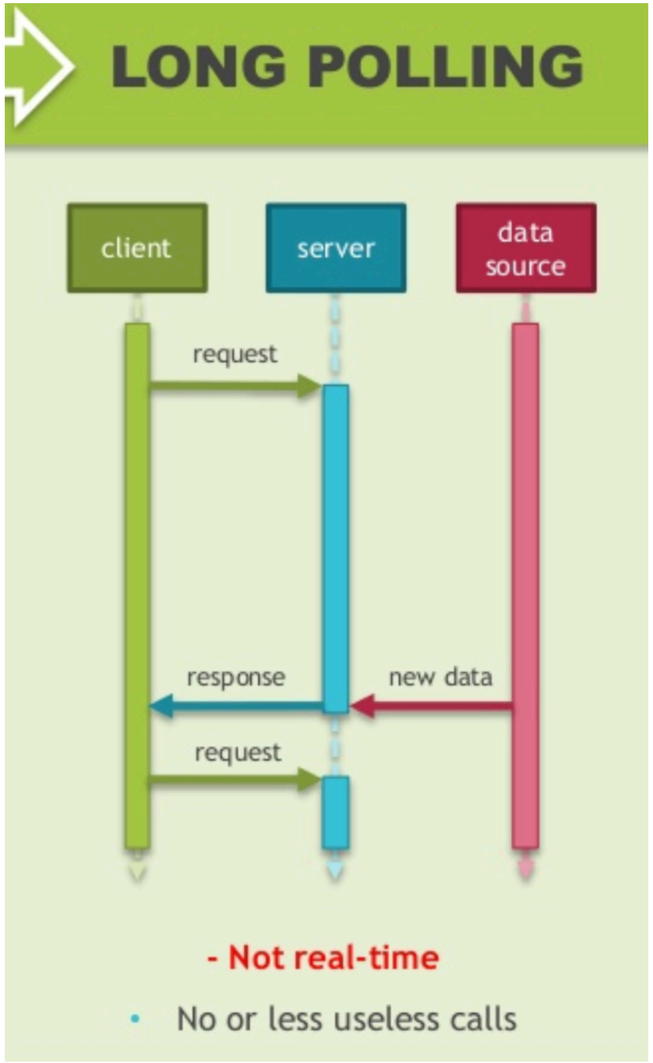

https://www.websocket.org/quantum.html

https://www.slideshare.net/kslisenko/best-practices-of-building-server-to-client-data-streaming

## Polling, Long-Polling, and Streaming

Long-polling:

1. The browser sends a request to the server
2. The server keeps the request open for a set period.
3. If there is new data, a response containing the message is sent to the client.
4. If there is no new data, the server sends a response to terminate the open request.




https://www.websocket.org/quantum.html

https://www.slideshare.net/kslisenko/best-practices-of-building-server-to-client-data-streaming

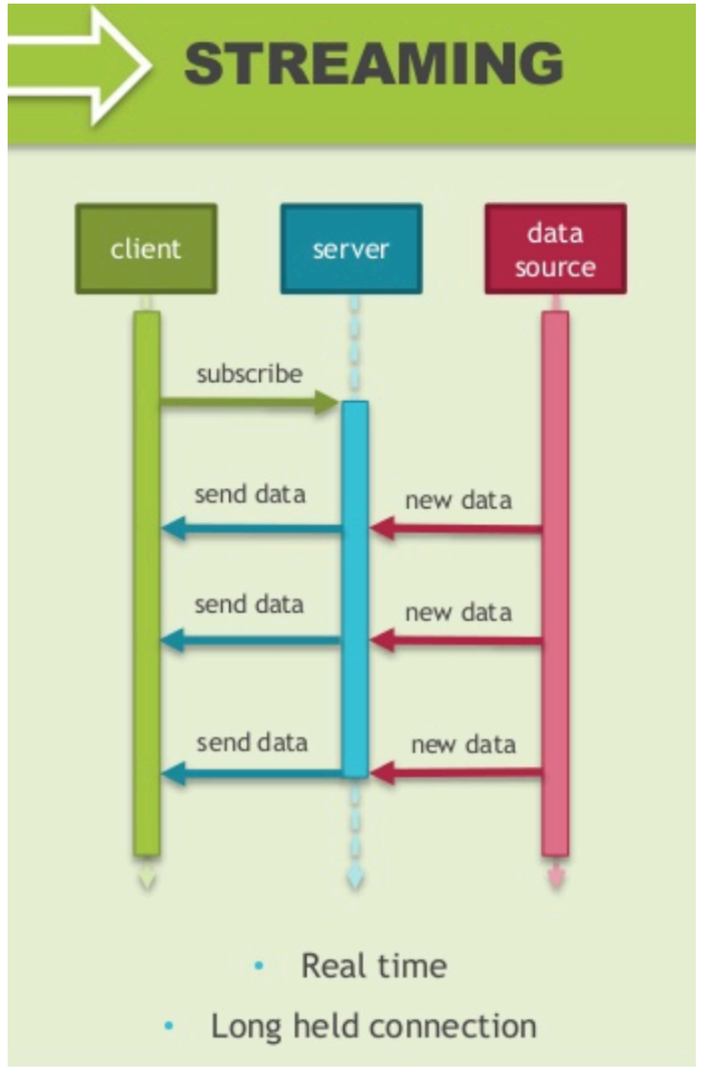
## Polling, Long-Polling, and Streaming

Streaming:

1. The browser sends a complete request
2. The server sends and maintains an open response.
3. The response is then updated whenever a message is ready to be sent, but the server never signals to complete the response.

However, since streaming is still encapsulated in HTTP, intervening firewalls and proxy servers may choose to buffer the response

https://www.websocket.org/quantum.html




https://www.slideshare.net/kslisenko/best-practices-of-building-server-to-client-data-streaming

## HTTP wasn't designed for real-time

All of these methods for providing real-time data involve HTTP that contains unnecessary header data and introduce latency.

To simulate full-duplex communication over half-duplex HTTP, many of today's solutions use two connections: one for the downstream and one for the upstream. However...

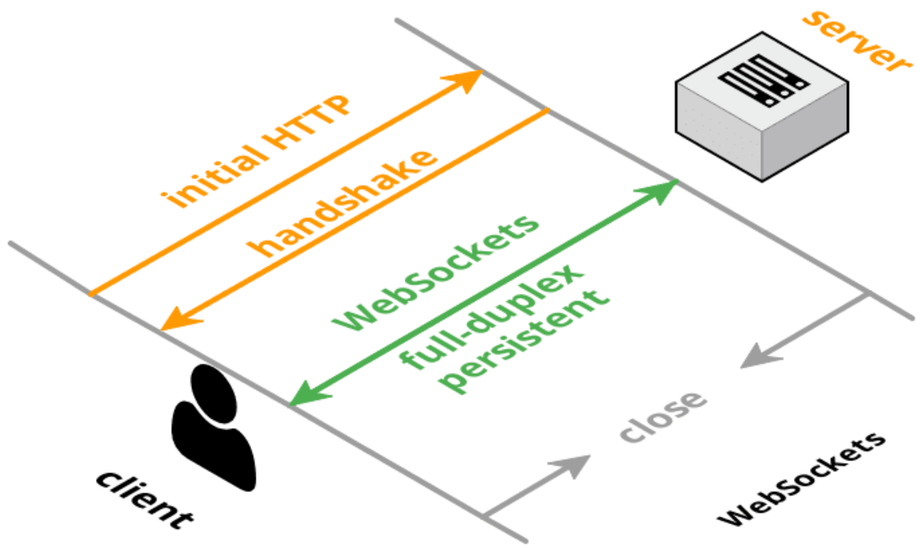
```
GET /PollingStock//PollingStock HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; en-US; rv:1.9.1.5)
Gecko/20091102 Firefox/3.5.5
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-us
Accept-Encoding: gzip,deflate
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer: http://www.example.com/PollingStock/
Cookie: showInheritedConstant=false; showInheritedProtectedConstant=false;
showInheritedProperty=false; showInheritedProtectedProperty=false;
showInheritedMethod=false; showInheritedProtectedMethod=false;
showInheritedEvent=false; showInheritedStyle=false; showInheritedEffect=false
```

https://www.websocket.org/quantum.html

## WebSockets

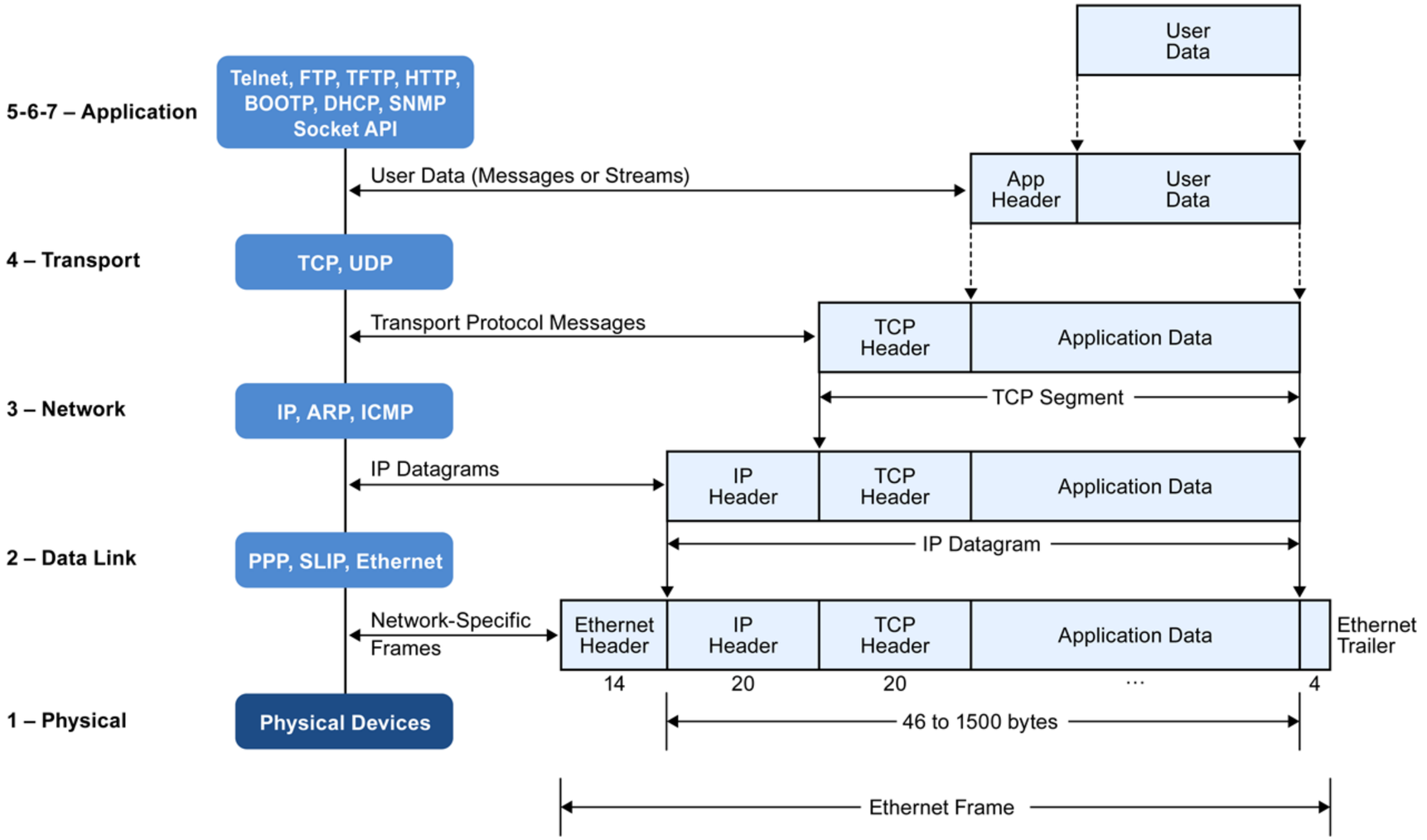
A full-duplex, bidirectional communications channel that operates through a single socket over the Web.

WebSockets provides a true standard that you can use to build scalable, real-time web applications.




https://www.websocket.org/quantum.html

## Recorderis: OSI seven layer model

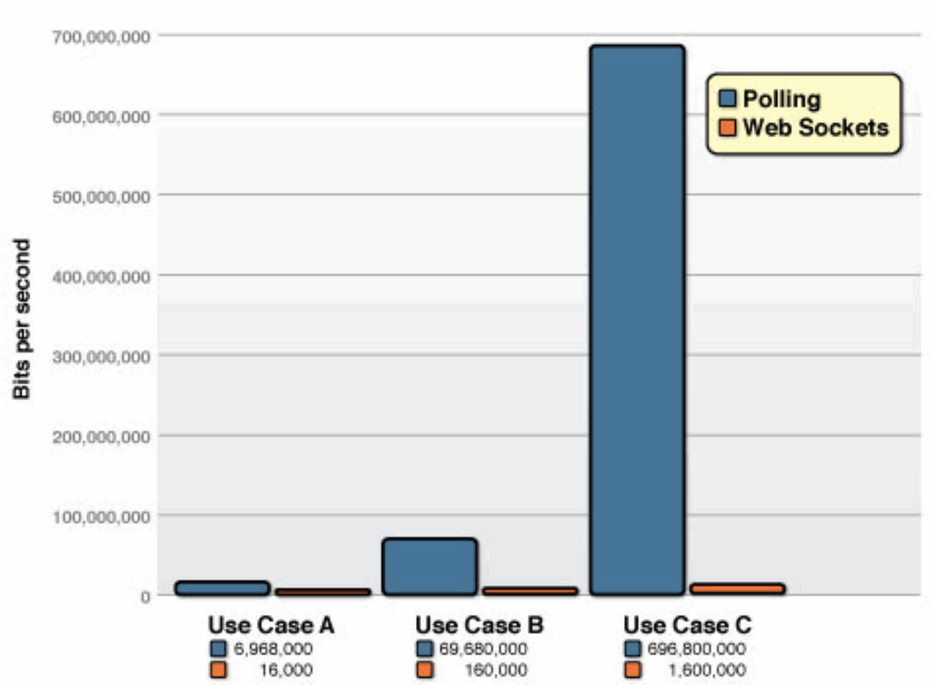


3

# WebSockets?

## WebSockets

The total HTTP request and response header information does not even include any data!
There are cases where header data exceeded 2000 bytes.
In a stock application for example, the data for a typical topic message is only about 20 characters long.



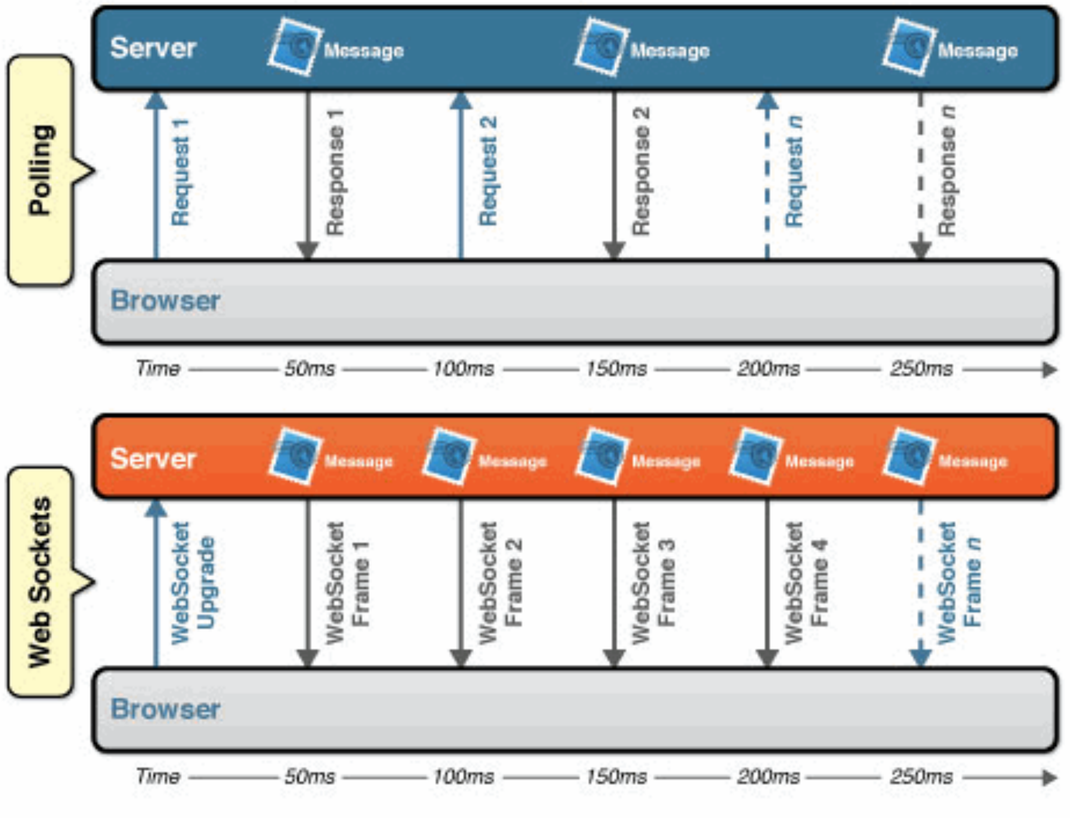

In WebSockets the data is minimally framed with just two bytes.

https://www.websocket.org/quantum.html

## WebSockets

If we assume that for a half-duplex polling solution takes 50 milliseconds for a message to travel from the server to the browser, then the polling application introduces a lot of extra latency, because a new request has to be sent to the server when the response is complete.

Once the connection is upgraded to WebSocket, messages can flow from the server to the browser the moment they arrive.




https://www.websocket.org/quantum.html

## WebSockets rocks

Web Sockets provides an enormous step forward in the scalability of the real-time web.

Web Sockets can provide a 500:1 or—depending on the size of the HTTP headers—even a 1000:1 reduction in unnecessary HTTP header traffic and 3:1 reduction in latency.

https://www.websocket.org/quantum.html

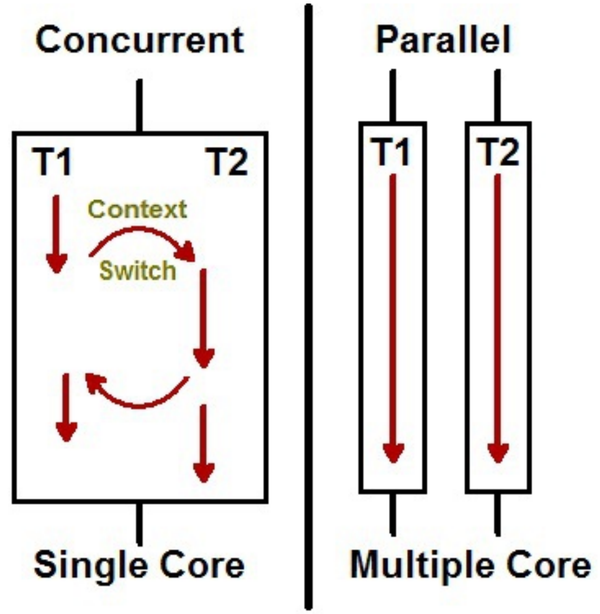
# Go Async - AsyncIO

4

## AsyncIO and Event Loop

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

Asynchronous IO is a language-agnostic paradigm.




AsyncIO provides a set of high-level APIs to:

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

## Couroutines

The heart of async IO are 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.")
```

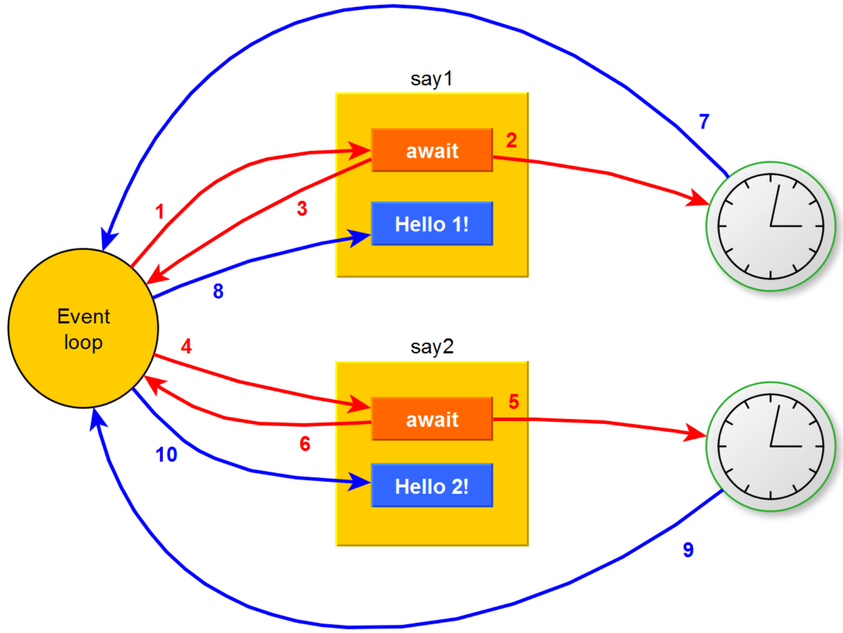
## Event Loop

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.

It is able to wake up an idle coroutine when whatever that coroutine is waiting on becomes available.

https://realpython.com/async-io-python



## Rules of AsyncIO

1. The syntax async def introduces a native coroutine.
2. The keyword await passes function control back to the event loop. (It suspends the execution of the surrounding coroutine.)
3. If Python encounters an await f() expression in the scope of g(), this is how await tells the event loop, “Suspend execution of g() until whatever I’m waiting on—the result of f()—is returned. In the meantime, go let something else run.”
4. To call a coroutine function, you must await it to get its results.

https://realpython.com/async-io-python

## Rules of AsyncIO

5. It is less common (and only recently legal in Python) to use yield in an async def block. This creates an asynchronous generator, which you iterate over with async for.
6. Anything defined with async def may not use yield from, which will raise a SyntaxError.
7. Just like it's a SyntaxError to use yield outside of a def function, it is a SyntaxError to use await outside of an async def coroutine. You can only use await in the body of coroutines.

https://realpython.com/async-io-python

## AsyncIO pretty good videos

Async Python Tutorial: Foundations for those with no prior async experience

https://www.youtube.com/watch?v=6kNzG0T44SI

Python tricks: Demystifying async, await, and asyncio:

https://www.youtube.com/watch?v=tSLDcRkgTsY

## WebSockets Example

Let’s create a WebSocket Server using websockets and asyncio:

```
pip install websockets asyncio
```

websockets package provide a function called serve, which receives a handler, the host and the port for the server creation

```
websockets.serve(ws_handler, ‘localhost’, 4000)
```

The handler must follow the following sintax:

```
async def ws_handler(websocket: websockets.WebSocketServerProtocol, uri: str) -> None:
```

https://medium.com/better-programming/how-to-create-a-websocket-in-python-b68d65dbd549
https://websockets.readthedocs.io/en/stable/api.html

## WebSockets Example

**ws_handler** is called each time a new connection is established, so the first thing we could do is to store the websocket client connection, for receiving and sending future messages.

```
class Server:
 clients = set()

 async def ws_handler(self, websocket: websockets.WebSocketServerProtocol, uri: str) -> None:
  self.register(websocket)

 def register(self, websocket: websockets.WebSocketServerProtocol) -> None:
  self.clients.add(websocket)
  logging.info(f'{websocket.remote_address} connects.')
```

## WebSockets Example

Now, how can we run this server? websockets.serve returns us a promise, so asyncio comes to the rescue

```
if __name__ == '__main__':
 server = Server()
 start_server = websockets.serve(server.ws_handler, 'localhost', 4000)

 loop = asyncio.get_event_loop()
 loop.run_until_complete(start_server)
 loop.run_forever()
```

The first thing is to get an event loop, this is issue by asyncio using get_event_loop(). Then we can init the server using run_until_complete. If you remove the last line it will init the server, but the code will finish, so the last line is (run_forever), it allows to keep the event loop running and waiting for new events.

## WebSockets Example

Let's add the main functionality of our server: distribute messages to all its clients.

```
async def ws_handler(self, websocket: websockets.WebSocketServerProtocol, uri: str) -> None:
    self.register(websocket)
    try:
        await self.distribute(websocket)

async def distribute(self, websocket: websockets.WebSocketServerProtocol) -> None:
    async for message in websocket:
        await self.sent_to_clients(message, websocket)

async def send_to_clients(self, message: str, websocket: websockets.WebSocketServerProtocol) -> None:
    if self.clients:
        await asyncio.gather(*[client.send(message) for client in self.clients])
```

Now the ws_handler adds a new coroutine to the event loop: distribute, and this coroutine adds async for in the event loop. So every time that the websocket connection has a message in it, the event loop will return the control to distribute, distribute will iterate over the messages and send each message.

## WebSockets Example

There is one last thing, when distribute finishes its try catch exception means that the connection is ended. We can put a finally to unregister that connection.

```
async def ws_handler(self, websocket: websockets.WebSocketServerProtocol, uri: str) -> None:
    await self.register(websocket)
    try:
        await self.distribute(websocket)
    finally:
        await self.unregister(websocket)

async def unregister(self, websocket: websockets.WebSocketServerProtocol) -> None:
    self.clients.remove(websocket)
    logging.info(f'{websocket.remote_address} disconnects.')
```

## WebSockets Example

Server is ready. Let's build a client. Basically all the magic is done by websockets.connect, that receives a url for connecting to the server.

```
async def connect(host: str, port: int) -> None:
    websocket_resource_url = f'ws://{host}:{port}'
    async with websockets.connect(websocket_resource_url) as websocket:
        await client_handler(websocket)
```

client_handler is a coroutine that do any stuff with the websocket connection created. To run this client we need again asyncio, get the event loop, and call run_until_complete. If connect finishes, script will finish too.

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(connect(host='localhost', port=4000))
```

## WebSockets Example

Server is ready. Let's build a client. Basically all the magic is done by websockets.connect, that receives a url for connecting to the server.

```
async def connect(host: str, port: int) -> None:
    websocket_resource_url = f'ws://{host}:{port}'
    async with websockets.connect(websocket_resource_url) as websocket:
        await client_handler(websocket)
```

client_handler is a coroutine that do any stuff with the websocket connection created. To run this client we need again asyncio, get the event loop, and call run_until_complete. If connect finishes, script will finish too.

```
if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(connect(host='localhost', port=4000))
```

## WebSockets Example

Let's complete our client:

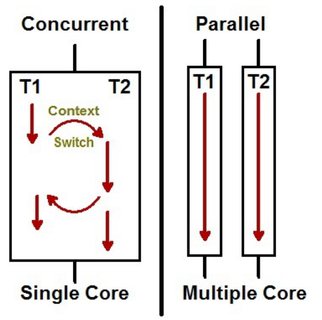
```
async def client_handler(websocket: websockets.WebSocketClientProtocol) -> None:
  while True:
  await asyncio.gather(*[receive_messages(websocket),
                         send_messages(websocket)])

async def send_messages(websocket: websockets.WebSocketClientProtocol):
  stdin, _ = await aioconsole.get_standard_streams()
  async for line in stdin:
    await websocket.send(line.decode())

async def receive_messages(websocket: websockets.WebSocketClientProtocol):
  async for message in websocket:
    log_message(message)

def log_message(message: str) -> None:
  logging.info(f'Message: {message}')
```

Gather allow us to group coroutines, and "parallelize" them. The correct word is run them concurrently.




aioconsole is used to get standard streams from the terminal. It is non-blocking: a promise.

## WebSockets Example

We can also create a simple sender, its messages will be broadcasted by the server.

```
async def send(message: str, host: str, port: int) -> None:
    websocket_resource_url = f'ws://{host}:{port}'
    async with websockets.connect(websocket_resource_url) as websocket:
        await websocket.send(message)

if __name__ == '__main__':
    loop = asyncio.get_event_loop()
    loop.run_until_complete(send(message='hi', host='localhost', port=4000))
```

It behaves actually as another client that connects to the websocket server and then sends a message.

# Quart and WebSockets

## Quart

Quart is a Python web micro-framework based on AsyncIO.

It is intended to provide the easiest way to use asyncio in a web context, especially with existing Flask apps.

https://pgjones.gitlab.io/quart/

## Quart

Quart is an evolution of the Flask API to work with Asyncio and to provide a number of features not present or possible in Flask, see Flask evolution.

For those of you familiar with Flask, Quart extends the Flask-API by adding support for:

- HTTP/1.1 request streaming.
- Websockets.
- HTTP/2 server push.

https://gitlab.com/pgjones/hypercorn

https://serverfault.com/questions/220046/why-is-setting-nginx-as-a-reverse-proxy-a-good-idea

## Quart Example

Creating a Quart app is as easy as creating a Flask one.

```
from quart import Quart, websocket, render_template

app = Quart(__name__)

@app.route('/')
async def index():
    return await render_template('index.html')

app.run()
```

https://pgjones.gitlab.io/quart/tutorials/websocket_tutorial.html

https://medium.com/@pgjones/websockets-in-quart-f2067788d1ee

## Quart Example

To add a websocket route set the websocket decorator with the path of the route.

```
@app.websocket('/ws')
async def ws():
    while True:
        data = await websocket.receive()
        await websocket.send(f"echo {data}")
```

This route establish the connection. Then a while True will keep running the following awaitable instructions.

https://pgjones.gitlab.io/quart/tutorials/websocket_tutorial.html

https://medium.com/@pgjones/websockets-in-quart-f2067788d1ee

## Quart Example

In the client, we will have a browser rendering an input for the message. Also, we will connect to the server using the javascript's WebSocket API.

```
<body>
  <div class="message-container">
    <input type="text" class="message-body" />
    <button type="button" class="send-button">Enviar</button>
  </div>
  <div class="echo-container"></div>
</body>
```

```
<script>
const ws = new WebSocket('ws://' + document.domain + ':' + location.port + '/ws');
…
```

https://pgjones.gitlab.io/quart/tutorials/websocket_tutorial.html

https://medium.com/@pgjones/websockets-in-quart-f2067788d1ee

## Quart Example

```
  <script>
    const ws = new WebSocket('ws://' + document.domain + ':' +
location.port + '/ws');
    const messageInput = document.querySelector('.message-body');
    const sendButton = document.querySelector('.send-button');
    const echoContainer = document.querySelector('.echo-container');

    sendButton.addEventListener('click', () => {
      ws.send(messageInput.value);
    })

    ws.onmessage = (event) => {
      const messageElement = document.createElement('P');
      console.log(messageElement)
      messageElement.innerHTML = event.data
      echoContainer.appendChild(messageElement)
    }
  </script>
```

After WebSocket connection, it gets DOM elements for the message body, send button and the container where echo messages will be displayed.

A listener will be added for send button for on click signal. Then, it will use Websocket connection to send the message.

When a new message is received from connection a new DOM child with the message will be append to the echo container body.

6

# Auth for WebSockets

## How to Auth - Example

Look for “2020-04-16 - Python Medellin - Creating a chat service with WebSockets/Chat” in the repo of this talk.

This example includes the following flow:

1. An app with an Auth and WebSocket blueprints is launched.
2. Websocket endpoint has now a decorator auth_required which validates a token_id and a user from the query string of the path /ws?token_id=&email=
3. Auth blueprints has two endpoints: register, which generates an User record in a Postgres database and login, which verifies the user and generates a token using PyJWT.
4. Database models and core is managed by SQLAlchemy.

https://medium.com/@pgjones/websockets-in-quart-f2067788d1ee

https://www.freecodecamp.org/news/how-to-secure-your-websocket-connections-d0be0996c556/

# Medium: williamegomezo

# williamegomezo.me

HUGE

# Done.

**Creating a chat service using WebSockets**

April 16, 2020