Web Sockets

websockets is a library for building WebSocket servers and clients in Python with a focus on correctness and simplicity.

Built on top of asyncio, Python’s standard asynchronous I/O framework, it provides an elegant coroutine-based API.

Here’s how a client sends and receives messages:

```
#!/usr/bin/env python
import asyncio
import websockets
async def hello():
    uri = "ws://localhost:8765"
   async with websockets.connect(uri) as websocket:
        await websocket.send("Hello world!")
       await websocket.recv()
asyncio.get_event_loop().run_until_complete(hello())
And here’s an echo server:
#!/usr/bin/env python
import asyncio
import websockets
async def echo(websocket, path):
    async for message in websocket:
        await websocket.send(message)
start_server = websockets.serve(echo, "localhost", 8765)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Requirements

websockets requires Python ≥ 3.6.1.

You should use the latest version of Python if possible. If you’re using an older version, be aware that for each minor version (3.x), only the latest bugfix release (3.x.y) is officially supported.

Installation

Install websockets with:

```
pip install websockets
```

Basic example

Here’s a WebSocket server example.

It reads a name from the client, sends a greeting, and closes the connection.

```
#!/usr/bin/env python# WS server exampleimport asyncio
import websockets
async def hello(websocket, path):
    name = await websocket.recv()
   print(f"< {name}")    greeting = f"Hello {name}!"    await websocket.send(greeting)
print(f"> {greeting}")


start_server = websockets.serve(
          hello, "localhost", 8765
)asyncio.get_event_loop().run_until_complete(start_server)asyncio.get_event_loop().run_forever()
```

On the server side, websockets executes the handler coroutine hello once for each WebSocket connection. It closes the connection when the handler coroutine returns.

Here’s a corresponding WebSocket client example.

```
#!/usr/bin/env python# WS client example

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}")
asyncio.get_event_loop().run_until_complete(hello())
```

Using connect() as an asynchronous context manager ensures the connection is closed before exiting the hello coroutine.

Secure example

Secure WebSocket connections improve confidentiality and also reliability because they reduce the risk of interference by bad proxies.

The WSS protocol is to WS what HTTPS is to HTTP: the connection is encrypted with Transport Layer Security (TLS) — which is often referred to as Secure Sockets Layer (SSL). WSS requires TLS certificates like HTTPS.

Here’s how to adapt the server example to provide secure connections. See the documentation of the ssl module for configuring the context securely.

```
#!/usr/bin/env python

# WSS (WS over TLS) server example, with a self-signed certificate

import asyncioimport pathlibimport sslimport websockets
async def hello(websocket, path):    name = await websocket.recv()      print(f"<
{name}")    greeting = f"Hello {name}!"  await websocket.send(greeting)    print(f">
{greeting}")
ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_SERVER)localhost_pem =
pathlib.Path(__file__).with_name("localhost.pem")ssl_context.load_cert_chain(localhost_
pem)
start_server = websockets.serve(    hello, "localhost", 8765, ssl=ssl_context)
asyncio.get_event_loop().run_until_complete(start_server)asyncio.get_event_loop().run_f
orever()
```

Here’s how to adapt the client.

```
#!/usr/bin/env python# WSS (WS over TLS) client example, with a self-signed

certificateimport asyncioimport pathlibimport sslimport websockets

ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)localhost_pem =

pathlib.Path(__file__).with_name("localhost.pem")ssl_context.load_verify_locations(local

host_pem)

async def hello():    uri = "wss://localhost:8765"    async with websockets.connect(

uri, ssl=ssl_context ) as websocket:        name = input("What's your name? ")  await

websocket.send(name)        print(f"> {name}")  greeting = await websocket.recv()

print(f"< {greeting}")


asyncio.get_event_loop().run_until_complete(hello())
```

This client needs a context because the server uses a self-signed certificate.

A client connecting to a secure WebSocket server with a valid certificate (i.e. signed by a CA that your Python installation trusts) can simply pass ssl=True to connect() instead of building a context.

Browser-based example

Here's an example of how to run a WebSocket server and connect from a browser.

Run this script in a console:

```
#!/usr/bin/env python# WS server that sends messages at random intervalsimport
asyncio
import datetime
import random
import websockets
async def time(websocket, path):
    while True:
        now = datetime.datetime.utcnow().isoformat() + "Z"
     await websocket.send(now)
    await asyncio.sleep(random.random() * 3)
start_server = websockets.serve(time, "127.0.0.1", 5678)
asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```

Then open this HTML file in a browser.

```
<!DOCTYPE html>
<html>
   <head>
```

```
            <title>WebSocket demo</title>
    </head>
    <body>
            <script>               var ws = new WebSocket("ws://127.0.0.1:5678/"),
                                    messages = document.createElement('ul');
         ws.onmessage = function (event) {
           var messages = document.getElementsByTagName('ul')[0],
                                message = document.createElement('li'),
                              content = document.createTextNode(event.data);
                message.appendChild(content);
                messages.appendChild(message);
              };document.body.appendChild(messages);
         </script>
    </body>
</html>
```

Synchronization example

A WebSocket server can receive events from clients, process them to update the application state, and synchronize the resulting state across clients.

Here’s an example where any client can increment or decrement a counter. Updates are propagated to all connected clients.

The concurrency model of asyncio guarantees that updates are serialized.

Run this script in a console:

```
#!/usr/bin/env python# WS server example that synchronizes state across
clientsimport asyncio
```

```
import jsonimport loggingimport websockets
logging.basicConfig()STATE = {"value": 0}USERS = set()def state_event():    return
json.dumps({"type": "state", **STATE})
def users_event():    return json.dumps({"type": "users", "count": len(USERS)})
async def notify_state():
    if USERS:
 # asyncio.wait doesn't accept an empty list
     message = state_event()        await asyncio.wait([user.send(message) for user in
USERS])
async def notify_users():    if USERS:
  # asyncio.wait doesn't accept an empty list   message = users_event()        await
asyncio.wait([user.send(message) for user in USERS])async def register(websocket):
USERS.add(websocket)    await notify_users()async def unregister(websocket):
USERS.remove(websocket)    await notify_users()
async def counter(websocket, path):    # register(websocket) sends user_event() to
websocket    await register(websocket)    try:            await
websocket.send(state_event())            async for message in websocket:
data = json.loads(message)                        if data["action"] == "minus":
                            STATE["value"] -= 1
                               await notify_state()         elif data["action"] == "plus":
            STATE["value"] += 1                          await notify_state()        else:
                        logging.error("unsupported event: %s", data)
            finally:         await unregister(websocket)start_server = websockets.serve(counter,
            "localhost", 6789)asyncio.get_event_loop().run_until_complete(start_server)
asyncio.get_event_loop().run_forever()
```
Then open this HTML file in several browsers.

```
<!DOCTYPE html>

<html>

        <head>

              <title>WebSocket demo</title>

             <style type="text/css">


                body {

                            font-family: "Courier New", sans-serif;

                            text-align: center;

         }

                .buttons {

                    font-size: 4em;

                    display: flex;

                    justify-content: center;

         }

         .button, .value {

                            line-height: 1;

                            padding: 2rem;

                            margin: 2rem;

                            border: medium solid;

                          min-height: 1em;

                            min-width: 1em;

                }

        .button {

                cursor: pointer;
```

```
                                user-select: none;
                    }
                    .minus {
                         color: red;
                    }
                    .plus {
                         color: green;
                  }
                    .value {
                         min-width: 2em;
                    }
                    .state {
                         font-size: 2em;
           }
               </style>
</head>
<body>
      <div class="buttons">
         <div class="minus button">-</div>
                    <div class="value">?</div>
                    <div class="plus button">+</div>
</div>
      <div class="state">
                    <span class="users">?</span> online
   </div>
```

```
        <script>
            var minus = document.querySelector('.minus'),
                plus = document.querySelector('.plus'),
                value = document.querySelector('.value'),
  users = document.querySelector('.users'),
  websocket = new WebSocket("ws://127.0.0.1:6789/");
minus.onclick = function (event) {
           websocket.send(JSON.stringify({action: 'minus'}));
}
          plus.onclick = function (event) {                websocket.send(JSON.stringify({action:
'plus'}));
         }
          websocket.onmessage = function (event) {
                data = JSON.parse(event.data);
                switch (data.type) {
                               case 'state':
                       value.textContent = data.value;
                                           break;
                                case 'users':
                       users.textContent = (
                                                         data.count.toString() + " user" + (data.count == 1 ? ""
: "s"));
                       break;
            default:                     console.error(                          "unsupported event", data
);               }              };
```

```
  </script>
 </body>
</html>
```

Common patterns

You will usually want to process several messages during the lifetime of a connection. Therefore you must write a loop. Here are the basic patterns for building a WebSocket server.

Consumer

For receiving messages and passing them to a consumer coroutine:

```
async def consumer_handler(websocket, path):
        async for message in websocket:
     await consumer(message)
```

In this example, consumer represents your business logic for processing messages received on the WebSocket connection.

Iteration terminates when the client disconnects.

Producer

For getting messages from a producer coroutine and sending them:

```
async def producer_handler(websocket, path):
   while True:
          message = await producer()
     await websocket.send(message)
```

In this example, producer represents your business logic for generating messages to send on the WebSocket connection.

send() raises a ConnectionClosed exception when the client disconnects, which breaks out of the while True loop.

Both

You can read and write messages on the same connection by combining the two patterns shown above and running the two tasks in parallel:

```
async def handler(websocket, path):
   consumer_task = asyncio.ensure_future(
              consumer_handler(websocket, path))
   producer_task = asyncio.ensure_future(
         producer_handler(websocket, path))
  done, pending = await asyncio.wait(
         [consumer_task, producer_task],
             return_when=asyncio.FIRST_COMPLETED,
  )
   for task in pending:
task.cancel()
```

Registration

As shown in the synchronization example above, if you need to maintain a list of currently connected clients, you must register them when they connect and unregister them when they disconnect.

```
connected = set()async def handler(websocket, path):
  # Register.
   connected.add(websocket)
   try:
               # Broadcast a message to all connected clients.
              await asyncio.wait([ws.send("Hello!") for ws in connected])
               await asyncio.sleep(10)
   finally:
               # Unregister.
      connected.remove(websocket)
```
This simplistic example keeps track of connected clients in memory. This only works as long as you run a single process. In a practical application, the handler may subscribe to some channels on a message broker, for example.