Examples

Creating Your First WebSocket Connection

If you want to connect to a websocket without writing any code yourself, you can try out the Getting Started wsdump.py script and the examples/ directory files.

You can create your first custom connection with this library using one of the simple examples below. Note that the first WebSocket example is best for a short-lived connection, while the WebSocketApp example is best for a long-lived connection.

WebSocket example

```
>>> import websocket
>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server")
19
>>> print(ws.recv())
echo.websocket.events sponsored by Lob.com
>>> ws.close()
```

WebSocketApp example

```
>>> import websocket
>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade",
on_message=on_message)
>>> wsapp.run_forever()
```

Debug and Logging Options

When you're first writing your code, you will want to make sure everything is working as you planned. The easiest way to view the verbose connection information is the use `websocket.enableTrace(True)`. For example, the following example shows how you can verify that the proper Origin header is set.

```
import websocket

websocket.enableTrace(True)
ws = websocket.WebSocket()
ws.connect("ws://echo.websocket.events/", origin="testing_websockets.com")
ws.send("Hello, Server")
print(ws.recv())
ws.close()
```

The output you will see will look something like this:

```
--- request header ---
GET / HTTP/1.1
Upgrade: websocket
Host: echo.websocket.events
Origin: testing_websockets.com
Sec-WebSocket-Key: GnuCGEiF3OuyRESXiVnsAQ==
Sec-WebSocket-Version: 13
Connection: Upgrade


-----------------------
--- response header ---
HTTP/1.1 101 Switching Protocols
Connection: Upgrade
Upgrade: websocket
Sec-Websocket-Accept: wvhwrjThsVAyr/V4Hzn5tWMSomI=
Via: 1.1 vegur
-----------------------
++Sent raw: b'\x81\x8d\xd4\xda9\xee\x9c\xbfU\x82\xbb\xf6\x19\xbd\xb1\xa8O\x8b\xa6'
++Sent decoded: fin=1 opcode=1 data=b'Hello, Server'
19
++Rcv raw: b'\x81*echo.websocket.events sponsored by Lob.com'
++Rcv decoded: fin=1 opcode=1 data=b'echo.websocket.events sponsored by Lob.com'
echo.websocket.events sponsored by Lob.com
++Sent raw: b'\x88\x82\xc9\x8c\x14\x99\xcad'
++Sent decoded: fin=1 opcode=8 data=b'\x03\xe8'
```

Using websocket-client with “with” statements

It is possible to use “with” statements, as outlined in PEP 343, to help manage the closing of WebSocket connections after a message is received. Below is one example of this being done with a short-lived connection:

Short-lived WebSocket using “with” statement

```
>>> from contextlib import closing
>>> from websocket import create_connection
>>> with closing(create_connection("wss://testnet.binance.vision/ws/btcusdt@trade")) as
conn:
...     print(conn.recv())

# Connection is now closed
```

Connection Options

After you can establish a basic WebSocket connection, customizing your connection using specific options is the next step. Fortunately, this library provides many options you can configure, such as:

- “Host” header value
- “Cookie” header value
- “Origin” header value
- WebSocket subprotocols
- Custom headers
- SSL or hostname verification
- Timeout value

For a more detailed list of the options available for the different connection methods, check out the source code comments for each:

- **WebSocket().connect() option docs**
    - Related: WebSocket.create_connection() option docs
- **WebSocketApp() option docs**
    - Related: WebSocketApp.run_forever docs

Setting Common Header Values

To modify the `Host`, `Origin`, `Cookie`, or `Sec-WebSocket-Protocol` header values of the WebSocket handshake request, pass the `host`, `origin`, `cookie`, or `subprotocols` options to your WebSocket connection. The first two examples show the Host, Origin, and Cookies headers being set, while the Sec-WebSocket-Protocol header is set separately in the following example. For debugging, remember that it is helpful to enable Debug and Logging Options.

WebSocket common headers example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events", cookie="chocolate",
... origin="testing_websockets-client.com", host="echo.websocket.events")
```

WebSocketApp common headers example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade",
... cookie="chocolate", on_message=on_message)
>>> wsapp.run_forever(origin="testing_websockets.com", host="127.0.0.1")
```

WebSocket subprotocols example

Use this to specify STOMP, WAMP, MQTT, or other values of the “Sec-WebSocket-Protocol” header. Be aware that websocket-client does not include support for these protocols, so your code must handle the data sent over the WebSocket connection.

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("wss://ws.kraken.com", subprotocols=["mqtt"])
```

WebSocketApp subprotocols example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://ws.kraken.com",
... subprotocols=["STOMP"], on_message=on_message)
>>> wsapp.run_forever()
```

Suppress Origin Header

There is a special `suppress_origin` option that can be used to remove the `Origi` connection handshake requests. The below examples illustrate how this can be u… debugging, remember that it is helpful to enable Debug and Logging Options.

WebSocket suppress origin example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events", suppress_origin=True)
```

WebSocketApp suppress origin example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade",
... on_message=on_message)
>>> wsapp.run_forever(suppress_origin=True)
```

Setting Custom Header Values

Setting custom header values, other than `Host`, `Origin`, `Cookie`, or `Sec-WebSocket-Protocol` (which are addressed above), in the WebSocket handshake request is similar to setting common header values. Use the `header` option to provide custom header values in a list or dict. For debugging, remember that it is helpful to enable Debug and Logging Options. There is no built-in support for “Sec-WebSocket-Extensions” header values as defined in RFC 7692.

WebSocket custom headers example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events",
... header={"CustomHeader1":"123", "NewHeader2":"Test"})
```

WebSocketApp custom headers example

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade"
... header={"CustomHeader1":"123", "NewHeader2":"Test"}, on_message=on_messa
>>> wsapp.run_forever()
```

Disabling SSL or Hostname Verification

See the relevant FAQ page for instructions.

Using a Custom Class

You can also write your own class for the connection, if you want to handle the nitty-gritty connection details yourself.

```
>>> import socket
>>> from websocket import create_connection, WebSocket

>>> class MyWebSocket(WebSocket):
...     def recv_frame(self):
...         frame = super().recv_frame()
...         print('yay! I got this frame: ', frame)
...         return frame
>>> ws = create_connection("ws://echo.websocket.events/",
... sockopt=((socket.IPPROTO_TCP, socket.TCP_NODELAY, 1),), class_=MyWebSocket)
```

Setting Timeout Value

The _socket.py file contains the functions `setdefaulttimeout()` and `getdefaulttimeout()`. These two functions set the global `_default_timeout` value, which sets the socket timeout value (in seconds). These two functions should not be confused with the similarly named `settimeout()` and `gettimeout()` functions found in the _core.py file. With WebSocketApp, the `run_forever()` function gets assigned the timeout from getdefaulttimeout(). When the timeout value is reached, the exception WebSocketTimeoutException is triggered by the _socket.py `send()` and `recv()` functions. Additional timeout values can be found in other locations in this library, including the `close()` function of the WebSocket class and the `create_connection()` function of the WebSocket class.

The WebSocket timeout example below shows how an exception is triggered after no response is received from the server after 5 seconds.

WebSocket timeout example

```
>>> import websocket

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events", timeout=5)
>>> # ws.send("Hello, Server") # Commented out to trigger WebSocketTimeoutException
>>> print(ws.recv())
# Program should end with a WebSocketTimeoutException
```

The WebSocketApp timeout example works a bit differently than the WebSocket example. Because WebSocketApp handles long-lived connections, it does not timeout after a certain amount of time without receiving a message. Instead, a timeout is triggered if no connection response is received from the server after the timeout interval (5 seconds in the example below).

WebSocketApp timeout example

```
>>> import websocket

>>> def on_error(wsapp, err):
...     print("EXAMPLE error encountered: ", err)
>>> websocket.setdefaulttimeout(5)
>>> wsapp = websocket.WebSocketApp("ws://nexus-websocket-a.intercom.io",
... on_error=on_error)
>>> wsapp.run_forever()
# Program should print a "timed out" error message
```

Connecting through a proxy

websocket-client supports proxied connections. The supported proxy protocols are HTTP, SOCKS4, SOCKS4a, SOCKS5, and SOCKS5h. If you want to route DNS requests through the proxy, use SOCKS4a or SOCKS5h. The proxy protocol should be specified in lowercase to the `proxy_type` parameter. The example below shows how to connect through a HTTP or SOCKS proxy. Proxy authentication is supported with the `http_proxy_auth` parameter, which should be a tuple of the username and password. Be aware that the current implementation of websocket-client uses the “CONNECT” method for HTTP proxies (though soon the HTTP proxy handling will use the same `python_socks` library currently enabled only for SOCKS proxies), and the HTTP proxy server must allow the “CONNECT” method. For example, the squid HTTP proxy only allows the “CONNECT” method on HTTPS ports by default. You may encounter problems if using SSL/TLS with your proxy.

WebSocket HTTP proxy with authentication example

```
import websocket

ws = websocket.WebSocket()
ws.connect("ws://echo.websocket.events",
  http_proxy_host="127.0.0.1", http_proxy_port="8080",
  proxy_type="http", http_proxy_auth=("username", "password123"))
ws.send("Hello, Server")
print(ws.recv())
ws.close()
```

WebSocket SOCKS4 (or SOCKS5) proxy example

```
import websocket

ws = websocket.WebSocket()
ws.connect("ws://echo.websocket.events",
  http_proxy_host="192.168.1.18", http_proxy_port="4444", proxy_type="socks4")
ws.send("Hello, Server")
print(ws.recv())
ws.close()
```

WebSocketApp proxy example

```
import websocket

ws = websocket.WebSocketApp("ws://echo.websocket.events")
wsapp.run_forever(proxy_type="socks5", http_proxy_host=proxy_ip, http_proxy_auth=
(proxy_username, proxy_password))
```

Connecting with Custom Sockets

You can also connect to a WebSocket server hosted on a specific socket using the `socket` option when creating your connection. Below is an example of using a unix domain socket.

```
import socket
from websocket import create_connection
my_socket = socket.socket(socket.AF_UNIX, socket.SOCK_STREAM)
my_socket.connect("/path/to/my/unix.socket")

ws = create_connection("ws://localhost/", # Dummy URL
                       socket = my_socket,
                       sockopt=((socket.SOL_SOCKET, socket.SO_KEEPALIVE, 1),))
```


Other socket types can also be used. The following example is for a AF_INET (IP address) socket.

```
import socket
from websocket import create_connection
my_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
my_socket.bind(("172.18.0.1", 3002))
my_socket.connect()

ws = create_connection("ws://127.0.0.1/", # Dummy URL
                       socket = my_socket)
```

Post-connection Feature Summary

Autobahn|TestSuite is an independent automated test suite to verify the compliance of WebSocket implementations.

Running the test suite against this library will produce a summary report of the conformant features that have been implemented.

A recently-run autobahn report (available as an .html file) is available in the /compliance directory.

Ping/Pong Usage

The WebSocket specification defines ping and pong message opcodes as part of the protocol. These can serve as a way to keep a connection active even if data is not being transmitted.

Pings may be sent in either direction. If the client receives a ping, a pong reply will be automatically sent.

However, if a blocking event is happening, there may be some issues with ping/pong. Below are examples of how ping and pong can be sent by this library.

You can get additional debugging information by using Wireshark to view the ping and pong messages being sent. In order for Wireshark to identify the WebSocket protocol properly, it should observe the initial HTTP handshake and the HTTP 101 response in cleartext (without encryption) - otherwise the WebSocket messages may be categorized as TCP or TLS messages. For debugging, remember that it is helpful to enable Debug and Logging Options.

WebSocket ping/pong example

This example is best for a quick test where you want to check the effect of a ping, or where situations where you want to customize when the ping is sent.

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.ping()
>>> ws.ping("This is an optional ping payload")
>>> ws.close()
```

WebSocketApp ping/pong example

This example, and `run_forever()` in general, is better for long-lived connections.

In this example, if a ping is received and a pong is sent in response, then the client is notified via `on_ping()`.

Further, a ping is transmitted every 60 seconds. If a pong is received, then the client is notified via `on_pong()`. If no pong is received within 10 seconds, then `run_forever()` will exit with a `WebSocketTimeoutException`.

```
>>> import websocket

>>> def on_message(wsapp, message):
...     print(message)
>>> def on_ping(wsapp, message):
...     print("Got a ping! A pong reply has already been automatically sent.")
>>> def on_pong(wsapp, message):
...     print("Got a pong! No need to respond")
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade",
... on_message=on_message, on_ping=on_ping, on_pong=on_pong)
>>> wsapp.run_forever(ping_interval=60, ping_timeout=10, ping_payload="This is an optional
ping payload")
```

Sending Connection Close Status Codes

RFC6455 defines various status codes that can be used to identify the reason for a close frame ending a connection. These codes are defined in the websocket/_abnf.py file. To view the code used to close a connection, you can enable logging to view the status code information. You can also specify your own status code in the .close() function, as seen in the examples below. Specifying a custom status code is necessary when using the custom status code values between 3000-4999.

WebSocket sending close() status code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server")
19
>>> print(ws.recv())
echo.websocket.events sponsored by Lob.com
>>> ws.close(websocket.STATUS_PROTOCOL_ERROR)
# Alternatively, use ws.close(status=1002)
```

WebSocketApp sending close() status code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def on_message(wsapp, message):
...     print(message)
...     wsapp.close(status=websocket.STATUS_PROTOCOL_ERROR) # Alternatively, use
wsapp.close(status=1002)
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade",
on_message=on_message)
>>> wsapp.run_forever(skip_utf8_validation=True)
```

Receiving Connection Close Status Codes

The RFC6455 spec states that it is optional for a server to send a close status code when closing a connection. The RFC refers to these codes as WebSocket Close Code Numbers, and their meanings are described in the RFC. It is possible to view this close code, if it is being sent, to understand why the connection is being close. One option to view the code is to enable logging to view the status code information. If you want to use the close status code in your program, examples are shown below for how to do this.

WebSocket receiving close status code example

```
>>> import websocket
>>> import struct

>>> websocket.enableTrace(True)
>>> ws = websocket.WebSocket()
>>> ws.connect("wss://tsock.us1.twilio.com/v3/wsconnect")
>>> ws.send("Hello")
11
>>> resp_opcode, msg = ws.recv_data()
>>> print("Response opcode: " + str(resp_opcode))
>>> if resp_opcode == 8 and len(msg) >= 2:
...     print("Response close code: " + str(struct.unpack("!H", msg[0:2])[0]))
...     print("Response message: " + str(msg[2:]))
... else:
...     print("Response message: " + str(msg))
```

WebSocketApp receiving close status code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def on_close(wsapp, close_status_code, close_msg):
...     # Because on_close was triggered, we know the opcode = 8
...     print("on_close args:")
...     if close_status_code or close_msg:
...         print("close status code: " + str(close_status_code))
...         print("close message: " + str(close_msg))
>>> def on_open(wsapp):
...     wsapp.send("Hello")
>>> wsapp = websocket.WebSocketApp("wss://tsock.us1.twilio.com/v3/wsconnect",
on_close=on_close, on_open=on_open)
>>> wsapp.run_forever()
```

Customizing frame mask

WebSocket frames use masking with a random value to add entropy. The masking value in websocket-client is normally set using os.urandom in the websocket/_abnf.py file. However, this value can be customized as you wish. One use case, outlined in issue #473, is to set the masking key to a null value to make it easier to decode the messages being sent and received. This is effectively the same as “removing” the mask, though the mask cannot be fully “removed” because it is a part of the WebSocket frame. Tools such as Wireshark can automatically remove masking from payloads to decode the payload message, but it may be easier to skip the demasking step in your custom project.

WebSocket custom masking key code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def zero_mask_key(_):
...     return "\x00\x00\x00\x00"
>>> ws = websocket.WebSocket()
>>> ws.set_mask_key(zero_mask_key)
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server")
>>> print(ws.recv())
>>> ws.close()
```

WebSocketApp custom masking key code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def zero_mask_key(_):
...     return "\x00\x00\x00\x00"
>>> def on_message(wsapp, message):
...     print(message)
>>> wsapp = websocket.WebSocketApp("wss://testnet.binance.vision/ws/btcusdt@trade",
on_message=on_message, get_mask_key=zero_mask_key)
>>> wsapp.run_forever()
```

Customizing opcode

WebSocket frames contain an opcode, which defines whether the frame contains text data, binary data, or is a special frame. The different opcode values are defined in RFC6455 section 11.8. Although the text opcode, 0x01, is the most commonly used value, the websocket-client library makes it possible to customize which opcode is used.

WebSocket custom opcode code example

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> ws = websocket.WebSocket()
>>> ws.connect("ws://echo.websocket.events")
>>> ws.send("Hello, Server", websocket.ABNF.OPCODE_TEXT)
>>> print(ws.recv())
>>> ws.send("This is a ping", websocket.ABNF.OPCODE_PING)
>>> ws.close()
```


WebSocketApp custom opcode code example

The WebSocketApp class contains different functions to handle different message opcodes. For instance, on_close, on_ping, on_pong, on_cont_message. One drawback of the current implementation (as of May 2021) is the lack of binary support for WebSocketApp, as noted by issue #351.

```
>>> import websocket
>>> websocket.enableTrace(True)

>>> def on_open(wsapp):
...     wsapp.send("Hello")

>>> def on_message(ws, message):
...     print(message)
...     ws.send("Send a ping", websocket.ABNF.OPCODE_PING)

>>> def on_pong(wsapp, message):
...     print("Got a pong! No need to respond")

>>> wsapp = websocket.WebSocketApp("ws://echo.websocket.events", on_open=on_open,
on_message=on_message, on_pong=on_pong)
>>> wsapp.run_forever()
```

Dispatching Multiple WebSocketApps

You can use an asynchronous dispatcher such as rel to run multiple WebSocketApps in the same application without resorting to threads.

WebSocketApp asynchronous dispatcher code example

```
>>> import websocket, rel

>>> addr = "wss://api.gemini.com/v1/marketdata/%s"
>>> for symbol in ["BTCUSD", "ETHUSD", "ETHBTC"]:
...     ws = websocket.WebSocketApp(addr % (symbol,), on_message=lambda w, m : print(m))
...     ws.run_forever(dispatcher=rel, reconnect=3)
>>> rel.signal(2, rel.abort)  # Keyboard Interrupt
>>> rel.dispatch()
```

README Examples

The examples are found in the README and are copied here for sphinx doctests to verify they run without errors.

Long-lived Connection

```
>>> import websocket
>>> import _thread
>>> import time
>>> import rel

>>> def on_message(ws, message):
...     print(message)

>>> def on_error(ws, error):
...     print(error)

>>> def on_close(ws, close_status_code, close_msg):
...     print("### closed ###")

>>> def on_open(ws):
...     print("Opened connection")

>>> if __name__ == "__main__":
...     websocket.enableTrace(True)
...     ws = websocket.WebSocketApp("wss://api.gemini.com/v1/marketdata/BTCUSD",
...                                 on_open=on_open,
...                                 on_message=on_message,
...                                 on_error=on_error,
...                                 on_close=on_close)

>>> ws.run_forever(dispatcher=rel, reconnect=5)  # Set dispatcher to automatic
reconnection, 5 second reconnect delay if connection closed unexpectedly
>>> rel.signal(2, rel.abort)  # Keyboard Interrupt
<Signal Object | Callback:"abort">
>>> rel.dispatch()
```

Short-lived Connection

```
>>> from websocket import create_connection

>>> ws = create_connection("ws://echo.websocket.events/")
>>> print(ws.recv())
echo.websocket.events sponsored by Lob.com
>>> print("Sending 'Hello, World'...")
Sending 'Hello, World'...
>>> ws.send("Hello, World")
18
>>> print("Sent")
Sent
>>> print("Receiving...")
Receiving...
>>> result =  ws.recv()
>>> print("Received '%s'" % result)
Received ...
>>> ws.close()
```

Real-world Examples

Other projects that depend on websocket-client may be able to illustrate more complex use cases for this library. A list of 600+ dependent projects is found on libraries.io, and a few of the projects using websocket-client are listed below:

- https://github.com/apache/airflow
- https://github.com/docker/docker-py
- https://github.com/scrapinghub/slackbot
- https://github.com/slackapi/python-slack-sdk
- https://github.com/wee-slack/wee-slack
- https://github.com/aluzzardi/wssh/
- https://github.com/llimllib/limbo
- https://github.com/miguelgrinberg/python-socketio
- https://github.com/invisibleroads/socketIO-client
- https://github.com/s4w3d0ff/python-poloniex
- https://github.com/Ape/samsungctl
- https://github.com/xchwarze/samsung-tv-ws-api
- https://github.com/andresriancho/websocket-fuzzer