websockets Documentation

Release 2.7

Aymeric Augustin

November 18, 2015

Contents

`websockets` is a library for developing WebSocket servers and clients in Python. It implements RFC 6455 with a focus on correctness and simplicity. It passes the Autobahn Testsuite.

Built on top of `asyncio`, Python's standard asynchronous I/O framework, it provides a straightforward API based on coroutines, making it easy to write highly concurrent applications.

CHAPTER 1

Installation

Installation is as simple as `pip install websockets`.

It requires Python ≥ 3.4 or Python 3.3 with the `asyncio` module, which is available with `pip install asyncio`.

CHAPTER 2

User guide

If you're new to `websockets`, Getting started describes usage patterns and provides examples.

If you've used `websockets` before and just need a quick reference, have a look at Cheat sheet.

If you need more details, the API documentation is for you.

If you're upgrading `websockets`, check the Changelog.

CHAPTER 3

Contributing

Bug reports, patches and suggestions welcome! Just open an issue or send a pull request.

3.1 Getting started

3.1.1 Basic example

Here’s a WebSocket server example. It reads a name from the client and sends a message.

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

import asyncio
import websockets

@asyncio.coroutine
def hello(websocket, path):
    name = yield from websocket.recv()
    print("< {}".format(name))
    greeting = "Hello {}!".format(name)

    yield from websocket.send(greeting)
    print("> {}".format(greeting))

start_server = websockets.serve(hello, 'localhost', 8765)

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

Here’s a corresponding client example.

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

import asyncio
import websockets

@asyncio.coroutine
def hello():
    websocket = yield from websockets.connect('ws://localhost:8765/')

    name = input("What's your name? ")
    yield from websocket.send(name)
    print("> {}".format(name))
    greeting = yield from websocket.recv()
    print("< {}".format(greeting))

    yield from websocket.close()

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

On the server side, the handler coroutine `hello` is executed once for each WebSocket connection. The connection is automatically closed when the handler returns.

3.1.2 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

import asyncio
import datetime
import random
import websockets

@asyncio.coroutine
def time(websocket, path):
    while True:
        now = datetime.datetime.utcnow().isoformat() + 'Z'
        if not websocket.open:
            return
        yield from websocket.send(now)
        yield from 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>
```

3.1.3 Common patterns

You will almost always want to process several messages during the lifetime of a connection. Therefore you must write a loop. Here are the recommended patterns to exit cleanly when the connection drops, either because the other side closed it or for any other reason.

Consumer

For receiving messages and passing them to a `consumer` coroutine:

```
@asyncio.coroutine
def handler(websocket, path):
    while True:
        message = yield from websocket.recv()
        if message is None:
            break
        yield from consumer(message)
```

`recv()` returns `None` when the connection is closed. In other words, `None` marks the end of the message stream. The handler coroutine should check for that case and return when it happens.

Producer

For getting messages from a `producer` coroutine and sending them:

```
@asyncio.coroutine
def handler(websocket, path):
    while True:
        message = yield from producer()
        if not websocket.open:
            break
        yield from websocket.send(message)
```

`send()` fails with an exception when it's called on a closed connection. Therefore the handler coroutine should check that the connection is still open before attempting to write and return otherwise.

Both

Of course, you can combine the two patterns shown above to read and write messages on the same connection:

```
@asyncio.coroutine
def handler(websocket, path):
    while True:
        listener_task = asyncio.ensure_future(websocket.recv())
        producer_task = asyncio.ensure_future(producer())
        done, pending = yield from asyncio.wait(
            [listener_task, producer_task],
            return_when=asyncio.FIRST_COMPLETED)

        if listener_task in done:
            message = listener_task.result()
            if message is None:
                break
            yield from consumer(message)
        else:
            listener_task.cancel()

        if producer_task in done:
            message = producer_task.result()
            if not websocket.open:
                break
            yield from websocket.send(message)
        else:
            producer_task.cancel()
```

(This code looks convoluted. If you know a more straightforward solution, please let me know about it!)

That's really all you have to know! `websockets` manages the connection under the hood so you don't have to.

3.2 Cheat sheet

3.2.1 Server

- Write a coroutine that handles a single connection. It receives a websocket protocol instance and the URI path in argument.
  - Call `recv()` and `send()` to receive and send messages at any time.
  - You may `ping()` or `pong()` if you wish but it isn't needed in general.
- Create a server with `serve()` which is similar to asyncio's `create_server()`.
  - The server takes care of establishing connections, then lets the handler execute the application logic, and finally closes the connection after the handler returns.
  - You may subclass `WebSocketServerProtocol` and pass it in the `klass` keyword argument for advanced customization.

3.2.2 Client

- Create a server with `connect()` which is similar to asyncio's `create_connection()`.
  - You may subclass `WebSocketClientProtocol` and pass it in the `klass` keyword argument for advanced customization.
- Call `recv()` and `send()` to receive and send messages at any time.
- You may `ping()` or `pong()` if you wish but it isn't needed in general.
- Call `close()` to terminate the connection.

3.2.3 Debugging

If you don't understand what `websockets` is doing, enable logging:

```
import logging
logger = logging.getLogger('websockets')
logger.setLevel(logging.INFO)
logger.addHandler(logging.StreamHandler())
```

The logs contains:

- Exceptions in the connection handler at the `ERROR` level
- Exceptions in the opening or closing handshake at the `INFO` level
- All frames at the `DEBUG` level — this can be very verbose

If you're new to `asyncio`, you will certainly encounter issues that are related to asynchronous programming in general rather than to `websockets` in particular. Fortunately Python's official documentation provides advice to develop with asyncio. Check it out: it's invaluable!

3.3 API

3.3.1 Design

`websockets` provides complete client and server implementations, as shown in the examples above. These functions are built on top of low-level APIs reflecting the two phases of the WebSocket protocol:

1. An opening handshake, in the form of an HTTP Upgrade request;
2. Data transfer, as framed messages, ending with a closing handshake.

The first phase is designed to integrate with existing HTTP software. `websockets` provides functions to build and validate the request and response headers.

The second phase is the core of the WebSocket protocol. `websockets` provides a standalone implementation on top of `asyncio` with a very simple API.

For convenience, public APIs can be imported directly from the `websockets` package, unless noted otherwise. Anything that isn't listed in this document is a private API.

3.3.2 High-level

Server

The *`websockets.server`* module defines a simple WebSocket server API.

`websockets.server.`**`serve`**(*ws_handler*, *host=None*, *port=None*, ***, *loop=None*, *klass=WebSocketServerProtocol*, *origins=None*, *subprotocols=None*, *extra_headers=None*, ***kwds*)

This coroutine creates a WebSocket server.

It's a wrapper around the event loop's `create_server()` method. `host`, `port` as well as extra keyword arguments are passed to `create_server()`. For example, you can set the `ssl` keyword argument to a `SSLContext` to enable TLS.

`ws_handler` is the WebSocket handler. It must be a coroutine accepting two arguments: a *`WebSocketServerProtocol`* and the request URI.

`serve()` accepts several optional arguments:

- `origins` defines acceptable Origin HTTP headers — include `''` if the lack of an origin is acceptable

- subprotocols is a list of supported subprotocols in order of decreasing preference
- extra_headers sets additional HTTP response headers — it can be a mapping, an iterable of (name, value) pairs, or a callable taking the request path and headers in arguments.

serve() yields a Server which provides:

- a close() method that closes open connections with status code 1001 and stops accepting new connections
- a wait_closed() coroutine that waits until closing handshakes complete and connections are closed.

Whenever a client connects, the server accepts the connection, creates a *WebSocketServerProtocol*, performs the opening handshake, and delegates to the WebSocket handler. Once the handler completes, the server performs the closing handshake and closes the connection.

Since there's no useful way to propagate exceptions triggered in handlers, they're sent to the 'websockets.server' logger instead. Debugging is much easier if you configure logging to print them:

```
import logging
logger = logging.getLogger('websockets.server')
logger.setLevel(logging.ERROR)
logger.addHandler(logging.StreamHandler())
```

class websockets.server.**WebSocketServerProtocol**(*ws_handler*, *ws_server*, ***, *origins=None*, *subprotocols=None*, *extra_headers=None*, *host=None*, *port=None*, *secure=None*, *timeout=10*, *max_size=2 ** 20*, *loop=None*)

Complete WebSocket server implementation as an asyncio.Protocol.

This class inherits most of its methods from *WebSocketCommonProtocol*.

For the sake of simplicity, it doesn't rely on a full HTTP implementation. Its support for HTTP responses is very limited.

handshake(*origins=None*, *subprotocols=None*, *extra_headers=None*)

Perform the server side of the opening handshake.

If provided, origins is a list of acceptable HTTP Origin values. Include '' if the lack of an origin is acceptable.

If provided, subprotocols is a list of supported subprotocols in order of decreasing preference.

If provided, extra_headers sets additional HTTP response headers. It can be a mapping or an iterable of (name, value) pairs. It can also be a callable taking the request path and headers in arguments.

Return the URI of the request.

select_subprotocol(*client_protos*, *server_protos*)

Pick a subprotocol among those offered by the client.

Client

The *websockets.client* module defines a simple WebSocket client API.

websockets.client.**connect**(*uri*, ***, *loop=None*, *klass=WebSocketClientProtocol*, *origin=None*, *subprotocols=None*, *extra_headers=None*, ***kwds*)

This coroutine connects to a WebSocket server.

It's a wrapper around the event loop's create_connection() method. Extra keyword arguments are passed to create_connection(). For example, you can set the ssl keyword argument to a SSLContext to enforce some TLS settings. When connecting to a `wss://` URI, if this argument isn't provided explicitly, it's set to `True`, which means Python's default `SSLContext` is used.

`connect()` accepts several optional arguments:

- `origin` sets the Origin HTTP header
- **`subprotocols` is a list of supported subprotocols in order of** decreasing preference
- `extra_headers` sets additional HTTP request headers – it can be a mapping or an iterable of (name, value) pairs

`connect()` yields a *`WebSocketClientProtocol`* which can then be used to send and receive messages.

It raises `InvalidURI` if `uri` is invalid and `InvalidHandshake` if the handshake fails.

class `websockets.client.`**`WebSocketClientProtocol`**(**, host=None, port=None, secure=None, timeout=10, max_size=2 ** 20, loop=None*)

Complete WebSocket client implementation as an `asyncio.Protocol`.

This class inherits most of its methods from *`WebSocketCommonProtocol`*.

`handshake`(*wsuri, origin=None, subprotocols=None, extra_headers=None*)

Perform the client side of the opening handshake.

If provided, `origin` sets the Origin HTTP header.

If provided, `subprotocols` is a list of supported subprotocols in order of decreasing preference.

If provided, `extra_headers` sets additional HTTP request headers. It must be a mapping or an iterable of (name, value) pairs.

Shared

The *`websockets.protocol`* module handles WebSocket control and data frames as specified in sections 4 to 8 of RFC 6455.

class `websockets.protocol.`**`WebSocketCommonProtocol`**(**, host=None, port=None, secure=None, timeout=10, max_size=2 ** 20, loop=None*)

This class implements common parts of the WebSocket protocol.

It assumes that the WebSocket connection is established. The handshake is managed in subclasses such as *`WebSocketServerProtocol`* and *`WebSocketClientProtocol`*.

It runs a task that stores incoming data frames in a queue and deals with control frames automatically. It sends outgoing data frames and performs the closing handshake.

The `host`, `port` and `secure` parameters are simply stored as attributes for handlers that need them.

The `timeout` parameter defines the maximum wait time in seconds for completing the closing handshake and, only on the client side, for terminating the TCP connection. *`close()`* will complete in at most this time on the server side and twice this time on the client side.

The `max_size` parameter enforces the maximum size for incoming messages in bytes. The default value is 1MB. `None` disables the limit. If a message larger than the maximum size is received, *`recv()`* will return `None` and the connection will be closed with status code 1009.

Once the handshake is complete, request and response HTTP headers are available:

- as a MIME `Message` in the `request_headers` and `response_headers` attributes

- as an iterable of (name, value) pairs in the `raw_request_headers` and `raw_response_headers` attributes

If a subprotocol was negotiated, it's available in the `subprotocol` attribute.

Once the connection is closed, the status code is available in the `close_code` attribute and the reason in `close_reason`.

`local_address`
Local address of the connection.

The address is a `(host, port)` tuple or `None` if the connection hasn't been established yet.

`remote_address`
Remote address of the connection.

The address is a `(host, port)` tuple or `None` if the connection hasn't been established yet.

`open`
This property is `True` when the connection is usable.

It may be used to handle disconnections gracefully.

`close`(*code=1000*, *reason=''*)
This coroutine performs the closing handshake.

It waits for the other end to complete the handshake. It doesn't do anything once the connection is closed.

It's safe to wrap this coroutine in `ensure_future()` since errors during connection termination aren't particularly useful.

`code` must be an `int` and `reason` a `str`.

`recv`()
This coroutine receives the next message.

It returns a `str` for a text frame and `bytes` for a binary frame.

When the end of the message stream is reached, or when a protocol error occurs, *`recv()`* returns `None`, indicating that the connection is closed.

`send`(*data*)
This coroutine sends a message.

It sends a `str` as a text frame and `bytes` as a binary frame.

It raises a `TypeError` for other inputs and *`InvalidState`* once the connection is closed.

`ping`(*data=None*)
This coroutine sends a ping.

It returns a `Future` which will be completed when the corresponding pong is received and which you may ignore if you don't want to wait.

A ping may serve as a keepalive.

`pong`(*data=b''*)
This coroutine sends a pong.

An unsolicited pong may serve as a unidirectional heartbeat.

Exceptions

exception `websockets.exceptions.`**`InvalidHandshake`**
Exception raised when a handshake request or response is invalid.

exception `websockets.exceptions.`**`InvalidOrigin`**
Exception raised when the origin in a handshake request is forbidden.

exception `websockets.exceptions.`**`InvalidState`**
Exception raised when an operation is forbidden in the current state.

exception `websockets.exceptions.`**`InvalidURI`**
Exception raised when an URI isn't a valid websocket URI.

exception `websockets.exceptions.`**`PayloadTooBig`**
Exception raised when a frame's payload exceeds the maximum size.

exception `websockets.exceptions.`**`WebSocketProtocolError`**
Internal exception raised when the remote side breaks the protocol.

3.3.3 Low-level

Opening handshake

The *`websockets.handshake`* module deals with the WebSocket opening handshake according to section 4 of RFC 6455.

It provides functions to implement the handshake with any existing HTTP library. You must pass to these functions:

- A `set_header` function accepting a header name and a header value,
- A `get_header` function accepting a header name and returning the header value.

The inputs and outputs of `get_header` and `set_header` are `str` objects containing only ASCII characters.

Some checks cannot be performed because they depend too much on the context; instead, they're documented below.

To accept a connection, a server must:

- Read the request, check that the method is GET, and check the headers with *`check_request()`*,
- Send a 101 response to the client with the headers created by *`build_response()`* if the request is valid; otherwise, send an appropriate HTTP error code.

To open a connection, a client must:

- Send a GET request to the server with the headers created by *`build_request()`*,
- Read the response, check that the status code is 101, and check the headers with *`check_response()`*.

`websockets.handshake.`**`build_request`**(*set_header*)
Build a handshake request to send to the server.

Return the `key` which must be passed to *`check_response()`*.

`websockets.handshake.`**`check_request`**(*get_header*)
Check a handshake request received from the client.

If the handshake is valid, this function returns the `key` which must be passed to *`build_response()`*.

Otherwise it raises an *`InvalidHandshake`* exception and the server must return an error like 400 Bad Request.

This function doesn't verify that the request is an HTTP/1.1 or higher GET request and doesn't perform Host and Origin checks. These controls are usually performed earlier in the HTTP request handling code. They're the responsibility of the caller.

`websockets.handshake.`**`build_response`**`(`*set_header*, *key*`)`
Build a handshake response to send to the client.

`key` comes from *`check_request()`*.

`websockets.handshake.`**`check_response`**`(`*get_header*, *key*`)`
Check a handshake response received from the server.

`key` comes from *`build_request()`*.

If the handshake is valid, this function returns `None`.

Otherwise it raises an *`InvalidHandshake`* exception.

This function doesn't verify that the response is an HTTP/1.1 or higher response with a 101 status code. These controls are the responsibility of the caller.

Data transfer

The *`websockets.framing`* module implements data framing as specified in section 5 of RFC 6455.

It deals with a single frame at a time. Anything that depends on the sequence of frames is implemented in *`websockets.protocol`*.

class `websockets.framing.`**`Frame`**
WebSocket frame.

- `fin` is the FIN bit
- `opcode` is the opcode
- `data` is the payload data

Only these three fields are needed by higher level code. The MASK bit, payload length and masking-key are handled on the fly by *`read_frame()`* and *`write_frame()`*.

`data`
Alias for field number 2

`fin`
Alias for field number 0

`opcode`
Alias for field number 1

`websockets.framing.`**`read_frame`**`(`*reader*, *mask*, ***, *max_size=None*`)`
Read a WebSocket frame and return a *`Frame`* object.

`reader` is a coroutine taking an integer argument and reading exactly this number of bytes, unless the end of file is reached.

`mask` is a `bool` telling whether the frame should be masked i.e. whether the read happens on the server side.

If `max_size` is set and the payload exceeds this size in bytes, *`PayloadTooBig`* is raised.

This function validates the frame before returning it and raises *`WebSocketProtocolError`* if it contains incorrect values.

`websockets.framing.`**`write_frame`**`(`*frame*, *writer*, *mask*`)`
Write a WebSocket frame.

`frame` is the *`Frame`* object to write.

`writer` is a function accepting bytes.

`mask` is a `bool` telling whether the frame should be masked i.e. whether the write happens on the client side.

This function validates the frame before sending it and raises *`WebSocketProtocolError`* if it contains incorrect values.

`websockets.framing.`**`parse_close`**`(`*data*`)`
Parse the data in a close frame.

Return `(code, reason)` when `code` is an `int` and `reason` a `str`.

Raise *`WebSocketProtocolError`* or `UnicodeDecodeError` if the data is invalid.

`websockets.framing.`**`serialize_close`**`(`*code*, *reason*`)`
Serialize the data for a close frame.

This is the reverse of *`parse_close()`*.

URI parser

The *`websockets.uri`* module implements parsing of WebSocket URIs according to section 3 of RFC 6455.

`websockets.uri.`**`parse_uri`**`(`*uri*`)`
This function parses and validates a WebSocket URI.

If the URI is valid, it returns a *`WebSocketURI`*.

Otherwise it raises an *`InvalidURI`* exception.

class `websockets.uri.`**`WebSocketURI`**
WebSocket URI.

- `secure` is the secure flag
- `host` is the lower-case host
- `port` if the integer port, it's always provided even if it's the default
- `resource_name` is the resource name, that is, the path and optional query

`host`
Alias for field number 1

`port`
Alias for field number 2

`resource_name`
Alias for field number 3

`secure`
Alias for field number 0

Utilities

The *`websockets.http`* module provides HTTP parsing functions. They're merely adequate for the WebSocket handshake messages.

These functions cannot be imported from `websockets`; they must be imported from *`websockets.http`*.

`websockets.http.`**`read_request`**`(`*stream*`)`
Read an HTTP/1.1 request from `stream`.

Return `(path, headers)` where `path` is a `str` and `headers` is a `Message`. `path` isn't URL-decoded.

Raise an exception if the request isn't well formatted.

The request is assumed not to contain a body.

`websockets.http.read_response(stream)`

Read an HTTP/1.1 response from `stream`.

Return `(status, headers)` where `status` is a `int` and `headers` is a `Message`.

Raise an exception if the request isn't well formatted.

The response is assumed not to contain a body.

3.4 Deployment

The author of `websockets` isn't aware of best practices for deploying network services based on `asyncio`.

He suggests running a Python script similar to the *server example*, perhaps inside a supervisor if you deem it useful.

If you can share knowledge on this topic, please file an issue. Thanks!

3.5 Limitations

Extensions aren't implemented. No extensions are registered at the time of writing.

The client doesn't attempt to guarantee that there is no more than one connection to a given IP adress in a CONNECTING state.

The client doesn't support connecting through a proxy.

3.6 Changelog

3.6.1 3.0

In development

3.6.2 2.7

- Added compatibility with Python 3.5.
- Refreshed documentation.

3.6.3 2.6

- Added `local_address` and `remote_address` attributes on protocols.
- Closed open connections with code 1001 when a server shuts down.
- Avoided TCP fragmentation of small frames.

3.6.4 2.5

- Improved documentation.
- Provided access to handshake request and response HTTP headers.
- Allowed customizing handshake request and response HTTP headers.
- Supported running on a non-default event loop.
- Returned a 403 error code instead of 400 when the request Origin isn't allowed.
- Cancelling `recv()` no longer drops the next message.
- Clarified that the closing handshake can be initiated by the client.
- Set the close status code and reason more consistently.
- Strengthened connection termination by simplifying the implementation.
- Improved tests, added tox configuration, and enforced 100% branch coverage.

3.6.5 2.4

- Added support for subprotocols.
- Supported non-default event loop.
- Added `loop` argument to `connect()` and `serve()`.

3.6.6 2.3

- Improved compliance of close codes.

3.6.7 2.2

- Added support for limiting message size.

3.6.8 2.1

- Added `host`, `port` and `secure` attributes on protocols.
- Added support for providing and checking Origin.

3.6.9 2.0

- Backwards-incompatible API change: `send()`, `ping()` and `pong()` are coroutines. They used to be regular functions.
- Added flow control.

3.6.10 1.0

- Initial public release.

3.7 License