websockets Documentation

Release 2.3.0

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 on Python's asynchronous I/O support introduced in PEP 3156, it provides an API based on coroutines, making it easy to write highly concurrent applications.

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` or in the Tulip repository.

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

CHAPTER 1

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))

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

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

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.

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.

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.

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

CHAPTER 2

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.

CHAPTER 3

High-level API

3.1 Server

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

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

This coroutine creates a WebSocket server.

It's a thin wrapper around the event loop's *create_server* method. *host*, *port* as well as extra keyword arguments are passed to *create_server*.

ws_handler is the WebSocket handler. It must be a coroutine accepting two arguments: a *WebSocketServerProtocol* and the request URI. If provided, *origin* is a list of acceptable Origin HTTP headers. Include `''` if the lack of an origin is acceptable.

It returns a *Server* object with a *close* method to stop the server.

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.DEBUG)
logger.addHandler(logging.StreamHandler())
```

class websockets.server.**WebSocketServerProtocol**(*ws_handler*, ***, *origins=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, this protocol doesn't inherit a proper HTTP implementation. Its support for HTTP responses is very limited.

handshake(*origins=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.

Return the URI of the request.

3.2 Client

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

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

This coroutine connects to a WebSocket server.

It accepts an `origin` keyword argument to set the Origin HTTP header.

It's a thin wrapper around the event loop's *create_connection* method. Extra keyword arguments are passed to *create_server*.

It returns 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.

Clients shouldn't close the WebSocket connection. Instead, they should wait until the server performs the closing handshake by yielding from the protocol's `worker` attribute.

connect() implements the sequence called "Establish a WebSocket Connection" in RFC 6455, except for the requirement that "there MUST be no more than one connection in a CONNECTING state."

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*)

Perform the client side of the opening handshake.

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

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

`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.

This is the expected way to terminate a connection on the server side.

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

It's usually safe to wrap this coroutine in *asyncio.async()* since errors during connection termination aren't particularly useful.

The *code* must be an `int` and the *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.

CHAPTER 4

Low-level API

4.1 Exceptions

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

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

exception websockets.exceptions.**InvalidURI**
Exception raised when an URI is invalid.

4.2 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 a 400.

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, usually 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.

4.3 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**(*fin*, *opcode*, *data*)

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, ie. 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, ie. 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()*.

4.4 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 namedtuple *(secure, host, port, resource_name)*

Otherwise, it raises an InvalidURI exception.

4.5 Utilities

The *websockets.http* module provides HTTP parsing functions. They're merely adequate for the WebSocket handshake messages. They're used by the sample client and servers.

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.

CHAPTER 5

Changelog

5.1 2.3

- Improved compliance of close codes.

5.2 2.2

- Added support for limiting message size.

5.3 2.1

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

5.4 2.0

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

5.5 1.0

- Initial public release.

CHAPTER 6

Limitations

Subprotocols and Extensions aren't implemented. Few subprotocols and no extensions are registered at the time of writing.

CHAPTER 7

License

W