websockets Documentation

Release 3.4

Aymeric Augustin

Aug 20, 2017

Contents

1 Installation **3**

2 User guide **5**

3 Contributing **7**

3.1 Getting started 7
3.2 Cheat sheet 11
3.3 API 12
3.4 Deployment 21
3.5 Limitations 24
3.6 Changelog 24
3.7 License 27

Python Module Index **29**

`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.

Getting started

Warning: This documentation is written for Python 3.5. If you're using Python 3.4 or 3.3, you will have to *adapt the code samples*.

Basic example

This section assumes Python 3.5. For older versions, read below. Here's a WebSocket server example. It reads a name from the client, sends a greeting, and closes the connection.

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

import asyncio
import websockets

async def hello(websocket, path):
    name = await websocket.recv()
    print("< {}".format(name))

    greeting = "Hello {}!".format(name)
    await 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()
```

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

Here's a corresponding client example.

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

import asyncio
import websockets

async def hello():
    async with websockets.connect('ws://localhost:8765') as websocket:
        name = input("What's your name? ")
        await websocket.send(name)
        print("> {}".format(name))

        greeting = await websocket.recv()
        print("< {}".format(greeting))

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

`async` and `await` aren't available in Python < 3.5. Here's how to adapt the client example for older Python versions.

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

import asyncio
import websockets

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

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

        greeting = yield from websocket.recv()
        print("< {}".format(greeting))

    finally:
        yield from websocket.close()

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

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
```

```
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>
```

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):
    while True:
        message = await websocket.recv()
        await consumer(message)
```

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

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)
```

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))
    producer_task = asyncio.ensure_future(producer_handler(websocket))
    done, pending = await asyncio.wait(
        [consumer_task, producer_task],
        return_when=asyncio.FIRST_COMPLETED,
    )

    for task in pending:
        task.cancel()
```

Registration

If you need to maintain a list of currently connected clients, you must register clients when they connect and unregister them when they disconnect.

```
connected = set()

async def handler(websocket, path):
    global connected
    # Register.
    connected.add(websocket)
    try:
        # Implement logic here.
        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.

That's all!

The design of the `websockets` API was driven by simplicity.

You don't have to worry about performing the opening or the closing handshake, answering pings, or any other behavior required by the specification.

`websockets` handles all this under the hood so you don't have to.

Python < 3.5

This documentation uses the `await` and `async` syntax introduced in Python 3.5.

If you're using Python 3.4 or 3.3, you must substitute:

```
async def ...
```

with:

```
@asyncio.coroutine
def ...
```

and:

```
await ...
```

with:

```
yield from ...
```

Otherwise you will encounter a `SyntaxError`.

Cheat sheet

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 exits normally or with an exception.
  - For advanced customization, you may subclass *`WebSocketServerProtocol`* and pass either this subclass or a factory function as the `create_protocol` argument.

Client

- Create a client with *`connect()`* which is similar to asyncio's `create_connection()`.
  - On Python 3.5, you can also use it as an asynchronous context manager.
  - For advanced customization, you may subclass `WebSocketClientProtocol` and pass either this subclass or a factory function as the `create_protocol` 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.
- If you aren't using *`connect()`* as a context manager, call *`close()`* to terminate the connection.

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 contain:

- 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!

Keeping connections open

Pinging the other side once in a while is a good way to check whether the connection is still working, and also to keep it open in case something kills idle connections after some time:

```
while True:
    try:
        msg = await asyncio.wait_for(ws.recv(), timeout=20)
    except asyncio.TimeoutError:
        # No data in 20 seconds, check the connection.
        try:
            await asyncio.wait_for(ws.ping(), timeout=10)
        except asyncio.TimeoutError:
            # No response to ping in 10 seconds, disconnect.
            break
    else:
        # do something with msg
        ...
```

API

Design

`websockets` provides complete client and server implementations, as shown in the *getting started guide*. 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.

High-level

Server

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

`websockets.server.`**`serve`**(*ws_handler*, *host=None*, *port=None*, ***, *create_protocol=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *origins=None*, *subprotocols=None*, *extra_headers=None*, ***kwds*)

Create, start, and return a *`WebSocketServer`* object.

`serve()` is a wrapper around the event loop's `create_server()` method. Internally, the function creates and starts a `Server` object by calling `create_server()`. The *`WebSocketServer`* keeps a reference to this object.

The returned *`WebSocketServer`* and its resources can be cleaned up by calling its *`close()`* and *`wait_closed()`* methods.

On Python 3.5 and greater, *`serve()`* can also be used as an asynchronous context manager. In this case, the server is shut down when exiting the context.

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

The `host` and `port` arguments, as well as unrecognized keyword arguments, are passed along to `create_server()`. For example, you can set the `ssl` keyword argument to a `SSLContext` to enable TLS.

The `create_protocol` parameter allows customizing the asyncio protocol that manages the connection. It should be a callable or class accepting the same arguments as *`WebSocketServerProtocol`* and returning a *`WebSocketServerProtocol`* instance. It defaults to *`WebSocketServerProtocol`*.

The behavior of the `timeout`, `max_size`, and `max_queue`, `read_limit`, and `write_limit` optional arguments is described in the documentation of *`WebSocketCommonProtocol`*.

`serve()` also accepts the following 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.

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.**WebSocketServer**(*loop*)

Wraps an underlying Server object.

This class provides the return type of *serve()*. This class shouldn't be instantiated directly.

Objects of this class store a reference to an underlying Server object returned by create_server(). The class stores a reference rather than inheriting from Server in part because create_server() doesn't support passing a custom Server class.

WebSocketServer supports cleaning up the underlying Server object and other resources by implementing the interface of asyncio.events.AbstractServer, namely its close() and wait_closed() methods.

close()

Close the underlying server, and clean up connections.

This calls close() on the underlying Server object, closes open connections with status code 1001, and stops accepting new connections.

wait_closed()

Wait until the underlying server and all connections are closed.

This calls wait_closed() on the underlying Server object and waits until closing handshakes are complete and all connections are closed.

This method must be called after *close()*.

class websockets.server.**WebSocketServerProtocol**(*ws_handler*, *ws_server*, ***, *host=None*, *port=None*, *secure=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *origins=None*, *subprotocols=None*, *extra_headers=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.

Raise *InvalidHandshake* or a subclass if the handshake fails.

Return the URI of the request.

process_request(*path*, *request_headers*)

Intercept the HTTP request and return an HTTP response if needed.

request_headers are a HTTPMessage.

If this coroutine returns None, the WebSocket handshake continues. If it returns a status code, headers and a optionally a response body, that HTTP response is sent and the connection is closed.

The HTTP status must be a HTTPStatus. HTTP headers must be an iterable of (name, value) pairs. If provided, the HTTP response body must be bytes.

(HTTPStatus was added in Python 3.5. Use a compatible object on earlier versions. Look at SWITCHING_PROTOCOLS in websockets.compatibility for an example.)

This method may be overridden to check the request headers and set a different status, for example to authenticate the request and return HTTPStatus.UNAUTHORIZED or HTTPStatus.FORBIDDEN.

It is declared as a coroutine because such authentication checks are likely to require network requests.

static 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*, ***, *create_protocol=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *loop=None*, *origin=None*, *subprotocols=None*, *extra_headers=None*, ***kwds*)
This coroutine connects to a WebSocket server at a given uri.

It yields a *WebSocketClientProtocol* which can then be used to send and receive messages.

connect() is a wrapper around the event loop's create_connection() method. Unknown 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.

The behavior of the timeout, max_size, and max_queue, read_limit, and write_limit optional arguments is described in the documentation of *WebSocketCommonProtocol*.

The create_protocol parameter allows customizing the asyncio protocol that manages the connection. It should be a callable or class accepting the same arguments as *WebSocketClientProtocol* and returning a *WebSocketClientProtocol* instance. It defaults to *WebSocketClientProtocol*.

connect() also accepts the following 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() raises InvalidURI if uri is invalid and InvalidHandshake if the opening handshake fails.

On Python 3.5, *connect()* can be used as a asynchronous context manager. In that case, the connection is closed when exiting the context.

class websockets.client.**WebSocketClientProtocol**(***, *host=None*, *port=None*, *secure=None*, *timeout=10*, *max_size=2 ** 20*, *max_queue=2 ** 5*, *read_limit=2 ** 16*, *write_limit=2 ** 16*, *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, max_queue=2 ** 5, read_limit=2 ** 16, write_limit=2 ** 16, 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 raise *`ConnectionClosed`* and the connection will be closed with status code 1009.

The `max_queue` parameter sets the maximum length of the queue that holds incoming messages. The default value is 32. 0 disables the limit. Messages are added to an in-memory queue when they're received; then *`recv()`* pops from that queue. In order to prevent excessive memory consumption when messages are received faster than they can be processed, the queue must be bounded. If the queue fills up, the protocol stops processing incoming data until *`recv()`* is called. In this situation, various receive buffers (at least in `asyncio` and in the OS) will fill up, then the TCP receive window will shrink, slowing down transmission to avoid packet loss.

Since Python can use up to 4 bytes of memory to represent a single character, each websocket connection may use up to `4 * max_size * max_queue` bytes of memory to store incoming messages. By default, this is 128MB. You may want to lower the limits, depending on your application's requirements.

The `read_limit` argument sets the high-water limit of the buffer for incoming bytes. The low-water limit is half the high-water limit. The default value is 64kB, half of asyncio's default (based on the current implementation of `StreamReader`).

The `write_limit` argument sets the high-water limit of the buffer for outgoing bytes. The low-water limit is a quarter of the high-water limit. The default value is 64kB, equal to asyncio's default (based on the current implementation of `_FlowControlMixin`).

As soon as the HTTP request and response in the opening handshake are processed, the request path is available in the `path` attribute, and the request and response HTTP headers are available:

- as a `HTTPMessage` 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

These attributes must be treated as immutable.

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`.

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. Thus it's idemptotent.

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, *`recv()`* raises *`ConnectionClosed`*. This can happen after a normal connection closure, a protocol error or a network failure.

Changed in version 3.0: *`recv()`* used to return `None` instead. Refer to the changelog for details.

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

It sends `str` as a text frame and `bytes` as a binary frame. It raises a `TypeError` for other inputs.

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 or as a check that the remote endpoint received all messages up to this point, with `yield from ws.ping()`.

By default, the ping contains four random bytes. The content may be overridden with the optional `data` argument which must be of type `str` (which will be encoded to UTF-8) or `bytes`.

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

An unsolicited pong may serve as a unidirectional heartbeat.

The content may be overridden with the optional `data` argument which must be of type `str` (which will be encoded to UTF-8) or `bytes`.

local_address
Local address of the connection.

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

remote_address
Remote address of the connection.

This 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 detect disconnections but this is discouraged per the EAFP principle. When `open` is `False`, using the connection raises a *`ConnectionClosed`* exception.

`state_name`
Current connection state, as a string.

Possible states are defined in the WebSocket specification: CONNECTING, OPEN, CLOSING, or CLOSED.

To check if the connection is open, use *`open`* instead.

Exceptions

exception `websockets.exceptions.`**`AbortHandshake`**(*status*, *headers*, *body=None*)
Exception raised to abort a handshake and return a HTTP response.

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

exception `websockets.exceptions.`**`InvalidMessage`**
Exception raised when the HTTP message in a handshake request is malformed.

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.`**`InvalidStatusCode`**(*status_code*)
Exception raised when a handshake response status code is invalid.

Provides the integer status code in its `status_code` attribute.

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

exception `websockets.exceptions.`**`ConnectionClosed`**(*code*, *reason*)
Exception raised when trying to read or write on a closed connection.

Provides the connection close code and reason in its `code` and `reason` attributes respectively.

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.

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 GET request from `stream`.

`stream` is an `StreamReader`.

Return `(path, headers)` where `path` is a `str` and `headers` is a list of `(name, value)` tuples.

`path` isn't URL-decoded or validated in any way.

Non-ASCII characters are represented with surrogate escapes.

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

Don't attempt to read the request body because WebSocket handshake requests don't have one. If the request contains a body, it may be read from `stream` after this coroutine returns.

`websockets.http.`**read_response**(*stream*)
Read an HTTP/1.1 response from `stream`.

`stream` is an `StreamReader`.

Return `(status_code, headers)` where `status_code` is a `int` and `headers` is a list of `(name, value)` tuples.

Non-ASCII characters are represented with surrogate escapes.

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

Don't attempt to read the response body, because WebSocket handshake responses don't have one. If the response contains a body, it may be read from `stream` after this coroutine returns.

Deployment

Backpressure

Note: This section discusses the concept of backpressure from the perspective of a server but the concepts also apply to clients. The issue is symmetrical.

With a naive implementation, if a server receives inputs faster than it can process them, or if it generates outputs faster than it can send them, data accumulates in buffers, eventually causing the server to run out of memory and crash.

The solution to this problem is backpressure. Any part of the server that receives inputs faster than it can it can process them and send the outputs must propagate that information back to the previous part in the chain.

`websockets` is designed to make it easy to get backpressure right.

For incoming data, `websockets` builds upon `StreamReader` which propagates backpressure to its own buffer and to the TCP stream. Frames are parsed from the input stream and added to a bounded queue. If the queue fills up, parsing halts until some the application reads a frame.

For outgoing data, `websockets` builds upon `StreamWriter` which implements flow control. If the output buffers grow too large, it waits until they're drained. That's why all APIs that write frames are asynchronous in websockets (since version 2.0).

Of course, it's still possible for an application to create its own unbounded buffers and break the backpressure. Be careful with queues.

Buffers

An asynchronous systems works best when its buffers are almost always empty.

For example, if a client sends frames too fast for a server, the queue of incoming frames will be constantly full. The server will always be 32 frames (by default) behind the client. This consumes memory and adds latency for no good reason.

If buffers are almost always full and that problem cannot be solved by adding capacity (typically because the system is bottlenecked by the output and constantly regulated by backpressure), reducing the size of buffers minimizes negative consequences.

By default `websockets` has rather high limits. You can decrease them according to your application's characteristics.

Bufferbloat can happen at every level in the stack where there is a buffer. The receiving side contains these buffers:

- OS buffers: you shouldn't need to tune them in general.
- `StreamReader` bytes buffer: the default limit is 64kB. You can set another limit by passing a `read_limit` keyword argument to *`connect()`* or *`serve()`*.
- `websockets` frame buffer: its size depends both on the size and the number of frames it contains. By default the maximum size is 1MB and the maximum number is 32. You can adjust these limits by setting the `max_size` and `max_queue` keyword arguments of *`connect()`* or *`serve()`*.

The sending side contains these buffers:

- `StreamWriter` bytes buffer: the default size is 64kB. You can set another limit by passing a `write_limit` keyword argument to *`connect()`* or *`serve()`*.
- OS buffers: you shouldn't need to tune them in general.

Deployment

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

You can run a script similar to the *server example*, inside a supervisor if you deem that useful.

You can also add a wrapper to daemonize the process. Third-party libraries provide solutions for that.

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

Graceful shutdown

You may want to close connections gracefully when shutting down the server, perhaps after executing some cleanup logic. There are two ways to achieve this with the object returned by *serve()*:

- using it as a asynchronous context manager, or
- calling its `close()` method, then waiting for its `wait_closed()` method to complete.

Tasks that handle connections will be cancelled, in the sense that *recv()* raises `CancelledError`.

On Unix systems, shutdown is usually triggered by sending a signal.

Here's a full example (Unix-only):

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

import asyncio
import signal
import websockets

async def echo(websocket, path):
    while True:
        try:
            msg = await websocket.recv()
        except websockets.ConnectionClosed:
            pass
        else:
            await websocket.send(msg)

async def echo_server(stop):
    async with websockets.serve(echo, 'localhost', 8765):
        await stop

loop = asyncio.get_event_loop()

# The stop condition is set when receiving SIGTERM.
stop = asyncio.Future()
loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)

# Run the server until the stop condition is met.
loop.run_until_complete(echo_server(stop))
```

`async`, `await`, and asynchronous context managers aren't available in Python < 3.5. Here's the equivalent for older Python versions:

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

import asyncio
import signal
import websockets

async def echo(websocket, path):
    while True:
        try:
```

```
            msg = await websocket.recv()
        except websockets.ConnectionClosed:
            pass
        else:
            await websocket.send(msg)

loop = asyncio.get_event_loop()

# Create the server.
start_server = websockets.serve(echo, 'localhost', 8765)
server = loop.run_until_complete(start_server)

# Run the server until SIGTERM.
stop = asyncio.Future()
loop.add_signal_handler(signal.SIGTERM, stop.set_result, None)
loop.run_until_complete(stop)

# Shut down the server.
server.close()
loop.run_until_complete(server.wait_closed())
```

It's more difficult to achieve the same effect on Windows. Some third-party projects try to help with this problem.

If your server doesn't run in the main thread, look at `call_soon_threadsafe()`.

Port sharing

The WebSocket protocol is an extension of HTTP/1.1. It can be tempting to serve both HTTP and WebSocket on the same port.

The author of `websockets` doesn't think that's a good idea, due to the widely different operational characteristics of HTTP and WebSocket.

If you need to respond to requests with a protocol other than WebSocket, for example TCP or HTTP health checks, run a server for that protocol on another port, within the same Python process, with `start_server()`.

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.

Changelog

3.5

In development

3.4

- Renamed *serve()* and *connect()*'s `klass` argument to `create_protocol` to reflect that it can also be a callable. For backwards compatibility, `klass` is still supported.
- *serve()* can be used as an asynchronous context manager on Python 3.5.
- Added support for customizing handling of incoming connections with *process_request()*.
- Made read and write buffer sizes configurable.
- Rewrote HTTP handling for simplicity and performance.
- Added an optional C extension to speed up low level operations.
- An invalid response status code during *connect()* now raises *InvalidStatusCode* with a `code` attribute.

3.3

- Reduced noise in logs caused by connection resets.
- Avoided crashing on concurrent writes on slow connections.

3.2

- Added `timeout`, `max_size`, and `max_queue` arguments to *connect()* and *serve()*.
- Made server shutdown more robust.

3.1

- Avoided a warning when closing a connection before the opening handshake.
- Added flow control for incoming data.

3.0

Warning: Version 3.0 introduces a backwards-incompatible change in the *recv()* API.

If you're upgrading from 2.x or earlier, please read this carefully.

recv() used to return `None` when the connection was closed. This required checking the return value of every call:

```
message = await websocket.recv()
if message is None:
    return
```

Now it raises a *ConnectionClosed* exception instead. This is more Pythonic. The previous code can be simplified to:

```
message = await websocket.recv()
```

When implementing a server, which is the more popular use case, there's no strong reason to handle such exceptions. Let them bubble up, terminate the handler coroutine, and the server will simply ignore them.

In order to avoid stranding projects built upon an earlier version, the previous behavior can be restored by passing `legacy_recv=True` to *`serve()`*, *`connect()`*, *`WebSocketServerProtocol`*, or *`WebSocketClientProtocol`*. `legacy_recv` isn't documented in their signatures but isn't scheduled for deprecation either.

Also:

- *`connect()`* can be used as an asynchronous context manager on Python 3.5.
- Updated documentation with `await` and `async` syntax from Python 3.5.
- *`ping()`* and *`pong()`* supports data passed as `str` in addition to `bytes`.
- Worked around an asyncio bug affecting connection termination under load.
- Made `state_name` atttribute on protocols a public API.
- Improved documentation.

2.7

- Added compatibility with Python 3.5.
- Refreshed documentation.

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.

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 status 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 code and reason more consistently.
- Strengthened connection termination by simplifying the implementation.
- Improved tests, added tox configuration, and enforced 100% branch coverage.

2.4

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

2.3

- Improved compliance of close codes.

2.2

- Added support for limiting message size.

2.1

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

2.0

Warning: Version 2.0 introduces a backwards-incompatible change in the *`send()`*, *`ping()`*, and *`pong()`* APIs.

If you're upgrading from 1.x or earlier, please read this carefully.

These APIs used to be functions. Now they're coroutines.

Instead of:

```
websocket.send(message)
```

you must now write:

```
await websocket.send(message)
```

Also:

- Added flow control for outgoing data.

1.0

- Initial public release.

License

```
Copyright (c) 2013-2015 Aymeric Augustin and contributors.
All rights reserved.

Redistribution and use in source and binary forms, with or without
modification, are permitted provided that the following conditions are met:

    * Redistributions of source code must retain the above copyright notice,
      this list of conditions and the following disclaimer.
    * Redistributions in binary form must reproduce the above copyright notice,
      this list of conditions and the following disclaimer in the documentation
      and/or other materials provided with the distribution.
    * Neither the name of websockets nor the names of its contributors may
      be used to endorse or promote products derived from this software without
      specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND
ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE
FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR
SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY,
OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE
OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

Python Module Index

W

websockets.client, 15
websockets.exceptions, 18
websockets.framing, 19
websockets.handshake, 18
websockets.http, 21
websockets.protocol, 16
websockets.server, 13
websockets.uri, 20

Index

A

AbortHandshake, 18

B

build_request() (in module websockets.handshake), 19
build_response() (in module websockets.handshake), 19

C

check_request() (in module websockets.handshake), 19
check_response() (in module websockets.handshake), 19
close() (websockets.protocol.WebSocketCommonProtocol method), 17
close() (websockets.server.WebSocketServer method), 14
connect() (in module websockets.client), 15
ConnectionClosed, 18

D

data (websockets.framing.Frame attribute), 20

F

fin (websockets.framing.Frame attribute), 20
Frame (class in websockets.framing), 19

H

handshake() (websockets.client.WebSocketClientProtocol method), 15
handshake() (websockets.server.WebSocketServerProtocol method), 14
host (websockets.uri.WebSocketURI attribute), 21

I

InvalidHandshake, 18
InvalidMessage, 18
InvalidOrigin, 18
InvalidState, 18
InvalidStatusCode, 18
InvalidURI, 18

L

local_address (websockets.protocol.WebSocketCommonProtocol attribute), 17

O

opcode (websockets.framing.Frame attribute), 20
open (websockets.protocol.WebSocketCommonProtocol attribute), 17

P

parse_close() (in module websockets.framing), 20
parse_uri() (in module websockets.uri), 20
PayloadTooBig, 18
ping() (websockets.protocol.WebSocketCommonProtocol method), 17
pong() (websockets.protocol.WebSocketCommonProtocol method), 17
port (websockets.uri.WebSocketURI attribute), 21
process_request() (websockets.server.WebSocketServerProtocol method), 14

R

read_frame() (in module websockets.framing), 20
read_request() (in module websockets.http), 21
read_response() (in module websockets.http), 21
recv() (websockets.protocol.WebSocketCommonProtocol method), 17
remote_address (websockets.protocol.WebSocketCommonProtocol attribute), 17
resource_name (websockets.uri.WebSocketURI attribute), 21

S

secure (websockets.uri.WebSocketURI attribute), 21
select_subprotocol() (websockets.server.WebSocketServerProtocol static method), 15

send() (websockets.protocol.WebSocketCommonProtocol method), 17
serialize_close() (in module websockets.framing), 20
serve() (in module websockets.server), 13
state_name (websockets.protocol.WebSocketCommonProtocol attribute), 18

W

wait_closed() (websockets.server.WebSocketServer method), 14
WebSocketClientProtocol (class in websockets.client), 15
WebSocketCommonProtocol (class in websockets.protocol), 16
WebSocketProtocolError, 18
websockets.client (module), 15
websockets.exceptions (module), 18
websockets.framing (module), 19
websockets.handshake (module), 18
websockets.http (module), 21
websockets.protocol (module), 16
websockets.server (module), 13
websockets.uri (module), 20
WebSocketServer (class in websockets.server), 13
WebSocketServerProtocol (class in websockets.server), 14
WebSocketURI (class in websockets.uri), 20
write_frame() (in module websockets.framing), 20