

## License

[MIT](#)

# ws: a Node.js WebSocket library

[Coverage Status](#)

ws is a simple to use, blazing fast, and thoroughly tested WebSocket client and server implementation.

Passes the quite extensive Autobahn test suite: [server](#), [client](#).

**Note:** This module does not work in the browser. The client in the docs is a reference to a backend with the role of a client in the WebSocket communication. Browser clients must use the native [WebSocket](#) object. To make the same code work seamlessly on Node.js and the browser, you can use one of the many wrappers available on npm, like [isomorphic-ws](#).

## Table of Contents

[Protocol support](#)

[Installing](#)

[Opt-in for performance](#)

[Legacy opt-in for performance](#)

[API docs](#)

[WebSocket compression](#)

[Usage examples](#)

[Sending and receiving text data](#)

- [Sending binary data](#)
- [Simple server](#)
- [External HTTP/S server](#)
- [Multiple servers sharing a single HTTP/S server](#)
- [Client authentication](#)
- [Server broadcast](#)
- [Round-trip time](#)
- [Use the Node.js streams API](#)
- [Other examples](#)
- [FAQ](#)
- [How to get the IP address of the client?](#)
- [How to detect and close broken connections?](#)
- [How to connect via a proxy?](#)
- [Changelog](#)
- [License](#)

## Protocol support

- **HyBi drafts 07-12** (Use the option `protocolVersion:`
- **HyBi drafts 13-17** (Current default, alternatively option `protocolVersion: 13`)

2

```
// instead of `WebSocket#close()`,
// Delay should be equal to the
// interval at which your server
// sends out pings plus a conservative
// assumption of the latency.
this.pingTimeout = setTimeout(() => {
  this.terminate();
}, 30000 + 1000);
}

const client = new WebSocket('wss://
websocket-echo.com/');

client.on('error', console.error);
client.on('open', heartbeat);
client.on('ping', heartbeat);
client.on('close', function clear() {
  clearTimeout(this.pingTimeout);
});
```

## How to connect via a proxy?

Use a custom http.Agent implementation like [https-proxy-agent](#) or [socks-proxy-agent](#).

## Changelog

We're using the GitHub [releases](#) for changelog entries.

19

```

});

const interval = setInterval(function
  ping() {
    wss.clients.forEach(function each(ws)
      {
        if (ws.isAlive === false) return
          ws.terminate();

        ws.isAlive = false;
        ws.ping();
      });
    }, 30000);

wss.on('close', function close() {
  clearInterval(interval);
});

```

Pong messages are automatically sent in response to ping messages as required by the spec.

Just like the server example above, your clients might as well lose connection without knowing it. You might want to add a ping listener on your clients to prevent that. A simple implementation would be:

```

import WebSocket from 'ws';

function heartbeat() {
  clearTimeout(this.pingTimeout);

  // Use `WebSocket#terminate()`, which
  // immediately destroys the
  // connection,

```

## Installing

```
npm install ws
```

### Opt-in for performance

[bufferutil](#) is an optional module that can be installed alongside the ws module:

```
npm install --save-optional bufferutil
```

This is a binary addon that improves the performance of certain operations such as masking and unmasking the data payload of the WebSocket frames. Prebuilt binaries are available for the most popular platforms, so you don't necessarily need to have a C++ compiler installed on your machine.

To force ws to not use bufferutil, use the [WS\\_NO\\_BUFFER\\_UTIL](#) environment variable. This can be useful to enhance security in systems where a user can put a package in the package search path of an application of another user, due to how the Node.js resolver algorithm works.

### Legacy opt-in for performance

If you are running on an old version of Node.js (prior to v18.14.0), ws also supports the [utf-8-validate](#) module:

```
npm install --save-optional utf-8-
validate
```

This contains a binary polyfill for [buffer.isUtf8\(\)](#).  
To force ws not to use utf-8-validate, use the [WS\\_NO\\_UTF\\_8\\_VALIDATE](#) environment variable.

## API docs

See [/doc/ws.md](#) for Node.js-like documentation of ws classes and utility functions.

## WebSocket compression

ws supports the [permessage-deflate extension](#) which enables the client and server to negotiate a compression algorithm and its parameters, and then selectively apply it to the data payloads of each WebSocket message.  
The extension is disabled by default on the server and enabled by default on the client. It adds a significant overhead in terms of performance and memory consumption so we suggest to enable it only if it is really needed.

Note that Node.js has a variety of issues with high-performance compression, where increased concurrency, especially on Linux, can lead to [catastrophic memory fragmentation](#) and slow performance. If you intend to use permessage-deflate in production, it is worthwhile to set up a test

```
ws.on('connection', function
  connection(ws, req) {
    const ip = req.headers['x-forwarded-
for'].split(',')[0].trim();
    ws.on('error', console.error);
  });
```

## How to detect and close broken connections?

Sometimes, the link between the server and the client can be interrupted in a way that keeps both the server and the client unaware of the broken state of the connection (e.g. when pulling the cord).  
In these cases, ping messages can be used as a means to verify that the remote endpoint is still responsive.

```
import { WebSocketServer } from 'ws';

function heartbeat() {
  this.isAlive = true;
}

const wss = new WebSocketServer({ port:
8080 });

wss.on('connection', function
  connection(ws) {
    ws.isAlive = true;
    ws.on('error', console.error);
    ws.on('pong', heartbeat);
```

## Other examples

For a full example with a browser client communicating with a ws server, see the examples folder.

Otherwise, see the test cases.

## FAQ

### How to get the IP address of the client?

The remote IP address can be obtained from the raw socket.

```
import { WebSocketServer } from 'ws';

const wss = new WebSocketServer({ port: 8080 });

wss.on('connection', function connection(ws, req) {
  const ip = req.socket.remoteAddress;

  ws.on('error', console.error);
});
```

When the server runs behind a proxy like NGINX, the de-facto standard is to use the X-Forwarded-For header.

representative of your workload and ensure Node.js/zlib will handle it with acceptable performance and memory usage.

Tuning of permessage-deflate can be done via the options defined below. You can also use `zlibDeflateOptions` and `zlibInflateOptions`, which is passed directly into the creation of [raw deflate/inflate streams](#).

See [the docs](#) for more options.

```
import WebSocket, { WebSocketServer }
  from 'ws';

const wss = new WebSocketServer({
  port: 8080,
  perMessageDeflate: {
    zlibDeflateOptions: {
      // See zlib defaults.
      chunkSize: 1024,
      memLevel: 7,
      level: 3
    },
    zlibInflateOptions: {
      chunkSize: 10 * 1024
    },
    // Other options settable:
    clientNoContextTakeover: true, // Defaults to negotiated value.
    serverNoContextTakeover: true, // Defaults to negotiated value.
    serverMaxWindowBits: 10, // Defaults to negotiated value.
    // Below options specified as
    // default values.
```

```

concurrencyLimit:
10, // Limits zlib concurrency
for perf.
threshold: 1024 // Size (in bytes)
below which messages
// should not be compressed if
context takeover is disabled.
}
}

```

The client will only use the extension if it is supported and enabled on the server. To always disable the extension on the client, set the perMessageDeflate option to false.

```

import WebSocket from 'ws';

const ws = new WebSocket('ws://
www.host.com/path', {
  perMessageDeflate: false
});

const ws = new WebSocket('ws://
www.host.com/path');

```

## Sending and receiving text data

## Usage examples

```

ws.on('message', function message(data)
{
  console.log(`Round-trip time: $
{Date.now() - data} ms`);
  setTimeout(function timeout() {
    ws.send(Date.now());
  }, 500);
});

```

## Use the Node.js streams API

```

import WebSocket,
{ createWebSocketStream } from
'ws';

const ws = new WebSocket('wss://
websocket-echo.com/');

const duplex = xduplex =
createWebSocketStream(ws, {
  encoding: 'utf8' });

duplex.on('error', console.error);

duplex.pipe(process.stdout);
process.stdin.pipe(duplex);

```

```

ws.on('message', function
  message(data, isBinary) {
    wss.clients.forEach(function
      each(client) {
        if (client !== ws &&
            client.readyState ===
              WebSocket.OPEN) {
          client.send(data, { binary:
            isBinary });
        }
      });
    });
  });
});

```

### Round-trip time

```

import WebSocket from 'ws';

const ws = new WebSocket('wss://
  websocket-echo.com/');

ws.on('error', console.error);

ws.on('open', function open() {
  console.log('connected');
  ws.send(Date.now());
});

ws.on('close', function close() {
  console.log('disconnected');
});

```

```

ws.on('error', console.error);

ws.on('open', function open() {
  ws.send('something');
});

ws.on('message', function message(data)
  {
    console.log('received: %s', data);
  });

```

### Sending binary data

```

import WebSocket from 'ws';

const ws = new WebSocket('ws://
  www.host.com/path');

ws.on('error', console.error);

ws.on('open', function open() {
  const array = new Float32Array(5);

  for (var i = 0; i < array.length; +
    +i) {
    array[i] = i / 2;
  }

  ws.send(array);
});

```

```
import { createServer } from 'https';
import { readFileSync } from 'fs';
import { WebSocketServer } from 'ws';

const server = createServer({
  cert: readFileSync('/path/to/
    cert.pem'),
  key: readFileSync('/path/to/key.pem')
});
```

### External HTTP/S server

```
import { WebSocketServer } from 'ws';

const wss = new WebSocketServer({ port:
    8080 });

wss.on('connection', function
    connection(ws) {
  ws.on('error', console.error);

  ws.on('message', function
    message(data) {
    console.log('received: %s', data);
  });

  ws.send('something');
});
```

### Simple server

```
import WebSocket, { WebSocketServer }
    from 'ws';

const wss = new WebSocketServer({ port:
    8080 });

wss.on('connection', function
    connection(ws) {
  ws.on('error', console.error);
```

A client WebSocket broadcasting to every other connected  
WebSocket clients, excluding itself.

```
const wss = new WebSocketServer({ port:
    8080 });

wss.on('connection', function
    connection(ws) {
  ws.on('error', console.error);

  ws.on('message', function
    message(data, isBinary) {
    wss.clients.forEach(function
      each(client) {
        if (client.readyState ===
            WebSocket.OPEN) {
          client.send(data, { binary:
              isBinary });
        }
      });
    });
  });
});
```

```

authenticate(request, function
  next(err, client) {
    if (err || !client) {
      socket.write('HTTP/1.1 401
        Unauthorized\r\n\r\n');
      socket.destroy();
      return;
    }

    socket.removeListener('error',
      onSocketError);

    wss.handleUpgrade(request, socket,
      head, function done(ws) {
        wss.emit('connection', ws,
          request, client);
      });
  });
});

server.listen(8080);

```

Also see the provided [example](#) using express-session.

## Server broadcast

A client WebSocket broadcasting to all connected WebSocket clients, including itself.

```

import WebSocket, { WebSocketServer }
  from 'ws';

```

```

const wss = new
  WebSocketServer({ server });

wss.on('connection', function
  connection(ws) {
    ws.on('error', console.error);

    ws.on('message', function
      message(data) {
        console.log('received: %s', data);
      });

    ws.send('something');
  });

server.listen(8080);

```

## Multiple servers sharing a single HTTP/S server

```

import { createServer } from 'http';
import { WebSocketServer } from 'ws';

const server = createServer();
const wss1 = new WebSocketServer({
  noServer: true });
const wss2 = new WebSocketServer({
  noServer: true });

wss1.on('connection', function
  connection(ws) {
    ws.on('error', console.error);

```

```

// ...
    }
    mss2.on('connection', function
        connection(ws) {
            ws.on('error', console.error);

            // ...
        })
    });

server.on('upgrade', function
    upgrade(request, socket, head) {
        const { pathname } = new
            URL(request.url, 'ws://
                base.url');
        if (pathname === '/foo') {
            mss1.handleUpgrade(request, socket,
                head, function done(ws) {
                    mss1.emit('connection', ws,
                        request);
                });
            } else if (pathname === '/bar') {
                mss2.handleUpgrade(request, socket,
                    head, function done(ws) {
                        mss2.emit('connection', ws,
                            request);
                    });
            }
        });
    });
server.listen(8080);

```

10

```

import { createServer } from 'http';
import { WebSocketServer } from 'ws';

function onSocketError(err) {
    console.error(err);
}

const server = createServer();
const wss = new WebSocketServer({
    noServer: true });

wss.on('connection', function
    connection(ws, request, client)
    {
        ws.on('error', console.error);
        ws.on('message', function
            message(data) {
                console.log('Received message $
                    {data} from user ${client}');
            });
        });
server.on('upgrade', function
    upgrade(request, socket, head) {
        socket.on('error', onSocketError);
        // This function is not defined on
        purpose. Implement it with your
        own logic.
    });

```

11

## Client authentication