ws: a Node.js WebSocket library

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 back end 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

- License

Protocol support

- **HyBi drafts 07-12** (Use the option `protocolVersion: 8`)
- **HyBi drafts 13-17** (Current default, alternatively option `protocolVersion: 13`)

Installing

```
npm install --save ws
```

Opt-in for performance and spec compliance

There are 2 optional modules that can be installed along side with the ws module. These modules are binary addons which improve certain operations. Prebuilt binaries are available for the most popular platforms so you don't necessarily need to have a C++ compiler installed on your machine.

- `npm install --save-optional bufferutil`: Allows to efficiently perform operations such as masking and unmasking the data payload of the WebSocket frames.
- `npm install --save-optional utf-8-validate`: Allows to efficiently check if a message contains valid UTF-8 as required by the spec.

API docs

See `/doc/ws.md` for Node.js-like docs for the ws classes.

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 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.

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({
  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.
  }
});
```

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`.

```
const WebSocket = require('ws');

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

Usage examples

Sending and receiving text data

```
const WebSocket = require('ws');

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

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

ws.on('message', function incoming(data) {
  console.log(data);
});
```

Sending binary data

```
const WebSocket = require('ws');

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

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);
});
```

Simple server

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

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

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

External HTTP/S server

```
const fs = require('fs');
const https = require('https');
const WebSocket = require('ws');
const server = new https.createServer({
  cert: fs.readFileSync('/path/to/cert.pem'),
  key: fs.readFileSync('/path/to/key.pem')
});
const wss = new WebSocket.Server({ server });

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

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

server.listen(8080);
```

Multiple servers sharing a single HTTP/S server

```
const http = require('http');
const WebSocket = require('ws');

const server = http.createServer();
const wss1 = new WebSocket.Server({ noServer: true });
const wss2 = new WebSocket.Server({ noServer: true });

wss1.on('connection', function connection(ws) {
  // ...
});

wss2.on('connection', function connection(ws) {
  // ...
});

server.on('upgrade', function upgrade(request, socket,
head) {
  const pathname = url.parse(request.url).pathname;
  if (pathname === '/foo') {
    wss1.handleUpgrade(request, socket, head, function
done(ws) {
      wss1.emit('connection', ws, request);
    });
  } else if (pathname === '/bar') {
    wss2.handleUpgrade(request, socket, head, function
done(ws) {
      wss2.emit('connection', ws, request);
    });
  } else {
    socket.destroy();
  }
});

server.listen(8080);
```

Server broadcast

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

// Broadcast to all.
wss.broadcast = function broadcast(data) {
  wss.clients.forEach(function each(client) {
    if (client.readyState === WebSocket.OPEN) {
      client.send(data);
    }
  });
};

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(data) {
    // Broadcast to everyone else.
    wss.clients.forEach(function each(client) {
      if (client !== ws && client.readyState ===
WebSocket.OPEN) {
        client.send(data);
      }
    });
  });
});
```

echo.websocket.org demo

```
const WebSocket = require('ws');

const ws = new WebSocket('wss://echo.websocket.org/', {
  origin: 'https://websocket.org'
});

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

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

ws.on('message', function incoming(data) {
  console.log(`Roundtrip time: ${Date.now() - data}
ms`);

  setTimeout(function timeout() {
    ws.send(Date.now());
  }, 500);
});
```

Other examples

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

Otherwise, see the test cases.

Error handling best practices

```
// If the WebSocket is closed before the following send
is attempted
ws.send('something');

// Errors (both immediate and async write errors) can
be detected in an optional
// callback. The callback is also the only way of being
notified that data has
// actually been sent.
ws.send('something', function ack(error) {
  // If error is not defined, the send has been
completed, otherwise the error
  // object will indicate what failed.
});

// Immediate errors can also be handled with
`try...catch`, but **note** that
// since sends are inherently asynchronous, socket
write failures will *not* be
// captured when this technique is used.
try { ws.send('something'); }
catch (e) { /* handle error */ }
```

FAQ

How to get the IP address of the client?

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

```
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: 8080 });

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

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

```
wss.on('connection', function connection(ws, req) {
  const ip = req.headers['x-forwarded-for'].split(/\s*,
\s*/)[0];
});
```

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.

```
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 8080 });

function noop() {}

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

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

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

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

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

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.