

[🏠](#) → [Network requests](#)

📅 28th November 2019

# WebSocket

The `WebSocket` protocol, described in the specification [RFC 6455](#) provides a way to exchange data between browser and server via a persistent connection. The data can be passed in both directions as “packets”, without breaking the connection and additional HTTP-requests.

WebSocket is especially great for services that require continuous data exchange, e.g. online games, real-time trading systems and so on.

## A simple example

To open a websocket connection, we need to create new `WebSocket` using the special protocol `ws` in the url:

```
1 let socket = new WebSocket("ws://javascript.info");
```

There's also encrypted `wss://` protocol. It's like HTTPS for websockets.

### Always prefer `wss://`

The `wss://` protocol not only encrypted, but also more reliable.

That's because `ws://` data is not encrypted, visible for any intermediary. Old proxy servers do not know about `WebSocket`, they may see “strange” headers and abort the connection.

On the other hand, `wss://` is `WebSocket` over TLS, (same as HTTPS is HTTP over TLS), the transport security layer encrypts the data at sender and decrypts at the receiver. So data packets are passed encrypted through proxies. They can't see what's inside and let them through.

Once the socket is created, we should listen to events on it. There are totally 4 events:

- **open** – connection established,
- **message** – data received,
- **error** – websocket error,
- **close** – connection closed.

...And if we'd like to send something, then `socket.send(data)` will do that.

Here's an example:

```
1 let socket = new WebSocket("wss://javascript.info/article/websocket/demo/he
2
```

```

3  socket.onopen = function(e) {
4      alert("[open] Connection established");
5      alert("Sending to server");
6      socket.send("My name is John");
7  };
8
9  socket.onmessage = function(event) {
10     alert(`[message] Data received from server: ${event.data}`);
11 };
12
13 socket.onclose = function(event) {
14     if (event.wasClean) {
15         alert(`[close] Connection closed cleanly, code=${event.code} reason=${e
16     } else {
17         // e.g. server process killed or network down
18         // event.code is usually 1006 in this case
19         alert('[close] Connection died');
20     }
21 };
22
23 socket.onerror = function(error) {
24     alert(`[error] ${error.message}`);
25 };

```

For demo purposes, there's a small server [server.js](#) written in Node.js, for the example above, running. It responds with "Hello from server, John", then waits 5 seconds and closes the connection.

So you'll see events `open` → `message` → `close`.

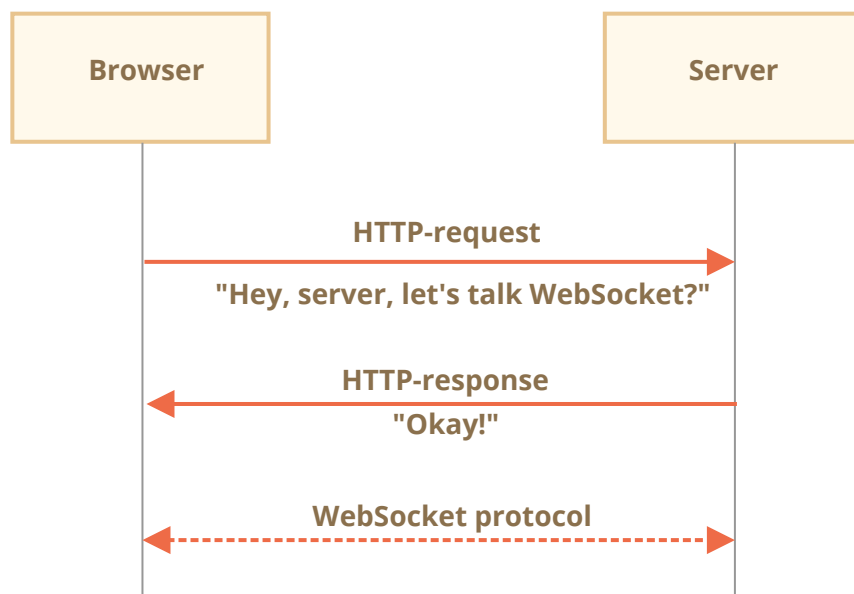
That's actually it, we can talk WebSocket already. Quite simple, isn't it?

Now let's talk more in-depth.

## Opening a websocket

When `new WebSocket(url)` is created, it starts connecting immediately.

During the connection the browser (using headers) asks the server: "Do you support WebSocket?" And if the server replies "yes", then the talk continues in WebSocket protocol, which is not HTTP at all.



Here's an example of browser headers for request made by `new WebSocket("wss://javascript.info/chat")`.

```
1 GET /chat
2 Host: javascript.info
3 Origin: https://javascript.info
4 Connection: Upgrade
5 Upgrade: websocket
6 Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
7 Sec-WebSocket-Version: 13
```

- `Origin` – the origin of the client page, e.g. `https://javascript.info`. `WebSocket` objects are cross-origin by nature. There are no special headers or other limitations. Old servers are unable to handle `WebSocket` anyway, so there are no compability issues. But `Origin` header is important, as it allows the server to decide whether or not to talk `WebSocket` with this website.
- `Connection: Upgrade` – signals that the client would like to change the protocol.
- `Upgrade: websocket` – the requested protocol is “websocket”.
- `Sec-WebSocket-Key` – a random browser-generated key for security.
- `Sec-WebSocket-Version` – `WebSocket` protocol version, 13 is the current one.

### **WebSocket handshake can't be emulated**

We can't use `XMLHttpRequest` or `fetch` to make this kind of HTTP-request, because JavaScript is not allowed to set these headers.

If the server agrees to switch to `WebSocket`, it should send code 101 response:

```
1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBULZAlC2g=
```

Here `Sec-WebSocket-Accept` is `Sec-WebSocket-Key`, recoded using a special algorithm. The browser uses it to make sure that the response corresponds to the request.

Afterwards, the data is transfered using `WebSocket` protocol, we'll see its structure (“frames”) soon. And that's not HTTP at all.

## Extensions and subprotocols

There may be additional headers `Sec-WebSocket-Extensions` and `Sec-WebSocket-Protocol` that describe extensions and subprotocols.

For instance:

- **Sec-WebSocket-Extensions:** deflate-frame means that the browser supports data compression. An extension is something related to transferring the data, functionality that extends WebSocket protocol. The header **Sec-WebSocket-Extensions** is sent automatically by the browser, with the list of all extensions it supports.
- **Sec-WebSocket-Protocol:** soap, wamp means that we'd like to transfer not just any data, but the data in [SOAP](#) or WAMP ("The WebSocket Application Messaging Protocol") protocols. WebSocket subprotocols are registered in the [IANA catalogue](#). So, this header describes data formats that we're going to use.

This optional header is set using the second parameter of `new WebSocket`. That's the array of subprotocols, e.g. if we'd like to use SOAP or WAMP:

```
1 let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
```

The server should respond with a list of protocols and extensions that it agrees to use.

For example, the request:

```
1 GET /chat
2 Host: javascript.info
3 Upgrade: websocket
4 Connection: Upgrade
5 Origin: https://javascript.info
6 Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
7 Sec-WebSocket-Version: 13
8 Sec-WebSocket-Extensions: deflate-frame
9 Sec-WebSocket-Protocol: soap, wamp
```

Response:

```
1 101 Switching Protocols
2 Upgrade: websocket
3 Connection: Upgrade
4 Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBULZAlC2g=
5 Sec-WebSocket-Extensions: deflate-frame
6 Sec-WebSocket-Protocol: soap
```

Here the server responds that it supports the extension "deflate-frame", and only SOAP of the requested subprotocols.

## Data transfer

WebSocket communication consists of "frames" – data fragments, that can be sent from either side, and can be of several kinds:

- "text frames" – contain text data that parties send to each other.
- "binary data frames" – contain binary data that parties send to each other.

- “ping/pong frames” are used to check the connection, sent from the server, the browser responds to these automatically.
- there’s also “connection close frame” and a few other service frames.

In the browser, we directly work only with text or binary frames.

**WebSocket .send() method can send either text or binary data.**

A call `socket.send(body)` allows `body` in string or a binary format, including `Blob`, `ArrayBuffer`, etc. No settings required: just send it out in any format.

**When we receive the data, text always comes as string. And for binary data, we can choose between `Blob` and `ArrayBuffer` formats.**

That’s set by `socket.bufferType` property, it’s “blob” by default, so binary data comes as `Blob` objects.

`Blob` is a high-level binary object, it directly integrates with `<a>`, `<img>` and other tags, so that’s a sane default. But for binary processing, to access individual data bytes, we can change it to “arraybuffer” :

```
1 socket.bufferType = "arraybuffer";
2 socket.onmessage = (event) => {
3   // event.data is either a string (if text) or arraybuffer (if binary)
4 };
```

## Rate limiting

Imagine, our app is generating a lot of data to send. But the user has a slow network connection, maybe on a mobile internet, outside of a city.

We can call `socket.send(data)` again and again. But the data will be buffered (stored) in memory and sent out only as fast as network speed allows.

The `socket.bufferedAmount` property stores how many bytes are buffered at this moment, waiting to be sent over the network.

We can examine it to see whether the socket is actually available for transmission.

```
1 // every 100ms examine the socket and send more data
2 // only if all the existing data was sent out
3 setInterval(() => {
4   if (socket.bufferedAmount == 0) {
5     socket.send(moreData());
6   }
7 }, 100);
```

## Connection close

Normally, when a party wants to close the connection (both browser and server have equal rights), they send a “connection close frame” with a numeric code and a textual reason.

The method for that is:

```
1 socket.close([code], [reason]);
```

- `code` is a special WebSocket closing code (optional)
- `reason` is a string that describes the reason of closing (optional)

Then the other party in `close` event handler gets the code and the reason, e.g.:

```
1 // closing party:
2 socket.close(1000, "Work complete");
3
4 // the other party
5 socket.onclose = event => {
6   // event.code === 1000
7   // event.reason === "Work complete"
8   // event.wasClean === true (clean close)
9 };
```

Most common code values:

- 1000 – the default, normal closure (used if no `code` supplied),
- 1006 – no way to such code manually, indicates that the connection was lost (no close frame).

There are other codes like:

- 1001 – the party is going away, e.g. server is shutting down, or a browser leaves the page,
- 1009 – the message is too big to process,
- 1011 – unexpected error on server,
- ...and so on.

The full list can be found in [RFC6455, §7.4.1](#).

WebSocket codes are somewhat like HTTP codes, but different. In particular, any codes less than 1000 are reserved, there'll be an error if we try to set such a code.

```
1 // in case connection is broken
2 socket.onclose = event => {
3   // event.code === 1006
4   // event.reason === ""
5   // event.wasClean === false (no closing frame)
6 };
```

## Connection state

To get connection state, additionally there's `socket.readyState` property with values:

- 0 – “CONNECTING”: the connection has not yet been established,
- 1 – “OPEN”: communicating,
- 2 – “CLOSING”: the connection is closing,
- 3 – “CLOSED”: the connection is closed.

# Chat example

Let's review a chat example using browser WebSocket API and Node.js WebSocket module <https://github.com/websockets/ws>. We'll pay the main attention to the client side, but the server is also simple.

HTML: we need a `<form>` to send messages and a `<div>` for incoming messages:

```

1 <!-- message form -->
2 <form name="publish">
3   <input type="text" name="message">
4   <input type="submit" value="Send">
5 </form>
6
7 <!-- div with messages -->
8 <div id="messages"></div>

```

From JavaScript we want three things:

1. Open the connection.
2. On form submission – `socket.send(message)` for the message.
3. On incoming message – append it to `div#messages`.

Here's the code:

```

1 let socket = new WebSocket("wss://javascript.info/article/websocket/chat/ws")
2
3 // send message from the form
4 document.forms.publish.onsubmit = function() {
5   let outgoingMessage = this.message.value;
6
7   socket.send(outgoingMessage);
8   return false;
9 };
10
11 // message received - show the message in div#messages
12 socket.onmessage = function(event) {
13   let message = event.data;
14
15   let messageElem = document.createElement('div');
16   messageElem.textContent = message;
17   document.getElementById('messages').prepend(messageElem);
18 }

```

Server-side code is a little bit beyond our scope. Here we'll use Node.js, but you don't have to. Other platforms also have their means to work with WebSocket.

The server-side algorithm will be:


1. Create `clients = new Set()` – a set of sockets.
2. For each accepted websocket, add it to the set `clients.add(socket)` and setup message event listener to get its messages.
3. When a message received: iterate over clients and send it to everyone.

4. When a connection is closed: `clients.delete(socket)` .

```
1  const ws = new require('ws');
2  const wss = new ws.Server({noServer: true});
3
4  const clients = new Set();
5
6  http.createServer((req, res) => {
7    // here we only handle websocket connections
8    // in real project we'd have some other code here to handle non-websocket r
9    wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
10 });
11
12 function onSocketConnect(ws) {
13   clients.add(ws);
14
15   ws.on('message', function(message) {
16     message = message.slice(0, 50); // max message length will be 50
17
18     for(let client of clients) {
19       client.send(message);
20     }
21   });
22
23   ws.on('close', function() {
24     clients.delete(ws);
25   });
26 }
```

Here's the working example:

Send



You can also download it (upper-right button in the iframe) and run locally. Just don't forget to install [Node.js](#) and `npm install ws` before running.

## Summary

WebSocket is a modern way to have persistent browser-server connections.

- WebSockets don't have cross-origin limitations.
- They are well-supported in browsers.
- Can send/receive strings and binary data.

The API is simple.

Methods:

- `socket.send(data)` ,



- `socket.close([code], [reason])` .

Events:

- `open` ,
- `message` ,
- `error` ,
- `close` .

WebSocket by itself does not include reconnection, authentication and many other high-level mechanisms. So there are client/server libraries for that, and it's also possible to implement these capabilities manually.

Sometimes, to integrate WebSocket into existing project, people run WebSocket server in parallel with the main HTTP-server, and they share a single database. Requests to WebSocket use `wss://ws.site.com` , a subdomain that leads to WebSocket server, while `https://site.com` goes to the main HTTP-server.

Surely, other ways of integration are also possible.



Previous lesson

Next lesson



Share  

 [Tutorial map](#)

## Comments

- If you have suggestions what to improve - please [submit a GitHub issue](#) or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>` , for more than 10 lines – use a sandbox ([plnkr](#), [JSBin](#), [codepen](#)...)