WebSockets

Living Standard — Last Updated 12 May 2023

Participate:

GitHub whatwg/websockets (new issue, open issues)
Chat on Matrix

Commits:

GitHub whatwg/websockets/commits
Snapshot as of this commit
@whatsockets

Tests:

web-platform-tests websockets/ (ongoing work)

Translations **(non-normative)**:

日本語

Abstract

This specification provides APIs to enable web applications to maintain bidirectional communications with server-side processes.

Table of Contents

1. Introduction §

This section is non-normative.

To enable web applications to maintain bidirectional communications with server-side processes, this specification introduces the `WebSocket` interface.

Note

This interface does not allow for raw access to the underlying network. For example, this interface could not be used to implement an IRC client without proxying messages through a custom server.

2. WebSocket protocol alterations §

Note

This section replaces part of the WebSocket protocol opening handshake client requirement to integrate it with algorithms defined in Fetch*. This way CSP, cookies, HSTS, and other* Fetch*-related protocols are handled in a single location. Ideally the RFC would be updated with this language, but it is never that easy. The* `WebSocket` *API, defined below, uses this language. [WSP] [FETCH]*

The way this works is by replacing The WebSocket Protocol's "establish a WebSocket connection" algorithm with a new one that integrates with Fetch*. "Establish a WebSocket connection" consists of three algorithms: setting up a connection, creating and transmiting a handshake request, and validating the handshake response. That layering is different from* Fetch*, which first creates a handshake, then sets up a connection and transmits the handshake, and finally validates the response. Keep that in mind while reading these alterations.*

2.1. Connections §

To **obtain a WebSocket connection**, given a *url*, run these steps:

1. Let *host* be *url*'s host.
2. Let *port* be *url*'s port.
3. Let *resource name* be U+002F (/), followed by the strings in *url*'s path (including empty strings), if any, separated from each other by U+002F (/).
4. If *url*'s query is non-empty, append U+003F (?), followed by *url*'s query, to *resource name*.
5. Let *secure* be false, if *url*'s scheme is "`http`"; otherwise true.
6. Follow the requirements stated in step 2 to 5, inclusive, of the first set of steps in section 4.1 of The WebSocket Protocol to establish a WebSocket connection, passing *host*, *port*, *resource name* and *secure*. [WSP]
7. If that established a connection, return it, and return failure otherwise.

Note

Although structured a little differently, carrying different properties, and therefore not shareable, a WebSocket connection is very close to identical to an "ordinary" connection.

2.2. Opening handshake §

To **establish a WebSocket connection**, given a *url*, *protocols*, and *client*, run these steps:

1. Let *requestURL* be a copy of *url*, with its scheme set to "`http`", if *url*'s scheme is "`ws`"; otherwise to "`https`".

   Note

   This change of scheme is essential to integrate well with fetching. E.g., HSTS would not work without it. There is no real reason for WebSocket to have distinct schemes, it's a legacy artefact. [HSTS]

2. Let *request* be a new request, whose URL is *requestURL*, client is *client*, service-workers mode is "`none`", referrer is "`no-referrer`", mode is "`websocket`", credentials mode is "`include`", cache mode is "`no-store`" , and redirect mode is "`error`".
3. Append (`` `Upgrade` ``, `` `websocket` ``) to *request*'s header list.
4. Append (`` `Connection` ``, `` `Upgrade` ``) to *request*'s header list.

5. Let *keyValue* be a nonce consisting of a randomly selected 16-byte value that has been forgiving-base64-encoded and isomor

✓ MDN

¶ Example

If the randomly selected value was the byte sequence 0x01 0x02 0x03 0x04 0x05 0x06 0x07 0x08 0x09 0x0a 0x0b 0x0c 0x0d 0x0e 0x0f 0x10, *keyValue* would be forgiving-base64-encoded to "`AQIDBAUGBwgJCgsMDQ4PEC==`" and isomorphic encoded to \``AQIDBAUGBwgJCgsMDQ4PEC==`\`.

6. Append (\``Sec-WebSocket-Key`\`, *keyValue*) to *request*'s header list.

7. Append (\``Sec-WebSocket-Version`\`, \``13`\`) to *request*'s header list.

8. For each *protocol* in *protocols*, combine (\``Sec-WebSocket-Protocol`\`, *protocol*) in *request*'s header list.

9. Let *permessageDeflate* be a user-agent defined "`permessage-deflate`" extension header value. [WSP]

¶ Example

\``permessage-deflate; client_max_window_bits`\`

10. Append (\``Sec-WebSocket-Extensions`\`, *permessageDeflate*) to *request*'s header list.

11. Fetch *request* with *useParallelQueue* set to true, and *processResponse* given *response* being these steps:

    1. If *response* is a network error or its status is not 101, fail the WebSocket connection.

    2. If *protocols* is not the empty list and extracting header list values given \``Sec-WebSocket-Protocol`\` and *response*'s header list results in null, failure, or the empty byte sequence, then fail the WebSocket connection.

       Note

       This is different from the check on this header defined by The WebSocket Protocol. That only covers a subprotocol not requested by the client. This covers a subprotocol requested by the client, but not acknowledged by the server.

    3. Follow the requirements stated step 2 to step 6, inclusive, of the last set of steps in section 4.1 of The WebSocket Protocol to validate *response*. This either results in fail the WebSocket connection or the WebSocket connection is established.

Fail the WebSocket connection and the WebSocket connection is established are defined by The WebSocket Protocol. [WSP]

⚠Warning!

The reason redirects are not followed and this handshake is generally restricted is because it could introduce serious security problems in a web browser context. For example, consider a host with a WebSocket server at one path and an open HTTP redirector at another. Suddenly, any script that can be given a particular WebSocket URL can be tricked into communicating to (and potentially sharing secrets with) any host on the internet, even if the script checks that the URL has the right hostname.

3. The WebSocket interface §

3.1. Interface definition §

The Web IDL definition for the WebSocket class is given as follows:

```
IDL
enum BinaryType { "blob", "arraybuffer" };

[Exposed=(Window,Worker)]
interface WebSocket : EventTarget {
  constructor(USVString url, optional (DOMString or sequence<DOMString>) protocols = []);
  readonly attribute USVString url;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long long bufferedAmount;

  // networking
  attribute EventHandler onopen;
  attribute EventHandler onerror;
  attribute EventHandler onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;
  undefined close(optional [Clamp] unsigned short code, optional USVString reason);

  // messaging
  attribute EventHandler onmessage;
  attribute BinaryType binaryType;
  undefined send((BufferSource or Blob or USVString) data);
};
```

Each WebSocket object has an associated **url**, which is a URL record.

Each WebSocket object has an associated **binary type**, which is a BinaryType. Initially it must be "blob".

Each WebSocket object has an associated **ready state**, which is a number representing the state of the connection. Initially it must be
can have the following values:

CONNECTING (numeric value 0)
: The connection has not yet been established.

OPEN (numeric value 1)
: The WebSocket connection is established and communication is possible.

CLOSING (numeric value 2)
: The connection is going through the closing handshake, or the close() method has been invoked.

CLOSED (numeric value 3)
: The connection has been closed or could not be opened.

For web developers (non-normative)

***socket* = new `WebSocket`(*url* [, *protocols*])**

Creates a new `WebSocket` object, immediately establishing the associated WebSocket connection.

url is a string giving the URL over which the connection is established. Only "`ws`", "`wss`", "`http`", and "`https`" schemes are allowe ✓ MDN cause a "`SyntaxError`" `DOMException`. URLs with fragments will always cause such an exception.

protocols is either a string or an array of strings. If it is a string, it is equivalent to an array consisting of just that string; if it is omitted, it is equivalent to the empty array. Each string in the array is a subprotocol name. The connection will only be established if the server reports that it has selected one of these subprotocols. The subprotocol names have to match the requirements for elements that comprise the value of \``Sec-WebSocket-Protocol`\` fields as defined by The WebSocket protocol. [WSP]

***socket*.`send`(*data*)**

Transmits *data* using the WebSocket connection. *data* can be a string, a `Blob`, an `ArrayBuffer`, or an `ArrayBufferView`.

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

Closes the WebSocket connection, optionally using *code* as the WebSocket connection close code and *reason* as the WebSocket connection close reason.

***socket*.`url`**

Returns the URL that was used to establish the WebSocket connection.

***socket*.`readyState`**

Returns the state of the WebSocket connection. It can have the values described above.

***socket*.`bufferedAmount`**

Returns the number of bytes of application data (UTF-8 text and binary data) that have been queued using `send()` but not yet been transmitted to the network.

If the WebSocket connection is closed, this attribute's value will only increase with each call to the `send()` method. (The number does not reset to zero once the connection closes.)

***socket*.`extensions`**

Returns the extensions selected by the server, if any.

***socket*.`protocol`**

Returns the subprotocol selected by the server, if any. It can be used in conjunction with the array form of the constructor's second argument to perform subprotocol negotiation.

***socket*.`binaryType`**

Returns a string that indicates how binary data from *socket* is exposed to scripts:

"`blob`"

Binary data is returned in `Blob` form.

"`arraybuffer`"

Binary data is returned in `ArrayBuffer` form.

The default is "`blob`".

socket*.`binaryType` = *value

Changes how binary data is returned.

The **`new WebSocket(`*`url`*`, `*`protocols`*`)`** constructor steps are:

1. Let *baseURL* be this's relevant settings object's API base URL.
2. Let *urlRecord* be the result of applying the URL parser to *url* with *baseURL*.
3. If *urlRecord* is failure, then throw a "`SyntaxError`" `DOMException`.
4. If *urlRecord*'s scheme is "`http`", then set *urlRecord*'s scheme to "`ws`".
5. Otherwise, if *urlRecord*'s scheme is "`https`", set *urlRecord*'s scheme to "`wss`".
6. If *urlRecord*'s scheme is not "`ws`" or "`wss`", then throw a "`SyntaxError`" `DOMException`.
7. If *urlRecord*'s fragment is non-null, then throw a "`SyntaxError`" `DOMException`.
8. If *protocols* is a string, set *protocols* to a sequence consisting of just that string.

9. If any of the values in *protocols* occur more than once or otherwise fail to match the requirements for elements that comprise the value of `` `Sec-WebSocket-Protocol` `` fields as defined by The WebSocket protocol, then throw a "`SyntaxError`" `DOMException`. [WSP]

10. Set this's url to *urlRecord*.

11. Let *client* be this's relevant settings object.

12. Run this step in parallel:

    1. Establish a WebSocket connection given *urlRecord*, *protocols*, and *client*. [FETCH]

    Note

    If the establish a WebSocket connection algorithm fails, it triggers the fail the WebSocket connection algorithm, which then invokes the close the WebSocket connection algorithm, which then establishes that the WebSocket connection is closed, which fires the `close` event as described below.

The **`url`** getter steps are to return this's url, serialized.

The **`readyState`** getter steps are to return this's ready state.

The **`extensions`** attribute must initially return the empty string. After the WebSocket connection is established, its value might change, as defined below.

The **`protocol`** attribute must initially return the empty string. After the WebSocket connection is established, its value might change, as defined below.

The **`close(code, reason)`** method steps are:

1. If *code* is present, but is neither an integer equal to 1000 nor an integer in the range 3000 to 4999, inclusive, throw an "`InvalidAccessError`" `DOMException`.

2. If *reason* is present, then run these substeps:

    1. Let *reasonBytes* be the result of encoding *reason*.

    2. If *reasonBytes* is longer than 123 bytes, then throw a "`SyntaxError`" `DOMException`.

3. Run the first matching steps from the following list:

    ↪ **If this's ready state is `CLOSING` (2) or `CLOSED` (3)**

    Do nothing.

    Note

    The connection is already closing or is already closed. If it has not already, a `close` event will eventually fire as described below.

    ↪ **If the WebSocket connection is not yet established [WSP]**

    Fail the WebSocket connection and set this's ready state to `CLOSING` (2). [WSP]

    Note

    The fail the WebSocket connection algorithm invokes the close the WebSocket connection algorithm, which then establishes that the WebSocket connection is closed, which fires the `close` event as described below.

    ↪ **If the WebSocket closing handshake has not yet been started [WSP]**

    Start the WebSocket closing handshake and set this's ready state to `CLOSING` (2). [WSP]

    If neither *code* nor *reason* is present, the WebSocket Close message must not have a body.

    Note

    The WebSocket Protocol erroneously states that the status code is required for the start the WebSocket closing handshake algorithm.

    If *code* is present, then the status code to use in the WebSocket Close message must be the integer given by *code*. [WSP]

If *reason* is also present, then *reasonBytes* must be provided in the Close message after the status code. [WSP]

Note

The start the WebSocket closing handshake algorithm eventually invokes the close the WebSocket connection algorithm, which then establishes that the WebSocket connection is closed, which fires the `close` event as described below.

↪ **Otherwise**

Set this's ready state to `CLOSING` (2).

Note

The WebSocket closing handshake is started, and will eventually invoke the close the WebSocket connection algorithm, which will establish that the WebSocket connection is closed, and thus the `close` event will fire, as described below.

Note

The `close()` method does not discard previously sent messages before starting the WebSocket closing handshake — even if, in practice, the user agent is still busy sending those messages, the handshake will only start after the messages are sent.

The **`bufferedAmount`** getter steps are to return the number of bytes of application data (UTF-8 text and binary data) that have been queued using `send()` but that, as of the last time the event loop reached step 1, had not yet been transmitted to the network. (This thus includes any text sent during the execution of the current task, regardless of whether the user agent is able to transmit text in the background in parallel with script execution.) This does not include framing overhead incurred by the protocol, or buffering done by the operating system or network hardware.

Example

In this simple example, the `bufferedAmount` attribute is used to ensure that updates are sent either at the rate of one update every 50ms, if the network can handle that rate, or at whatever rate the network *can* handle, if that is too fast.

```
var socket = new WebSocket('ws://game.example.com:12010/updates');
socket.onopen = function () {
  setInterval(function() {
    if (socket.bufferedAmount == 0)
      socket.send(getUpdateData());
  }, 50);
};
```

The `bufferedAmount` attribute can also be used to saturate the network without sending the data at a higher rate than the network can handle, though this requires more careful monitoring of the value of the attribute over time.

The **`binaryType`** getter steps are to return this's binary type.

The `binaryType` setter steps are to set this's binary type to the given value.

Note

User agents can use the binary type as a hint for how to handle incoming binary data: if it is "`blob`", it is safe to spool it to disk, and if it is "`arraybuffer`", it is likely more efficient to keep the data in memory. Naturally, user agents are encouraged to use more subtle heuristics to decide whether to keep incoming data in memory or not, e.g. based on how big the data is or how common it is for a script to change the attribute at the last minute. This latter aspect is important in particular because it is quite possible for the attribute to be changed after the user agent has received the data but before the user agent has fired the event for it.

The **`send(*data*)`** method steps are:

1. If this's ready state is `CONNECTING`, then throw an "`InvalidStateError`" `DOMException`.
2. Run the appropriate set of steps from the following list:

If *data* is a string

If the WebSocket connection is established and the WebSocket closing handshake has not yet started, then the user agent must send a WebSocket Message comprised of the *data* argument using a text frame opcode; if the data cannot be sent, e.g. because it would need to be buffered but the buffer is full, the user agent must flag the WebSocket as full and then close the WebSocket connection. Any invocation of this method with a string argument that does not throw an exception must increase the `bufferedAmount` attribute by the number of bytes needed to express the argument as UTF-8. [UNICODE] [ENCODING] [WSP]

If *data* is a `Blob` object

If the WebSocket connection is established, and the WebSocket closing handshake has not yet started, then the user agent must send a WebSocket Message comprised of *data* using a binary frame opcode; if the data cannot be sent, e.g. because it would need to be buffered but the buffer is full, the user agent must flag the WebSocket as full and then close the WebSocket connection. The data to be sent is the raw data represented by the `Blob` object. Any invocation of this method with a `Blob` argument that does not throw an exception must increase the `bufferedAmount` attribute by the size of the `Blob` object's raw data, in bytes. [WSP] [FILEAPI]

If *data* is an `ArrayBuffer`

If the WebSocket connection is established, and the WebSocket closing handshake has not yet started, then the user agent must send a WebSocket Message comprised of *data* using a binary frame opcode; if the data cannot be sent, e.g. because it would need to be buffered but the buffer is full, the user agent must flag the WebSocket as full and then close the WebSocket connection. The data to be sent is the data stored in the buffer described by the `ArrayBuffer` object. Any invocation of this method with an `ArrayBuffer` argument that does not throw an exception must increase the `bufferedAmount` attribute by the length of the `ArrayBuffer` in bytes. [WSP]

If *data* is an `ArrayBufferView`

If the WebSocket connection is established, and the WebSocket closing handshake has not yet started, then the user agent must send a WebSocket Message comprised of *data* using a binary frame opcode; if the data cannot be sent, e.g. because it would need to be buffered but the buffer is full, the user agent must flag the WebSocket as full and then close the WebSocket connection. The data to be sent is the data stored in the section of the buffer described by the `ArrayBuffer` object that *data* references. Any invocation of this method with this kind of argument that does not throw an exception must increase the `bufferedAmount` attribute by the length of *data*'s buffer in bytes. [WSP]

The following are the event handlers (and their corresponding event handler event types) that must be supported, as event handler IDL attributes, by all objects implementing the `WebSocket` interface:

Event handler	Event handler event type
`onopen`	`open`
`onmessage`	`message`
`onerror`	`error`
`onclose`	`close`

4. Feedback from the protocol §

When the WebSocket connection is established, the user agent must queue a task to run these steps:

1. Change the ready state to `OPEN` (1).
2. Change the `extensions` attribute's value to the extensions in use, if it is not the null value. [WSP]
3. Change the `protocol` attribute's value to the subprotocol in use, if it is not the null value. [WSP]
4. Fire an event named **`open`** at the `WebSocket` object.

Note

Since the algorithm above is queued as a task, there is no race condition between the WebSocket connection being established and the script setting up an event listener for the `open` event.

When a WebSocket message has been received with type *type* and data *data*, the user agent must queue a task to follow these steps: [WSP]

1. If ready state is not `OPEN` (1), then return.
2. Let *dataForEvent* be determined by switching on *type* and binary type:

   ↪ ***type* indicates that the data is Text**

   a new `DOMString` containing *data*

   ↪ ***type* indicates that the data is Binary and binary type is `"blob"`**

   a new `Blob` object, created in the relevant Realm of the `WebSocket` object, that represents *data* as its raw data [FILEA

   ↪ ***type* indicates that the data is Binary and binary type is `"arraybuffer"`**

   a new `ArrayBuffer` object, created in the relevant Realm of the `WebSocket` object, whose contents are *data*

3. Fire an event named **`message`** at the `WebSocket` object, using `MessageEvent`, with the `origin` attribute initialized to the serialization of the `WebSocket` object's url's origin, and the `data` attribute initialized to *dataForEvent*.

✓ MDN

✓ MDN

✓ MDN

✓ MDN

Note

User agents are encouraged to check if they can perform the above steps efficiently before they run the task, picking tasks from other task queues while they prepare the buffers if not. For example, if the binary type is "`blob`" when the data arrived, and the user agent spooled all the data to disk, but just before running the above task for this particular message the script switched binary type to "`arraybuffer`", the user agent would want to page the data back to RAM before running this task so as to avoid stalling the main thread while it created the `ArrayBuffer` object.

Example

Here is an example of how to define a handler for the `message` event in the case of text frames:

```
mysocket.onmessage = function (event) {
  if (event.data == 'on') {
    turnLampOn();
  } else if (event.data == 'off') {
    turnLampOff();
  }
};
```

The protocol here is a trivial one, with the server just sending "on" or "off" messages.

When the WebSocket closing handshake is started, the user agent must queue a task to change the ready state to `CLOSING` (2). (If the `close()` method was called, the ready state will already be set to `CLOSING` (2) when this task runs.) [WSP]

When the WebSocket connection is closed, possibly cleanly, the user agent must queue a task to run the following substeps:

1. Change the ready state to `CLOSED` (3).
2. If the user agent was required to fail the WebSocket connection, or if the WebSocket connection was closed after being **flagged as full**, fire an event named **`error`** at the `WebSocket` object. [WSP]
3. Fire an event named **`close`** at the `WebSocket` object, using `CloseEvent`, with the `wasClean` attribute initialized to true if the connection closed cleanly and false otherwise, the `code` attribute initialized to the WebSocket connection close code, and the `reason` attribute initialized to the result of applying UTF-8 decode without BOM to the WebSocket connection close reason. [WSP]

⚠Warning!

User agents must not convey any failure information to scripts in a way that would allow a script to distinguish the following situations:

- ***A server whose host name could not be resolved.***
- ***A server to which packets could not successfully be routed.***
- ***A server that refused the connection on the specified port.***
- ***A server that failed to correctly perform a TLS handshake (e.g., the server certificate can't be verified).***
- ***A server that did not complete the opening handshake (e.g. because it was not a WebSocket server).***
- ***A WebSocket server that sent a correct opening handshake, but that specified options that caused the client to drop the connection (e.g. the server specified a subprotocol that the client did not offer).***
- ***A WebSocket server that abruptly closed the connection after successfully completing the opening handshake.***

In all of these cases, the WebSocket connection close code would be 1006, as required by **WebSocket Protocol**. ***[WSP]***

Allowing a script to distinguish these cases would allow a script to probe the user's local network in preparation for an attack.

Note

In particular, this means the code 1015 is not used by the user agent (unless the server erroneously uses it in its close frame, of course).

The task source for all tasks queued in this section is the **WebSocket task source**.

5. Ping and Pong frames §

The WebSocket protocol defines Ping and Pong frames that can be used for keep-alive, heart-beats, network status probing, latency instrumentation, and so forth. These are not currently exposed in the API.

User agents may send ping and unsolicited pong frames as desired, for example in an attempt to maintain local network NAT mappings, to detect failed connections, or to display latency metrics to the user. User agents must not use pings or unsolicited pongs to aid the server; it is assumed that servers will solicit pongs whenever appropriate for the server’s needs.

6. The `CloseEvent` interface §

`WebSocket` objects use the `CloseEvent` interface for their `close` events:

```
IDL
[Exposed=(Window,Worker)]
interface CloseEvent : Event {
  constructor(DOMString type, optional CloseEventInit eventInitDict = {});

  readonly attribute boolean wasClean;
  readonly attribute unsigned short code;
  readonly attribute USVString reason;
};

dictionary CloseEventInit : EventInit {
  boolean wasClean = false;
  unsigned short code = 0;
  USVString reason = "";
};
```

For web developers (non-normative)

event **.** **`wasClean`**

Returns true if the connection closed cleanly; false otherwise.

event **.** **`code`**

Returns the WebSocket connection close code provided by the server.

event **.** **`reason`**

Returns the WebSocket connection close reason provided by the server.

The **`wasClean`** attribute must return the value it was initialized to. It represents whether the connection closed cleanly or not.

The **`code`** attribute must return the value it was initialized to. It represents the WebSocket connection close code provided by the server.

The **`reason`** attribute must return the value it was initialized to. It represents the WebSocket connection close reason provided by the server.

✓ MDN

✓ MDN

7. Garbage collection §

A WebSocket object whose ready state was set to CONNECTING (0) as of the last time the event loop reached step 1 must not be garbage collected if there are any event listeners registered for open events, message events, error events, or close events.

A WebSocket object whose ready state was set to OPEN (1) as of the last time the event loop reached step 1 must not be garbage collected if there are any event listeners registered for message events, error, or close events.

A WebSocket object whose ready state was set to CLOSING (2) as of the last time the event loop reached step 1 must not be garbage c

✓ MDN

any event listeners registered for error or close events.

A WebSocket object with an established connection that has data queued to be transmitted to the network must not be garbage collected. [WSP]

If a WebSocket object is garbage collected while its connection is still open, the user agent must start the WebSocket closing handshak

✓ MDN

, with no status code for the Close message. [WSP]

✓ MDN

If a user agent is to **make disappear** a WebSocket object (this happens when a Document object goes away), the user agent must follow the first appropriate set of steps from the following list:

↪ **If the WebSocket connection is not yet established [WSP]**

Fail the WebSocket connection. [WSP]

↪ **If the WebSocket closing handshake has not yet been started [WSP]**

Start the WebSocket closing handshake, with the status code to use in the WebSocket Close message being 1001. [WSP]

↪ **Otherwise**

Do nothing.

Acknowledgments §

Until the creation of this standard in 2021, the text here was maintained in the HTML Standard and Fetch Standard. Thanks to all of the contributors to those repositories who helped develop the specification, especially Ian Hickson and Anne van Kesteren as the respective original authors.

Thanks to devsnek and 平野裕 (Yutaka Hirano) for their contributions after the creation of the WebSockets Standard.

This standard is written by Adam Rice (Google, ricea@chromium.org).

Intellectual property rights §

Index §

Terms defined by this specification §

Terms defined by reference §

- [DOM] defines the following terms:
    - Document
    - Event
    - EventInit
    - EventTarget
    - fire an event
- [ENCODING] defines the following terms:
    - utf-8 decode without bom
    - utf-8 encode
- [FETCH] defines the following terms:
    - append
    - cache mode
    - client
    - combine
    - connection
    - credentials mode
    - extracting header list values
    - fetch
    - header list
    - header value
    - mode
    - network error
    - processresponse
    - redirect mode
    - referrer
    - request
    - service-workers mode
    - status
    - url
    - useparallelqueue
- [FILEAPI] defines the following terms:
    - Blob
- [HTML] defines the following terms:
    - EventHandler
    - MessageEvent
    - api base url
    - data
    - event handler
    - event handler event type
    - event handler idl attribute
    - event loop
    - in parallel
    - origin
    - queue a task
    - relevant realm
    - relevant settings object
    - step 1
    - task
    - task queues
    - task source
- [INFRA] defines the following terms:
    - forgiving-base64 encode
    - isomorphic encode
- [URL] defines the following terms:
    - fragment
    - host
    - origin
    - path
    - port
    - query
    - scheme

  - url
  - url parser
  - url record
  - url serializer
- [WEBIDL] defines the following terms:
  - ArrayBuffer
  - ArrayBufferView
  - BufferSource
  - Clamp
  - DOMException
  - DOMString
  - Exposed
  - InvalidAccessError
  - InvalidStateError
  - SyntaxError
  - USVString
  - boolean
  - sequence
  - the given value
  - this
  - undefined
  - unsigned long long
  - unsigned short
- [WSP] defines the following terms:
  - a websocket message has been received
  - cleanly
  - close the websocket connection
  - established
  - extensions in use
  - fail the websocket connection
  - sec-websocket-protocol
  - send a websocket message
  - start the websocket closing handshake
  - subprotocol in use
  - the websocket closing handshake is started
  - the websocket connection close code
  - the websocket connection close reason
  - the websocket connection is closed
  - the websocket connection is established
  - ws
  - wss

References §

Normative References §

[DOM]

Anne van Kesteren. *DOM Standard*. Living Standard. URL: https://dom.spec.whatwg.org/

[ENCODING]

Anne van Kesteren. *Encoding Standard*. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]

Anne van Kesteren. *Fetch Standard*. Living Standard. URL: https://fetch.spec.whatwg.org/

[FILEAPI]

Marijn Kruisselbrink. *File API*. URL: https://w3c.github.io/FileAPI/

[HSTS]

J. Hodges; C. Jackson; A. Barth. *HTTP Strict Transport Security (HSTS)*. November 2012. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6797

[HTML]

Anne van Kesteren; et al. *HTML Standard*. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]

Anne van Kesteren; Domenic Denicola. *Infra Standard*. Living Standard. URL: https://infra.spec.whatwg.org/

[UNICODE]

The Unicode Standard. URL: https://www.unicode.org/versions/latest/

[URL]

Anne van Kesteren. *URL Standard*. Living Standard. URL: https://url.spec.whatwg.org/

[WEBIDL]

Edgar Chen; Timothy Gu. *Web IDL Standard*. Living Standard. URL: https://webidl.spec.whatwg.org/

[WSP]

I. Fette; A. Melnikov. *The WebSocket Protocol*. December 2011. Proposed Standard. URL: https://www.rfc-editor.org/rfc/rfc6455

IDL Index §

```
IDL
enum BinaryType { "blob", "arraybuffer" };

[Exposed=(Window,Worker)]
interface WebSocket : EventTarget {
  constructor(USVString url, optional (DOMString or sequence<DOMString>) protocols = []);
  readonly attribute USVString url;

  // ready state
  const unsigned short CONNECTING = 0;
  const unsigned short OPEN = 1;
  const unsigned short CLOSING = 2;
  const unsigned short CLOSED = 3;
  readonly attribute unsigned short readyState;
  readonly attribute unsigned long long bufferedAmount;

  // networking
  attribute EventHandler onopen;
  attribute EventHandler onerror;
  attribute EventHandler onclose;
  readonly attribute DOMString extensions;
  readonly attribute DOMString protocol;
  undefined close(optional [Clamp] unsigned short code, optional USVString reason);

  // messaging
  attribute EventHandler onmessage;
  attribute BinaryType binaryType;
  undefined send((BufferSource or Blob or USVString) data);
};

[Exposed=(Window,Worker)]
interface CloseEvent : Event {
  constructor(DOMString type, optional CloseEventInit eventInitDict = {});

  readonly attribute boolean wasClean;
  readonly attribute unsigned short code;
  readonly attribute USVString reason;
};

dictionary CloseEventInit : EventInit {
  boolean wasClean = false;
  unsigned short code = 0;
  USVString reason = "";
};
```