



The WebSocket API

Editor's Draft 23 April 2013

Latest Published Version:

<http://www.w3.org/TR/websockets/>

Latest Editor's Draft:

<http://dev.w3.org/html5/websockets/>

Previous Versions:

<http://www.w3.org/TR/2009/WD-websockets-20090423/>

<http://www.w3.org/TR/2009/WD-websockets-20091029/>

Editor:

[Ian Hickson](#), Google, Inc.

Copyright © 2012 W3C® ([MIT](#), [ERCIM](#), [Keio](#)), All Rights Reserved. W3C [liability](#), [trademark](#) and [document use](#) rules apply.
The bulk of the text of this specification is also available in the WHATWG [Web Applications 1.0](#) specification, under a license that permits reuse of the specification text.

Abstract

This specification defines an API that enables Web pages to use the WebSocket protocol (defined by the IETF) for two-way communication with a remote host.

Status of This document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the most recently formally published revision of this technical report can be found in the [W3C technical reports index](http://www.w3.org/TR/) at <http://www.w3.org/TR/>.

If you wish to make comments regarding this document, you can enter feedback using this form:

Feedback Comments

Please enter your feedback, carefully indicating the title of the section for which you are submitting feedback, quoting the text that's wrong today if appropriate. If you're suggesting a new feature, it's really important to say *what* the problem you're trying to solve is. That's more important than the solution, in fact.

Note: Please don't use section numbers as these tend to change rapidly and make your feedback harder to understand.

(Note: Your IP address and user agent will be publicly recorded for spam prevention purposes.)

You can also e-mail feedback to public-webapps@w3.org ([subscribe](#), [archives](#)), or whatwg@whatwg.org ([subscribe](#), [archives](#)). All feedback is welcome.

Implementors should be aware that this specification is not stable. **Implementors who are not taking part in the discussions are likely to find the specification changing out from under them in incompatible ways.** Vendors interested in implementing this specification before it eventually reaches the Candidate Recommendation stage should join the aforementioned mailing lists and take part in the discussions.

The latest stable version of the editor's draft of this specification is always available on [the W3C CVS server](#) and in the [WHATWG Subversion repository](#). The [latest editor's working copy](#) (which may contain unfinished text in the process of being prepared) contains the latest draft text of this specification (amongst others). For more details, please see the [WHATWG FAQ](#).

Notifications of changes to this specification are sent along with notifications of changes to related specifications using the following mechanisms:

E-mail notifications of changes

Commit-Watchers mailing list (complete source diffs):

<http://lists.whatwg.org/listinfo.cgi/commit-watchers-whatwg.org>

Browsable version-control record of all changes:

CVSWeb interface with side-by-side diffs: <http://dev.w3.org/cvsweb/html5/>

Annotated summary with unified diffs: <http://html5.org/tools/web-apps-tracker>

Raw Subversion interface: `svn checkout http://svn.whatwg.org/webapps/`

The W3C [Web Applications Working Group](#) is the W3C working group responsible for this specification's progress along the W3C Recommendation track. This specification is the 23 April 2013 Editor's Draft.

This specification is being developed in conjunction with an RFC for a wire protocol, the WebSocket Protocol, available from the following location:

- RFC 6455: The WebSocket Protocol: <http://tools.ietf.org/html/rfc6455>

This document was produced by a group operating under the [5 February 2004 W3C Patent Policy](#). W3C maintains a [public list of any patent disclosures](#) made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains [Essential Claim\(s\)](#) must disclose the information in accordance with [section 6 of the W3C Patent Policy](#).

This document was produced by a group operating under the [5 February 2004 W3C Patent Policy](#). W3C maintains a [public list of any patent disclosures](#) made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains [Essential Claim\(s\)](#) must disclose the information in accordance with [section 6 of the W3C Patent Policy](#).

Table of Contents

[1 Introduction](#)

[2 Conformance requirements](#)

[2.1 Dependencies](#)

[3 Terminology](#)

[4 The `WebSocket` interface](#)

[5 Feedback from the protocol](#)

[6 Ping and Pong frames](#)

[7 Parsing `WebSocket` URLs](#)

[8 Event definitions](#)

[9 Garbage collection](#)

[References](#)

[Acknowledgements](#)

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 Conformance requirements

All diagrams, examples, and notes in this specification are non-normative, as are all sections explicitly marked non-normative. Everything else in this specification is normative.

The key words "MUST", "MUST NOT", "REQUIRED", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in the normative parts of this document are to be interpreted as described in RFC2119. For readability, these words do not appear in all uppercase letters in this specification. [\[RFC2119\]](#)

Requirements phrased in the imperative as part of algorithms (such as "strip any leading space characters" or "return false and abort these steps") are to be interpreted with the meaning of the key word ("must", "should", "may", etc) used in introducing the algorithm.

Some conformance requirements are phrased as requirements on attributes, methods or objects. Such requirements are to be interpreted as requirements on user agents.

Conformance requirements phrased as algorithms or specific steps may be implemented in any manner, so long as the end result is equivalent. (In particular, the algorithms defined in this specification are intended to be easy to follow, and not intended to be performant.)

The only conformance class defined by this specification is user agents.

User agents may impose implementation-specific limits on otherwise unconstrained inputs, e.g. to prevent denial of service attacks, to guard against running out of memory, or to work around platform-specific limitations.

When support for a feature is disabled (e.g. as an emergency measure to mitigate a security problem, or to aid in development, or for performance reasons), user agents must act as if they had no support for the feature whatsoever, and as if the feature was not mentioned in this specification. For example, if a particular feature is accessed via an attribute in a Web IDL interface, the attribute itself would be omitted from the objects that implement that interface — leaving the attribute on the object but making it return null or throw an exception is insufficient.

2.1 Dependencies

This specification relies on several other underlying specifications.

HTML

Many fundamental concepts from HTML are used by this specification. [\[HTML\]](#)

WebIDL

The IDL blocks in this specification use the semantics of the WebIDL specification. [\[WEBIDL\]](#)

3 Terminology

The construction "a `Foo` object", where `Foo` is actually an interface, is sometimes used instead of the more accurate "an object implementing the interface `Foo`".

The term DOM is used to refer to the API set made available to scripts in Web applications, and does not necessarily imply the existence of an actual `Document` object or of any other `Node` objects as defined in the DOM specifications. [\[DOM\]](#)

An IDL attribute is said to be *getting* when its value is being retrieved (e.g. by author script), and is said to be *setting* when a new value is assigned to it.

4 The [WebSocket](#) interface

IDL

```
enum BinaryType { "blob", "arraybuffer" };
[Constructor(DOMString url, optional (DOMString or DOMString[])
protocols)]
interface WebSocket : EventTarget {
    readonly attribute DOMString 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 bufferedAmount;

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

    // messaging
        attribute EventHandler onmessage;
        attribute BinaryType binaryType;
    void send(DOMString data);
    void send(Blob data);
    void send(ArrayBuffer data);
    void send(ArrayBufferView data);
};
```

The **WebSocket**(*url*, *protocols*) constructor takes one or two arguments. The first argument, *url*, specifies the URL to which to connect. The second, *protocols*, if present, 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 must all be strings that match the requirements for elements that comprise the value of `Sec-WebSocket-Protocol` header fields as defined by the WebSocket protocol specification. [\[WSP\]](#)

When the `WebSocket()` constructor is invoked, the UA must run these steps:

1. [Parse a WebSocket URL's components](#) from the *url* argument, to obtain *host*, *port*, *resource name*, and *secure*. If this fails, throw a `SyntaxError` exception and abort these steps. [\[WSP\]](#)
2. If *secure* is false but the origin of the entry script has a scheme component that is itself a secure protocol, e.g. HTTPS, then throw a `SecurityError` exception and abort these steps.
3. If *port* is a port to which the user agent is configured to block access, then throw a `SecurityError` exception and abort these steps. (User agents typically block access to well-known ports like SMTP.)

Access to ports 80 and 443 should not be blocked, including the unlikely cases when *secure* is false but *port* is 443 or *secure* is true but *port* is 80.

4. If *protocols* is absent, let *protocols* be an empty array.

Otherwise, if *protocols* is present and a string, let *protocols* instead be an array consisting of just that string.

5. 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` header fields as defined by the WebSocket protocol specification, then throw a `SyntaxError` exception and abort these steps. [\[WSP\]](#)
6. Let *origin* be the ASCII serialization of the origin of the entry script, converted to ASCII lowercase.
7. Return a new [WebSocket](#) object, but continue these steps asynchronously.
8. *Establish a WebSocket connection* given the set (*host*, *port*, *resource name*, *secure*), along with the *protocols* list, an empty list for the extensions, and *origin*. The *headers to send appropriate cookies* must be a `Cookie` header whose value is the *cookie-string* computed from the user's cookie store and the URL *url*; for these purposes this is *not* a "non-HTTP" API. [\[WSP\]](#) [\[COOKIES\]](#)

When the user agent *validates the server's response* during the "establish a WebSocket connection" algorithm, if the status code received from the server is not 101 (e.g. it is a redirect), the user agent must *fail the WebSocket connection*.

⚠Warning! Following HTTP procedures here 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.

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` attribute must return the result of resolving the URL that was passed to the constructor. (It doesn't matter what it is resolved relative to, since we already know it is an absolute URL.)

The `readyState` attribute represents the state of the connection. It 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.

When the object is created its `readyState` must be set to [CONNECTING](#) (0).

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

Note: The `extensions` attribute returns the extensions selected by the server, if any. (Currently this will only ever be the empty string.)

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

Note: The `protocol` attribute 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.

The `close()` method must run the following steps:

1. If the method's first argument is present but is neither an integer equal to 1000 nor an integer in the range 3000 to 4999, throw an `InvalidAccessError` exception and abort these steps.
2. If the method's second argument is present, then run these substeps:
 1. Let *raw reason* be the method's second argument.
 2. Let *Unicode reason* be the result of converting *raw reason* to a sequence of Unicode characters.
 3. Let *reason* be the result of encoding *Unicode reason* as UTF-8.
 4. If *reason* is longer than 123 bytes, then throw a `SyntaxError` exception and abort these steps. [\[RFC3629\]](#)
3. Run the first matching steps from the following list:

↪ If the `readyState` attribute is in the `CLOSING` (2) or `CLOSED` (3) state
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 the `readyState` attribute's value 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 the `readyState` attribute's value to `CLOSING` (2). [\[WSP\]](#)

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

If the second argument is also present, then *reason* must be provided in the **Close** message after the status code. [\[RFC3629\]](#) [\[WSP\]](#)

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 the `readyState` attribute's value 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.

The `bufferedAmount` attribute must 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 started executing a task, 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 asynchronously with script execution.) This does not include framing overhead incurred by the protocol, or buffering done by the operating system or network hardware. If the 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).

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.

When a `WebSocket` object is created, its `binaryType` IDL attribute must be set to the string `"blob"`. On getting, it must return the last value it was set to. On setting, the user agent must set the IDL attribute to the new value.

Note: This attribute allows authors to control how binary data is exposed to scripts. By setting the attribute to `"blob"`, binary data is returned in `Blob` form; by setting it to `"arraybuffer"`, it is returned in `ArrayBuffer` form. User agents can use this as a hint for how to handle incoming binary data: if the attribute is set to `"blob"`, it is safe to spool it to disk, and if it is set to `"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 transmits data using the connection. If the `readyState` attribute is `CONNECTING`, it must throw an `InvalidStateError` exception. Otherwise, the user agent must run the appropriate set of steps from the following list:

If the argument is a string

Let *data* be the result of converting the *data* argument to a sequence of Unicode characters. 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 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 *close the WebSocket connection with prejudice*. 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. [\[UNICODE\]](#) [\[RFC3629\]](#) [\[WSP\]](#)

If the argument 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 *close the WebSocket connection with prejudice*. 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 the argument is an ArrayBuffer 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 *close the WebSocket connection with prejudice*. 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. [\[WSP\]](#) [\[TYPEDARRAY\]](#)

If the argument is an ArrayBufferView 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 *close the WebSocket connection with prejudice*. The data to be sent is the data stored in the section of the buffer described by the `ArrayBuffer` object that the `ArrayBufferView` object references. Any invocation of this method with an `ArrayBufferView` argument that does not throw an exception must increase the `bufferedAmount` attribute by the length of the `ArrayBufferView` in bytes. [\[WSP\]](#) [\[TYPEDARRAY\]](#)

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

Event handler	Event handler event type
<code>onopen</code>	<code>open</code>
<code>onmessage</code>	<code>message</code>
<code>onerror</code>	<code>error</code>

Event handler	Event handler event type
<code>onclose</code>	<code>close</code>

5 Feedback from the protocol

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

1. Change the [readyState](#) attribute's value to [OPEN](#) (1).
2. Change the [extensions](#) attribute's value to the *extensions in use*, if is not the null value. [\[WSP\]](#)
3. Change the [protocol](#) attribute's value to the *subprotocol in use*, if is not the null value. [\[WSP\]](#)
4. Act as if the user agent had received a set-cookie-string consisting of the *cookies set during the server's opening handshake*, for the URL *url* given to the [WebSocket\(\)](#) constructor. [\[COOKIES\]](#) [\[RFC3629\]](#) [\[WSP\]](#)
5. Fire a simple event named `open` at the [WebSocket](#) object.

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 the [readyState](#) attribute's value is not [OPEN](#) (1), then abort these steps.
2. Let *event* be a newly created trusted event that uses the `MessageEvent` interface, with the event type `message`, which does not bubble, is not cancelable, and has no default action. [\[HTML\]](#)
3. Initialize *event*'s `origin` attribute to the Unicode serialization of the origin of the URL that was passed to the [WebSocket](#) object's constructor.
4. If *type* indicates that the data is Text, then initialize *event*'s `data` attribute to *data*.

If *type* indicates that the data is Binary, and [binaryType](#) is set to `"blob"`, then initialize *event*'s `data` attribute to a new `Blob` object that represents *data* as its raw data. [\[FILEAPI\]](#)

If *type* indicates that the data is Binary, and [binaryType](#) is set to `"arraybuffer"`, then initialize *event*'s `data` attribute to a new read-only `ArrayBuffer` object whose contents are *data*. [\[TYPEDARRAY\]](#)

5. Dispatch *event* at the [WebSocket](#) object.

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 [binaryType](#) attribute was set to `"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 [binaryType](#) 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.

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 `readyState` attribute's value to `CLOSING` (2). (If the `close()` method was called, the `readyState` attribute's value 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 `readyState` attribute's value to `CLOSED` (3).
2. If the user agent was required to *fail the WebSocket connection* or *the WebSocket connection is closed with prejudice*, fire a simple event named `error` at the `WebSocket` object. [\[WSP\]](#)
3. Create a trusted event that uses the `CloseEvent` interface, with the event type `close`, which does not bubble, is not cancelable, has no default action, whose `wasClean` attribute is initialized to true if the connection closed *cleanly* and false otherwise, whose `code` attribute is initialized to *the WebSocket connection close code*, and whose `reason` attribute is initialized to the result of applying the UTF-8 decoder to *the WebSocket connection close reason*, and dispatch the event at the `WebSocket` object. [\[WSP\]](#)

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 the WebSocket connection close code would be 1006, as required by the WebSocket Protocol specification. [\[WSP\]](#)

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**.

6 Ping and Pong frames

The WebSocket protocol specification 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.

7 Parsing WebSocket URLs

The steps to **parse a WebSocket URL's components** from a string *url* are as follows. These steps return either a *host*, a *port*, a *resource name*, and a *secure* flag, or they fail.

1. If the *url* string is not an absolute URL, then fail this algorithm.
2. Resolve the *url* string, with the URL character encoding set to UTF-8. [\[RFC3629\]](#)

Note: It doesn't matter what it is resolved relative to, since we already know it is an absolute URL at this point.

3. If the resulting parsed URL does not have a scheme component whose value is either "ws" or "wss", then fail this algorithm.
4. If the resulting parsed URL has a non-null fragment component, then fail this algorithm.
5. If the scheme component of the resulting parsed URL is "ws", set *secure* to false; otherwise, the scheme component is "wss", set *secure* to true.
6. Let *host* be the value of the resulting parsed URL's host component.
7. If the resulting parsed URL has a port component that is not the empty string, then let *port* be that component's value; otherwise, there is no explicit *port*.
8. If there is no explicit *port*, then: if *secure* is false, let *port* be 80, otherwise let *port* be 443.
9. Let *resource name* be the value of the resulting parsed URL's path component (which might be empty).
10. If *resource name* is the empty string, set it to a single character U+002F SOLIDUS (/).
11. If the resulting parsed URL has a non-null query component, then append a single U+003F QUESTION MARK character (?) to *resource name*, followed by the value of the query component.
12. Return *host*, *port*, *resource name*, and *secure*.

8 Event definitions

IDL

```
[Constructor(DOMString type, optional CloseEventInit
eventInitDict)]
interface CloseEvent : Event {
    readonly attribute boolean wasClean;
    readonly attribute unsigned short code;
    readonly attribute DOMString reason;
};

dictionary CloseEventInit : EventInit {
    boolean wasClean;
    unsigned short code;
    DOMString reason;
};
```

The **wasClean** attribute must return the value it was initialized to. When the object is created, this attribute must be initialized to false. It represents whether the connection closed cleanly or not.

The **code** attribute must return the value it was initialized to. When the object is created, this attribute must be initialized to zero. It represents the WebSocket connection close code provided by the server.

The **reason** attribute must return the value it was initialized to. When the object is created, this attribute must be initialized to empty string. It represents the WebSocket connection close reason provided by the server.

9 Garbage collection

A [WebSocket](#) object whose [readyState](#) attribute's value was set to [CONNECTING](#) (0) as of the last time the event loop started executing a task 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 [readyState](#) attribute's value was set to [OPEN](#) (1) as of the last time the event loop started executing a task must not be garbage collected if there are any event listeners registered for `message` events, `error`, or `close` events.

A [WebSocket](#) object whose [readyState](#) attribute's value was set to [CLOSING](#) (2) as of the last time the event loop started executing a task must not be garbage collected if there are 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 handshake*, with no status code for the Close message. [\[WSP\]](#)

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.

References

All references are normative unless marked "Non-normative".

[COOKIES]

[*HTTP State Management Mechanism*](#), A. Barth. IETF.

[DOM]

[*DOM*](#), A. van Kesteren, A. Gregor, Ms2ger. WHATWG.

[FILEAPI]

[*File API*](#), A. Ranganathan. W3C.

[HTML]

[*HTML*](#), I. Hickson. WHATWG.

[RFC2119]

[*Key words for use in RFCs to Indicate Requirement Levels*](#), S. Bradner. IETF.

[RFC3629]

[*UTF-8, a transformation format of ISO 10646*](#), F. Yergeau. IETF.

[TYPEDARRAY]

[*Typed Array Specification*](#), D. Herman, K. Russell. Khronos.

[UNICODE]

[*The Unicode Standard*](#). Unicode Consortium.

[WEBIDL]

[*Web IDL*](#), C. McCormack. W3C.

[WSP]

[*The WebSocket protocol*](#), I. Fette, A. Melnikov. IETF.

Acknowledgements

For a full list of acknowledgements, please see the HTML specification. [\[HTML\]](#)