# HTML

## Living Standard — Last Updated 8 September 2023

← 9 Communication — Table of Contents — 9.3 Cross-document messaging →

## 9.2 Server-sent events §

### 9.2.1 Introduction §

*This section is non-normative.*

To enable servers to push data to web pages over HTTP or using dedicated server-push protocols, this specification introduces the `EventSource` interface.

Using this API consists of creating an `EventSource` object and registering an event listener.

```
var source = new EventSource('updates.cgi');
source.onmessage = function (event) {
  alert(event.data);
};
```

On the server-side, the script ("`updates.cgi`" in this case) sends messages in the following form, with the `text/event-stream` MIME type:

```
data: This is the first message.

data: This is the second message, it
data: has two lines.

data: This is the third message.
```

Authors can separate events by using different event types. Here is a stream that has two event types, "add" and "remove":

```
event: add
data: 73857293

event: remove
data: 2153

event: add
data: 113411
```

The script to handle such a stream would look like this (where `addHandler` and `removeHandler` are functions that take one argument, the event):

```
var source = new EventSource('updates.cgi');
source.addEventListener('add', addHandler, false);
```

File an issue about the selected text

```
    source.addEventListener('remove', removeHandler, false);
```

The default event type is "message".

Event streams are always decoded as UTF-8. There is no way to specify another character encoding.

Event stream requests can be redirected using HTTP 301 and 307 redirects as with normal HTTP requests. Clients will reconnect if the connection is closed; a client can be told to stop reconnecting using the HTTP 204 No Content response code.

Using this API rather than emulating it using `XMLHttpRequest` or an `iframe` allows the user agent to make better use of network resources in cases where the user agent implementer and the network operator are able to coordinate in advance. Amongst other benefits, this can result in significant savings in battery life on portable devices. This is discussed further in the section below on connectionless push.

### 9.2.2 The `EventSource` interface §

IDL
```
[Exposed=(Window,Worker)]
interface EventSource : EventTarget {
  constructor(USVString url, optional EventSourceInit eventSourceInitDict = {});

  readonly attribute USVString url;
  readonly attribute boolean withCredentials;

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

  // networking
  attribute EventHandler onopen;
  attribute EventHandler onmessage;
  attribute EventHandler onerror;
  undefined close();
};

dictionary EventSourceInit {
  boolean withCredentials = false;
};
```

Each `EventSource` object has the following associated with it:

- A **url** (a URL record). Set during construction.
- A **request**. This must initially be null.
- A **reconnection time**, in milliseconds. This must initially be an implementation-defined value, probably in the region of a few seconds.
- A **last event ID string**. This must initially be the empty string.

Apart from url these are not currently exposed on the `EventSource` object.

For web developers (non-normative)

***source* = new EventSource( *url* [, { withCredentials: true } ])**

Creates a new `EventSource` object.

*url* is a string giving the URL that will provide the event stream.

Setting `withCredentials` to true will set the credentials mode for connection requests to *url* to "`include`".

***source*.close()**

Aborts any instances of the fetch algorithm started for this `EventSource` object, and sets the `readyState` attribute to `CLOSED`.

***source*.url**

Returns the URL providing the event stream.

***source*.withCredentials**

Returns true if the credentials mode for connection requests to the URL providing the event stream is set to "`include`", and false otherwise.

File an issue about the selected text

*source*.`readyState`

Returns the state of this `EventSource` object's connection. It can have the values described below.

The `EventSource(url, eventSourceInitDict)` constructor, when invoked, must run these steps:

1. Let *ev* be a new `EventSource` object.
2. Let *settings* be *ev*'s relevant settings object.
3. Let *urlRecord* be the result of parsing *url* with *settings*'s API base URL and *settings*'s API URL character encoding.
4. If *urlRecord* is failure, then throw a "`SyntaxError`" `DOMException`.
5. Set *ev*'s url to *urlRecord*.
6. Let *corsAttributeState* be Anonymous.
7. If the value of *eventSourceInitDict*'s `withCredentials` member is true, then set *corsAttributeState* to Use Credentials and set *ev*'s `withCredentials` attribute to true.
8. Let *request* be the result of creating a potential-CORS request given *urlRecord*, the empty string, and *corsAttributeState*.
9. Set *request*'s client to *settings*.
10. User agents may set (`` `Accept` ``, `` `text/event-stream` ``) in *request*'s header list.
11. Set *request*'s cache mode to "`no-store`".
12. Set *request*'s initiator type to "`other`".
13. Set *ev*'s request to *request*.
14. Let *processEventSourceEndOfBody* given response *res* be the following step: if *res* is not a network error, then reestablish the connection.
15. Fetch *request*, with *processResponseEndOfBody* set to *processEventSourceEndOfBody* and *processResponse* set to the following steps given response *res*:
    1. If *res* is an aborted network error, then fail the connection.
    2. Otherwise, if *res* is a network error, then reestablish the connection, unless the user agent knows that to be futile, in which case the user agent may fail the connection.
    3. Otherwise, if *res*'s status is not 200, or if *res*'s `` `Content-Type` `` is not `` `text/event-stream` ``, then fail the connection.
    4. Otherwise, announce the connection and interpret *res*'s body line by line.
16. Return *ev*.

The **`url`** attribute's getter must return the serialization of this `EventSource` object's url.

The **`withCredentials`** attribute must return the value to which it was last initialized. When the object is created, it must be initialized to false.

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, or it was closed and the user agent is reconnecting.

**`OPEN` (numeric value 1)**
: The user agent has an open connection and is dispatching events as it receives them.

**`CLOSED` (numeric value 2)**
: The connection is not open, and the user agent is not trying to reconnect. Either there was a fatal error or the `close()` method was invoked.

When the object is created its `readyState` must be set to `CONNECTING` (0). The rules given below for handling the connection define when the value changes.

The **`close()`** method must abort any instances of the fetch algorithm started for this `EventSource` object, and must set the `readyState` attribute to `CLOSED`.

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 `EventSource` interface:

| Event handler | Event handler event type |
|---|---|
| `onopen` | `open` |
| `onmessage` | `message` |
| `onerror` | `error` |

File an issue about the selected text

### 9.2.3 Processing model §

When a user agent is to **announce the connection**, the user agent must queue a task which, if the readyState attribute is set to a value other than CLOSED, sets the readyState attribute to OPEN and fires an event named open at the EventSource object.

When a user agent is to **reestablish the connection**, the user agent must run the following steps. These steps are run in parallel, not as part of a task. (The tasks that it queues, of course, are run like normal tasks and not themselves in parallel.)

1. Queue a task to run the following steps:
    1. If the readyState attribute is set to CLOSED, abort the task.
    2. Set the readyState attribute to CONNECTING.
    3. Fire an event named error at the EventSource object.
2. Wait a delay equal to the reconnection time of the event source.
3. Optionally, wait some more. In particular, if the previous attempt failed, then user agents might introduce an exponential backoff delay to avoid overloading a potentially already overloaded server. Alternatively, if the operating system has reported that there is no network connectivity, user agents might wait for the operating system to announce that the network connection has returned before retrying.
4. Wait until the aforementioned task has run, if it has not yet run.
5. Queue a task to run the following steps:
    1. If the EventSource object's readyState attribute is not set to CONNECTING, then return.
    2. Let *request* be the EventSource object's request.
    3. If the EventSource object's last event ID string is not the empty string, then:
        1. Let *lastEventIDValue* be the EventSource object's last event ID string, encoded as UTF-8.
        2. Set (`Last-Event-ID`, *lastEventIDValue*) in *request*'s header list.
    4. Fetch *request* and process the response obtained in this fashion, if any, as described earlier in this section.

When a user agent is to **fail the connection**, the user agent must queue a task which, if the readyState attribute is set to a value other than CLOSED, sets the readyState attribute to CLOSED and fires an event named error at the EventSource object. **Once the user agent has failed the connection, it does *not* attempt to reconnect.**

The task source for any tasks that are queued by EventSource objects is the **remote event task source**.

### 9.2.4 The `Last-Event-ID` header §

The **Last-Event-ID**` HTTP request header reports an EventSource object's last event ID string to the server when the user agent is to reestablish the connection.

See whatwg/html issue #7363 to define the value space better. It is essentially any UTF-8 encoded string, that does not contain U+0000 NULL, U+000A LF, or U+000D CR.

### 9.2.5 Parsing an event stream §

This event stream format's MIME type is text/event-stream.

The event stream format is as described by the `stream` production of the following ABNF, the character set for which is Unicode. [ABNF]

```
stream        = [ bom ] *event
event         = *( comment / field ) end-of-line
comment       = colon *any-char end-of-line
field         = 1*name-char [ colon [ space ] *any-char ] end-of-line
end-of-line   = ( cr lf / cr / lf )

; characters
lf            = %x000A ; U+000A LINE FEED (LF)
cr            = %x000D ; U+000D CARRIAGE RETURN (CR)
space         = %x0020 ; U+0020 SPACE
File an issue about the selected text  A ; U+003A COLON (:)
```

```
bom         = %xFEFF ; U+FEFF BYTE ORDER MARK
name-char   = %x0000-0009 / %x000B-000C / %x000E-0039 / %x003B-10FFFF
                ; a scalar value other than U+000A LINE FEED (LF), U+000D CARRIAGE RETURN (CR), or U+003A COLON (:)
any-char    = %x0000-0009 / %x000B-000C / %x000E-10FFFF
                ; a scalar value other than U+000A LINE FEED (LF) or U+000D CARRIAGE RETURN (CR)
```

Event streams in this format must always be encoded as UTF-8. [ENCODING]

Lines must be separated by either a U+000D CARRIAGE RETURN U+000A LINE FEED (CRLF) character pair, a single U+000A LINE FEED (LF) character, or a single U+000D CARRIAGE RETURN (CR) character.

Since connections established to remote servers for such resources are expected to be long-lived, UAs should ensure that appropriate buffering is used. In particular, while line buffering with lines are defined to end with a single U+000A LINE FEED (LF) character is safe, block buffering or line buffering with different expected line endings can cause delays in event dispatch.

### 9.2.6 Interpreting an event stream §

Streams must be decoded using the UTF-8 decode algorithm.

Note

*The UTF-8 decode algorithm strips one leading UTF-8 Byte Order Mark (BOM), if any.*

The stream must then be parsed by reading everything line by line, with a U+000D CARRIAGE RETURN U+000A LINE FEED (CRLF) character pair, a single U+000A LINE FEED (LF) character not preceded by a U+000D CARRIAGE RETURN (CR) character, and a single U+000D CARRIAGE RETURN (CR) character not followed by a U+000A LINE FEED (LF) character being the ways in which a line can end.

When a stream is parsed, a *data* buffer, an *event type* buffer, and a *last event ID* buffer must be associated with it. They must be initialized to the empty string.

Lines must be processed, in the order they are received, as follows:

↪ **If the line is empty (a blank line)**

Dispatch the event, as defined below.

↪ **If the line starts with a U+003A COLON character (:)**

Ignore the line.

↪ **If the line contains a U+003A COLON character (:)**

Collect the characters on the line before the first U+003A COLON character (:), and let *field* be that string.

Collect the characters on the line after the first U+003A COLON character (:), and let *value* be that string. If *value* starts with a U+0020 SPACE character, remove it from *value*.

Process the field using the steps described below, using *field* as the field name and *value* as the field value.

↪ **Otherwise, the string is not empty but does not contain a U+003A COLON character (:)**

Process the field using the steps described below, using the whole line as the field name, and the empty string as the field value.

Once the end of the file is reached, any pending data must be discarded. (If the file ends in the middle of an event, before the final empty line, the incomplete event is not dispatched.)

The steps to **process the field** given a field name and a field value depend on the field name, as given in the following list. Field names must be compared literally, with no case folding performed.

↪ **If the field name is "event"**

Set the *event type* buffer to field value.

↪ **If the field name is "data"**

Append the field value to the *data* buffer, then append a single U+000A LINE FEED (LF) character to the *data* buffer.

↪ **If the field name is "id"**

If the field value does not contain U+0000 NULL, then set the *last event ID* buffer to the field value. Otherwise, ignore the field.

↪ **If the field name is "retry"**

If the field value consists of only ASCII digits, then interpret the field value as an integer in base ten, and set the event stream's reconnection time to that integer. Otherwise, ignore the field.

File an issue about the selected text

↪ **Otherwise**

The field is ignored.

When the user agent is required to **dispatch the event**, the user agent must process the *data* buffer, the *event type* buffer, and the *last event ID* buffer using steps appropriate for the user agent.

For web browsers, the appropriate steps to dispatch the event are as follows:

1. Set the last event ID string of the event source to the value of the *last event ID* buffer. The buffer does not get reset, so the last event ID string of the event source remains set to this value until the next time it is set by the server.
2. If the *data* buffer is an empty string, set the *data* buffer and the *event type* buffer to the empty string and return.
3. If the *data* buffer's last character is a U+000A LINE FEED (LF) character, then remove the last character from the *data* buffer.
4. Let *event* be the result of creating an event using `MessageEvent`, in the relevant realm of the `EventSource` object.
5. Initialize *event*'s `type` attribute to "`message`", its `data` attribute to *data*, its `origin` attribute to the serialization of the origin of the event stream's final URL (i.e., the URL after redirects), and its `lastEventId` attribute to the last event ID string of the event source.
6. If the *event type* buffer has a value other than the empty string, change the type of the newly created event to equal the value of the *event type* buffer.
7. Set the *data* buffer and the *event type* buffer to the empty string.
8. Queue a task which, if the `readyState` attribute is set to a value other than `CLOSED`, dispatches the newly created event at the `EventSource` object.

Note

*If an event doesn't have an "id" field, but an earlier event did set the event source's last event ID string, then the event's `lastEventId` field will be set to the value of whatever the last seen "id" field was.*

For other user agents, the appropriate steps to dispatch the event are implementation dependent, but at a minimum they must set the *data* and *event type* buffers to the empty string before returning.

Example

The following event stream, once followed by a blank line:

```
data: YHOO
data: +2
data: 10
```

...would cause an event `message` with the interface `MessageEvent` to be dispatched on the `EventSource` object. The event's `data` attribute would contain the string "`YHOO\n+2\n10`" (where "`\n`" represents a newline).

This could be used as follows:

```
var stocks = new EventSource("https://stocks.example.com/ticker.php");
stocks.onmessage = function (event) {
  var data = event.data.split('\n');
  updateStocks(data[0], data[1], data[2]);
};
```

...where `updateStocks()` is a function defined as:

```
function updateStocks(symbol, delta, value) { ... }
```

...or some such.

Example

The following stream contains four blocks. The first block has just a comment, and will fire nothing. The second block has two fields with names "data" and "id" respectively; an event will be fired for this block, with the data "first event", and will then set the last event ID to "1" so that if the connection died between this block and the next, the server would be sent a ``Last-Event-ID`` header with the value ``1``. The third block fires an event with data "second event", and also has an "id" field, this time with no value, which resets the last event ID to the empty string (meaning no ``Last-Event-ID`` header will now be sent in the event of a reconnection being attempted). Finally, the last block just fires an event with the data " third event" (with a single leading space character). Note that the last still has to end with a blank line, the end of the stream is not enough to trigger the dispatch of the last event.

```
: test stream

data: first event
id: 1

data:second event
```

File an issue about the selected text

```
id

data:  third event
```

Example

The following stream fires two events:

```
data

data
data

data:
```

The first block fires events with the data set to the empty string, as would the last block if it was followed by a blank line. The middle block fires an event with the data set to a single newline character. The last block is discarded because it is not followed by a blank line.

Example

The following stream fires two identical events:

```
data:test

data: test
```

This is because the space after the colon is ignored if present.

### 9.2.7 Authoring notes §

Legacy proxy servers are known to, in certain cases, drop HTTP connections after a short timeout. To protect against such proxy servers, authors can include a comment line (one starting with a ':' character) every 15 seconds or so.

Authors wishing to relate event source connections to each other or to specific documents previously served might find that relying on IP addresses doesn't work, as individual clients can have multiple IP addresses (due to having multiple proxy servers) and individual IP addresses can have multiple clients (due to sharing a proxy server). It is better to include a unique identifier in the document when it is served and then pass that identifier as part of the URL when the connection is established.

Authors are also cautioned that HTTP chunking can have unexpected negative effects on the reliability of this protocol, in particular if the chunking is done by a different layer unaware of the timing requirements. If this is a problem, chunking can be disabled for serving event streams.

Clients that support HTTP's per-server connection limitation might run into trouble when opening multiple pages from a site if each page has an `EventSource` to the same domain. Authors can avoid this using the relatively complex mechanism of using unique domain names per connection, or by allowing the user to enable or disable the `EventSource` functionality on a per-page basis, or by sharing a single `EventSource` object using a shared worker.

### 9.2.8 Connectionless push and other features §

User agents running in controlled environments, e.g. browsers on mobile handsets tied to specific carriers, may offload the management of the connection to a proxy on the network. In such a situation, the user agent for the purposes of conformance is considered to include both the handset software and the network proxy.

Example

For example, a browser on a mobile device, after having established a connection, might detect that it is on a supporting network and request that a proxy server on the network take over the management of the connection. The timeline for such a situation might be as follows:

1. Browser connects to a remote HTTP server and requests the resource specified by the author in the `EventSource` constructor.
2. The server sends occasional messages.
3. In between two messages, the browser detects that it is idle except for the network activity involved in keeping the TCP connection alive, and decides to switch to sleep mode to save power.
4. The browser disconnects from the server.
5. The browser contacts a service on the network, and requests that the service, a "push proxy", maintain the connection instead.
6. The "push proxy" service contacts the remote HTTP server and requests the resource specified by the author in the `EventSource` constructor (possibly including a `` `Last-Event-ID` `` HTTP header, etc.).
7. The browser allows the mobile device to go to sleep.

File an issue about the selected text :her message.

9. The "push proxy" service uses a technology such as OMA push to convey the event to the mobile device, which wakes only enough to process the event and then returns to sleep.

This can reduce the total data usage, and can therefore result in considerable power savings.

As well as implementing the existing API and `text/event-stream` wire format as defined by this specification and in more distributed ways as described above, formats of event framing defined by other applicable specifications may be supported. This specification does not define how they are to be parsed or processed.

### 9.2.9 Garbage collection §

While an `EventSource` object's `readyState` is `CONNECTING`, and the object has one or more event listeners registered for `open`, `message` or `error` events, there must be a strong reference from the `Window` or `WorkerGlobalScope` object that the `EventSource` object's constructor was invoked from to the `EventSource` object itself.

While an `EventSource` object's `readyState` is `OPEN`, and the object has one or more event listeners registered for `message` or `error` events, there must be a strong reference from the `Window` or `WorkerGlobalScope` object that the `EventSource` object's constructor was invoked from to the `EventSource` object itself.

While there is a task queued by an `EventSource` object on the remote event task source, there must be a strong reference from the `Window` or `WorkerGlobalScope` object that the `EventSource` object's constructor was invoked from to that `EventSource` object.

If a user agent is to **forcibly close** an `EventSource` object (this happens when a `Document` object goes away permanently), the user agent must abort any instances of the fetch algorithm started for this `EventSource` object, and must set the `readyState` attribute to `CLOSED`.

If an `EventSource` object is garbage collected while its connection is still open, the user agent must abort any instance of the fetch algorithm opened by this `EventSource`.

### 9.2.10 Implementation advice §

*This section is non-normative.*

User agents are strongly urged to provide detailed diagnostic information about `EventSource` objects and their related network connections in their development consoles, to aid authors in debugging code using this API.

For example, a user agent could have a panel displaying all the `EventSource` objects a page has created, each listing the constructor's arguments, whether there was a network error, what the CORS status of the connection is and what headers were sent by the client and received from the server to lead to that status, the messages that were received and how they were parsed, and so forth.

Implementations are especially encouraged to report detailed information to their development consoles whenever an `error` event is fired, since little to no information can be made available in the events themselves.

← 9 Communication — Table of Contents — 9.3 Cross-document messaging →

File an issue about the selected text