Server-Sent Events

Editor's Draft 14 May 2014

Latest Published Version:
http://www.w3.org/TR/eventsource/
Latest Editor's Draft:
http://dev.w3.org/html5/eventsource/
Previous Versions:
http://www.w3.org/TR/2011/WD-eventsource-20110310/
http://www.w3.org/TR/2011/WD-eventsource-20110208/
http://www.w3.org/TR/2009/WD-eventsource-20091222/
http://www.w3.org/TR/2009/WD-eventsource-20091029/
http://www.w3.org/TR/2009/WD-eventsource-20090423/
Editor:
Ian Hickson, Google, Inc.

Abstract

This specification defines an API for opening an HTTP connection for receiving push notifications from a server in the form of DOM events. The API is designed such that it can be extended to work with other push notification schemes such as Push SMS.

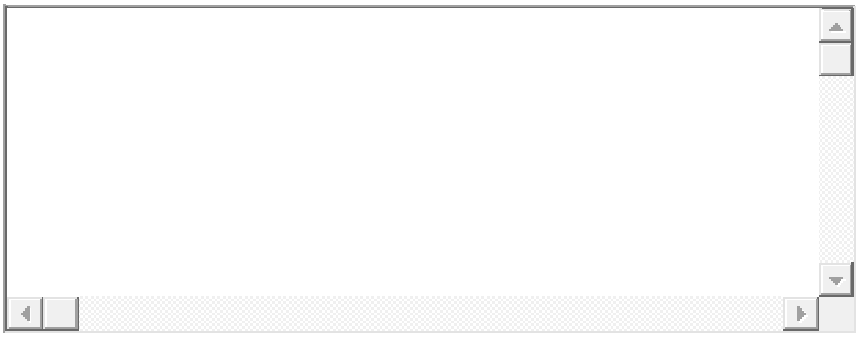
Status of This document

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 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.



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

Submit feedback (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 14 May 2014 Editor's Draft.

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

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);
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 implementor 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.

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 `EventSource` interface

```
[Constructor(DOMString url, optional EventSourceInit
eventSourceInitDict), Exposed=Window,Worker]
interface EventSource : EventTarget {
  readonly attribute DOMString 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;
  void close();
};

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

The **`EventSource()`** constructor takes one or two arguments. The first specifies the URL to which to connect. The second specifies the settings, if any, in the form of an `EventSourceInit` dictionary. When the `EventSource()` constructor is invoked, the UA must run these steps:

1. Resolve the URL specified in the first argument, relative to the API base URL specified by the entry settings object.

2. If the previous step failed, then throw a `SyntaxError` exception and abort these steps.

3. Create a new `EventSource` object.

4. Let *CORS mode* be Anonymous.

5. If the second argument is present, and the `withCredentials` dictionary member has the value true, then set *CORS mode* to Use Credentials and initialise the new `EventSource` object's `withCredentials` attribute to true.

6. Return the new `EventSource` object, but continue these steps asynchronously.

7. Do a potentially CORS-enabled fetch of the resulting absolute URL using the API referrer source specified by the entry settings object, with the *mode* being *CORS mode*, and the *origin* being the origin specified by the entry settings object, and process the resource obtained in this fashion, if any, as described below.

The definition of the fetching algorithm (which is used by CORS) is such that if the browser is already fetching the resource identified by the given absolute URL, that connection can be reused, instead of a new connection being established. All messages received up to this point are dispatched immediately, in this case.

The **`url`** attribute must return the absolute URL that resulted from resolving the value that was passed to the constructor.

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

In addition to the above, each `EventSource` object has the following associated with it:

- A **reconnection time**, in milliseconds. This must initially be a user-agent-defined value, probably in the region of a few seconds.
- A **last event ID string**. This must initially be the empty string.

These values are not currently exposed on the interface.

5 Processing model

The resource indicated in the argument to the `EventSource` constructor is fetched when the constructor is run.

For HTTP connections, the `Accept` header may be included; if included, it must contain only formats of event framing that are supported by the user agent (one of which must be `text/event-stream`, as described below).

If the event source's last event ID string is not the empty string, then a `Last-Event-ID` HTTP header must be included with the request, whose value is the value of the event source's last event ID string, encoded as UTF-8.

User agents should use the `Cache-Control: no-cache` header in requests to bypass any caches for requests of event sources. (This header is not a custom request header, so the user agent will still use the CORS simple cross-origin request mechanism.) User agents should ignore HTTP cache headers in the response, never caching event sources.

As data is received, the tasks queued by the networking task source to handle the data must act as follows.

HTTP 200 OK responses with a Content-Type header specifying the type `text/event-stream`, ignoring any MIME type parameters, must be processed line by line as described below.

When a successful response with a supported MIME type is received, such that the user agent begins parsing the contents of the stream, the user agent must announce the connection.

The task that the networking task source places on the task queue once the fetching algorithm for such a resource (with the correct MIME type) has completed must cause the user agent to asynchronously reestablish the connection. This applies whether the connection is closed gracefully or unexpectedly (but does not apply when the fetch algorithm is canceled by the user agent, e.g. in response to `window.stop()`, since in those cases the final task is actually discarded). It doesn't apply for the error conditions listed below except where explicitly specified.

HTTP 200 OK responses that have a Content-Type specifying an unsupported type, or that have no Content-Type at all, must cause the user agent to fail the connection.

HTTP 305 Use Proxy, 401 Unauthorized, and 407 Proxy Authentication Required should be treated transparently as for any other subresource.

HTTP 301 Moved Permanently, 302 Found, 303 See Other, and 307 Temporary Redirect responses are handled by the fetching and CORS algorithms. In the case of 301 redirects, the user agent must also remember the new URL so that subsequent requests for this resource for this `EventSource` object start with the URL given for the last 301 seen for requests for this object.

Network errors that prevents the connection from being established in the first place (e.g. DNS errors), must cause the user agent to asynchronously reestablish the connection.

Any other HTTP response code not listed here, as well as the cancelation of the fetch algorithm by the user agent (e.g. in response to `window.stop()` or the user canceling the network connection manually) must cause the user agent to fail the connection.

For non-HTTP protocols, UAs should act in equivalent ways.

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 a simple 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 asynchronously, not as part of a task. (The tasks that it queues, of course, are run like normal tasks and not asynchronously.)

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 a simple 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 `readyState` attribute is not set to `CONNECTING`, abort these steps.
    2. Perform a potentially CORS-enabled fetch of the absolute URL of the event source resource, using the same *referrer source*, and with the same *mode* and *origin*, as those used in the original request triggered by the `EventSource()` constructor, and process the resource 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 a simple 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**.

6 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
colon         = %x003A ; U+003A COLON (:)
bom           = %xFEFF ; U+FEFF BYTE ORDER MARK
name-char     = %x0000-0009 / %x000B-000C / %x000E-0039 / %x003B-10FFFF
                ; a Unicode character 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 Unicode character 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.

7 Interpreting an event stream

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

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 initialised 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"

Set the *last event ID* buffer to the field value.

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.

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 abort these steps.

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. Create an event that uses the `MessageEvent` interface, with the event type `message`, which does not bubble, is not cancelable, and has no default action. The `data` attribute must be initialized to the value of the *data* buffer, the `origin` attribute must be initialised to the Unicode serialization of the origin of the event stream's final URL (i.e. the URL after redirects), and the `lastEventId` attribute must be initialised to the last event ID string of the event source. This event is not trusted.

5. 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.

6. Set the *data* buffer and the *event type* buffer to the empty string.

7. 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.

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.

> 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("http://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.

> 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
> id
>
> data:  third event
> ```

> 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.

The following stream fires two identical events:

```
data:test

data: test
```

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

7.1 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. Where possible, chunking should be disabled for serving event streams unless the rate of messages is high enough for this not to matter.

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.

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.

> 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 that 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.
> 8. The server sends another 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 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`.

It's possible for one active network connection to be shared by multiple `EventSource` objects and their fetch algorithms, which is why the above is phrased in terms of aborting the fetch algorithm and not the actual underlying download.

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.

11 IANA considerations

11.1 `text/event-stream`

This registration is for community review and will be submitted to the IESG for review, approval, and registration with IANA.

Type name:
text
Subtype name:
event-stream
Required parameters:
No parameters
Optional parameters:

`charset`
The `charset` parameter may be provided. The parameter's value must be "`utf-8`". This parameter serves no purpose; it is only allowed for compatibility with legacy servers.

Encoding considerations:
8bit (always UTF-8)
Security considerations:
An event stream from an origin distinct from the origin of the content consuming the event stream can result in information leakage. To avoid this, user agents are required to apply CORS semantics. [FETCH]

Event streams can overwhelm a user agent; a user agent is expected to apply suitable restrictions to avoid depleting local resources because of an overabundance of information from an event stream.

Servers can be overwhelmed if a situation develops in which the server is causing clients to reconnect rapidly. Servers should use a 5xx status code to indicate capacity problems, as this will prevent conforming clients from reconnecting automatically.

Interoperability considerations:
Rules for processing both conforming and non-conforming content are defined in this specification.
Published specification:
This document is the relevant specification.
Applications that use this media type:
Web browsers and tools using Web services.
Additional information:

Magic number(s):
No sequence of bytes can uniquely identify an event stream.

File extension(s):
No specific file extensions are recommended for this type.

Macintosh file type code(s):

No specific Macintosh file type codes are recommended for this type.

Person & email address to contact for further information:
Ian Hickson <ian@hixie.ch>

Intended usage:
Common

Restrictions on usage:
This format is only expected to be used by dynamic open-ended streams served using HTTP or a similar protocol. Finite resources are not expected to be labeled with this type.

Author:
Ian Hickson <ian@hixie.ch>

Change controller:
W3C

Fragment identifiers have no meaning with `text/event-stream` resources.

11.2 `Last-Event-ID`

This section describes a header for registration in the Permanent Message Header Field Registry. [RFC3864]

Header field name:
Last-Event-ID

Applicable protocol:
http

Status:
standard

Author/Change controller:
W3C

Specification document(s):
This document is the relevant specification.

Related information:
None.

References

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

[ABNF]
Augmented BNF for Syntax Specifications: ABNF, D. Crocker, P. Overell. IETF.

[DOM]
DOM, A. van Kesteren, A. Gregor, Ms2ger. WHATWG.

[ENCODING]
Encoding, A. van Kesteren, J. Bell. WHATWG.

[FETCH]
Fetch, A. van Kesteren. WHATWG.

[HTML]
HTML, I. Hickson. WHATWG.

[RFC2119]
Key words for use in RFCs to Indicate Requirement Levels, S. Bradner. IETF.

[RFC3864]
Registration Procedures for Message Header Fields, G. Klyne, M. Nottingham, J. Mogul. IETF.

[WEBIDL]
Web IDL, C. McCormack. W3C.

Acknowledgements

For a full list of acknowledgements, please see the WHATWG HTML standard.