7th October 2019

Server Sent Events

The Server-Sent Events specification describes a built-in class `EventSource`, that keeps connection with the server and allows to receive events from it.

Similar to `WebSocket`, the connection is persistent.

But there are several important differences:

`WebSocket`	`EventSource`
Bi-directional: both client and server can exchange messages	One-directional: only server sends data
Binary and text data	Only text
WebSocket protocol	Regular HTTP

`EventSource` is a less-powerful way of communicating with the server than `WebSocket`.

Why should one ever use it?

The main reason: it's simpler. In many applications, the power of `WebSocket` is a little bit too much.

We need to receive a stream of data from server: maybe chat messages or market prices, or whatever. That's what `EventSource` is good at. Also it supports auto-reconnect, something we need to implement manually with `WebSocket`. Besides, it's a plain old HTTP, not a new protocol.

Getting messages

To start receiving messages, we just need to create `new EventSource(url)`.

The browser will connect to `url` and keep the connection open, waiting for events.

The server should respond with status 200 and the header `Content-Type: text/event-stream`, then keep the connection and write messages into it in the special format, like this:

```
data: Message 1

data: Message 2

data: Message 3
data: of two lines
```

- A message text goes after `data:`, the space after the colon is optional.
- Messages are delimited with double line breaks `\n\n`.

- To send a line break `\n`, we can immediately send one more `data:` (3rd message above).

In practice, complex messages are usually sent JSON-encoded. Line-breaks are encoded as `\n` within them, so multiline `data:` messages are not necessary.

For instance:

```
data: {"user":"John","message":"First line\n Second line"}
```

…So we can assume that one `data:` holds exactly one message.

For each such message, the `message` event is generated:

```
let eventSource = new EventSource("/events/subscribe");

eventSource.onmessage = function(event) {
  console.log("New message", event.data);
  // will log 3 times for the data stream above
};

// or eventSource.addEventListener('message', ...)
```

Cross-origin requests

`EventSource` supports cross-origin requests, like `fetch` any other networking methods. We can use any URL:

```
let source = new EventSource("https://another-site.com/events");
```

The remote server will get the `Origin` header and must respond with `Access-Control-Allow-Origin` to proceed.

To pass credentials, we should set the additional option `withCredentials`, like this:

```
let source = new EventSource("https://another-site.com/events", {
  withCredentials: true
});
```

Please see the chapter Fetch: Cross-Origin Requests for more details about cross-origin headers.

Reconnection

Upon creation, `new EventSource` connects to the server, and if the connection is broken – reconnects.

That's very convenient, as we don't have to care about it.

There's a small delay between reconnections, a few seconds by default.

The server can set the recommended delay using `retry:` in response (in milliseconds):

```
retry: 15000
data: Hello, I set the reconnection delay to 15 seconds
```

The `retry:` may come both together with some data, or as a standalone message.

The browser should wait that many milliseconds before reconnecting. Or longer, e.g. if the browser knows (from OS) that there’s no network connection at the moment, it may wait until the connection appears, and then retry.

- If the server wants the browser to stop reconnecting, it should respond with HTTP status 204.
- If the browser wants to close the connection, it should call `eventSource.close()` :

```
let eventSource = new EventSource(...);

eventSource.close();
```

Also, there will be no reconnection if the response has an incorrect `Content-Type` or its HTTP status differs from 301, 307, 200 and 204. In such cases the `"error"` event will be emitted, and the browser won’t reconnect.

Please note:

When a connection is finally closed, there’s no way to “reopen” it. If we’d like to connect again, just create a new `EventSource` .

Message id

When a connection breaks due to network problems, either side can’t be sure which messages were received, and which weren’t.

To correctly resume the connection, each message should have an `id` field, like this:

```
data: Message 1
id: 1

data: Message 2
id: 2

data: Message 3
data: of two lines
id: 3
```

When a message with `id:` is received, the browser:

- Sets the property `eventSource.lastEventId` to its value.
- Upon reconnection sends the header `Last-Event-ID` with that `id` , so that the server may re-send following messages.

Put `id:` after `data:`

Please note: the `id` is appended below message `data` by the server, to ensure that `lastEventId` is updated after the message is received.

Connection status: readyState

The `EventSource` object has `readyState` property, that has one of three values:

```
EventSource.CONNECTING = 0; // connecting or reconnecting
EventSource.OPEN = 1;       // connected
EventSource.CLOSED = 2;     // connection closed
```

When an object is created, or the connection is down, it's always `EventSource.CONNECTING` (equals `0`).

We can query this property to know the state of `EventSource`.

Event types

By default `EventSource` object generates three events:

- `message` – a message received, available as `event.data`.
- `open` – the connection is open.
- `error` – the connection could not be established, e.g. the server returned HTTP 500 status.

The server may specify another type of event with `event: ...` at the event start.

For example:

```
event: join
data: Bob

data: Hello

event: leave
data: Bob
```

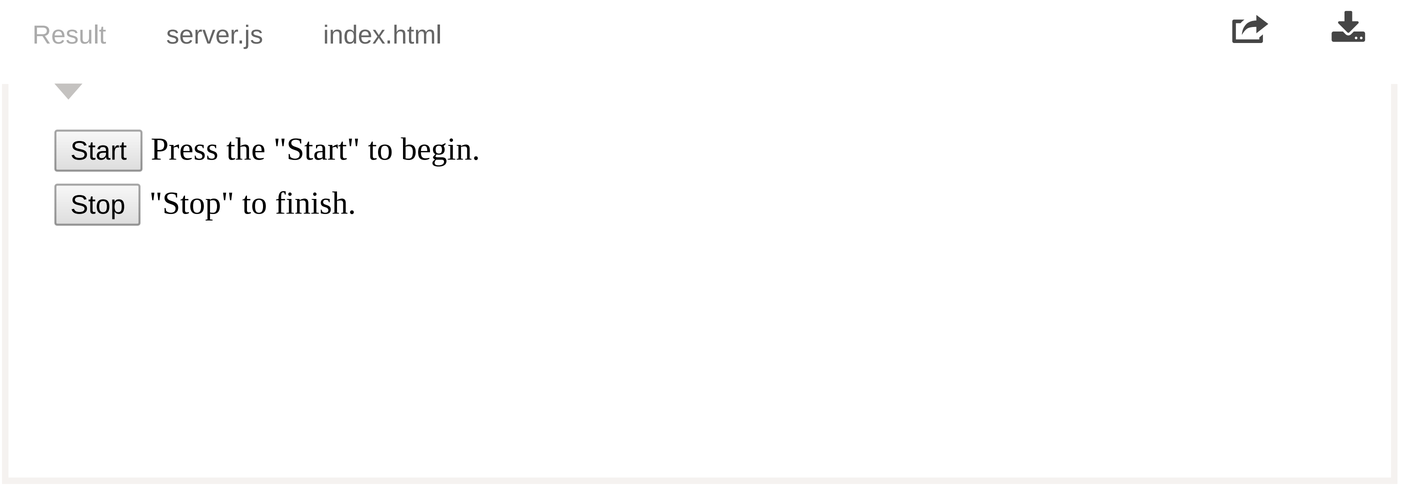
To handle custom events, we must use `addEventListener`, not `onmessage`:

```
eventSource.addEventListener('join', event => {
  alert(`Joined ${event.data}`);
});

eventSource.addEventListener('message', event => {
  alert(`Said: ${event.data}`);
});

eventSource.addEventListener('leave', event => {
  alert(`Left ${event.data}`);
});
```

Full example

Here’s the server that sends messages with `1`, `2`, `3`, then `bye` and breaks the connection.

Then the browser automatically reconnects.



Press the "Start" to begin.

"Stop" to finish.

Summary

`EventSource` object automatically establishes a persistent connection and allows the server to send messages over it.

It offers:

- Automatic reconnect, with tunable `retry` timeout.
- Message ids to resume events, the last received identifier is sent in `Last-Event-ID` header upon reconnection.
- The current state is in the `readyState` property.

That makes `EventSource` a viable alternative to `WebSocket`, as it’s more low-level and lacks such built-in features (though they can be implemented).

In many real-life applications, the power of `EventSource` is just enough.

Supported in all modern browsers (not IE).

The syntax is:

```
let source = new EventSource(url, [credentials]);
```

The second argument has only one possible option: `{ withCredentials: true }`, it allows sending cross-origin credentials.

Overall cross-origin security is same as for `fetch` and other network methods.

Properties of an `EventSource` object

`readyState`

The current connection state: either `EventSource.CONNECTING (=0)` , `EventSource.OPEN (=1)` or `EventSource.CLOSED (=2)` .

`lastEventId`

The last received `id` . Upon reconnection the browser sends it in the header `Last-Event-ID` .

Methods

`close()`

Closes the connection.

Events

`message`

Message received, the data is in `event.data` .

`open`

The connection is established.

`error`

In case of an error, including both lost connection (will auto-reconnect) and fatal errors. We can check `readyState` to see if the reconnection is being attempted.

The server may set a custom event name in `event:` . Such events should be handled using `addEventListener` , not `on<event>` .

Server response format

The server sends messages, delimited by `\n\n` .

A message may have following fields:

- `data:` – message body, a sequence of multiple `data` is interpreted as a single message, with `\n` between the parts.
- `id:` – renews `lastEventId` , sent in `Last-Event-ID` on reconnect.
- `retry:` – recommends a retry delay for reconnections in ms. There's no way to set it from JavaScript.
- `event:` – event name, must precede `data:` .

A message may include one or more fields in any order, but `id:` usually goes the last.

Previous lesson Next lesson

Share Tutorial map

Comments

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