Server-Sent Events (SSE)

BROWSER APIS AND PROTOCOLS, CHAPTER 16

Server-Sent Events enables efficient server-to-client streaming of text-based event data—e.g., real-time notifications or updates generated on the server. To meet this goal, SSE introduces two components: a new EventSource interface in the browser, which allows the client to receive push notifications from the server as DOM events, and the "event stream" data format, which is used to deliver the individual updates.

The combination of the EventSource API in the browser and the well-defined event stream data format is what makes SSE both an efficient and an indispensable tool for handling real-time data in the browser:

- Low latency delivery via a single, long-lived connection
- Efficient browser message parsing with no unbounded buffers
- Automatic tracking of last seen message and auto reconnect
- Client message notifications as DOM events

Under the hood, SSE provides an efficient, cross-browser implementation of XHR streaming; the actual delivery of the messages is done over a single, long-lived HTTP connection. However, unlike dealing XHR streaming on our own, the browser handles all the connection management and message parsing, allowing our applications to focus on the business logic! In short, SSE makes working with real-time data simple and efficient. Let's take a look under the hood.

EventSource API §

The EventSource interface abstracts all the low-level connection establishment and message parsing behind a simple browser API. To get started, we simply need to specify the URL of the SSE event stream resource and register the appropriate JavaScript event listeners on the object:

```
var source = new EventSource("/path/to/stream-url"); 1

source.onopen = function () { ... }; 2
source.onerror = function () { ... }; 3

source.addEventListener("foo", function (event) { 4
  processFoo(event.data);
});

source.onmessage = function (event) {  5
  log_message(event.id, event.data);
```

```
    source.close(); 6
  }
}
```

1. Open new SSE connection to stream endpoint
2. Optional callback, invoked when connection is established
3. Optional callback, invoked if the connection fails
4. Subscribe to event of type "foo"; invoke custom logic
5. Subscribe to all events without an explicit type
6. Close SSE connection if server sends a "CLOSE" message ID

Note

EventSource can stream event data from remote origins by leveraging the same CORS permission and opt-in workflow as a regular XHR.

That's all there is to it for the client API. The implementation logic is handled for us: the connection is negotiated on our behalf, received data is parsed incrementally, message boundaries are identified, and finally a DOM event is fired by the browser. EventSource interface allows the application to focus on the business logic: open new connection, process received event notifications, terminate stream when finished.

Note

SSE provides a memory-efficient implementation of XHR streaming. Unlike a raw XHR connection, which buffers the full received response until the connection is dropped, an SSE connection can discard processed messages without accumulating all of them in memory.

As icing on the cake, the EventSource interface also provides auto-reconnect and tracking of the last seen message: if the connection is dropped, EventSource will automatically reconnect to the server and optionally advertise the ID of the last seen message, such that the stream can be resumed and lost messages can be retransmitted.

How does the browser know the ID, type, and boundary of each message? This is where the event stream protocol comes in. The combination of a simple client API and a well-defined data format is what allows us to offload the bulk of the work to the browser. The two go hand in hand, even though the low-level data protocol is completely transparent to the application in the browser.

Emulating EventSource with Custom JavaScript §

browser. For the latest status, see caniuse.com/eventsource ⊘ .

However, the good news is the EventSource interface is simple enough such that it can be emulated via an optional JavaScript library (i.e., a "polyfill") for browsers that do not support it natively. Similarly, the delivery of the event stream can be implemented on top of existing XHR mechanisms:

```
if (!window.EventSource) {
  // load JavaScript polyfill library
}

var source = new EventSource("/event-stream-endpoint");
...
```

The benefit of using a polyfill library is that it allows our applications to, once again, focus on the application logic, instead of worrying about the browser quirks and implementation status. Having said that, while a polyfill will provide a consistent API, be aware that the underlying XHR transport will not be as efficient:

- XHR polling will incur message delays and high request overhead.
- XHR long-polling minimizes latency delays but has high request overhead.
- XHR streaming support is limited and buffers all the data in memory.

Without native support for efficient XHR streaming of event stream data, the polyfill library can fallback to polling, long-polling, or XHR streaming, each of which has its own performance costs. For a full discussion, refer to Real-Time Notifications and Delivery.

In short, check the implementation of your polyfill library to ensure that it meets your performance goals! Many of the most popular libraries (e.g., jQuery.EventSource) use XHR polling to emulate the SSE transport—simple, but also an inefficient transport.

Event Stream Protocol §

An SSE event stream is delivered as a streaming HTTP response: the client initiates a regular HTTP request, the server responds with a custom *"text/event-stream"* content-type, and then streams the UTF-8 encoded event data. However, even that sounds too complicated, so an example is in order:

```
=> Request
GET /stream HTTP/1.1 (1)
Host: example.com
Accept: text/event-stream

<= Response
HTTP/1.1 200 OK (2)
```

```
Transfer-Encoding: chunked

retry: 15000 (3)

data: First message is a simple string. (4)

data: {"message": "JSON payload"} (5)

event: foo (6)
data: Message of type "foo"

id: 42 (7)
event: bar
data: Multi-line message of
data: type "bar" and id "42"

id: 43 (8)
data: Last message, id "43"
```

1. Client connection initiated via EventSource interface
2. Server response with "text/event-stream" content-type
3. Server sets client reconnect interval (15s) if the connection drops
4. Simple text event with no message type
5. JSON payload with no message type
6. Simple text event of type "foo"
7. Multiline event with message ID and type
8. Simple text event with optional ID

The event-stream protocol is trivial to understand and implement:

- Event payload is the value of one or more adjacent `data` fields.
- Event may carry an optional `ID` and an `event` type string.
- Event boundaries are marked by newlines.

On the receiving end, the EventSource interface parses the incoming stream by looking for newline separators, extracts the payload from data fields, checks for optional ID and type, and finally dispatches a DOM event to notify the application. If a type is present, then a custom DOM event is fired, and otherwise the generic "onmessage" callback is invoked; see EventSource API for both cases.

UTF-8 Encoding and Binary Transfers with SSE §

EventSource does not perform any additional processing on the actual payload: the message is extracted from one or more data fields, concatenated together and passed directly to the

Having said that, note that all event source data is UTF-8 encoded: SSE is not meant as a mechanism for transferring binary payloads! If necessary, one could base64 encode an arbitrary binary object to make it SSE friendly, but doing so would incur high (33%) byte overhead; see Resource Inlining.

Concerned by high overhead of UTF-8 on the wire? An SSE connection is a streaming HTTP response, which means that it can be compressed (i.e., gziped), just as any other HTTP response while in flight! While SSE is not meant for delivery of binary data, it is nonetheless an efficient transport: ensure that your server is applying gzip compression on the SSE stream.

Lack of support for binary streaming is not an oversight. SSE was specifically designed as a simple, efficient, server-to-client transport for text-based data. If you need to transfer binary payloads, then a WebSocket is the right tool for the job.

Finally, in addition to automatic event parsing, SSE provides built-in support for reestablishing dropped connections, as well as recovery of messages the client may have missed while disconnected. By default, if the connection is dropped, then the browser will automatically reestablish the connection. The SSE specification recommends a 2–3 second delay, which is a common default for most browsers, but the server can also set a custom interval at any point by sending a `retry` command to the client.

Similarly, the server can also associate an arbitrary ID string with each message. The browser automatically remembers the last seen ID and will automatically append a "Last-Event-ID" HTTP header with the remembered value when issuing a reconnect request. Here's an example:

```
(existing SSE connection)
retry: 4500 1

id: 43 2
data: Lorem ipsum

(connection dropped)
(4500 ms later)

=> Request
GET /stream HTTP/1.1 3
Host: example.com
Accept: text/event-stream
Last-Event-ID: 43

<= Response
HTTP/1.1 200 OK 4
Content-Type: text/event-stream
Connection: keep-alive
Transfer-Encoding: chunked
```

1. Server sets the client reconnect interval to 4.5 seconds
2. Simple text event, ID: 43
3. Automatic client reconnect request with last seen event ID
4. Server response with "text/event-stream" content-type
5. Simple text event, ID: 44

The client application does not need to provide any extra logic to reestablish the connection or remember the last seen event ID. The entire workflow is handled by the browser, and we rely on the server to handle the recovery. Specifically, depending on the requirements of the application and the data stream, the server can implement several different strategies:

- If lost messages are acceptable, then no event IDs or special logic is required: simply let the client reconnect and resume the stream.
- If message recovery is required, then the server needs to specify IDs for relevant events, such that the client can report the last seen ID when reconnecting. Also, the server needs to implement some form of a local cache to recover and retransmit missed messages to the client.

The exact implementation details of how far back the messages are persisted are, of course, specific to the requirements of the application. Further, note that the ID is an optional event stream field. Hence, the server can also choose to checkpoint specific messages or milestones in the delivered event stream. In short, evaluate your requirements, and implement the appropriate logic on the server.

SSE Use Cases and Performance §

SSE is a high-performance transport for server-to-client streaming of text-based real-time data: messages can be pushed the moment they become available on the server (low latency), there is minimum message overhead (long-lived connection, event-stream protocol, and gzip compression), the browser handles all the message parsing, and there are no unbounded buffers. Add to that a convenient EventSource API with auto-reconnect and message notifications as DOM events, and SSE becomes an indispensable tool for working with real-time data!

There are two key limitations to SSE. First, it is server-to-client only and hence does not address the request streaming use case—e.g., streaming a large upload to the server. Second, the event-stream protocol is specifically designed to transfer UTF-8 data: binary streaming, while possible, is inefficient.

Having said that, the UTF-8 limitation can often be resolved at the application layer: SSE delivers a notification to the application about a new binary asset available on the server, and the

It also has the benefit of leveraging the numerous services provided by the XHR: response caching, transfer-encoding (compression), and so on. If an asset is streamed, it cannot be cached by the browser cache.

Note

Real-time push, just as polling, can have a large negative impact on battery life. First, consider batching messages to avoid waking up the radio. Second, eliminate unnecessary keepalives; an SSE connection is not "dropped" while the radio is idle. For more details, see Eliminate Periodic and Inefficient Data Transfers.

SSE Streaming over TLS §

SSE provides a simple and convenient real-time transport on top of a regular HTTP connection, which makes it simple to deploy on the server and to polyfill on the client. However, existing network middleware, such as proxy servers and firewalls, which are not SSE aware, may still cause problems: intermediaries may choose to buffer the event-stream data, which will translate to increased latency or an outright broken SSE connection.

As a result, if you experience this or similar problems, you may want to consider delivering an SSE event-stream over a TLS connection; see Proxies, Intermediaries, TLS, and New Protocols on the Web.

« Back to the Table of Contents