Stream Updates with Server-Sent Events

By Eric Bidelman

Published: November 30th, 2010
Comments: 3

Introduction

I wouldn't be surprised if you've stumbled on this article wondering, "What the heck are Server-Sent Events (SSEs)?" Many people have never heard of them, and rightfully so. Over the years, the specification has seen significant changes, and the API has taken somewhat of a backseat to newer, sexier communication protocols such as the WebSocket API. The idea behind SSEs may be familiar: a web app "subscribes" to a stream of updates generated by a server and, whenever a new event occurs, a notification is sent to the client. But to really understand Server-Sent Events, we need to understand the limitations of its AJAX predecessors, which includes:

Polling is a traditional technique used by the vast majority of AJAX applications. The basic idea is that the application repeatedly polls a server for data. If you're familiar with the HTTP protocol, you know that fetching data revolves around a request/response format. The client makes a request and waits for the server to respond with data. If none is available, an empty response is returned. So what's the big deal with polling? Extra polling creates greater HTTP overhead.

Long polling (Hanging GET / COMET) is a slight variation on polling. In long polling, if the server does not have data available, the server holds the request open until new data is made available. Hence, this technique is often referred to as a "Hanging GET". When information becomes available, the server responds, closes the connection, and the process is repeated. The effect is that the server is constantly responding with new data as it becomes available. The shortcoming is that the implementation of such a procedure typically involves hacks such as appending script tags to an 'infinite' iframe. We can do better than hacks!

Server-Sent Events on the other hand, have been designed from the ground up to be efficient. When communicating using SSEs, a server can push data to your app whenever it wants, without the need to make an initial request. In other words, updates can be streamed from server to client as they happen. SSEs open a single unidirectional channel between server and client.

The main difference between Server-Sent Events and long-polling is that SSEs are handled directly by the browser and the user simply has to listen for messages.

Server-Sent Events vs. WebSockets

Why would you choose Server-Sent Events over WebSockets? Good question.

One reason SSEs have been kept in the shadow is because later APIs like WebSockets provide a richer protocol to perform bi-directional, full-duplex communication. Having a two-way channel is more attractive for things like games, messaging apps, and for cases where

you need near real-time updates in both directions. However, in some scenarios *data doesn't need to be sent from the client*. You simply need updates from some server action. A few examples would be friends' status updates, stock tickers, news feeds, or other automated data push mechanisms (e.g. updating a client-side Web SQL Database or IndexedDB object store). If you'll need to send data to a server, `XMLHttpRequest` is always a friend.

SSEs are sent over traditional HTTP. That means they *do not require a special protocol or server implementation* to get working. WebSockets on the other hand, require full-duplex connections and new Web Socket servers to handle the protocol. In addition, Server-Sent Events have a variety of features that WebSockets lack by design such as *automatic reconnection*, *event IDs*, and the ability to *send arbitrary events*.

JavaScript API

To subscribe to an event stream, create an `EventSource` object and pass it the URL of your stream:

```
if (!!window.EventSource) {
  var source = new EventSource('stream.php');
} else {
  // Result to xhr polling :(
}
```

Note: If the URL passed to the `EventSource` constructor is an absolute URL, its origin (scheme, domain, port) must match that of the calling page.

Next, set up a handler for the `message` event. You can optionally listen for `open` and `error`:

```
source.addEventListener('message', function(e) {
  console.log(e.data);
}, false);

source.addEventListener('open', function(e) {
  // Connection was opened.
}, false);

source.addEventListener('error', function(e) {
  if (e.readyState == EventSource.CLOSED) {
    // Connection was closed.
  }
}, false);
```

When updates are pushed from the server, the `onmessage` handler fires and new data is be available in its `e.data` property. The magical part is that whenever the connection is closed, the browser will automatically reconnect to the source after ~3 seconds. Your server implementation can even have control over this reconnection timeout. See Controlling the reconnection-timeout in the next section.

That's it. Your client is now ready to process events from `stream.php`.

Event Stream Format

Sending an event stream from the source is a matter of constructing a plaintext response, served with a `text/event-stream` Content-Type, that follows the SSE format. In its basic

form, the response should contain a "`data:`" line, followed by your message, followed by two "\n" characters to end the stream:

```
data: My message\n\n
```

Multiline Data

If your message is longer, you can break it up by using multiple "`data:`" lines. Two or more consecutive lines beginning with "`data:`" will be treated as a single piece of data, meaning only one `message` event will be fired. Each line should end in a single "\n" (except for the last, which should end with two). The result passed to your `message` handler is a single string concatenated by newline characters. For example:

```
data: first line\n
data: second line\n\n
```

will produce "first line\nsecond line" in `e.data`. One could then use `e.data.split('\n').join('')` to reconstruct the message sans "\n" characters.

Send JSON Data

Using multiple lines makes it easy to send JSON without breaking syntax:

```
data: {\n
data: "msg": "hello world",\n
data: "id": 12345\n
data: }\n\n
```

and possible client-side code to handle that stream:

```
source.addEventListener('message', function(e) {
  var data = JSON.parse(e.data);
  console.log(data.id, data.msg);
}, false);
```

Associating an ID with an Event

You can send a unique id with an stream event by including a line starting with "`id:`":

```
id: 12345\n
data: GOOG\n
data: 556\n\n
```

Setting an ID lets the browser keep track of the last event fired so that if, the connection to the server is dropped, a special HTTP header (`Last-Event-ID`) is set with the new request. This lets the browser determine which event is appropriate to fire. The `message` event contains a `e.lastEventId` property.

Controlling the Reconnection-timeout

The browser attempts to reconnect to the source roughly 3 seconds after each connection is closed. You can change that timeout by including a line beginning with "`retry:`", followed by the number of milliseconds to wait before trying to reconnect.

The following example attempts a reconnect after 10 seconds:

```
retry: 10000\n
data: hello world\n\n
```

Specifying an event name

A single event source can generate different types events by including an event name. If a line beginning with "`event:`" is present, followed by a unique name for the event, the event is associated with that name. On the client, an event listener can be setup to listen to that particular event.

For example, the following server output sends three types of events, a generic 'message' event, 'userlogon', and 'update' event:

```
data: {"msg": "First message"}\n\n
event: userlogon\n
data: {"username": "John123"}\n\n
event: update\n
data: {"username": "John123", "emotion": "happy"}\n\n
```

With event listeners setup on the client:

```
source.addEventListener('message', function(e) {
  var data = JSON.parse(e.data);
  console.log(data.msg);
}, false);

source.addEventListener('userlogon', function(e) {
  var data = JSON.parse(e.data);
  console.log('User login:' + data.username);
}, false);

source.addEventListener('update', function(e) {
  var data = JSON.parse(e.data);
  console.log(data.username + ' is now ' + data.emotion);
}, false);
```

Server Examples

A simple server implementation in PHP:

```
<?php
header('Content-Type: text/event-stream');
header('Cache-Control: no-cache'); // recommended to prevent caching
of event data.

/**
 * Constructs the SSE data format and flushes that data to the client.
 *
 * @param string $id Timestamp/id of this connection.
 * @param string $msg Line of text that should be transmitted.
 */
```

```
function sendMsg($id, $msg) {
  echo "id: $id" . PHP_EOL;
  echo "data: $msg" . PHP_EOL;
  echo PHP_EOL;
  ob_flush();
  flush();
}

$serverTime = time();

sendMsg($serverTime, 'server time: ' . date("h:i:s", time()));
```

Download the code

Here's a similiar implementation using Node JS:

```
var http = require('http');
var sys = require('sys');
var fs = require('fs');

http.createServer(function(req, res) {
  //debugHeaders(req);

  if (req.headers.accept && req.headers.accept == 'text/event-stream')
{
    if (req.url == '/events') {
      sendSSE(req, res);
    } else {
      res.writeHead(404);
      res.end();
    }
  } else {
    res.writeHead(200, {'Content-Type': 'text/html'});
    res.write(fs.readFileSync(__dirname + '/sse-node.html'));
    res.end();
  }
}).listen(8000);

function sendSSE(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
  });

  var id = (new Date()).toLocaleTimeString();

  // Sends a SSE every 5 seconds on a single connection.
  setInterval(function() {
    constructSSE(res, id, (new Date()).toLocaleTimeString());
  }, 5000);

  constructSSE(res, id, (new Date()).toLocaleTimeString());
}

function constructSSE(res, id, data) {
  res.write('id: ' + id + '\n');
  res.write("data: " + data + '\n\n');
}

function debugHeaders(req) {
  sys.puts('URL: ' + req.url);
  for (var key in req.headers) {
    sys.puts(key + ': ' + req.headers[key]);
```

```
  }
  sys.puts('\n\n');
}
```

Download the code

sse-node.html:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
</head>
<body>
  <script>
    var source = new EventSource('/events');
    source.onmessage = function(e) {
      document.body.innerHTML += e.data + '<br>';
    };
  </script>
</body>
</html>
```

Cancel an Event Stream

Normally, the browser auto-reconnects to the event source when the connection is closed, but that behavior can be canceled from either the client or server.

To cancel a stream from the client, simply call:

```
source.close();
```

To cancel a stream from the server, respond with a non "`text/event-stream`" `Content-Type` or return an HTTP status other than `200 OK` (e.g. `404 Not Found`).

Both methods will prevent the browser from re-establishing the connection.

A Word on Security

From the WHATWG's section on Cross-document messaging security:

> *Authors should check the origin attribute to ensure that messages are only accepted from domains that they expect to receive messages from. Otherwise, bugs in the author's message handling code could be exploited by hostile sites.*

So, as an extra level of precaution, be sure to verify `e.origin` in your `message` handler matches your app's origin:

```
source.addEventListener('message', function(e) {
  if (e.origin != 'http://example.com') {
    alert('Origin was not http://example.com');
    return;
  }
```

```
  ...
}, false);
```

Another good idea is to check the integrity of the data you receive:

> *Furthermore, even after checking the origin attribute, authors should also check that the data in question is of the expected format....*

Demo

I've written a demo app in PHP that keeps a clock up to date form the server.

References

- Server-Sent Events specification
- Cross-document messaging security