

## 10 Minutes Tutorial For Server-Sent Events

- by Alex
- .
- 09 Jan 2019
- .
- Topics: Server-Sent Events
- .
- 9664 Views

Except [WebSocket](#), there is another method for the server to push information to the browser: Server-Sent Events (SSE). And I'll introduce its usage in this article.

### 1. What's SSE?

Strictly speaking, the HTTP protocol doesn't allow the server to push information actively. However, there is a workaround: The server declares to the client what will be sent next is streaming.

In other words, what will be sent continuously is a stream of data instead of a one-off packet. At this time, the client won't close the connection, but will wait for the new data stream sent by the server. Video playback is an example of this. Essentially, this kind of communication is to achieve a long-time download by the way of streaming.

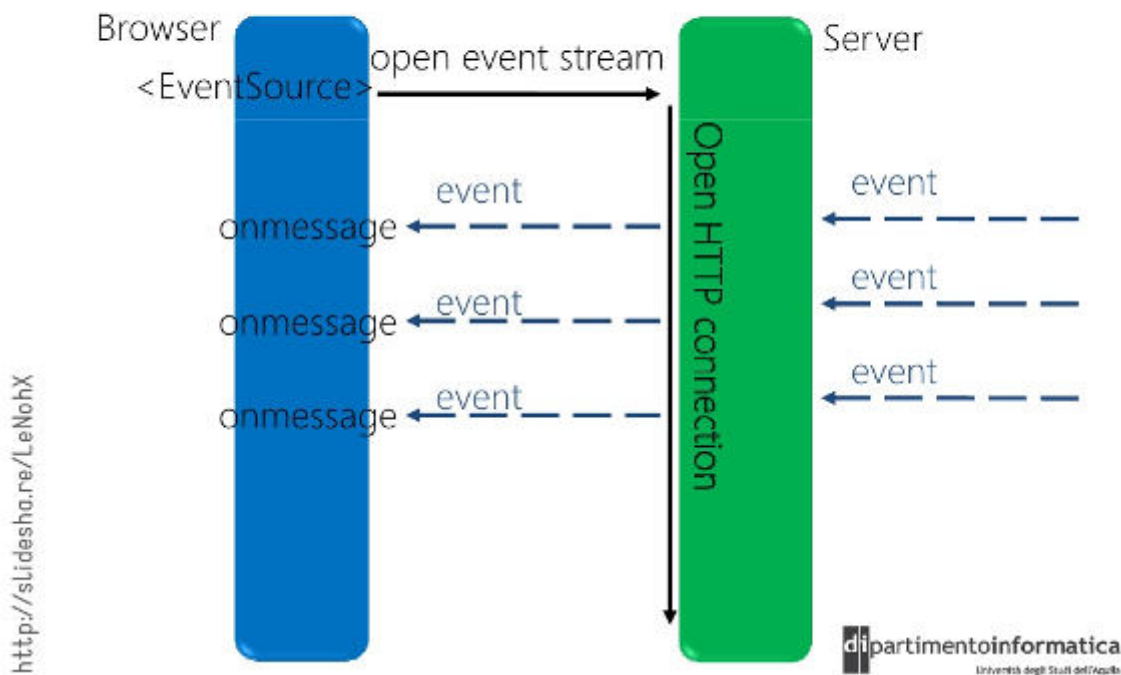
And SSE uses this mechanism to make streaming push information to the browser. It is based on the HTTP protocol and is supported by other browsers currently except for IE/Edge.

### 2. The Features of SSE

SSE is similar to WebSocket for that they both establish a communication channel between the browser and the server, and then the server push information to the browser.

Overall, WebSockets is more powerful and flexible. WebSockets is a full-duplex channel, so it can communicate in both directions; SSE is a one-way channel and can only be sent by the server to the browser, because the streaming is a download essentially. And if the browser sends information to the server, it then becomes another HTTP request.

# Real-time Web? Server-Sent Events



However, SSE has its own advantages.

- SSE uses the HTTP protocol and is supported by all the existing servers; and WebSocket is a standalone protocol.
- SSE is lightweight and easy to use; and the WebSocket protocol is relatively complex.
- SSE supports reconnection by default; and WebSocket needs to be implemented by itself.
- SSE is generally only used to transfer text, and binary data needs to be encoded first and then transmitted; and WebSocket supports binary data transfer by default.
- SSE supports the message types for being customized and then sent.

Therefore, each has its own characteristics and is suitable for different scenarios.

## 3. Client API

### 3.1 EventSource object

The SSE client API is deployed on the EventSource object. The following code can be used to detect if the browser supports SSE.

```
if ('EventSource' in window) {
  // ...
}
```

When using SSE, the browser will generate an instance of EventSource first to initiate a connection to the server.

```
var source = new EventSource(url);
```

The above url can be in the same domain as the current URL, or it can cross-domain as well. When crossing-domain, you can specify a second parameter and open the withCredentials property, which is used to indicate whether to send the Cookie together.

```
var source = new EventSource(url, { withCredentials: true });
```

The `readyState` property of the `EventSource` instance is used to indicate the current state of the connection. This property is read-only and the values for the property are as follows.

- 0: is equivalent to the constant `EventSource.CONNECTING`, indicating that the connection has not been established, or the disconnection is reconnecting.
- 1: is equivalent to the constant `EventSource.OPEN`, indicating that the connection has been established and you can send data now.
- 2: is equivalent to the constant `EventSource.CLOSED`, indicating that the connection has been broken and will not be reconnected.

## 3.2 Basic usage

Once the connection is established, the `open` event will be triggered and you can define the callback function in the `onopen` property.

```
source.onopen = function (event) {  
    // ...  
};  
  
// another way  
source.addEventListener('open', function (event) {  
    // ...  
}, false);
```

When the client receives the data sent by the server, it will trigger the `message` event, and you can define the callback function in the `onmessage` property.

```
source.onmessage = function (event) {  
    var data = event.data;  
    // handle message  
};  
  
// another way  
source.addEventListener('message', function (event) {  
    var data = event.data;  
    // handle message  
}, false);
```

In the above code, the `data` property of the event object is the data (in text format) returned by the server.

If a communication error occurs (such as the connection break), the `error` event will be triggered, and the callback function can be defined in the `onerror` property.

```
source.onerror = function (event) {  
    // handle error event  
};  
  
// another way  
source.addEventListener('error', function (event) {  
    // handle error event  
}, false);
```

The `close` method is used to close the SSE connection.

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

## 3.3 Custom events

By default, the data sent by the server will always trigger the message event of the EventSource instance. However, developers can customize SSE events, in which case the data sent back will not trigger the message event.

```
source.addEventListener('foo', function (event) {  
    var data = event.data;  
    // handle message  
}, false);
```

The browser listens to the foo event in the above code.

## 4. How to Implement The Server

### 4.1 Data format

The SSE data sent by the server to the browser must be UTF-8 encoded text with the following HTTP header.

```
Content-Type: text/event-stream  
Cache-Control: no-cache  
Connection: keep-alive
```

Among the above three lines, the Content-Type of the first line should specify the MIME type as event-stream.

The information sent each time consists of several message, and each message is separated by `\n\n`. Each message is composed of several lines of code internally, and each line should be written as follows.

```
[field]: value\n
```

The above field can take the following four values.

- data
- event
- id
- retry

In addition, you can also have a line beginning with a colon to indicate this is a comment. Generally, the server will send a comment to the browser at regular intervals, keeping the connection uninterrupted.

```
: This is a comment
```

Here is an example.

```
: this is a test stream\n\n  
data: some text\n\n  
data: another message\n  
data: with two lines \n\n
```

### 4.2 data

The data field is represented for data.

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

If the data is long, it can be divided into multiple lines, and the last line should be ended with `\n\n`, and the previous lines ended with `\n`.

```
data: begin message\n
data: continue message\n\n
```

Here is an example of sending JSON data.

```
data: {\n
data: "foo": "bar",\n
data: "baz", 555\n
data: }\n\n
```

## 4.3 id

The `id` field is represented for data identifier, which is equivalent to the serial number of each piece of data.

```
id: msg1\n
data: message\n\n
```

The browser reads the value with the `lastEventId` property. Once the connection is broken, the browser will send an HTTP header containing a special `Last-Event-ID` header which will send the value back to help the server re-establish the connection. Therefore, the header can be considered as a synchronization mechanism.

## 4.4 event

The `event` field is represented for a custom event type, and it's the `message` event by default. The browser can listen for the event with `addEventListener()`.

```
event: foo\n
data: a foo event\n\n

data: an unnamed event\n\n

event: bar\n
data: a bar event\n\n
```

It creates three pieces of information in the above code. The name for the first one is `foo`, triggering the browser's `foo` event; the second one is `unnamed`, indicating it is the default type, and it will trigger the browser's `message` event; the third is `bar`, triggering the browser's `bar` event.

Here's another example.

```
event: userconnect
data: {"username": "bobby", "time": "02:33:48"}

event: usermessage
data: {"username": "bobby", "time": "02:34:11", "text": "Hi everyone."}

event: userdisconnect
data: {"username": "bobby", "time": "02:34:23"}

event: usermessage
data: {"username": "sean", "time": "02:34:36", "text": "Bye, bobby."}
```

## 4.5 retry

The server can use the `retry` field to specify the interval at which the browser re-initiates the connection.

```
retry: 10000\n
```

There are two situations causing the browser to re-initiate the connection: one is the expiration for the time interval, and the other is the connection error due to a network error or the like.

## 5. A Node Server Instance

SSE requires the server to stay connected to the browser. The resources consumed are different for different server software. For the Apache server, each connection is a thread, so if you want to maintain a large number of connections, you have to consume a lot of resources definitely. For Node, all connections use the same thread, so the resources consumed are much smaller, but it requires that each connection can not contain time-consuming operations, such as IO operations on disk.

Here is the SSE server instance for Node.

```
var http = require("http");

http.createServer(function (req, res) {
  var fileName = "." + req.url;

  if (fileName === "./stream") {
    res.writeHead(200, {
      "Content-Type": "text/event-stream",
      "Cache-Control": "no-cache",
      "Connection": "keep-alive",
      "Access-Control-Allow-Origin": '*',
    });
    res.write("retry: 10000\n");
    res.write("event: connecttime\n");
    res.write("data: " + (new Date()) + "\n\n");
    res.write("data: " + (new Date()) + "\n\n");

    interval = setInterval(function () {
      res.write("data: " + (new Date()) + "\n\n");
    }, 1000);

    req.connection.addListener("close", function () {
      clearInterval(interval);
    }, false);
  }
}).listen(8844, "127.0.0.1");
```

Save the above code as `server.js` and execute the following command.

```
$ node server.js
```

The above command will open an HTTP service on port 8844 of the machine.

Then, [open the page](#) and you can have a look at the client code and run it.

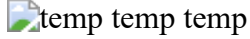
## 6. Reference

- Colin Ihrig, [Implementing Push Technology Using Server-Sent Events](#)
- Colin Ihrig, [The Server Side of Server-Sent Events](#)
- Eric Bidelman, [Stream Updates with Server-Sent Events](#)
- MDN, [Using server-sent events](#)
- Segment.io, [Server-Sent Events: The simplest realtime browser spec](#)

Share To :

## 2 Comments

temp



useful tutorial. thanks



sxwgf 10 Feb 2019

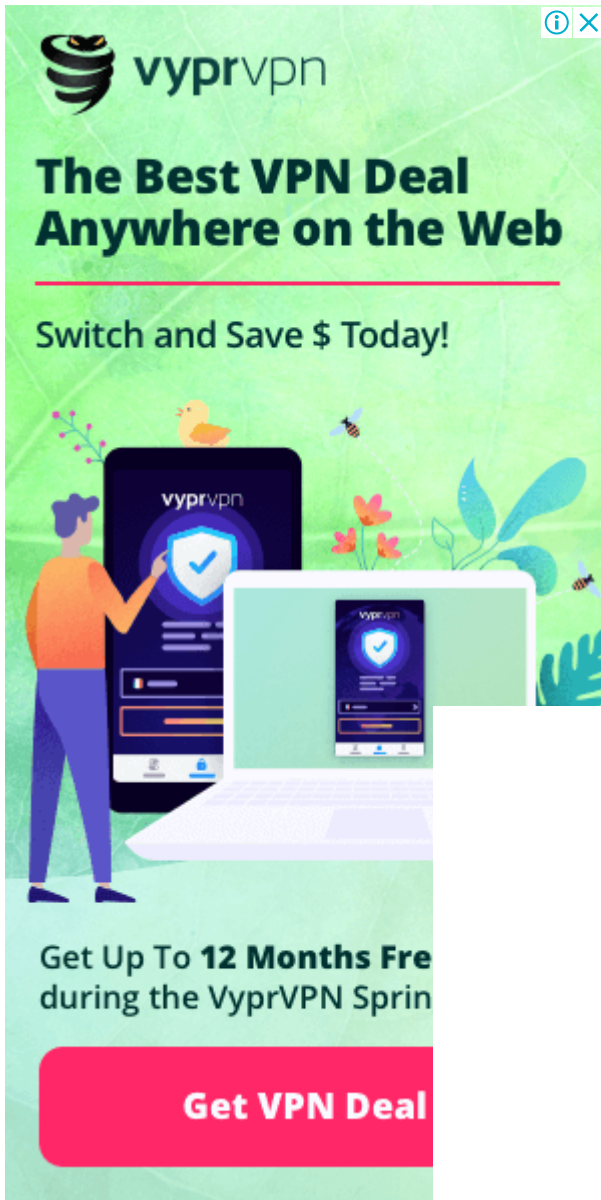
I think this is the most important part of the SSE - *"SSE requires the server to stay connected to the browser. The resources consumed are different for different server software. For the Apache server, each connection is a thread, so if you want to maintain a large number of connections, you have to consume a lot of resources definitely. For Node, all connections use the same thread, so the resources consumed are much smaller, but it requires that each connection can not contain time-consuming operations, such as IO operations on disk."*



ElnurShabanov 08 Nov 2019

Login to post a comment

[Login With Github](#)



The advertisement banner for VyprVPN features a green background with a subtle leaf pattern. At the top left is the VyprVPN logo, and at the top right are information and close buttons. The main headline reads "The Best VPN Deal Anywhere on the Web" in bold black text, followed by a red horizontal line and the sub-headline "Switch and Save \$ Today!". The central illustration shows a person in an orange shirt and blue pants interacting with a large smartphone displaying the VyprVPN app interface, which includes a shield icon and a checkmark. A laptop in the foreground also displays the app. The background is decorated with a small yellow bird, a bee, and various colorful flowers. At the bottom, the text "Get Up To 12 Months Free during the VyprVPN Spring" is displayed above a prominent red button labeled "Get VPN Deal".