Building Real-time Apps with Websockets & Server-Sent Events

By Mark Brown June 06, 2016

This article was peer reviewed by Craig Bilner and Dan Prince. Thanks to all of SitePoint's peer reviewers for making SitePoint content the best it can be!

An important part of writing rich internet applications is reacting to data changes. Consider the following quote by Guillermo Rauch, taken from his 2014 BrazilJS talk, The 7 Principles of Rich Web Applications.

> When data changes on the server, let the clients know without asking. This is a form of performance improvement that frees the user from manual refresh actions (F5, pull to refresh). New challenges: (re)connection management, state reconciliation.

In this article we'll look at examples of how to use the raw WebSocket API as well as the lesser known EventSource for server-sent events (SSE) to build "real-time" UI's that are self-updating. If you're unsure what I mean by that, I recommend watching the video referenced above, or reading th
corresponding blog post

A Brief History

In the past we had to simulate server-push, the most notable method being long polling. This involved the client making a long request that would remain open until the server was ready to push a message. After receiving a message the request would be closed and a new request would b made. Other solutions involved `<iframe>` hacks and Flash. This was not ideal.

Then, in 2006, Opera introduced server-sent events (SSE) from the WHATWG Web Applications 1.0 specification.
SSE allowed you to stream events continuously from your web server to the visitor's browser. Other browsers followed suit and started implementing SSE in 2011 as part of the HTML5 spec.

Things continued to get interesting in 2011 when the WebSocket protocolwas standardised. WebSockets allow you to open a two-way persistent connection between client and server, giving yo the ability to push data back to the clients whenever data changes on the server without the client having to request it. This is hugely important for the responsiveness of an application with a lot of concurrent connections and quickly changing content—a multiplayer online game for example. However, it wasn't until socket.io—the most prominent effort to bring WebSockets to the masses—wa released in 2014 that we saw a lot more experimentation happening with real time communication.

Suffice to say, that today we have much simpler ways of achieving server-push without issuing new requests or relying on non-standard plugins. These technologies give you the ability to stream data back to the client the moment things happen on the server.

WebSockets

The easiest way to understand what a persistent connection allows you to do is to run a working demo, we'll step through the code later but for now download the demo and have a play.

Demo

```
git clone https://github.com/sitepoint-editors/websocket-demo.git
cd websocket-demo
npm install
npm start
```

Open http://localhost:8080/ in multiple browser windows and observe the logs in both the browser an the server to see messages going back and forth. More importantly note the time it takes to receive a message on the server and for the rest of the connected clients to be made aware of the change.

The Client

The `WebSocket` constructor initiates a connection with the server over the `ws` or `wss`(Secure) protocols. It has a `send` method for pushing data to the server and you can provide an `onmessage` handler for receiving data from the server.

Here's an annotated example showing all of the important events:

```
// Open a connection
var socket = new WebSocket('ws://localhost:8081/');

// When a connection is made
socket.onopen = function() {
  console.log('Opened connection ');

  // send data to the server
  var json = JSON.stringify({ message: 'Hello ' });
  socket.send(json);
}

// When data is received
socket.onmessage = function(event) {
  console.log(event.data);
}

// A connection could not be made
socket.onerror = function(event) {
  console.log(event);
}
```

```
// A connection was closed
socket.onclose = function(code, reason) {
  console.log(code, reason);
}

// Close the connection when the window is closed
window.addEventListener('beforeunload', function() {
  socket.close();
});
```

The Server

By far, the most popular Node library for working with WebSockets on the server is ws, we'll use that to simplify things as writing WebSocket serversis not a trivial task.

```
 var WSS = require('ws').Server;

// Start the server
var wss = new WSS({ port: 8081 });

// When a connection is established
wss.on('connection', function(socket) {
  console.log('Opened connection ');

  // Send data back to the client
  var json = JSON.stringify({ message: 'Gotcha' });
  socket.send(json);

  // When data is received
  socket.on('message', function(message) {
    console.log('Received: ' + message);
  });

  // The connection was closed
  socket.on('close', function() {
    console.log('Closed Connection ');
  });

});
```

```
// Every three seconds broadcast "{ message: 'Hello hello!' }" to all connected
var broadcast = function() {
  var json = JSON.stringify({
    message: 'Hello hello!'
  });

  // wss.clients is an array of all connected clients
  wss.clients.forEach(function each(client) {
    client.send(json);
    console.log('Sent: ' + json);
  });
}
setInterval(broadcast, 3000);
```

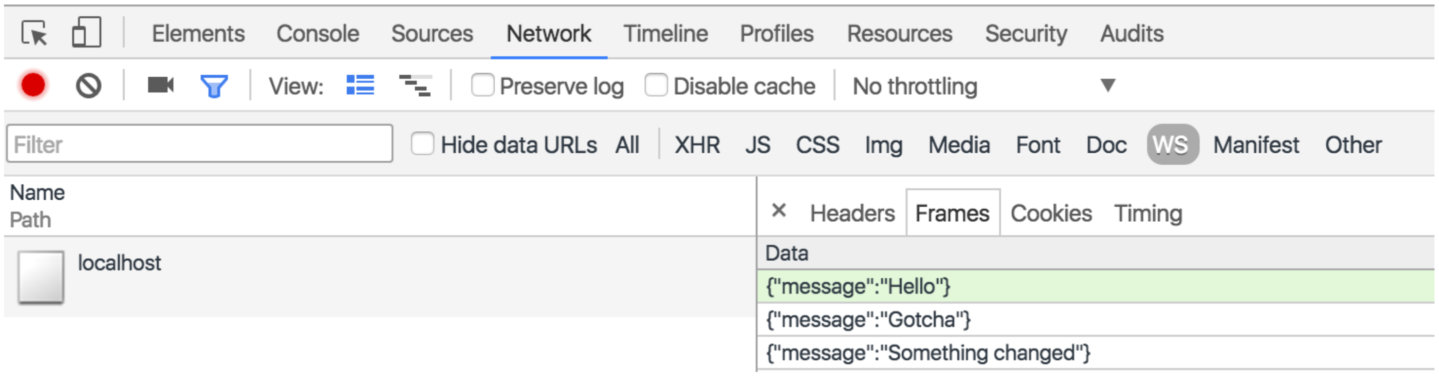
The `ws` package makes building a WebSocket enabled server simple, you should read up on WebSocket Security if you're using them in production though.

Browser Compatibility

Browser support for WebSockets is solid, the exceptions being Opera Mini and IE9 and below, there' a polyfill available for older IE's which uses Flash behind the scenes.

Debugging

In Chrome you can inspect messages sent and received under Network > WS > Frames, sent messages show up in green.



```
{"message":"Hello"}
{"message":"Gotcha"}
{"message":"Something changed"}
```

WebSocket debugging in Firefox is possible using the Websocket Monitor addon for the Firefox Dev Tools. It is developed by the Firebug development team.

Server-Sent Events

Like WebSockets, SSE opens a persistent connection that allows you to send data back to the connected clients the second something is changed on the server. The only caveat is that it doesn't allow messages to go the other direction. That's not really a problem though, we still have good old fashioned Ajax techniques for that.

Demo

```
git clone https://github.com/sitepoint-editors/server-sent-events-demo.git
cd server-sent-events-demo
npm install
npm start
```

As before, open http://localhost:8080/ in multiple browser windows and observe the logs in both the browser and the server to see messages going back and forth.

The Client

The `EventSource` function initiates a connection with the server over good old HTTP or HTTPS. It the server. Here’s an annotated example showing all of the important events.

```
 // Open a connection
var stream = new EventSource("/sse");

// When a connection is made
stream.onopen = function() {
  console.log('Opened connection ');
};

// A connection could not be made
stream.onerror = function (event) {
  console.log(event);
};

// When data is received
stream.onmessage = function (event) {
  console.log(event.data);
};

// A connection was closed
stream.onclose = function(code, reason) {
  console.log(code, reason);
}

// Close the connection when the window is closed
window.addEventListener('beforeunload', function() {
  stream.close();
});
```

The Server

There’s a neat little wrapper sse for creating server-sent events. We’ll use that to simplify things at fir but sending events from the server *is simple enough* to do ourselves so we’ll explain how SSE on th server works later.

```
 var SSE = require('sse');
var http = require('http');
var server = http.createServer();
var clients = [];

server.listen(8080, '127.0.0.1', function() {
  // initialize the /sse route
  var sse = new SSE(server);

  // When a connection is made
  sse.on('connection', function(stream) {
    console.log('Opened connection ');
    clients.push(stream);

    // Send data back to the client
    var json = JSON.stringify({ message: 'Gotcha' });
    stream.send(json);
    console.log('Sent: ' + json);

    // The connection was closed
    stream.on('close', function() {
      clients.splice(clients.indexOf(stream), 1);
      console.log('Closed connection ');
    });
  });
});

// Every three seconds broadcast "{ message: 'Hello hello!' }" to all connected
var broadcast = function() {
  var json = JSON.stringify({ message: 'Hello hello!' });

  clients.forEach(function(stream) {
    stream.send(json);
    console.log('Sent: ' + json);
  });
}
setInterval(broadcast, 3000)
```

Sending Events from the Server

As mentioned above, sending events from the server is simple enough to do ourselves. Here's how:

When a HTTP request comes in from `EventSource` it will have an `Accept`header of `text/event-stream`, we need to respond with headers that keep the HTTP connection alive, then when we are ready to send data back to the client we write data to the `Response` object in a special format `data: <data>\n\n`.

```
http.createServer(function(req, res) {

  // Open a long held http connection
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
  });

  // Send data to the client
  var json = JSON.stringify({ message: 'Hello ' });
  res.write("data: " + json + "\n\n");

}).listen(8000);
```

In addition to the `data` field you can also send event, id and retry fields if you need them e.g.

```
 event: SOMETHING_HAPPENED
data: The thing
id: 123
retry: 300

event: SOMETHING_ELSE_HAPPENED
data: The thing
id: 124
retry: 300
```

Although SSE is wonderfully simple to implement on both the client and the server, as mentioned above, its one caveat is that it doesn't provide a way to send data from the client to the server. Luckil we can already do that with `XMLHttpRequest` or `fetch`. Our new found superpower is to be able to push from the server to the client.

For security, as it's HTTP the standard Cross-Origin rules apply so you should always whitelist origin
on both the server and the client:

```
stream.onmessage = function(event) {
  if (e.origin != 'http://example.com') return;
}
```

Then we can still push to the server as usual with good old Ajax:

```
document.querySelector('#send').addEventListener('click', function(event) {
  var json = JSON.stringify({ message: 'Hey there' });

  var xhr = new XMLHttpRequest();
  xhr.open('POST', '/api', true);
  xhr.setRequestHeader('Content-Type', 'application/json');
  xhr.send(json);

  log('Sent: ' + json);
});
```

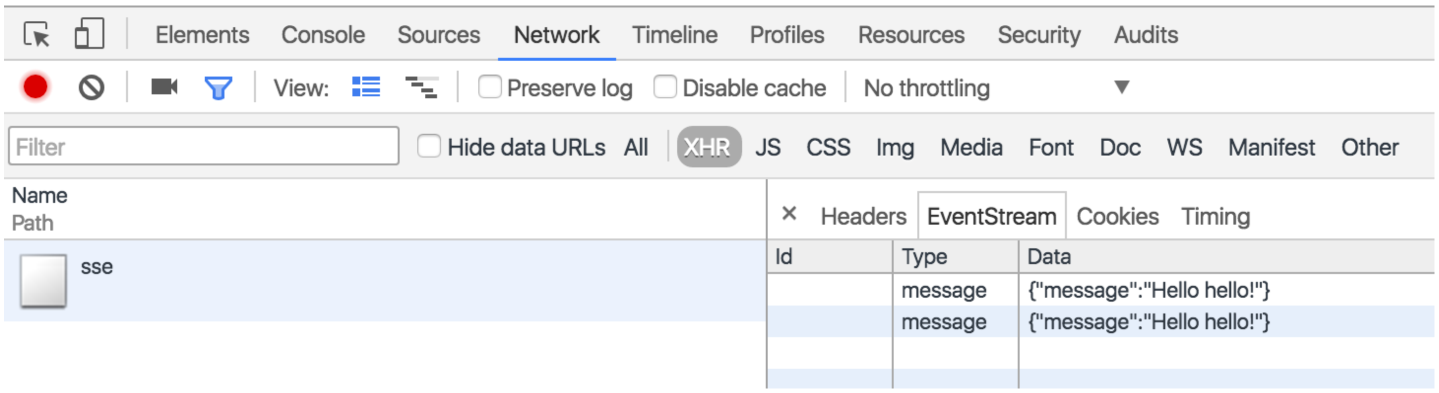
Browser Compatibility

Browser support for SSE is lower than WebSocket due to Microsoft never having shipped a browser that supports it, there is a bug report for it and you should all vote for SSE to help make it a priority fo the next release.

If you need to get SSE working in IE and Edge today you can use a Polyfill for EventSource.

Debugging

In Chrome you can inspect messages received under Network > XHR > EventStream



Id	Type	Data
	message	{"message":"Hello hello!"}
	message	{"message":"Hello hello!"}

Challenges

In Guillermo Rauch's article quoted at the beginning (re)connection management and state reconciliation are mentioned as *new* challenges that these persistent connections have introduced. He's right, you need to think about what should occur when the connection is lost and when it is re-

`EventSource` has a built-in re-connection mechanism, it will attempt to reconnect every 3 seconds if connection is lost automatically. You can test this out in the SSE demo by making a connection in the browser and stopping the server with `Ctrl` + `C`, you'll see errors being logged until you start the server back up again with `npm start`, it keeps calm and carries on.

`WebSocket` doesn't have this ability, if a connection is lost you'll need to create a new one and wire u the events again if you want that same behaviour.

State reconciliation is the practice of synchronising the client with the server when a re-connection occurs. One way to do this is to keep track of the time that a disconnection happened and upon re-connection send all of the events that particular client had missed out on whilst disconnected.

The solutions to these challenges vary depending on what type of app you're building:

- If you're building a multiplayer online game you may need to halt the game until reconnection happens.
- In a Single Page App you may want to start saving changes locally and then send bulk updates to the server on reconnection.
- If you have a traditional app with only a couple of "real-time" pages you may not care if a connection is lost as things will be eventually consistent.

Frameworks

It's fair to say that the era of WebSockets is upon us. No matter what programming language you rur on the server there will be a framework that includes methods for handling persistent connections an broadcasting to connected clients.

- socket.io
- Meteor
- Phoenix – Channels
- Rails 5 – ActionCable

On the client-side these frameworks give you methods for addressing the challenges of (re)connectic management and state reconciliation and give you a simple way to subscribe to different "channels". On the server-side they offer you the pooling of open connections and give you broadcast mechanisms.

When implementing a real-time feature in your app, there's no need to throw away what you know clients can subscribe to, something that would benefit from being updated in real-time. Treat it as a performance improvement for both the client and server, the client is instantly updated the moment something happens and the server doesn't need to respond to the tedious polling:

> Are we there yet? Are we there yet?

Now, the server can respond at the start.

> I'll tell you when we're there

Links

- 7 Principles of Rich Web Applications
- WebSocket
- EventSource
- ws – npm
- sse – npm

Are you using WebSockets or server-sent events in production? Is there a framework I've missed tha
deserves a mention? Be sure to let me know in the comments.

Was this helpful?

More: Eventsource, real-time apps, rich web applications, Server-Sent Events, WebSockets