How to track client-side JavaScript errors

JAVASCRIPT - WEBSOCKET

Published on April 20, 2021 - By Christophe Boulet

This article explains how to use a WebSocket to report the client-side JavaScript errors of a web application, and take advantage of this new channel to monitor the user's activity as well.

Why is it crucial to track client-side errors ?

When most of the code of a web application is executed in the client's browser, it's important to watch for JavaScript errors: most users won't notify the application's owner of an unexpected behavior, and will just go elsewhere. If you work on such web application and don't monitor the client-side errors, you might never hear of these problems. This leads to disappointed and leaving users.

But why only errors ? The channel can be used to send other events to track the user's activity, like which buttons are clicked, which keyboard shortcuts are pressed, time spent on some actions... anything which is significant to understand how the users are interacting with the application. This information will be valuable to improve the product.

Let's look how to use a WebSocket for all this. This article is split into 5 sections:

- Collect JavaScript errors
- Set up a WebSocket - client-side
- Set up a WebSocket - server-side
- Use the WebSocket to track user activity
- Summary

Collect JavaScript errors

The browser triggers a Window: error event when there's an unhandled JavaScript exception. This event provides some useful information about the error : the file, line number and position where the error occurred, and even a stacktrace. This will be of great help to understand and reproduce the error.

```
window.addEventListener('error', function(e) {
  let stacktrace = e.stack;
  if (!stacktrace && e.error) {
    stacktrace = e.error.stack;
  }

  // For now, just print the error
  console.log(e.message + ', ' + e.filename + ', ' + e.lineno + ':' + e.colno)
  if (stacktrace) {
    console.log('Stacktrace: ' + stacktrace);
  }
});
```

The details of the error will be sent through a WebSocket in the next section.
Fun fact: this code catches the JavaScript errors generated by the web application, but also the ones generated by the browser's add-ons ! Fortunately the filename is here to quickly identify the culprit 😃

Set up a WebSocket - client-side

Setting up a WebSocket on the client and sending a message to the server is only 3 JavaScript lines.

```
const socketUrl = 'ws://localhost:3099';
const socket = new WebSocket(socketUrl);
// Here give some time to the WS to get ready
// or use socket.addEventListener('open', ...) to send a message immediately
socket.send('My first message');
```

Let's encapsulate this code into a global `reporter` object and initialize the WebSocket when the page loads. The `reporter.event()` method is called by the error event handler of the first section. It will also be used in the last section to send user activity. The messages are sent in JSON, with an `eventCode` so they can be filtered by the server.

```
const reportUrl = 'TODO'; // Url to the WebSocket server "wss://[host]:[port]"

// The reporter object encapsulating the WebSocket
const reporter = {
  socket: null,

  init: function() {
    this.socket = new WebSocket(reportUrl);
  },

  event: function(eventCode, message) {
    const isReady = this.socket && this.socket.readyState === WebSocket.OPEN;
    if (isReady) { // Messages triggered before the WebSocket is ready are ignored
      this.socket.send(JSON.stringify({eventCode: eventCode, message: message}));
    }
  }
};

// Start reporter immediately
reporter.init();

// Collect unhandled JavaScript errors and send them to the server
window.addEventListener('error', function(e) {
  reporter.event('JAVASCRIPT_ERROR', e.message + ', ' + e.filename + ', ' + e.lineno + ':' + e.colno);

  let stacktrace = e.stack;
  if (!stacktrace && e.error) {
    stacktrace = e.error.stack;
  }
  if (stacktrace) {
    reporter.event('JAVASCRIPT_ERROR_STACKTRACE', stacktrace);
  }
});
```

Why do I set up the WebSocket at page loading instead of waiting for the first event ?
Because the lifetime of the WebSocket will give me the time spent by the user on the web application (the socket is automatically closed by the browser when the user leaves the page). This is part of tracking user

activity, and is also the reason I chose to use a permanent WebSocket instead of sending an individual Ajax request per event.

The above implementation could be improved to handle errors of disconnection/reconnection : if the WebSocket is unexpectedly closed (network failure, long inactivity, server restarted, ...), the next events are lost; They could be stored temporarily until the WebSocket is up again, by retrying connection every 30 seconds.

Set up a WebSocket - server-side

The WebSocket server may or may not be part of the main application. Format Express is built with Ruby on Rails, and I could have use Action Cable for the WebSocket server. Yet I chose to separate it from the main application, and use Node.js. The ws package is a nice implementation of a WebSocket server:

```
const WebSocket = require('ws');   // Install it with "npm install ws"
const wss = new WebSocket.Server({ port: 3099 });

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });
});
```

Now arise the question of what to do with the notifications of JavaScript errors ? There is no right answer, it's up to you to decide which solution will make them noticed (and ultimately fixed): you could log them in a file, store them in a DB, send them to a chat server, email them to the team, ... They should be treated as well as server-side errors, and so join the same workflow.

Let's see a full NodeJS script for the WebSocket server. I chose to write the events in a log file. A `connectionId` is randomly generated to identify all the events belonging to the same connection.

reporter-server.js

```
const { v4: uuidv4 } = require('uuid'); // "npm install uuid"

// Open log file
const fs = require('fs');
const log_file = fs.createWriteStream(__dirname + '/events.log', {flags : 'a'});
function handleEvent(message, connectionId) {
  log_file.write(`${timestamp()} [${connectionId}] ${message}\n`);
}

// Start server
const yargs = require('yargs');  // To read command line arguments
const argv = yargs.option('port', { type: 'number' }).argv;
const port = argv.port || 3099;  // The listening port can be changed via command line
const WebSocket = require('ws');
const wss = new WebSocket.Server({ port: port });

wss.on('connection', function connection(ws) {
  // Handle start of connection
  const startedAt = Date.now();
  const connectionId = uuidv4(); // Random UUID
  handleEvent('START_PAGE_VIEW', connectionId);

  // Handle message
```

```
  ws.on('message', function incoming(message) {
    const json = JSON.parse(message);
    handleEvent(`${json.eventCode} ${json.message}`, connectionId);
  });

  // Handle end of connection
  ws.on('close', function() {
    handleEvent('END_PAGE_VIEW ' + millisecondsToStr(Date.now() - startedAt), connectionId);
  });
  ws.on('error', function() {
    handleEvent('END_PAGE_VIEW [error] ' + millisecondsToStr(Date.now() - startedAt), connectionId);
  });
});

console.log(`WebSocket server running at http://localhost:${port}/`);

// ----- help functions -------

// Current time YYYY-MM-DD HH-MM-SS
function timestamp() {
  const d = new Date();
  return pad(d.getFullYear()) + '-' + pad(d.getMonth() + 1) + '-' + pad(d.getDate()) + ' ' +
         pad(d.getHours()) + ':' + pad(d.getMinutes()) + ':' + pad(d.getSeconds());
}
function pad(n) { return n < 10 ? "0" + n : n}

// Human readable duration: "2 minutes", "34 seconds". From https://stackoverflow.com/a/8212878/1128103
function millisecondsToStr(milliseconds) {
  let temp = Math.floor(milliseconds / 1000);
  const hours = Math.floor((temp %= 86400) / 3600);
  if (hours) {
    return hours + ' hour' + numberEnding(hours);
  }
  const minutes = Math.floor((temp %= 3600) / 60);
  if (minutes) {
    return minutes + ' minute' + numberEnding(minutes);
  }
  const seconds = temp % 60;
  if (seconds) {
    return seconds + ' second' + numberEnding(seconds);
  }
  return 'less than a second';
}
function numberEnding(number) { return (number > 1) ? 's' : ''; }
```

Note: The `connectionId` is generated for each WebSocket, so when a user goes to another page, a new `connectionId` is generated. To track the user activity across multiple pages, the `connectionId` could be stored in a session cookie or a persistent cookie (the latter requires GDPR consent from the user).

Let's have a look at the log file generated for the scenario of a user which encountered an error. At the end of the connection, the total time spent by the user on the page is logged.

```
2021-04-06 08:56:10 [1b9d6b-...] START_SESSION
2021-04-06 08:58:27 [1b9d6b-...] JAVASCRIPT_ERROR TypeError: null is not an object (evaluating 'this.response.size'), ht
2021-04-06 08:58:27 [1b9d6b-...] JAVASCRIPT_ERROR_STACKTRACE getLinesCount@http://localhost:3000/assets/application.js:4
                                                             readResponse@http://localhost:3000/assets/application.js:36
                                                             processFormat@http://localhost:3000/assets/application.js:2
```

```
2021-04-06 08:58:35 [1b9d6b-...] END_SESSION 2 minutes
```

Mission accomplished ! A JavaScript error occurred client-side, and I have all the details of the error server-side to help me fix it. Yet I'm missing some context that may be helpful: *what was the user doing prior to the error ?*

Use the WebSocket to track user activity

In the example above, the user stayed around 2 minutes, yet I have no idea of what he was doing. Aside from the error tracking, it would be great to know which features are used by the users, and which ones are not.

The `reporter` can be used to report any activity of the users. Here are some short generic examples. Each application would have its own specific events to understand the behavior of its users.

Which buttons do they use ?

```
document.querySelectorAll('button').forEach( button => {
  button.addEventListener('click', () => { reporter.event('CLICK_BUTTON', button.id); });
});
```

Are they using keyboard shortcut ?

```
window.addEventListener('keydown', event => {
  if (event.ctrlKey) {
     reporter.event('KEYPRESS', buildKeyPressMessage(event));
  }
});

function buildKeyPressMessage(event) {
  return (event.ctrlKey ? 'Ctrl + ' : '') +
         (event.shiftKey ? 'Shift + ' : '') +
         (event.altKey ? 'Alt + ' : '') +
         event.key + ' [' + event.keyCode + ']';
}
```

How long takes some long task in the user's browser ?

```
const startTime = new Date();
executeTask1();  // long task...
reporter.event('TASK1', 'duration: ' + (new Date() - startTime) + 'ms');
```

Even for this article, right now, you're tracked ! With the following code, I know if you read all the article or just the first 2 sentences, using the `scroll` event and computing percentage of the page scrolled.

```
const READ_PROGRESS_MILESTONES = [10, 25, 50, 75, 90, 100];

function reportReadProgress() {
  const scrollPct = getPageScrollPercentage();
  while (READ_PROGRESS_MILESTONES.length > 0 && READ_PROGRESS_MILESTONES[0] <= scrollPct) {
    reporter.event('READ_PROGRESS', READ_PROGRESS_MILESTONES[0] + '%');
    READ_PROGRESS_MILESTONES.shift();
  }
}
```

```
window.addEventListener('scroll', reportReadProgress);

// ----- help functions -------

// From https://stackoverflow.com/questions/2387136/cross-browser-method-to-determine-vertical-scroll-percentage-in-java
function getPageScrollPercentage() {
  let root;
  if (document.documentElement.scrollTop) {
    root = document.documentElement;
  } else if (document.body.scrollTop) {
    root = document.body;
  }
  return root ? (root.scrollTop / (root.scrollHeight - root.clientHeight) * 100) : 0;
}
```

Summary

In this article, I showed you how to monitor the client-side JavaScript errors, and also track the user activity via a WebSocket. Do not neglect the importance of these reports, particularly for the errors. It's a source of frustration for the users. If they're paying customers or professionals, you may hear sooner or later that there's a problem; If they're just visitors, they're certainly already looking for an alternative. Very few people will take the time to signal an error.

The user activity helps to give some context for the errors (once I could identify an error that occurred only when the search bar was opened, even if at first the problem seemed unrelated). Furthermore tracking the activity helps to understand the behavior of the users, and improve the web application later.
You may have noticed that some metrics could have been retrieved with Google Analytics or an alternative (total time on the page, custom events, ...). On Format Express, I deliberately chose to not include any external resource, and these metrics are a cheap substitute and preserve anonymity.

The WebSocket has been used for communication one-way only: from the browser to the server. It could have been replaced with Ajax requests, and that's totally fine. Yet the WebSocket is a bidirectional channel, and so it may be used also for communication from the server to the clients: for example to ask users to refresh the page when a new version of the web application is available or to make important announcements to the users. That may be treated in another article.