Essay: Large-scale JavaScript error tracking

Julian Nalenz
nalenz@kth.se

May 2, 2022

1 Introduction

With the emergence of JavaScript as a programming language for full-stack web development for both the server side with Node.js and the client side, mechanics for monitoring the successful execution of code are essential. Especially when code execution is not directly observable by the developer, such as in customers' web browsers or on backend servers, a consistent way of representing, storing and analyzing exceptions needs to be found. Having access to good information about occurred exception generally greatly simplifies debugging and thus allows developers to solve issues more rapidly.

But what actually characterizes an exception in JavaScript? Which components does such an object have? What are the different ways of handling exceptions in JavaScript? And how can monitoring exceptions be scaled across large infrastructures?

This essay will first give a general introduction into this field and explain how exceptions in JavaScript work. Afterwards, the tool Sentry [1], used for application monitoring and error tracking, will be discussed and how its functionality solves the challenges developers are confronted with in this domain. Finally, the conclusion provides a short summary and reflection on these topics.

2 The anatomy of JavaScript exceptions

As JavaScript is a language that has been syntactically influenced by other imperative programming languages like C, Java or Python, similarties in exception handling can be found. However, a distinction must be made between synchronous code, where exceptions are propagated upwards the call tree as traditionally known in, for example, Java and asynchronous code, which is based on Promise objects as well as the modern async-await syntax. Promises have their own mechanisms of exception handling, which are explained below. A highly valuable resource in this context is "A study of JavaScript error handling", which delves deeply into the various ubiquitous JavaScript environments and how they approach exception handling.

2.1 Exceptions in synchronous code

The "Error" constructor forms the very first entry point into JavaScript exception handling. Mainly due to backwards compatibility reasons, it has only one parameter which is truly cross-browser and cross-environment compatible: `message`, that simply stands for the error message [2].

```
function test() {
    try {
        throw new Error("This is an error!");
    } catch(e) {
        console.error(e);
    }
}

test();

// Output in Node.js 16.15.0:
// Error: This is an error!
//     at test (/home/user/test.js:3:15)
//     at Object.<anonymous> (/home/user/test.js:9:1)
//     at Module._compile (node:internal/modules/cjs/loader:1105:14)
//     at Object.Module._extensions..js (node:internal/modules/cjs/loader:1159:10)
//     at Module.load (node:internal/modules/cjs/loader:981:32)
//     at Function.Module._load (node:internal/modules/cjs/loader:822:12)
//     at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:77:12)
//     at node:internal/main/run_main_module:17:47
```

Figure 1: Example of synchronously catching an Error.

The code provided in Figure 2 showcases the most basic form of error handling: throwing and catching an exception. As apparent in the exemplary output produced in Node.js 16.15.0, upon construction of an `Error` object, the stack trace associated with it is stored as well. Through this, the programmer can retrace where the exception happened and which function calls led to it.

Additionally, it is possible to extend the Error object, for instance with a syntax construct like `class ValidationError extends Error { ... }`. This allows passing arbitrary additional parameters to a self-defined constructor, for example to enrich an exception with additional helpful information.

2.2 Exceptions in asynchronous code

In general, asynchronous code commonly arises when working with user interfaces, network communication, filesystem I/O and other non-blocking operations. As specified in the respective specification [3], as soon as a `Promise` object is constructed, its execution starts immediately and it enters its *pending* state. When the operation was completed successfully, it eventually enters the *fulfilled* state, otherwise it becomes *rejected*. The lifecycle has been further explained in formalized detail by Loring et al. in [4].

JavaScript's modern `async-await` syntax provides a useful abstraction for this, so that it is no longer necessary to rely on callbacks. Promise rejections can be then be caught with a `try-catch` construct, as if the code would be running

synchronously. However, if such a rejection is not caught, environments like Node.js detect this and report an `UnhandledPromiseRejection` [5]:

```
function sleep(ms, fail) {
    return new Promise((resolve, reject) => {
        setTimeout(fail ? reject : resolve, ms);
    });
}

async function test() {
    console.log("start");
    await sleep(1000, false);
    console.log("sleep␣1␣finished");
    await sleep(1000, true);
}

test();

// Output in Node.js 16.15.0:
// start
// sleep 1 finished
// node:internal/process/promises:279
//             triggerUncaughtException(err, true /* fromPromise */);
//             ^
//
// [UnhandledPromiseRejection: This error originated either by throwing inside
// of an async function without a catch block, or by rejecting a promise which
// was not handled with .catch(). The promise rejected with the reason "undefined".] {
//   code: 'ERR_UNHANDLED_REJECTION'
// }
```

Figure 2: Example of an `UnhandledPromiseRejection` in Node.js.

A structured approach to finding broken promises in asynchronous JavaScript programs has been further investigated by Alimadadi et al. [6].

3 Sentry

As its core functionality, the software `Sentry` focuses on automated error reporting. Most importantly, this means that unhandled exceptions, both synchronous and asynchronous, are detected automatically. Following this, Sentry collects all available information about the error (message, stack trace etc.) as well as some meta information about the user or environment (browser User-Agent, IP address etc.) and sends this to a previously defined Sentry instance. Sentry can be run both as Software as s Service (hosted by the company behind it) and fully self-hosted.

In [7], Santos explains how global error handling works. Most importantly, JavaScript environments provide different mechanisms to globally catch all *unhandled* exceptions, for instance by assigning a callback to `window.onerror` in client-side JavaScript or using `process.on('uncaughtException', ...)` in Node.js.

When now setting up Sentry for Node.js as described in its documentation and replacing the line `console.error(e)` by `Sentry.captureException()` in

Figure 2, that error is reported in the Sentry "Issues" frontend [8] as showcased in Figure 3.

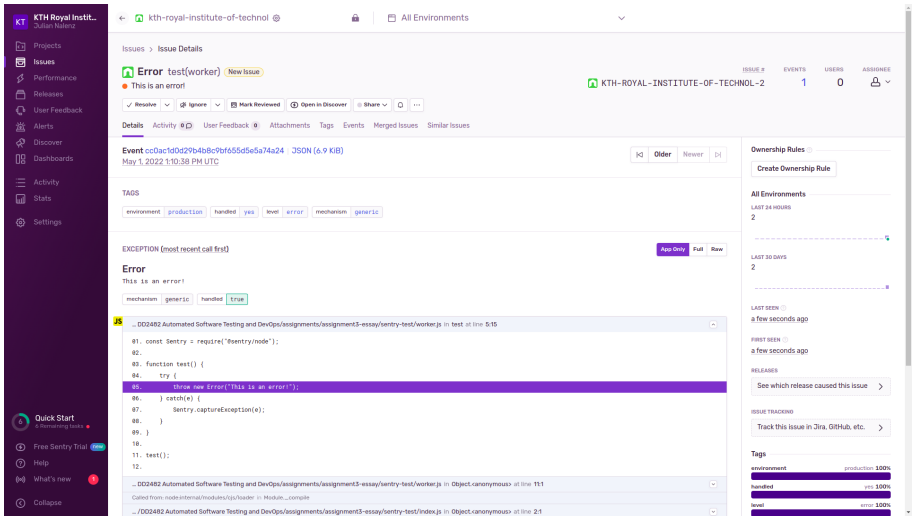

Figure 3: Example of an exception reported via `Sentry.captureException`.

In this case, Sentry is even able to automatically read out the source code of the relevant file that caused the exception. In case of a transpiled application like commonly found in the context of React, in order to be able to demangle the minified source, Sentry also supports uploading source maps [9] [10].

3.1 Exception fingerprinting

Sentry documents this topic unter "Issue Grouping", where it is introduced with "A fingerprint is a way to uniquely identify an error, and all events have one. Events with the same fingerprint are grouped together into an issue." [11]. Most notably, Sentry uses the following properties of an exception to do the grouping:

- By default, Sentry primarily looks at the stack trace. Conceptually, this means that when two errors originated in the same place, they will be grouped together. Sentry then shows only one entry in the "Issues" dashboard and assigns both events to this entry.
- If no stack trace is available, Sentry looks at the reported exception object itself. This is normally irrelevant for JavaScript, because its `Error` objects always contain a stack trace.
- If not even a more detailed exception object is available, Sentry simply considers the reported error message itself.

Furthermore, Sentry allows changing the concrete rules for which stack traces are grouped together, the so-called *fingerprint rules*. For instance, grouping events together is possible, even if their stack traces are different, so that two separate exception names can be treated as the same. Additionally, matching on log messages, stack frame path or module, log level and many other properties is possible.

3.2 Integrations

Another powerful feature of Sentry are integrations into other tools. For instance, Sentry can be integrated into Slack, so that an automated message is sent by a bot whenever a new event occurs. An example of this can be seen in Figure 4.

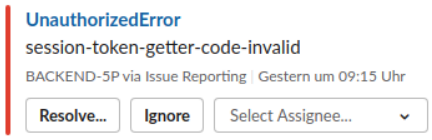

Figure 4: An automated message sent by Sentry into a Slack channel due to a new exception. From this interface, developers can interact with the issue without leaving the Slack UI.

As documented at [12], the various available Sentry integrations can be grouped into the following categories:

- notification & indicents (e.g. sending an automated message when a new type of event occurs)
- source code management and issue tracking (e.g. automatically creating GitHub issues from reported exceptions)
- deployment (e.g. automatically upload source maps of a React app to Sentry or become able to identify commits that caused a new error)
- data visualization (e.g. visualize Sentry issues and statistics in Grafana)
- cloud monitoring (e.g. add Sentry automatically to AWS Lambda functions)

3.3 Distributed tracing

Modern web application often consist of many more services than just a frontend and a backend server. For example, applications often have a database server and a separate authentication server or moving certain parts of an application's backend into different microservices are also viable options. One advantage of such a microservices architecture is improved separation of concerns, so that

it becomes clearer and more lightweight who in a company is responsible for which functionality and to avoid large, monolithic repositories. However, when processing an end user's request requires working with several different internal microservices, there are many more possible points of failure. To be able to retrace where exactly an error happened, methods that can be used in different frameworks and programming languages can be used must be found.

As defined in [13]: "Distributed tracing is a monitoring technique that links the operations and requests occurring between multiple services. This allows developers to "trace" the path of an end-to-end request as it moves from one service to another, letting them pinpoint errors or performance bottlenecks in individual services that are negatively affecting the overall system." Traces consist of *spans*, which describe the individual operations a trace consists of. To solve the issue of retraceability, a so-called `trace_id` can be assigned to each request, so that issues happening in different services can be grouped together [13].

Sentry also supports distributed tracing via this method, for both exception and performance monitoring [14].

4 Conclusion and reflection

Sentry has a broad scope of programming languages and error reporting, monitoring and tracing functionality and can thus be seen as a highly valuable choice for developers, so that they get more transparency about crashes in real-life environments, such as directly at the end customer.

Beyond pure, factual error reporting, educating developers on how exceptions are handled correctly is just as important. Further suggested reading includes [7], who also discusses how developers often deal with errors in JavaScript applications as well as various related research questions that relate to analyzing real-life production source code. Moreover, Hsieh et al. provide a comprehensible list of common exception handling "code smells", i.e. bad practices like ignoring exceptions, no global uncaught exception handler, throwing symbols instead of error objects or dummy handlers, as well as appropriate refactorings [15].

References

[1] Functional Software, Inc. *Application Monitoring and Error Tracking Software*. URL: `https://sentry.io/welcome/` (visited on 04/30/2022).

[2] MDN Web Docs. *Error - JavaScript*. URL: `https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Error` (visited on 04/30/2022).

[3] ECMAScript 2023 Language Specification. *Promise Objects*. URL: `https://tc39.es/ecma262/multipage/control-abstraction-objects.html#sec-promise-objects` (visited on 05/01/2022).

[4] M. C. Loring, M. Marron, and D. Leijen. "Semantics of Asynchronous JavaScript". In: *SIGPLAN Not.* 52.11 (Oct. 2017), pp. 51–62. ISSN: 0362-1340. DOI: 10.1145/3170472.3133846.

[5] V. Karpov. *Unhandled Promise Rejections in Node.js*. Apr. 4, 2017. URL: https://thecodebarbarian.com/unhandled-promise-rejections-in-node.js.html (visited on 04/30/2022).

[6] S. Alimadadi, D. Zhong, M. Madsen, and F. Tip. "Finding Broken Promises in Asynchronous JavaScript Programs". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276532.

[7] L. M. dos Santos. "A study of JavaScript error handling". PhD thesis. Universidade Federal de Pernambuco, Feb. 14, 2019.

[8] Sentry Documentation. *Issues*. URL: https://docs.sentry.io/product/issues/ (visited on 04/30/2022).

[9] Sentry Documentation. *Source Maps for React*. URL: https://docs.sentry.io/platforms/javascript/guides/react/sourcemaps/ (visited on 05/01/2022).

[10] J. Lenz and N. Fitzgerald. *Source Map Revision 3 Proposal*. URL: https://sourcemaps.info/spec.html (visited on 04/30/2022).

[11] Sentry Documentation. *Issue Grouping*. URL: https://docs.sentry.io/product/data-management-settings/event-grouping/ (visited on 04/30/2022).

[12] Sentry Documentation. *Integrations*. URL: https://docs.sentry.io/product/integrations/ (visited on 04/30/2022).

[13] B. Vinegar. *Distributed Tracing 101 for Full Stack Developers*. Aug. 12, 2021. URL: https://blog.sentry.io/2021/08/12/distributed-tracing-101-for-full-stack-developers (visited on 04/30/2022).

[14] Sentry Documentation. *Distributed Tracing and the Root of Every Problem*. URL: https://sentry.io/features/distributed-tracing/ (visited on 05/02/2022).

[15] C.-Y. Hsieh, C. Le My, K. T. Ho, and Y. C. Cheng. "Identification and refactoring of exception handling code smells in JavaScript". In: *Journal of Internet Technology* 18.6 (2017), pp. 1461–1471.