The Ultimate Guide to

JavaScript Error Monitoring

RAYGUN

Table of Contents

The Ultimate Guide to JavaScript Error Monitoring by Raygun

Published by Raygun.com - www.raygun.com

SECTION 1

Introduction to JavaScript error monitoring

JavaScript, as the default scripting language for websites and web applications in the browser, is an important part of modern software development. Due to its history and the open nature of the web, client browsers can vary wildly with their implementations of basic JavaScript APIs, even for modern user-agents (let alone legacy versions). As application code can frequently involve critical business logic including purchasing and handling sensitive data, or must provide a first-class user experience where UI jank and latency need to be minimized as much as possible, deploying robust client-side JavaScript code is a necessity for today's web.

This, combined with the fact that widely-supported JavaScript (ECMAScript 5) as a language itself is commonly considered to contain several gotchas that don't assist in the writing of bug-free code, software developers need all the help they can get to ensure that no bugs make it through to production websites/applications. As is common in frameworks and runtimes, most browsers fortunately implement the **window.onerror** handler which receives an Error object. With this, an error tracking tool like *Raygun Crash Reporting* can package up the error and send it to a dashboard, where multiple occurrences of each error are grouped together to minimize noise and allow developers to fix bugs with a sane workflow.

Further to this, as modern ('evergreen') browsers provide additional data such as column numbers and source maps, tools such as *Raygun* can use these to transform production minified stack traces, producing the effect of

having debugger tools attached locally, but for every error that occurs in end user's browsers, every time they use your website/app.

As the browser is an unknown environment, monitoring JavaScript errors in your code should also take into consideration the effect of extensions and security policies such as CORS that may disrupt or remove data for errors that occur. These concerns will be discussed in detail in this guide.

The popularity of client-side frameworks such as Angular, Ember, React and others, has provided opportunities for structure and guarantees previously unavailable with lower-level libraries. Various approaches and code snippets are available throughout this guide to aid in the setup of JavaScript error monitoring for these frameworks.

SECTION 2

What is an error and how do you create one?

The **Error** object in JavaScript is at the core of raising and handling errors. Instances of the Error object are created and thrown when runtime errors occur in your code.

To create a new error in JavaScript, instantiate one with a string message, then throw it with the **throw** keyword:

```
throw new Error('My custom error');
```

Besides the generic Error object, there are six other core error types that can be used in the same way as the generic Error object.

RangeError

The **RangeError** is thrown when a number is used as an argument to a function where the number is outside of a valid range. This can be encountered when attempting to create an array of an illegal length, or when passing a bad value to methods such as Number.toFixed(). For example, this method expects a number from 0 to 20. A number outside of this range will cause a RangeError.

ReferenceError

The **ReferenceError** is thrown when a non-existent (or misspelled) variable

is referenced. In the example below we try to call a method on a variable that does not exist. Note that the variable “brain” does not exist in the current or global scope.

```
function openEyes(){
brain.trigger(‘eyes.open’);
}
```

SyntaxError

A **SyntaxError** is thrown when the JavaScript engine encounters malformed JavaScript code. This type of error is only found when the code is executed and is unique in that you cannot recover from this type of error. The following example throws a SyntaxError due to the missing function name before the function parameters.

```
function (brain) {
brain.trigger(‘eyes.open’);
}
```

TypeError

A **TypeError** is thrown when a value is encountered that is not of the expected type. A common cause of this error is trying to call a method on an object that does not exist. In the example below we try to call the trigger method on the brain object, but we have not defined a trigger method on this object.

```
var brain = {};
brain.trigger(‘eyes.open’);
```

URIError

A **URIError** is thrown when global URI handling functions, such as encodeURI() and decodeURI(), are passed a malformed URI.

```
var uri = decodeURI('%');
```

EvalError

The **EvalError** exception is thrown when usage of the eval() function results in an error. This exception is not thrown in recent versions of JavaScript, however the EvalError object remains for compatibility.

SECTION 3

Why don’t users report software errors and crashes?

Remember when you used to encounter those Windows pop up messages each time your computer crashed? The ones synonymous with the Windows operating system were regularly popping up every time things imploded. You lost your work, smashed your keyboard and had a good cry. Just me?

If I click to ‘send this error report’, what actually happens to it? I’m pretty sure it’ll go one of two ways.

What I think will happen when I click on ‘Send Error Report’

A friendly employee will get notified of my error and pass this information onto an eagerly awaiting development team who will take this problem extremely seriously. Next up I’ll get an email from these lovely people. They’ll apologize, tell me not to worry, I am important to them. They make me feel amazing. I’m awash with kind thoughts and positive things to say about their company and their customer service. I’ll never use anyone else. They care about me and I actually helped them make better software by reporting this issue!

What actually happens when I click on ‘Send Error Report’

Nothing.

This behaviour and expectation has become embedded in software user’s

minds for many years. Usual responses from people when they encounter software errors and crashes are:
What's the point of submitting a bug report? They never go anywhere.

- That's a hassle, I can't be bothered. They don't care.
- I don't have time to explain it all. They will just ignore it.

Usually this is the case for the developer too. Why should they spend their time going back and forth with customers over email to get screenshots, operating systems, browsers and associated versions, all to end up with a bug they can't replicate!

What could happen in the future (right now actually)

By plugging an automatic error monitoring solution like Raygun into your application, you don't need the user to report the issue. Raygun monitors your software for problems and notifies you in real time, so without the need for the user to do anything, you can see who was affected, what browser they were using, what operating system and even what line of code the error happened on!

You can even reach out to the user from the Raygun app to say 'Hey, I saw you had that issue yesterday, we just wanted to let you know that we're sorry about that and have fixed the issue now.' See? It's the future, but right now.

Real world, real problems

So I'll tell you a story, a real world example of this phenomenon. About six months ago the management at my partner's school (where she is a teacher)

implemented a new software system to create, collaborate and send out reports to parents.

It was designed to take the hassle out of report writing and showed great promise. The school was perhaps their tenth or so school to implement the system, so it was kind of in a Beta format and they were encouraged to give feedback.

All was good until bugs started appearing. They encountered several big issues with the system ranging from photos disappearing, posts not saving, clicking save and things disappearing, inviting users to the account and no invitation emails showing up – you get the picture.

This went on for months and I was hearing of all these frustrations and problems with the system.

'Have you reported these problems?", I asked. "Well, no, actually. What's the point?"

It seemed amazing to me that these people were having all these problems yet never took the step to report the problems to the developer, and it turns out this is far more common than you'd think.

Here at Raygun, we obviously get real time error reports from our own software. We have found that only 1% of users actually report to us the errors that we know they experienced, and these poor teachers (and they can't be the only ones who do this), had found novel ways to navigate around the errors from occurring rather than report them, by navigating

through different screens to do the tasks or saving their writing to their clipboard before hitting the save button just incase it lost it all, rather than report the problems and face the hassle.

The developer on the other hand had no idea that these problems were happening. Absolutely no clue. We found this out when we emailed him to see if he'd like to try Raygun to fix the issues the teachers were having. He was almost taken aback that someone was suggesting the software had errors, and in hindsight you can see how he would be. You won't know about things that are happening in your software if you're not told about them, and you certainly can't expect your users to be the ones to tell you. They won't.

SECTION 4

Writing error messages in JavaScript

In the modern development shop, back and front end developers are expected to do a lot more than they ever did before: tooling, unit testing and even functional testing. At the same time, the release cycles are reduced and the number of feature requests are increasing. Is it possible to maintain quality and performance with all these new burdens?

The answer is yes. It is possible, and some teams are already doing it.

Great tools allow more to be done with less effort, especially when QA and operations teams are transitioned or reduced and the responsibilities of development starts growing quickly. But a tools first approach can quickly become a problem.

Tools solve nothing if the perception of how they will be adopted is wrong. For example, jumping on an analytics tool as a method to track errors is not only a misuse of the technology but has some serious technical and operational limitations. In this case, there will be long delays before errors can be identified, and the developer who needs to use the analytics tool will not have ownership.

Despite this, log analysis tools are often pitched as a way to advance the team, help in unit testing and handle bugs without additional work. Both of these processes take quite a bit of energy. They are at the front end of

development and on the first line of defense for bugs, but often developers don't realize that effort spent here saves time and angry team members at their desk later.

For front-end developers working on new functionality, unit tests are not only annoying, but they also take up brain-cycles. While the premise around test-driven development is fantastic, it's also a mental distraction, requiring the creativity to both code the feature and develop the methods of testing.

Detailed tests should be written for each function/class, but they require a great deal of coding and thinking about test cases. If done correctly, this can be a very complex process because developers naturally have a tendency to write tests so they pass—which obviously drives release managers crazy!

Test driven development (TDD) is not new, but it is as an addition to functional testing. If developers code unit tests, and then selenium code when features are complete, they have now crossed three different activities. Not surprisingly, this may require testing efforts that exceed the amount of time spent on development.

The key to performance in this scenario is to keep both tools and log analysis. Asking developers to do the testing and the QA team to facilitate and automate is extremely beneficial for catching bugs and responding to them more quickly. And by reducing the complexity of the test cases, time will be saved in the long run. One way to reduce complexity is to pick tools that have a great interface, notify you instantly on commits or releases and allow you to quickly prioritize issues without doing anything special.

Error monitoring - a line of code that never changes and is put in every error handling block - is a great example of this type of tool. The cloud interface associated with error monitoring identifies critical errors and pushes those errors out to users instantly. Common errors, the ones that pop up in every release but have little impact, will be low priority whereas a net new error type would be highlighted.

There is no way to avoid the burden that necessary testing early in the delivery pipeline puts on the developers. The issue must be addressed but not by choosing tools on a whim or making an existing tool do the job. Picking purpose-built tools that back up the increasing testing load will reduce the complexity of testing cases and allow developers the freedom to code rather than spend their time contemplating how they will catch and respond to bugs.

As errors are frequently due to some invalid state or data, it is tempting to concatenate this data onto the end of the error message string. There are problems with this approach, namely that it removes the type of the data by stringifying it, removing it from its separate variable, making it unnecessarily difficult to manipulate. This is fine when logging to console during debugging, but for proper development, staging and production usage, extra state data should be delivered separately to the error message.

For instance with Raygun:

```
var error = new Error("My custom error");
  Raygun.send(error, { custom_state: 'From user 01', dateTime: new
  Date() });
```

The data can then be displayed separately, and more accurate error grouping can be conducted.

Handling unhandled errors

When using an automatic error monitoring solution, you are able to catch all errors (whether handled or not) in your codebase.

Applying a custom global error handler:

```
window.onerror = function (message, source, lineno, colno, error) {
// Add your custom final error handling logic here
}
```

Raygun4JS contains a reliable global error handler out-of-the-box, and can also automatically catch errors in jQuery Ajax callback functions.

SECTION 5

Security issues and best practices

An important concern for practically all software, besides being feature-complete, maintainable and performant, is security. This is especially so for software that exists in networked environments, and this includes client-side code running in the browser.

Due to the huge scope of the internet and the potential for adversaries to compromise your application code in order to extract sensitive data that can cause huge damage to your clients/customers as well as your business or reputation, developing JS code in a security-minded manner is as important as code running on the server.

Just like writing maintainable, scalable code requires architectural discipline, and building fast applications requires a constant eye towards performance, creating secure applications requires developers to be mindful of security at all times.

Teams should foster an environment where topics around security can and should be discussed, as well as peer reviews considering how secure a piece of code is or how it could be improved. If such a culture is created, processes for building more secure web applications will naturally fall out, reducing the likelihood of damaging events. Like many of the trickier problems in information science, security practices aren't always apparent or easy to bestow on team members. It may also be difficult to justify adding time on projects to accommodate security as a separate requirement, but as mentioned above this can be avoided if it is part of the culture to begin with.

Practically, there are several areas and aspects to consider when building safe and secure web applications. Many of these techniques aren't difficult to apply to new or existing code as the code paths they apply to are often localized (e.g to authentication flows or template rendering), thus hardening such code isn't that time consuming. Listed below are some areas to investigate and keep in mind when building JS applications, but is by no means an exhaustive list. We encourage you to keep abreast of industry developments and best practices, as security pitfalls can pop up rapidly and potentially be quite problematic (e.g Heartbleed).

An excellent third-party resource we recommend for these topics is OWASP - the Open Web Application Security Project, at https://www.owasp.org/index.php/Main_Page.

HTTPS by default

Ensuring that the HTML document that loads your JavaScript files is delivered by HTTPS only is a highly recommended practice. This applies to all pages including public, unauthenticated (e.g marketing) ones, and login pages, before the credentials are entered. This mitigates man-in-the-middle attacks, and as of 2016 is rewarded and recommended by Google's ranking algorithms.

This should be enforced by the HTTP server or reverse proxy using a 301 redirect. These are fairly simple to configure using popular servers such as Nginx, Apache or IIS. In this way, no insecure HTTP traffic can be sniffed and the baseline security of your user's data is higher.

Cross site scripting

As many modern single page web applications accept and render user input using templating, sanitizing this user input is key to preventing malicious script tags from being parsed and executed inside unknowing user's browsers, where it can exfiltrate their data.

All inputs should be URL encoded and persisted, then rendered out to the browser later using the encoded form. If your field accepts some form of markup, this should be parsed and rendered using a popular library, and never using a hand-rolled solution (there are many edge cases which clever casing and unicode characters can exploit to insert real script tags).

If you need to output raw text, ideally you should control it end-to-end and not have it originating from strings inputted by the user.

Sites should use and respect CORS, and be mindful of how techniques such as JSONP designed extend and workaround the restrictions implied. In particular if an API server on another domain is used to receive/send XHR payloads, that server should be whitelisted in the Access-Control-Allow-Origin HTTP header, in most cases.

Cookies

If your application use case maps nicely to HTTP cookies, these should have the 'secure' flag set. This is done so at the HTTP server level
- see OWASP for how to do so for many common servers.

The effect of this is to only transmit cookies encrypted, and not in cleartext,

as they frequently contain sensitive user data. This is naturally available when the document is served with HTTPS - see 'HTTPS By Default' above.

Passwords and authentication

Passwords should **never** be persisted in the backend in cleartext (unencrypted). Hand-rolling your own encryption library for encrypting passwords is also very highly discouraged, as they can and will contain many subtle bugs and backdoors which an adversary could crack easily. Popular, verified implementation libraries of secure encryption algorithms are available for all code bases.

As of 2016, we recommend using the **bcrypt** library (which uses the Blowfish algorithm) to encrypt cleartext passwords, then persisting the hash only for later login verification. This should be implemented server-side, with cleartext credentials only delivered to it via a secure HTTPS session. If you are writing server-side JS using Node, you may look into bcrypt-nodejs from NPM for a good, easy-to-use implementation.

After login, a session token should be generated, persisted then delivered back to the client for storage (e.g in-memory, with Local Storage or similar). This token should then be the piece of data that is checked and sent back with every action to the server, and never the password or its hash.

SECTION 6

Source maps

When dealing with minified JavaScript, the information supplied by an error's stack trace does not provide useful line/column information and enable you to track down the source of an error. To deal with this you can utilize a source map which will provide the translation between the minified JavaScript and the original source code.

Source Maps are a JSON based mapping format which can be used by any processed file to create a mapping relationship between source and the processed output. Source maps can be utilized for minified JavaScript, CoffeeScript, Less and Sass!

With source maps in place, when an error occurs, instead of a difficult to interpret error on line 1, the browser will take the following actions to provide you with an interpreted stack trace.

Here is a quick overview of what the browser is doing to make this happen:

1. Your browser loads the initial page and downloads source.min.js as it is referenced
2. An error occurs in the JavaScript contained within source.min.js, typically this will be handled by window.onError
3. The developer console goes to render out this error and the associated stack trace. On inspection of source.min.js it notices there is an end of file comment which conforms to the format to indicate a source map file is present
4. Your browser now downloads source.min.js.map, if this file exists the

map file will be processed and we will start attempting to decode the stack based on the mappings contained in this file. If there is a successful mapping, the updated reference will point to a line number, column number and symbol in source.js rather than source.min.js

UglifyJS 2 will generate sources maps during minification when provided with the --source-map flag and a specified output file.

Raygun Crash Reporting provides a service which maps minified stack traces for errors reported to the service from the JavaScript provider. Assets required for the mapping are downloaded automatically from the locations specified in the stack trace, minified and map files.

As an alternative to automatic asset retrieval Raygun can additionally utilize minified JavaScript and map files uploaded into the Raygun Source Map Center, this is provided as a means for users who do not wish for their code to be publicly accessible to use the source mapping capabilities.

In addition to mapping file names, method names and line numbers for the stack trace, Raygun can inject source code snippets to give additional context to errors. This feature is afforded to anyone who generates their source maps with the sourcesContent array populated. UglifyJS 2 will populate the sourcesContent array when provided the --source-map-include-sources flag.

SECTION 7

Practices to avoid errors in development

Semicolons

Automatic Semicolon Insertion is a controversial feature of JavaScript which allows programmers to omit semicolons from terminating statements. This feature makes learning JavaScript easier for developers learning their first programming language.

Example without Semicolons

```
var i = 0
i++
alert(i)
```

Example with Semicolons

```
var i = 0;
  i++;
  alert(i);
```

New line characters in the first example are treated as line terminators and so the Interpreter will terminate the previous statement.

Whilst ASI is stable with browsers implementing it as stated in ECMAScript specification it is commonly recommended not to be relied upon because:

- The JavaScript interpreter will insert semicolons automatically in some circumstances
- Bugs can be introduced which are harder to locate and debug

Eval

Eval is a global function allowing strings passed to be executed as code. Whilst not directly related to error prevention, use of eval isn't recommended as it can open up your website to malicious third parties.

Example

```
eval('alert("Hello World")');
```

Alternatives

There are cases where injecting strings of code into a JavaScript environment may be needed. In this case it maybe good to look into the JS Interpreter project which helps isolate the script from the main environment.

Dynamic Typing

JavaScript is a loosely typed language and so a variable can have different types assigned.

Example

```
var age = "25";
  if( age > 21 ) { // True
```

```
  // Allowed to drink!
  }
  var age = 25;
  if( age > 21 ) { // True
  // Allowed to drink!
}
```

While this is powerful feature often it needlessly complicates the code, and that in turn often leads to more bugs in the future.

Remembering Data Types

It can hard to recall a variable's type when you have been jumping around a large codebase. A variable naming scheme to help solve this issue is Hungarian Notation, where you prefix the variable name with its data type.

Example

```
var iAge = 25;
var sName = "Jill";
var bIsAwesome = true;
```

Type Coercion

Type Coercion issues can appear when you are evaluating variables of different types. Generally these occur when you are doing comparison operations and can lead to hard to find, subtle errors later in your code.

Example

```
if( 5 == “5”) { }   // true
if( 5 === “5”) { }  // false
if( 1 == true) { }  // true
if( 1 === true) { } // false
if( undefined == null ) { } // true
if( undefined === null ) { } // false
```

Browser Support

API’s are constantly being added to JavaScript and while it is tempting to use them, you may find that not all of your users use the latest version of Google Chrome or Firefox, meaning if you are developing an app using the latest ECMAScript 6 features, whilst supporting a legacy version of IE, some of your users are bound to encounter errors.

It isn’t recommended to enable/disable features based on browser detection scripts because new browsers/versions are created all the time. So having a robust browser detection library isn’t practical. Instead, enable features using feature detection scripts.

Feature Detection

One approach to stop such issues is to use a feature detection library like Modernizr where you can then enable the feature if it is supported.

Example

```
if(Modernizr.awesomeNewFeature) {
showOffAwesomeNewFeature();
} else {
getTheOldLameExperience();
}
```

Modernizr (and others) can have custom builds, enabling you to only include detection scripts for the features you use. Saving on bandwidth for your users.

Polyfills

Another option is to include a Polyfill. A Polyfill is a piece of code which enables features, by providing a fallback alternative in browsers when it detects a particular feature isn't supported.

A useful list of Polyfills can be found here.

The Global Namespace

Multiple JavaScript files can be included in a single page, which is a great thing, but an issue not obvious to developers are that variables declared outside of a function will be global in scope.

Example

```
pizza = 'Meat lovers';
    alert( window.pizza ); // Result: Meat lovers

    var burger = 'Cheese';
```

```
alert( window.burger ); // Result: Cheese
window.pasta = 'Rotelle';
alert( pasta ); // Result: Rotelle
```

Note: Make sure you also declare variables using `var`. The example above is only added to show that undeclared variables are added to the window object.

This may not seem a big deal when dealing with small websites, but it can become an issue in large apps since they can include external JavaScript libraries and a multiple of scripts for each page.

Issues with global variables are:

1. Global namespace can get polluted. Meaning you may accidentally override variables used in other files
2. Stops the Garbage Collector from removing variables because it can't be sure if it will ever be used again or not
3. They are slower to lookup than local variables
4. Naming collisions with libraries

Immediately-invoked function expression

An immediately-invoked function is a function which is executed straight away. This is a JavaScript design pattern many programmers implement to prevent pollution of the global namespace.

Example

```
(function() {
```

```
var pizza = “Meat lovers”;
alert( window.pizza ); //
})();
```

Namespace

Whilst keeping scripts modular is good approach to take, it is inevitable that you will need to use some data across files. We still don’t want to declare lots of global variables, so to solve this issue you can create a single global object for your app and have all the functions/variables as properties of that object.

Example

```
var MyApp = {
        propertyA: true,
        propertyB: false,
        init: function() {
            // Insert code here
    }
            // Continue function/variable declarations
};
```

Strict mode

Example

```
function awesome() {
  “use strict”;
  //Strict mode enforced
}
```

SECTION 8

How to setup basic monitoring of errors in JavaScript

With just a single, short code snippet, every error that occurs in your software can be detected and instantly diagnosed with an automatic error monitoring tool like Raygun.

The most basic way to handle an error in JavaScript is to wrap your code in a **try** - **catch** - **finally block**. This will catch the error and prevent it from 'bubbling' up to the top-level **window.onerror** handler:

```
try {
  throw new Error("My custom error");
} catch (err) {
  // Handle the cause of the error, potentially by returning default data, or send the error to Raygun:
 Raygun.send(err);
}
finally {
  // code that always gets executed
}
```

About Raygun Crash Reporting

Raygun's error and crash reporting software silently monitors your web and mobile applications, collecting all error and crash events that are affecting your customers. When issues are found they are presented on your crash reporting dashboard, with detailed diagnostic reports about every single error and crash, making digging through log files and trying to replicate issues a thing of the past.

Contextual information about the problem is made available to your entire team instantly, including which specific users of your application have been affected. Raygun then provides your team with a seamless workflow to solve the problem quickly. Should the inevitable happen and users are exposed to outages, bugs, errors, crashes or bad deploys, your entire team will know about it instantly with Raygun's smart alerts via email or team chat notifications, giving you all the diagnostic information you need to fix the problem quickly and efficiently.

"Without Raygun we had no clear window into what errors our applications were throwing. We only knew by digging through logs when we got the time, or by our customers letting us know something was wrong. Raygun takes the unknown and makes it known"

Daniel Hoenig - Software Architect at Schneider Electric

SECTION 9

Further reading / resources

Getting started with JavaScript source maps:

https://raygun.com/blog/2014/02/getting-started-with-javascript-source-maps/

The Error object reference guide:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Error

Exception handling statements, throw, try-catch-finally:

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Control_flow_and_error_handling#Exception_handling_statements