raven-js Documentation

Release 1.1.19

Matt Robenolt

August 19, 2015

Contents

Raven.js is a tiny standalone JavaScript client for Sentry. It can be used to report errors from browser and node environments. The quality of reporting will heavily depend on the environment the data is submitted from.

CHAPTER 1

Installation

Raven.js is distributed in a few different methods, and should get included after any other libraries are included, but before your own scripts. For all details see Installation. For just getting started, you can use our CDN:

```
<script src="//cdn.ravenjs.com/1.1.19/raven.min.js"></script>
```

CHAPTER 2

Configuring the Project

We must first configure Sentry to allow certain hosts to report errors. This prevents abuse so somebody else couldn't start sending errors to your account from their site.

This can be found under the **Project Settings** page in Sentry. You'll need to add each domain that you plan to report from into the **Allowed Domains** box. Alternatively if you're not worried about CORS security, you can simply enter $\star$ to whitelist all domains.

CHAPTER 3

Configuring the Client

Now need to set up Raven.js to use your Sentry DSN:

```
Raven.config('___PUBLIC_DSN___').install()
```

At this point, Raven is ready to capture any uncaught exception.

Although, this technically works, this is not going to yield the greatest results. It's highly recommended to next check out Configuration and Usage after you have it up and running to improve your results.

CHAPTER 4

Reporting Errors

The simplest way, is to try and explicitly capture and report potentially problematic code with a `try...catch` block and `Raven.captureException`.

```
try {
    doSomething(a[0])
} catch(e) {
    Raven.captureException(e)
}
```

There are more ways to report errors. For a complete guide on this see *Reporting Errors Correctly*.

CHAPTER 5

Adding Context

While a user is logged in, you can tell Sentry to associate errors with user data. This data is then submitted with each error which allows you to figure out which users are affected.

```
Raven.setUserContext({
    email: 'matt@example.com',
    id: '123'
})
```

If at any point, the user becomes unauthenticated, you can call `Raven.setUserContext()` with no arguments to remove their data.

Other similar methods are `Raven.setExtraContext` and `Raven.setTagsContext` as well as `Raven.context`. See *Passing Additional Data* for more info.

CHAPTER 6

Dealing with Minified Source Code

Raven and Sentry support Source Maps. If you provide source maps in addition to your minified files that data becomes available in Sentry. For more information see *Source Maps*.

CHAPTER 7

Deep Dive

For more detailed information about how to get most out of Raven.js there is additional documentation available that covers all the rest:

7.1 Installation

Raven is distributed in a few different methods, and should get included after any other libraries are included, but before your own scripts.

So for example:

```
<script src="jquery.js"></script>
<script src="//cdn.ravenjs.com/1.1.19/jquery,native/raven.min.js"></script>
<script>Raven.config('___PUBLIC_DSN___').install();</script>
<script src="app.js"></script>
```

This allows the ability for Raven's plugins to instrument themselves. If included before something like jQuery, it'd be impossible to use for example, the jquery plugin.

7.1.1 Using our CDN

We serve our own builds off of Fastly. They are accessible over both http and https, so we recommend leaving the protocol off.

Our CDN distributes builds with and without plugins.

```
<script src="//cdn.ravenjs.com/1.1.19/raven.min.js"></script>
```

We highly recommend trying out a plugin or two since it'll greatly improve the chances that we can collect good information.

This version does not include any plugins. See ravenjs.com for more information about plugins and getting other builds.

7.1.2 Bower

We also provide a way to deploy Raven via bower. Useful if you want serve your own scripts instead of depending on our CDN and mantain a `bower.json` with a list of dependencies and versions (adding the `--save` flag would automatically add it to `bower.json`).

```
$ bower install raven-js --save
```

```
<script src="/bower_components/raven-js/dist/raven.js"></script>
```

Also note that the file is uncompresed but is ready to pass to any decent JavaScript compressor like uglify.

7.1.3 npm

Raven is published to npm as well. https://www.npmjs.com/package/raven-js

```
$ npm install raven-js --save
```

7.1.4 Requirements

Raven expects the browser to provide *window.JSON* and *window.JSON.stringify*. In Internet Explorer 8+ these are available in standards mode. You can also use json2.js to provide the JSON implementation in browsers/modes which doesn't support native JSON

7.2 Plugins

Plugins extend the functionality of Raven.js to cover common libraries and environments automatically.

7.2.1 What are plugins?

In Raven.js, plugins are little snippets of code to augment functionality for a specific application/framework. It is highly recommended to checkout the list of plugins and use what apply to your project.

In order to keep the core small, we have opted to only include the most basic functionality by default, and you can pick and choose which plugins are applicable for you.

7.2.2 Why are plugins needed at all?

JavaScript is pretty restrictive when it comes to exception handling, and there are a lot of things that make it difficult to get relevent information, so it's important that we inject code and wrap things magically so we can extract what we need. See Usage for tips regarding that.

7.2.3 All Plugins

- https://github.com/getsentry/raven-js/tree/master/plugins
- Download

7.3 Configuration

We must first configure the client to allow certain hosts to report errors. This prevents abuse so somebody else couldn't start sending errors to your account from their site.

Note: Without setting this, all messages will be rejected!

This can be found under the *Project Details* page in Sentry.

Now need to set up Raven.js to use your Sentry DSN.

```
Raven.config('___PUBLIC_DSN___').install()
```

At this point, Raven is ready to capture any uncaught exception.

Although, this technically works, this is not going to yield the greatest results. It's highly recommended to next check out Usage.

7.3.1 Optional settings

`Raven.config()` can be passed an optional object for extra configuration.

`logger`
The name of the logger used by Sentry. Default: `javascript`

```
{
  logger: 'javascript'
}
```

`release`
Track the version of your application in Sentry.

```
{
  release: '721e41770371db95eee98ca2707686226b993eda'
}
```

Can also be defined with `Raven.setReleaseContext('721e41770371db95eee98ca2707686226b993eda')`.

`tags`
Additional tags to assign to each event.

```
{
  tags: {git_commit: 'c0deb10c4'}
}
```

`whitelistUrls`
The inverse of `ignoreUrls`. Only report errors from whole urls matching a regex pattern or an exact string. `whitelistUrls` should match the url of your actual JavaScript files. It should match the url of your site if and only if you are inlining code inside `<script>` tags.

Does not affect `captureMessage` or when non-error object is passed in as argument to captureException.

```
{
  whitelistUrls: [/getsentry\.com/, /cdn\.getsentry\.com/]
}
```

`ignoreErrors`
Very often, you will come across specific errors that are a result of something other than your application, or errors that you're completely not interested in. *ignoreErrors* is a list of these messages to be filtered out before being sent to Sentry as either regular expressions or strings.

Does not affect captureMessage or when non-error object is passed in as argument to captureException.

```
{
  ignoreErrors: ['fb_xd_fragment']
}
```

ignoreUrls

The inverse of `whitelistUrls` and similar to `ignoreErrors`, but will ignore errors from whole urls matching a regex pattern or an exact string.

```
{
  ignoreUrls: [/graph\.facebook\.com/, 'http://example.com/script2.js']
}
```

Does not affect captureMessage or when non-error object is passed in as argument to `captureException`.

includePaths

An array of regex patterns to indicate which urls are a part of your app in the stack trace. All other frames will appear collapsed inside Sentry to make it easier to discern between frames that happened in your code vs other code. It'd be suggested to add the current page url, and the host for your CDN.

```
{
    includePaths: [/https?:\/\/getsentry\.com/, /https?:\/\/cdn\.getsentry\.com/]
}
```

dataCallback

A function that allows mutation of the data payload right before being sent to Sentry.

```
{
    dataCallback: function(data) {
      // do something to data
      return data;
    }
}
```

shouldSendCallback

A callback function that allows you to apply your own filters to determine if the message should be sent to Sentry.

```
{
    shouldSendCallback: function(data) {
      return false;
    }
}
```

maxMessageLength

By default, raven truncates messages to a max length of 100 characters. You can customize the max length with this parameter.

7.3.2 Putting it all together

```
<!doctype html>
<html>
<head>
    <title>Awesome stuff happening here</title>
</head>
<body>
    ...
    <script src="jquery.min.js"></script>
    <script src="//cdn.ravenjs.com/1.1.18/jquery,native/raven.min.js"></script>
    <script>
        Raven.config('___PUBLIC_DSN___', {
            logger: 'my-logger',
            whitelistUrls: [
```

```
                /disqus\.com/,
                /getsentry\.com/
            ],
            ignoreErrors: [
                'fb_xd_fragment',
                /ReferenceError:.*/
            ],
            includePaths: [
                /https?:\/\/(www\.)?getsentry\.com/
            ]
        }).install();
    </script>
    <script src="myapp.js"></script>
</body>
</html>
```

7.3.3 TraceKit specific optional settings

Usually there is no need to touch these settings, but they exist in case you need to tweak something.

fetchContext
Enable TraceKit to attempt to fetch source files to look up anonymous function names, this can be useful to enable if you don't get the context for some entries in the stack trace. Default value is `false`.

```
{
    fetchContext: true
}
```

linesOfContext
The count of lines surrounding the error line that should be used as context in the stack trace, default value is `11`. Only applicable when `fetchContext` is enabled.

```
{
    linesOfContext: 11
}
```

collectWindowErrors
Enable or disable the TraceKit `window.onerror` handler, default value is `true`.

```
{
    collectWindowErrors: true
}
```

7.4 Usage

By default, Raven makes a few efforts to try its best to capture meaningful stack traces, but browsers make it pretty difficult.

The easiest solution is to prevent an error from bubbling all of the way up the stack to `window`.

7.4.1 Reporting Errors Correctly

There are different methods to report errors and this all depends a little bit on circumstances.

try ... catch

The simplest way, is to try and explicitly capture and report potentially problematic code with a `try...catch` block and `Raven.captureException`.

```
try {
    doSomething(a[0])
} catch(e) {
    Raven.captureException(e)
}
```

Do not throw strings! Always throw an actual `Error` object. For example:

```
throw new Error('broken')  // good
throw 'broken'  // bad
```

It's impossible to retrieve a stack trace from a string. If this happens, Raven transmits the error as a plain message.

context/wrap

`Raven.context` allows you to wrap any function to be immediately executed. Behind the scenes, Raven is just wrapping your code in a `try...catch` block to record the exception before re-throwing it.

```
Raven.context(function() {
    doSomething(a[0])
})
```

`Raven.wrap` wraps a function in a similar way to `Raven.context`, but instead of executing the function, it returns another function. This is especially useful when passing around a callback.

```
var doIt = function() {
    // doing cool stuff
}

setTimeout(Raven.wrap(doIt), 1000)
```

7.4.2 Tracking Users

While a user is logged in, you can tell Sentry to associate errors with user data.

```
Raven.setUserContext({
    email: 'matt@example.com',
    id: '123'
})
```

If at any point, the user becomes unauthenticated, you can call `Raven.setUserContext()` with no arguments to remove their data. *This would only really be useful in a large web app where the user logs in/out without a page reload.*

This data is generally submitted with each error or message and allows you to figure out which errors are affected by problems.

7.4.3 Capturing Messages

```
Raven.captureMessage('Broken!')
```

7.4.4 Passing Additional Data

`captureException`, `context`, `wrap`, and `captureMessage` functions all allow passing additional data to be tagged onto the error, such as `tags` or `extra` for additional context.

```
Raven.captureException(e, {tags: { key: "value" }})

Raven.captureMessage('Broken!', {tags: { key: "value" }})

Raven.context({tags: { key: "value" }}, function(){ ... })

Raven.wrap({logger: "my.module"}, function(){ ... })

Raven.captureException(e, {extra: { foo: "bar" }})
```

You can also set context variables globally to be merged in with future exceptions with `setExtraContext` and `setTagsContext`.

```
Raven.setExtraContext({ foo: "bar" })
Raven.setTagsContext({ key: "value" })
```

7.4.5 Getting Back an Event ID

An event id is a globally unique id for the event that was just sent. This event id can be used to find the exact event from within Sentry.

This is often used to display for the user and report an error to customer service.

```
Raven.lastEventId()
```

`Raven.lastEventId()` will be undefined until an event is sent. After an event is sent, it will contain the string id.

```
Raven.captureMessage('Broken!')
alert(Raven.lastEventId())
```

7.4.6 Verify Raven Setup

If you need to conditionally check if raven needs to be initialized or not, you can use the *isSetup* function. It will return *true* if Raven is already initialized:

```
Raven.isSetup()
```

7.4.7 Dealing with Minified Source Code

Raven and Sentry support Source Maps.

We have provided some instructions to creating Source Maps over at https://www.getsentry.com/docs/sourcemaps/. Also, checkout our Gruntfile for a good example of what we're doing.

You can use Source Map Validator to help verify that things are correct.

7.4.8 CORS

If you're hosting your scripts on another domain and things don't get caught by Raven, it's likely that the error will bubble up to `window.onerror`. If this happens, the error will report some ugly `Script error` and Raven will drop it on the floor since this is a useless error for everybody.

To help mitigate this, we can tell the browser that these scripts are safe and we're allowing them to expose their errors to us.

In your `<script>` tag, specify the `crossorigin` attribute:

```
<script src="//cdn.example.com/script.js" crossorigin="anonymous"></script>
```

And set an `Access-Control-Allow-Origin` HTTP header on that file.

```
Access-Control-Allow-Origin: *
```

Note: both of these steps need to be done or your scripts might not even get executed

7.4.9 Custom Grouping Behavior

In some cases you may see issues where Sentry groups multiple events together when they should be separate entities. In other cases, Sentry simply doesn't group events together because they're so sporadic that they never look the same.

Both of these problems can be addressed by specifying the `fingerprint` attribute.

For example, if you have HTTP 404 (page not found) errors, and you'd prefer they deduplicate by taking into account the URL:

```
Raven.captureException(ex, {fingerprint: ['{{ default }}', 'http://my-url/']});
```

7.5 Source Maps

In various browsers Sentry supports deminifying JavaScript via source maps. A source map is a file generated by your minifier which compresses a mapping of the minified file to the original uncompressed version(s).

One important thing to note is that even though we support mapping files, a users browser may not actually be able to collect the required information for the server to generate a sourcemap.

Sentry requires the following to be able to map tracebacks to their source:

- A source map header or footer
- A publicly accessible uncompressed version of the source
- A line of context that includes a line number, column number, and filename

The first two items are the responsibility of you, the end-user, and you can take care of publishing them as part of your build process. The latter however, with an individual line of context, is severely crippled in many browsers.

One thing to note is that Sentry will attempt to process the source map before storing (or grouping) an event. This ensures that if we are able to deminify the source, we'll be able to more effectively group similar events over time.</p>

7.5.1 Browser Support

In our experiences, the only browser that routinely delivers usable error reports is **Google Chrome**.

For additional information, see this more up-to-date wiki page.

7.5.2 Generating a Source Map

While there are several compression libraries which support source maps, as of writing our recommendation is to use UglifyJS. That said, many tools such as webpack and browserify.

As an example, we can look at how we used to do things with Sentry (pre-webpack):

```
node_modules/uglify-js/bin/uglifyjs {input} \
  --source-map-root={relroot}/ \
  --source-map-url={name}.map.js \
  --source-map={relpath}/{name}.map.js -o {output}
```

We won't attempt to go into the complexities of source maps, so we recommend taking a stab at compiling them, and running them against a validator.

7.5.3 Validating a Source Map

We maintain an online validation tool that can be used to test your source (and sourcemaps) against: sourcemaps.io.

7.5.4 Uploading Source Maps to Sentry

In many cases your application may sit behind firewalls or you simply can't expose source code to the public. Sentry provides an abstraction called **Releases** which you can attach source artifacts to.

The release API is intended to allow you to store source files (and sourcemaps) within Sentry. This removes the requirement for them to be web-accessible, and also removes any inconsistency that could come from network flakiness (on either your end, or Sentry's end).

- Start by creating a new API key under your organization's API Keys nav (on the home).
- Ensure you you have `project:write` selected under scopes.
- You'll use HTTP basic auth with the api key being your username, and an empty value for the password.

Now you need to setup your build system to create a release, and attach the various source files. You will want to upload all dist files (i.e. the minified/shipped JS), the referenced sourcemaps, and the files that those sourcemaps point to.

```
# Create a new release
$ curl https://app.getsentry.com/api/0/projects/:organization_slug/:project_slug/releases/ \
  -u [api_key]: \
  -X POST \
  -d '{"version": "abcdef"}' \
  -H 'Content-Type: application/json'

{
  "dateCreated": "2015-03-06T04:51:32.723Z",
  "version": "2da95dfb052f477380608d59d32b4ab9"
}
```

```
# Upload a file for the given release
# Note: The filename should be the *full* url that this
# would be referenced as in production.
$ curl https://app.getsentry.com/api/0/projects/:organization_slug/:project_slug/releases/2da95dfb052
  -u [api_key]: \
  -X POST \
  -F file=@README.rst \
  -F name="http://example.com/readme.rst"

{
  "dateCreated": "2015-03-06T04:53:00.308Z",
  "headers": {
    "Content-Type": "application/octet-stream"
  },
  "id": "1",
  "name": "http://example.com/readme.rst",
  "sha1": "22591348ed129fe016c535654f6493737f0f9df6",
  "size": 452
}
```

```
# If you make a mistake, you can also simply clear out the release
$ curl https://app.getsentry.com/api/0/projects/:organization_slug/:project_slug/releases/2da95dfb052
  -u [api_key]: \
  -X DELETE
```

Additionally, you'll need to configure the client to send the `release`:

```
Raven.config({
    release: '2da95dfb052f477380608d59d32b4ab9'
});
```

Note: You dont *have* to upload the source files (ref'd by sourcemaps), but without them the grouping algorithm will not be as strong, and the UI will not show any contextual source.

Additional information can be found in the Releases API documentation.

7.6 Tips and Tricks

These are some general recommendations and tips for how to get the most out of Raven.js and Sentry.

7.6.1 Decluttering Sentry

The first thing to do is to consider constructing a whitelist of domains in which might raise acceptable exceptions.

If your scripts are loaded from `cdn.example.com` and your site is `example.com` it'd be reasonable to set `whitelistUrls` to:

```
whitelistUrls: [
  /https?:\/\/((cdn|www)\.)?example\.com/
]
```

Since this accepts a regular expression, that would catch anything *.example.com or example.com exactly. See also: *Config: whitelistUrls*.

Next, checkout the list of plugins we provide and see which are applicable to you.

The community has compiled a list of common ignore rules for common things, like Facebook, Chrome extensions, etc. So it's recommended to at least check these out and see if they apply to you. Check out the original gist.

```
var ravenOptions = {
    ignoreErrors: [
      // Random plugins/extensions
      'top.GLOBALS',
      // See: http://blog.errorception.com/2012/03/tale-of-unfindable-js-error. html
      'originalCreateNotification',
      'canvas.contentDocument',
      'MyApp_RemoveAllHighlights',
      'http://tt.epicplay.com',
      'Can\'t find variable: ZiteReader',
      'jigsaw is not defined',
      'ComboSearch is not defined',
      'http://loading.retry.widdit.com/',
      'atomicFindClose',
      // Facebook borked
      'fb_xd_fragment',
      // ISP "optimizing" proxy - `Cache-Control: no-transform` seems to
      // reduce this. (thanks @acdha)
      // See http://stackoverflow.com/questions/4113268
      'bmi_SafeAddOnload',
      'EBCallBackMessageReceived',
      // See http://toolbar.conduit.com/Developer/HtmlAndGadget/Methods/JSInjection.aspx
      'conduitPage'
    ],
    ignoreUrls: [
      // Facebook flakiness
      /graph\.facebook\.com/i,
      // Facebook blocked
      /connect\.facebook\.net\/en_US\/all\.js/i,
      // Woopra flakiness
      /eatdifferent\.com\.woopra-ns\.com/i,
      /static\.woopra\.com\/js\/woopra\.js/i,
      // Chrome extensions
      /extensions\//i,
      /^chrome:\/\//i,
      // Other plugins
      /127\.0\.0\.1:4001\/isrunning/i,  // Cacaoweb
      /webappstoolbarba\.texthelp\.com\//i,
      /metrics\.itunes\.apple\.com\.edgesuite\.net\//i
    ]
};
```

7.6.2 Sampling Data

It happens frequently that errors sent from your frontend can be overwhelming. One solution here is to only send a sample of the events that happen. You can do this via the `shouldSendCallback` setting:

```
shouldSendCallback: function(data) {
    // only send 10% of errors
    var sampleRate = 10;
    return (Math.random() * 100 <= sampleRate);
}
```

7.6.3 jQuery AJAX Error Reporting

For automatically AJAX errors from jQuery the following tip might come in helpful. However depending on the type of request you might have to do slightly different things.

7.6.4 Same Origin

Whenever an Ajax request completes with an error, jQuery triggers the `ajaxError` event, passing the `event` object, the `jqXHR` object (prior to jQuery 1.5, the `XHR` object), and the `settings` object that was used in the creation of the request. When an HTTP error occurs, the fourth argument (`thrownError`) receives the textual portion of the HTTP status, such as "Not Found" or "Internal Server Error."

You can use this event to globally handle Ajax errors:

```
$(document).ajaxError(function(event, jqXHR, ajaxSettings, thrownError) {
    Raven.captureMessage(thrownError || jqXHR.statusText, {
        extra: {
            type: ajaxSettings.type,
            url: ajaxSettings.url,
            data: ajaxSettings.data,
            status: jqXHR.status,
            error: thrownError || jqXHR.statusText,
            response: jqXHR.responseText.substring(0, 100)
        }
    });
});
```

Note:

- This handler is not called for cross-domain script and cross-domain JSONP requests.
- If `$.ajax()` or `$.ajaxSetup()` is called with the `global` option set to `false`, the `.ajaxError()` method will not fire.
- As of jQuery 1.8, the `.ajaxError()` method should only be attached to document.

7.6.5 Cross Origin

Due security reasons most web browsers are not giving permissions to access error messages for cross domain scripts. This is not jQuery issue but an overall javascript limitation.

Depending on your situation you have different options now:

When you control the backend

If you have access to the backend system you are calling, you can set response headers to allow a cross domain call:

```
Access-Control-Allow-Origin: http://domain1.com, http://domain2.com
```

Script tags have now got a new non-standard attribute called `crossorigin` (read more). The most secure value for this would be `anonymous`. So, you'll have to modify your script tags to look like the following:

```
<script src="http://sub.domain.com/script.js" crossorigin="anonymous"></script>
```

When you have no access to the backend

If you have no access to the backend, you could try a workaround, which is basically adding a timeout on the Ajax call. This is however very dirty, and will fail on slow connection or long response time:

```
$.ajax({
    url: 'http:/mysite/leaflet.js',
    success: function() { ... },
    error: function() { ... },
    timeout: 2000, // 2 seconds timeout before error function will be called
    dataType: 'script',
    crossDomain: true
});
```

Development info:

7.7 Contributing

This part of the documentation gives you a basic overview of how to help with the development of Raven.js.

7.7.1 Setting up an Environment

To run the test suite and run our code linter, node.js and npm are required. If you don't have node installed, get it here first. Installing all other dependencies is as simple as:

```
$ npm install
```

And if you don't have Grunt already, feel free to install that globally:

```
$ npm install -g grunt-cli
```

Running the Test Suite

The test suite is powered by Mocha and can both run from the command line, or in the browser.

From the command line:

```
$ grunt test
```

From your browser:

```
$ grunt run:test
```

Then visit: http://localhost:8000/test/

Compiling Raven.js

The simplest way to compile your own version of Raven.js is with the supplied grunt command:

```
$ grunt build
```

By default, this will compile raven.js and all of the included plugins.

If you only want to compile the core raven.js:

```
$ grunt build.core
```

Files are compiled into `build/`.

Contributing Back Code

Please, send over suggestions and bug fixes in the form of pull requests on GitHub. Any nontrivial fixes/features should include tests. Do not include any changes to the `dist/` folder or bump version numbers yourself.

7.7.2 Documentation

The documentation is written using reStructuredText, and compiled using Sphinx. If you don't have Sphinx installed, you can do it using following command (assuming you have Python already installed in your system):

```
$ pip install sphinx
```

Documentation can be then compiled by running:

```
$ make docs
```

Afterwards you can view it in your browser by running following command and than pointing your browser to http://127.0.0.1:8000/:

```
$ grunt run:docs
```

Releasing New Version

- Bump version numbers in both `package.json` and `bower.json`.
- `$ grunt dist` This will compile a new version and update it in the `dist/` folder.
- Confirm that build was fine, etc.
- Commit new version, create a tag. Push to GitHub.
- `$ grunt publish` to recompile all plugins and all permutations and upload to S3.
- `$ npm publish` to push to npm.
- Confirm that the new version exists behind `cdn.ravenjs.com`
- Update version in the `gh-pages` branch specifically for http://ravenjs.com/.
- glhf

Resources:

- Downloads and CDN
- Bug Tracker
- Github Project