RFC 723: Unified logger service for improved client-side monitoring

Recommender: Mihir Chaturvedi
Date: Jul 20, 2022
Status: Draft
Decider: Mihir Chaturvedi, Valery Bugakov
Input providers: frontend-platform-team
Approvers: Jason Gornall, Taylor Sperry, Valery Bugakov, Tom Ross
Approvals:
Team: Frontend Platform

Summary

Provide a unified/standard logger service across Sourcegraph clients to report and process errors throughout the application, and make it so that developers working on the clients primarily and solely make use of only this service for all their logging needs.

Background

The need for a “unified logger service” was first identified here as a good use-case for uniformly reporting errors on the client-side to prevent rogue and unexplained `console.error`s. Further discussion regarding other use-cases and potential opportunities brought by such a service took place in this Slack thread.

Problem

The current state of client-side monitoring and error handling across Sourcegraph’s different consumer applications is unstructured, and prone to leaving errors and exceptions unhandled, ignored, or improperly processed.

Outlined below are some of the different problems that (can) arise:

1. **Errors ignored or unprocessed**
   Errors inside `catch` blocks or `.catch()` method calls on promises are often ignored, many times without an explanatory comment as to why ignoring the error might be safe or valid.
   - Common patterns around ignoring or improperly processing the errors include:
     - Empty `catch` blocks or `.catch()` method calls
     - `noop` or similar functions passed into `.catch()`

        - Errors logged to the browser console, and not further sent or processed
        - Unhelpful comments such as `// ignored` inside these blocks.
    - The above patterns are difficult to identify using a linter, and writing rules to warn against such code becomes impractical.

2. **Unique environment constraints for different consumer applications**
   Each consumer application (sourcegraph.com, various managed instances, browser extension, IDE extensions, etc) have unique constraints depending on their execution environment. Depending on the environment, errors must be conditionally processed and sent to third-party tools. Writing such branching logic becomes tedious, especially if it's expected from individual developers when reporting errors.

3. **Vendor lock-in, hard to modify third-party reporting**
   Currently, we will need to carry out a large-scale refactor across the code-base if we need to switch to or add a different/new third-party tool to report errors and carry out general monitoring.

4. **Pre/post-processing of logs is difficult**
   We lack a scalable way to trigger custom events or conditionally ignore logs based on log type. Instead, we have to rely on monkey-patching the Console class's methods to incorporate our modifications.

Proposal

We propose a unified logger service accessible by all consumer applications to log errors, warnings, and arbitrary messages to. The logger service will make it so that Sourcegraph developers will primarily and solely make use of this service, without having to worry about the underlying error processing.

Such a service fixes the above mentioned issues:

1. **Structured method to report errors**
   Developers will be required to report errors using only the method or hook supplied by the logger service. The logging function will provide a structured and shared interface for the developers to follow, and will allow for ignoring errors *if supplied with an explanation for the same*.
   Creating linter rules for such a logging function is simpler and quicker. We will only need to check for the presence of the function inside a `catch` block or `.catch()` method call.

2. **Handling of environment constraints abstracted away**
   Developers won't be bothered to individually report errors to Sentry or other third-party tools, and instead have the assurance that the underlying logic in the logger function will handle that for them, including support for branching logic for different environment requirements.

3. **Minimal to none vendor lock-in, easy to modify/add third-party tooling**
   Since reports to third-party tools (such as Sentry) will be handled in a single location inside the core logger function, switching over to or adding another tool or library will be as simple as modifying a couple lines of code only, instead of making a large refactor.

4. **Scalable way to incorporate custom events**
   Akin to preventing vendor lock-in, the logger service will provide a single place to trigger custom events based on log type, filter out unnecessary logs, etc.

React hook vs. direct imports

The service will be available as a React hook for use by `.tsx` files, and also as via direct import of the core logger function for use in regular `.ts` files which cannot leverage the hook.

The advantage of using a React hook is in its ability to provide a contextual, environment-specific logger function to the consumer. As Valery Bugakov mentioned here:

> *Injecting the logger service into applications gives us more control over code location and its extensibility. Imagine the core implementation of the logger service that provides a consistent interface and basic implementation that can be shared between all client applications and multiple env-specific loggers that comply with this interface and add additional logic where needed. This approach gives us:*
>
> 1. *Freedom to use logger method in shared components and shared logic without thinking about the execution environment.*
> 2. *Simpler implementation of the core logger without execution environment branching.*
> 3. *More control over application-specific changes given to application teams. E.g., the browser extension team owns the logger in their browser package as long as they comply with the core interface.*
> 4. *Simpler dependency graph. We would be able to colocate code with consumer applications which will become important once we introduce strict package APIs and control over inter-package dependencies: core logger -> used by the application logger -> used by the application. No circular dependency here 🎉.*

For places that are not able to use the hooks (e.g. regular .ts files), the core logger methods will still be available for use by directly importing them. This is akin to Apollo Client's `useClient` hook and React Router's `useHistory` hook — when not available we directly import `ApolloClient` and `history from 'history'`. The intention behind the React hook is to keep it as the default/preferable way of using it. But if we need it, we can import and use it anywhere. [ref]

Definition of success

How do we know if this proposal was successful? Are there any metrics we need to start tracking?