Speedometer 3 - Editing Text Scenario

Brian Grinstead, Feb 2023

Context: This is a research document for a Speedometer "scenario" as outlined here. The goal is to figure out the essential elements of this scenario (libraries, patterns, etc) and one or more workloads that capture those.

Motivation: Lots of people edit text content in the browser. Lots of that content, like WYSIWYG content or code, is too rich or complex to represent well with a `<textarea>`. Sites typically rely on advanced editor libraries for this, and we should make sure browsers perform well with them. *Note: this analysis is **not** meant as a benchmark or judgment on the libraries themselves, but as input to the benchmark.*

Tech we expect it to exercise: Forms / Editing, Infinite scrolling / virtualization. The workload "shell" could be adapted easily to test inline SVG icons or additional layout & CSS features.

TLDR: The most salient technical aspect here is the library. These are quite complex and ~nobody is rolling their own rich editors but lots of people want to have the feature. While there are a relatively small number of very popular libraries there is not a single winner. Additionally, initial analysis indicates that they each perform differently enough that it wouldn't be redundant to test more than one. So we should test a number of popular libraries with a similar workload for each (using something like the design of the Monaco editor workload in GrandPrix).

Results from library research

An initial list of libraries[1] was compiled based on online searches for popular code and WYSIWYG libraries, and lists on related GitHub Topics. For each library the current number of weekly NPM downloads, number of GitHub stars, and other project indicators were gathered.

Then a prototype test harness (source code) was created to integrate each using information from project documentation and online searching when necessary. The harness renders each editor in a variety of configurations:

- Viewport size (small, medium, large)
- Length of text / code (small, large)
- WYSIWYG formatting / code highlighting (on/off)

That harness was then used for manual testing across browsers and capturing a profile in Firefox. Note that the specific details about performance in Firefox are not important. The relevant observation from this work is that there's a good amount of variety in the shape of the

[1] Monaco, CodeMirror, Ace | TipTap, Quill, EditorJS, TinyMCE

work done within popular libraries, which indicates there's value in including more than just the Monaco test. Here are some observations about each:

- **Monaco** is JS and worker heavy, and the test does cause some jank across browsers. It seems like an obvious candidate for inclusion as it's extremely popular and the de facto option for in-browser IDEs these days. There's a busy worker that runs even after rendering is complete (presumably still trying to parse the large JS file for syntax highlighting).
- **CodeMirror** is JS heavy, along with some layout and DOM. *Note: this integration revealed additional Firefox-specific jank in DOM which is being investigated, but is not relevant to whether we include it or not.*
- **Ace** is JS heavy, along with some layout and GC. I have not integrated this perfectly, as it's throwing an error at "http://localhost:5173/worker-javascript.js".
- **TipTap** is reflow heavy. It also seems to be a popular newer choice for WYSIWYG and is likely worth including.
- **Quill** spends a lot of time in DOM. It's using MutationEvents which is not recommended, and should probably be omitted for that reason.
- **EditorJS** is causing the most jank. It's quite different from the others, being a "block editor", so it's likely not being integrated in a typical way (with long stretches of normal paragraph text). Also unsure if this would not be a good proxy for "block editor" style applications.
- **TinyMCE** is layout heavy. Good candidate for inclusion as it's very popular, and seems to be an extremely thin wrapper around contentEditable (i.e. you call execCommand directly for formatting operations). This is probably not an option though due to licensing. CKEditor wasn't tested but may have similar characteristics and includes MPL so we may consider if everyone would be OK with that. One more option could be to directly use contentEditable.

All of these create interesting profiles, but the most likely candidates for new workloads based on this analysis are Monaco, CodeMirror and/or Ace (selecting only one if we want to limit representation of code editing), TipTap, and some sort of thinner contentEditable library dependant on licensing.

Workload Limitations

It appears to not be possible within the framework to test actual text editing (by simulating key events etc), so "editing" is restricted to `setValue` and `formatRange` style API calls to the editors. This is not fully realistic to real world usage, although these API calls are typically required for initial rendering and interactivity and likely slower than incremental modifications as a result of key presses.

Appendix: Research Notes

Following are raw notes from the investigation. They're left for transparency, but probably aren't interesting on their own.

Potential elements to test

- Rich text vs code
  - Q: Are these different enough performance-wise to include both?
    - A test harness page for research which loaded the top N of each kind and allowed us to profile while setting text in each would be very helpful to assess this.
      - [bgrins] I've started working on this at https://github.com/bgrins/editor-tests / https://bgrins.github.io/editor-tests/ . Here's a profile as of Feb 9 covering CodeMirror, Ace, TipTap, Quill, and Editor.js https://share.firefox.dev/3JXc6rw. Monaco to follow.
- Editing vs viewing
  - Q: GrandPrix workload only loads the text, should we also simulate edits? Can we effectively simulate edits across all rich editors using the framework?
    - Smaug: Some kind of input could get through using document.execCommand. Key events are trickier, since the testing framework can't create trusted events
    - Bgrins: Unfortunately our best bet may be to rely on `setValue` type functions for each library. I suspect this will hit most of the expensive work that would happen with real key events. This can be used to capture "length and complexity of text" below
- User chrome (toolbars, buttons, etc)
  - Unlikely to have a material impact on scores. It's not a bad idea to add a gdocs or vscode-like chrome to an app shell to exercise inline SVG icons and additional layout, but I wouldn't make it a variable for tests and I also wouldn't prioritize that over core workload development.
- Library choice
  - This is likely the best candidate to generate **multiple workloads**. There appear to be some widely popular choices, and my hypothesis is that they will have fairly different performance characteristics, but we need to validate that.
- Length and complexity of text
  - I think multiple variations can be supported in a single workload (as per above this could be done by setting and then resetting the value in an editor as separate steps). My intuition is that we should have 2: a "relatively simple" and "relatively complex" set of text to load in each workload.
- Formatting (syntax highlighting, B/I/U)

Resources

- https://www.wappalyzer.com/technologies/rich-text-editors/
    - TinyMCE, Ace, CKEditor, …
    - There are likely some methodological issues with collection here. Hypothesis - due to reliance on scraping and rich editing features are likely to be behind logins or at least some user interaction. Library detection can also be tricky. However this is a good starting point and indicates that we should look at some of the older and more direct contentEditable libraries like TinyMCE. Will add to the test page.
- Switching Rich Text Editors (Feb 2022) https://www.ashbyhq.com/blog/engineering/tiptap-part-1. They went with TipTap after evaluating the following: Slate 0.63.0 (20.6k ⭐), Quill 1.3.7 (29.9k ⭐), ProseMirror (5.7k ⭐), ReMirror1.0.0-next.60 (1.1k ⭐), tiptap 2.0.0-beta60 (10.8k ⭐)
- A good resource comparing popular editors from Quill's doc https://github.com/quilljs/quill/blob/795dd1e407d404d0fb0a1918dc0197f6cb937267/website/content/guides/comparison-with-other-rich-text-editors.mdx
    - CKEditor and TinyMCE are widely used but expose contentEditable directly to users. Quill and others maintain document state and provide a higher level abstraction.
    - Unlike Quill, Draft has a React dependency and it also does block editing.
    - Quill and ProseMirror and Trix are roughly aligned

Popular libraries

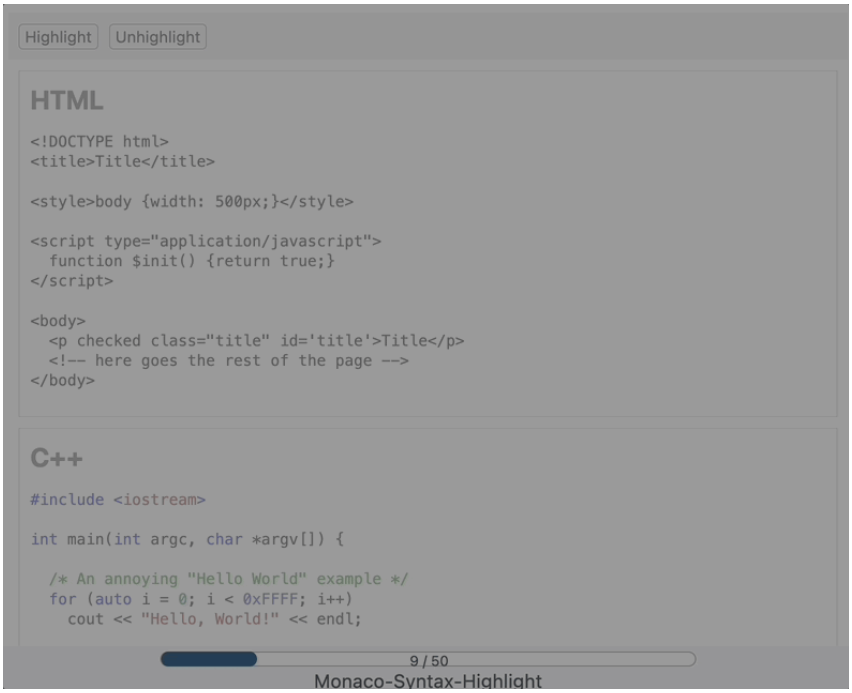
- Popular code editors: My sense is that there are 2-3 primary underlying libraries here
    - **CodeMirror** (3m weekly npm downloads, 3.5k github stars). Used by jsfiddle, codepen,
        - https://github.com/codemirror/dev/
        - https://www.npmjs.com/package/codemirror
        - [bgrins] Does this current version use contentEditable?
    - **Monaco** (700k weekly npm downloads, 32k github stars). Used by GitHub, StackBlitz, CodeSandbox, Repl.it. Probably the most obvious inclusion in the category.
        - https://www.npmjs.com/package/monaco-editor
        - https://github.com/microsoft/monaco-editor
        - Question: Does this use contentEditable or something self managed? According to this it uses a hidden textarea. Is that still true?
        - Integration with vite uses module workers (only supported in Firefox Nightly atm) https://twitter.com/youyuxi/status/1355316139144970240?lang=en
    - **Ace** (500k weekly npm downloads, 25k stars)
        - https://github.com/ajaxorg/ace
        - https://www.npmjs.com/package/ace-builds
        - Appears to use a hidden textarea for input + normal DOM for rendering

- Rich text editors: My sense is that there are less clear winners here, although I believe many are built on top of underlying libraries like ProseMirror. From a quick audit it appears that these are pretty much all built on top of contentEditable, so it's possible that they may have similar performance characteristics. Many maintain their own document state on top of that and so it's possible they act quite differently, especially with large docs or lots of edits.
    - Sourced from https://github.com/topics/rich-text-editor & https://github.com/topics/wysiwyg.
        - **Quill** (35k stars, 1.1m downloads). last published on npm 3 years ago. Also uses Mutation Events.
            - https://github.com/quilljs/quill
            - https://www.npmjs.com/package/quill
            - Note: used by hrblock
        - **Slate** (25k stars, 550k weekly downloads). "You can think of it like a pluggable implementation of contenteditable built on top of React. It was inspired by libraries like Draft.js, Prosemirror and Quill."
            - https://github.com/ianstormtaylor/slate
            - https://www.npmjs.com/package/slate
            - Marked as "in beta" in the repo
            - Structure is specified as an object (similar to Editor.js), see example at https://github.com/ianstormtaylor/slate/blob/d0d1cb981b469d52316d5d134df395f02f3b3bf7/site/examples/richtext.tsx
        - **Trix** (17k stars, 180k weekly downloads). Appears to wrap over contentEditable as per README.
            - https://github.com/basecamp/trix
            - https://www.npmjs.com/package/trix
        - **Editor.js** (21k stars, 40k weekly). Block style. Appears to use contenteditable for each block.
            - https://github.com/codex-team/editor.js https://www.npmjs.com/package/@editorjs/editorjs
        - **TipTap** (18k stars, ~200k npm downloads). Uses ProseMirror under the hood, so contenteditable
            - https://github.com/ueberdosis/tiptap
            - https://www.npmjs.com/package/@tiptap/react - React is more popular wrapper
            - https://www.npmjs.com/package/@tiptap/vue-3
            -
            - [bgrins] I've used this one and it has lots of good docs / examples
        - **ProseMirror** typically used as a dependency for higher level editors (see TipTap for example). Uses contenteditable.
            - https://github.com/ProseMirror/prosemirror
        - **TinyMCE** (12k stars, 400k downloads). One of the OGs, uses contentEditable under the hood
            - https://github.com/tinymce/tinymce

        - https://www.npmjs.com/package/tinymce
    - **Gutenberg** (8.5k stars, 25k downloads, GPL license) Used by Wordpress. A broader scope than some of these, intended to enable full site editing.
        - https://github.com/WordPress/gutenberg
        - https://www.npmjs.com/package/@wordpress/block-editor
    - ~~https://github.com/yabwe/medium-editor (15k stars) No recent updates~~

Existing Workloads

- GrandPrix has a Monaco test https://github.com/GoogleChromeLabs/GrandPrix/tree/main/resources/benchmarks/monaco-editor. It opens various code files with and without syntax highlighting (see screencast below)
    - I think we can add this to tentative/ in a fork as a test harness for the libraries above
- I've set up a harness for evaluating a number of the popular libraries at https://github.com/bgrins/editor-tests



Profile findings

Running https://bgrins.github.io/editor-tests/ in Firefox, I observe the following profile https://share.firefox.dev/41jRpfm and callstacks (gathered by setting the selection to each usertiming duration and switching to Call Tree tab).

Code Editors

Monaco	CodeMirror	Ace
389ms JS > Layout > DOM	726ms DOM > JS > Layout Note that the DOM timing may be Firefox-specific jank	216ms JS > Idle > Layout > GC I haven't integrated the language Worker properly
nsThread::ProcessNextEvent xpcom/threads/nsThread.cpp **Call node details** Traced running ... 389ms Traced self time — Running samples 100% 384 Self samples — — **Categories** Running sample count JavaScript 68% 261 Layout 14% 53 DOM 6.3% 24 Other 4.7% 18 GC / CC 3.6% 14 Graphics 1.6% 6 Idle 1.0% 4 Profiler 1.0% 4 **Implementation** Running sample ... Native code 32% 123 JS interpreter 6.3% 24	**nsThread::ProcessNextEvent** xpcom/threads/nsThread.cpp **Call node details** Traced running ... 726ms Traced self time — Running samples 100% 721 Self samples — — **Categories** Running sample count DOM 34% 246 JavaScript 28% 200 Layout 27% 194 Idle 6.5% 47 GC / CC 2.1% 15 Profiler 1.5% 11 Other 0.6% 4 Graphics 0.6% 4 **Implementation** Running sample ... Native code 72% 521 JS interpreter 2.5% 18	**nsThread::ProcessNextEvent** xpcom/threads/nsThread.cpp **Call node details** Traced running ... 216ms Traced self time — Running samples 100% 212 Self samples — — **Categories** Running sample count JavaScript 43% 92 Idle 18% 39 Layout 14% 30 GC / CC 11% 24 DOM 7.1% 15 Graphics 2.8% 6 Profiler 1.9% 4 Other 0.9% 2 **Implementation** Running sample ... Native code 57% 121 JS interpreter 8.0% 17

WYSIWYG Editors

TipTap	Quill	EditorJS	TinyMC
315ms Layout > JS > GC	367ms JS > Layout > GC	1714ms JS > Layout > DOM	316ms JS > Layout > Idle

nsThread::ProcessNextEvent
xpcom/threads/nsThread.cpp

Call node details

Traced running ...		315ms
Traced self time		—
Running samples	100%	313
Self samples	—	—

Categories	Running sample count	
Layout	49%	152
JavaScript	16%	49
GC / CC	15%	47
Idle	11%	35
DOM	5.8%	18
Other	1.9%	6
Graphics	1.3%	4
Profiler	0.6%	2

Implementation	Running sample ...	
Native code	85%	266
JS interpreter	2.2%	7

nsThread::ProcessNextEvent
xpcom/threads/nsThread.cpp

Call node details

Traced running ...		367ms
Traced self time		—
Running samples	100%	367
Self samples	—	—

Categories	Running sample count	
JavaScript	49%	180
Layout	20%	73
GC / CC	11%	39
Idle	6.8%	25
DOM	6.5%	24
Profiler	4.4%	16
Other	2.2%	8
Graphics	0.5%	2

Implementation	Running sample ...	
Native code	51%	187
JS interpreter	1.1%	4

nsThread::ProcessNextEvent
xpcom/threads/nsThread.cpp

Call node details

Traced runni...		1,714ms
Traced self ti...		—
Running sam...	100%	1,706
Self samples	—	—

Categories	Running sample count	
JavaScript	70%	1,191
Layout	14%	235
DOM	6.2%	106
GC / CC	5.3%	91
Other	1.9%	32
Idle	1.2%	20
Profiler	1.1%	18
Network	0.4%	7
Graphics	0.4%	6

Implementation	Running sample ...	
Native code	31%	527

nsThread::ProcessNextEvent
xpcom/threads/nsThread.cpp

Call node details

Traced running ...		316ms
Traced self time		—
Running samples	100%	316
Self samples	—	—

Categories	Running sample count	
JavaScript	35%	111
Layout	35%	110
Idle	10%	32
DOM	7.0%	22
GC / CC	6.3%	20
Other	4.7%	15
Profiler	1.6%	5
Graphics	0.3%	1

Implementation	Running sample ...	
Native code	65%	205
JS interpreter	5.1%	16

Learnings

- Be more systematic with library metadata collection (license etc) and selection for inclusion in a test harness
- Actually integrating code into a webpage is invaluable. The Vite boilerplate for https://github.com/bgrins/editor-tests/ was good enough - it's been iterated on in https://github.com/bgrins/data-ui-tests, and I've also documented how to host it easily on gh-pages with https://briangrinstead.com/blog/publishing-vite-project-to-github-pages/
- Async initialization should be handled better (try to simulate what Speedometer does). For example the Monaco language worker.