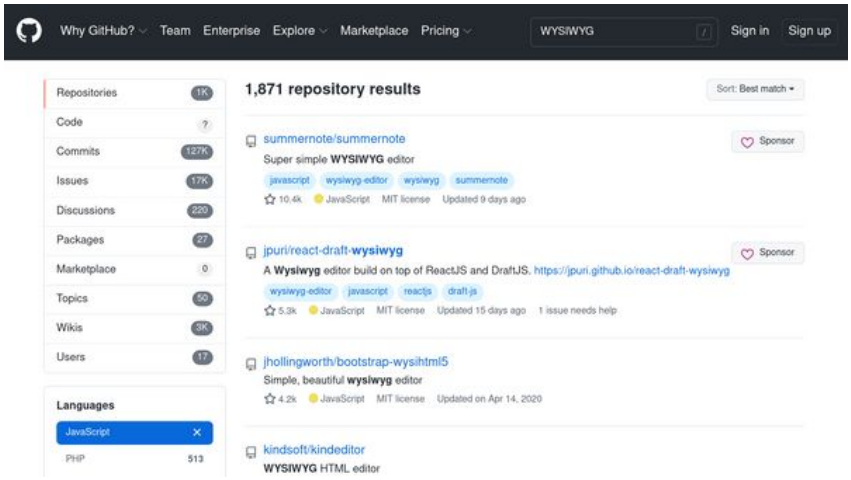
My pain building a WYSIWYG editor with contenteditable

In my years of frontend experience, this project gave me the wildest ride in terms of rare bugs and unexpected behavior.

6 min. read View original

development 17/09/2021

Naturally, when a text editor comes up in a discussion, creating a custom solution is not the first thing that comes to mind. So I went ahead, and I started looking for what reliable WYSIWYG editors I could use. There are quite a few ready solutions on the internet. Some are famous and used by well-known platforms such as WordPress. But many are also abandoned and are left to collect dust in GitHub repositories.



After lots of time searching and testing editors online, it turns out that we have some rather unique requirements. None of the available editors met our criteria, so I was set on an adventure to code a custom WYSIWYG editor.

In my years of frontend experience, this project gave me the wildest ride in terms of rare bugs and unexpected behavior. I've googled so hard that I ended up finding obscure WebKit bugs that aren't closed since a decade ago!

I want to clear out that I have only used modern Chrome and Firefox. Contenteditable in Safari and older browsers is another world that I will cover in future blog posts.

A custom WYSIWYG editor in 2021?

You might ask yourself, do we need an in-house solution when there are hundreds of options out there? And the answer is surprising, yes - we do!

The nature of our services asks for it; we need a custom data type to store and process the

contents of a contenteditable for later use in our product line. For example, the content you create through our editor must be searchable, and also it is rendered in all of our products in very diverse styles - such as in the helpdesk widget, search widget, or our knowledge base platform.

The current solutions don't offer too much control in processing the data from the contenteditable - the output is always dirty HTML, and storing and processing HTML generated by a contenteditable in the database is a no-go.

Oh, and we also needed as many elements as possible, including custom ones which not only the current libraries didn't support - but also it's not possible to extend functionality or register new ones unless you fork the whole thing.

Our editor currently supports:

- Bold
- Italic
- Underline
- Headings

- Bulleted List
- URLs
- Inline Code
- Images
- Table
- Horizontal Dividers
- Blockquote
- Code block (with syntax highlights and multi-language support)
- Youtube videos
- and a few more

Starting the development

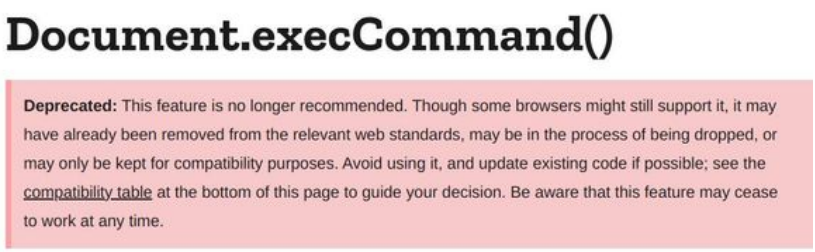
My thought process was to use the contenteditable attribute to create a straightforward editor first - by following best practices available online. The kind of editor that only supports bold, italic, and underline. And then figure out a way to extend functionality and add custom elements as we need them.

My thinking took that direction because when I see open source projects such as Firefox or Chromium, "inconsistent mess" is not the first thought that goes in my mind. I thought the

current contenteditable attribute could handle such a simple editor - but I was wrong.

Pain #1: execCommand is deprecated

To make a text bold with contenteditable, you have to use the JavaScript method `execCommand` which, if you look up into the MDN, it's a deprecated method! And there is no other method available. It's the only one you can use to style text in a contenteditable!

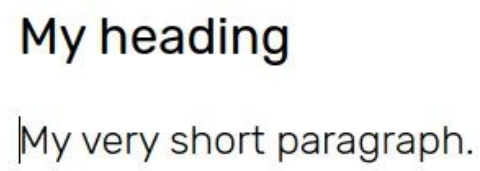


Frankly, this is quite a red flag, but I decided to push forward anyway.

Many editors on the planet right now are at risk of simply stopping working in the next version of Chrome or Firefox when the browser vendor decides to drop `execCommand` altogether. I haven't seen any editors that don't use the `execCommand` method yet.

Pain #2: Backspace merge from hell

Picture this scenario: Your cursor is at the beginning of a paragraph. Just before this paragraph, there is a heading (H2), and you hit backspace. What should happen?



In layman's terms: the paragraph should join the heading. And in more technical terms: the contents of the P tag should go inside the contents of the H2 tag. But this is not what was happening when I looked in dev tools.

```
<!-- current HTML -->
<h2>My heading</h2>
<p>My very short paragraph.</p>

<!-- expected outcome -->
<h2>My headingMy very short paragraph.</h2>
```

Upon hitting backspace, the browser decides to convert the P tag into a SPAN - and then it applies some inline CSS to this SPAN to make it look like the H2 tag and then puts this new SPAN element inside the H2 tag.

```
<!-- actual outcome -->
<h2>My heading<span style="font-size: 20px; line-
height: 25px;">My very short paragraph.</span></h2>
```

To add more to this complexity, if the paragraph has styles such as bold and italic - it will also convert the B and I elements into a SPAN and preserve the text style with inline CSS through the `text-style` property.

When I found out that contenteditable is committing sins with HTML elements, I questioned whether I would finish this project.

But this is only pain #2, so keep reading to find out more!

Pain #3: Funny line breaks

The most well-known way to add a line break (vertical space) since Firefox 1 is by using the BR tag.

When the cursor is in the middle of a chunk of text, and you hit Enter twice - you would naturally think that chunk of text gets separated in two, and then contenteditable adds two BR tags in between.

```
<!-- current HTML -->
<p>.. some very long paragraph with multiple <u>styles
such as, <b>bold</b> and <i>itallic</i></u></p>

<!-- expected outcome -->
<p>.. some very long paragraph with multiple <u>styles
such as,</u></p>
<br>
<br>
<p><u><b>bold</b> and <i>italic</i></u></p>
```

Instead, what happens is that the two BR tags do get added, but for some reason, also, contenteditable wraps the BRs around the tags that happened to wrap the text before you hit enter twice - resulting in some of the fanciest BR tags I've seen!

```
<!-- actual outcome -->
<p>.. some very long paragraph with multiple <u>styles
such as,</u></p>
<p><u><br></u></p>
<p><u><br></u></p>
<p><u><b>bold</b> and <i>italic</i></u></p>
```

No matter how nested the markup is, these funny br tags will follow.

For example:

- `<p><b><br></b></p>`
- `<p><i><u><br></u></i></p>`
- `<h2><b><u><br></u></b></h2>`

This behavior seems like a bug at first because the outcome is simply voodoo. But (I think) the browser is trying to preserve the text style, just in case you decide to write text inside these line breaks.

Pain #4: You can style everything!

As long as you can select a piece of text in contenteditable, the shortcut CTRL + B will make the text bold, and it doesn't matter where it sits or for what reason.

Column 1	Column 2	Column 3
you **shouldn't** be *able* to style this table	A2	A3
but you can!	B2	B3

As I mentioned at the beginning of this blog post, our editor has custom elements such as tables or code highlights. It made no sense to allow text styling for many of these custom elements, but it was possible anyway!

I got around this issue, though, by adding `contenteditable=false` to the wrapper DIV of our custom elements.

What's next

The default behavior in contenteditable is not fun. It feels like it's a hack on top of a hack - it doesn't think too far ahead, and if you allow it, it will nest your HTML infinitely as long as the content looks good on-screen.

I'm creating an open-source version of a new editor - one that I would build after getting all this experience and knowledge from working with the editor at Answerly.

This new editor would handle all keystroke events manually and would allow for some very consistent configurations and features such as:

- Choosing what elements to use for text styles
- Consistent line breaks
- Better merge
- Adding and editing custom elements
- Configuring the outcome of the contenteditable (Ie. JSON, Markdown)

I will be sharing my development story and code in this blog.

The future of the real contenteditable

Contenteditable will be consistent, eventually, whether with existing proposals or something completely new. But say it updated tomorrow, it would take a lot of time until browser adoption grows - so until now, we're stuck with the current solutions.

If you search the internet, two propositions seem to have the most traction and promise to improve the current contenteditable.

contenteditable=minimal

This proposition from Ben Peters @ Microsoft suggests a minimalistic version of contenteditable that only supports caret drawing and moving, some keyboard events, and typing characters. This contenteditable would allow the user to handle the styling manually, something similar to what I will be doing with my open source project.

Polymer elements and Shadow DOM

An old proposition includes rebuilding contenteditable with Polymer Elements and the Shadow DOM. It would work as the current contenteditable but allow for a lot more customization and more events.

Many blogposts cover this last proposition according to my research, but none include a source, so I'd take it with a pinch of salt.

References:

- contenteditable=minimal proposition in w3
- Merge bug in webkit
- execCommand is deprecated
- contenteditable in MDN