ChatGPT

Refining Inline Annotation Behavior for Safari Cursor Issues

Problem Overview

In Safari (WebKit), clicking in the **middle of an annotated inline span** (e.g. a TipTap/ProseMirror mark or inline node) often fails to show the text cursor at that spot. Users report that **Safari only displays the caret when clicking at the end of the annotated text**, whereas Firefox correctly places the cursor wherever you click. This is a known browser quirk – Safari tends to treat a styled or non-editable inline element as a single unit for cursor placement, leading to the **inability to position the caret in the middle of the span**. The goal is to adjust the HTML/CSS for these annotation spans so that Safari will allow a caret **at the exact click location**, without disrupting behavior in other browsers (Firefox, Chrome).

Below we evaluate several refinement strategies and their effects:

1. Using `display: inline-block` on Annotation Spans

Applying `display: inline-block` to the annotation's `<span>` forces it to behave more like an atomic object in layout, which surprisingly **helps Safari place the caret correctly inside the span**. This approach has been recommended by ProseMirror's author for inline nodes: it prevents Chrome/Safari from collapsing caret positions at span boundaries [1]. In practice, developers confirmed that making an inline annotation span an inline-block **fixed the cursor placement issues in Chrome and Safari (while Firefox was unaffected)** [2] [3]. With this CSS, clicking anywhere on the annotated text in Safari should immediately show the caret at that position (instead of only at the edges).

Side Effects: The main side effect observed is an **increased caret (cursor) height** in some cases. Because an inline-block element establishes its own line box, the blinking cursor inside it may stretch to the full height of the element or line. For example, one developer noted that after adding `display:inline-block` to a contenteditable span, **the text cursor became abnormally tall in Safari and Firefox** [4]. This often happens if the span has padding or if line-height is not adjusted, making the caret look too large. To mitigate this, you can ensure the span's CSS doesn't inadvertently enlarge the line-height. Techniques like setting `vertical-align` or explicitly defining a matching `line-height` for the span can normalize the caret size. In most cases, however, the benefit of correct cursor placement outweighs the minor styling tweaks needed to address caret height.

Compatibility: Firefox and other browsers generally tolerate the inline-block change. Firefox already handled the caret correctly; it continues to do so with inline-block, since that CSS does not break editing. Chrome also benefits similarly to Safari (as noted in ProseMirror discussions [1]). Overall editing behavior (typing, deleting text within the span) remains unchanged – the span still contains text and is editable. Thus, `display:inline-block` is a **straightforward and effective fix** for Safari's cursor issue, with only cosmetic adjustments needed for the caret.

2. Adding a Small Right Margin to Annotation Spans

Another tweak is to give each annotation span a tiny **right-side margin (e.g. `margin-right: 1px`)**. This does not directly affect clicking *within* a single span, but it addresses a related Safari quirk when **two annotated spans are adjacent with no normal text between**. In Safari (and Chrome), when two inline elements abut, the browser can have trouble placing the caret in the narrow gap between them. By adding a 1px margin, you ensure there is a sliver of space that the user can click, making Safari treat the spans as separate clickable targets. A ProseMirror user reported that **he could not place a cursor between two back-to-back inline nodes until he added a small right margin, after which Safari allowed a caret between them** [5]. This trick effectively gives the caret a "landing zone" between touching spans.

Side Effects: The margin introduces a tiny visual gap. In most cases a 1px space is barely noticeable, but if your annotations have a background or underline, users might see a slight break. It's usually acceptable for inline annotations like mentions or highlighted terms – the gap is a reasonable trade-off to enable cursor positioning. If a completely seamless look is required, one could consider using a zero-width space (next approach) instead of a visible margin, though that has its own trade-offs.

Margin does slightly alter text layout (e.g. it can affect where line-breaks occur by adding a hair of extra width per span). However, 1px is typically negligible for layout, and other browsers (Firefox, Chrome) handle it fine. In Firefox, you likely could also place the caret between adjacent spans even without the margin, but the margin won't break anything – it just ensures **consistency across browsers** for selecting the gap.

Summary: `margin-right:1px` is a useful supplementary fix, especially when multiple annotation spans appear in succession. It **helps Safari register clicks between spans as intended**. On a single isolated span, this has no effect on clicking in the middle of the text (since that involves internal span behavior), but it does ensure clicking immediately after an annotated span (at its boundary) works. Typically, this margin is used *in combination* with other fixes (like inline-block or zero-width characters) for full coverage.

3. Inserting a Zero-Width Space inside the Span

Inserting an **invisible character** – specifically a zero-width space (ZWSP, Unicode `U+200B` `&#8203;`) – at strategic points in the annotated text is a classic contenteditable hack. The idea is to break the span's text node into multiple pieces so that Safari has an internal position to place the caret. For example, you might wrap the annotated text like: `<span class="annotated">Hel&#8203;lo</span>`, inserting a zero-width space in the middle (or at the start/end). This creates an extra potential caret position that Safari might utilize when you click around that area.

Effect on Safari: By splitting the text node, Safari may no longer treat the entire span as one continuous unbreakable unit for selection. If the issue was that Safari wasn't drawing a cursor mid-span, a ZWSP could nudge it to do so by providing a text boundary. This approach is somewhat hit-or-miss – it often helps with **caret placement at the very start or end** of an inline element. In fact, Atlassian Confluence uses a related tactic by injecting a zero-width **no-break space** (`U+FEFF`) *around* mention tags to ensure the caret can sit before/after them [6] [7]. By analogy, putting a zero-width character inside could give Safari a place to anchor the cursor at that point.

Side Effects: The major drawback is that you are adding phantom characters into the document. Although zero-width spaces are invisible, they are still part of the text content. **Users can accidentally delete them** or move the cursor through them while editing. For instance, one source notes that adding invisible characters inside an otherwise non-editable span "works, but the invisible characters **can be deleted**" by the user during editing [8]. If a ZWSP is removed, the Safari issue might reappear at that spot. Another concern is copy-paste: if the user copies text containing these characters, they might carry over (usually harmless, but they could affect things like string length or certain text processing logic).

From a maintenance perspective, littering the content with zero-width spaces can become messy. It may also confuse algorithms that don't expect them. Therefore, while this method **can make the Safari caret appear in more places**, it's a bit of a fragile solution. It's often considered a last resort when other CSS approaches fail, because it directly manipulates the content data. If used, it might be better to insert ZWSPs *only at boundaries* (start or end of the span) rather than between every character, to minimize clutter. For example, appending a `&#8203;` at the end of the span text can ensure Safari can place the caret *just inside* the end of the span, appearing as if in the middle when clicked (though in reality it's at that zero-width character position).

Recommendation: Use sparingly. This can be combined with marking the ZWSP as non-editable (e.g., wrap it in a `<span contenteditable="false">&#8203;</span>` so that the user can't easily delete it). But at that point, one might consider approach 4 instead (explicit non-editable wrappers). In summary, zero-width spaces do **allow additional cursor positions in Safari**, but they introduce hidden content management overhead.

4. Making the Annotation Span Non-Editable (with Editable Buffers)

This approach treats the entire annotated segment as an **atomic, uneditable node**, and uses tiny editable buffers (like zero-width spaces or other elements) just before and after it to catch the cursor. Essentially, you would set `contenteditable="false"` on the annotation `<span>` itself, meaning Safari (and other browsers) will not allow a caret inside that span at all. To compensate, you ensure there are caret positions immediately adjacent to the span by inserting, for example, an editable zero-width space on each side. The structure might look like:

```
<span class="caret-buffer" contenteditable="true">&#8203;</span>
<span class="annotated" contenteditable="false">[Annotated Text]</span>
<span class="caret-buffer" contenteditable="true">&#8203;</span>
```

With this setup, when you click "in the middle" of the annotated text, the browser will actually place the caret into one of the adjacent invisible buffer spans (either just before or just after the non-editable span, whichever is closer to the click). The effect is that the cursor appears next to the annotation, which for a user clicking the middle might be visually indistinguishable from "in the middle" of it (since you can't actually enter the non-editable span). This ensures the caret is visible near where the user clicked, and any typing will happen just outside the annotated block.

Safari Behavior: This method is quite robust for Safari's issues, because it **sidesteps Safari's internal quirks entirely by removing the editable region inside the problematic span**. Safari will always position

the caret in an editable element, so it ends up in one of the zero-width siblings. For instance, Confluence's example for an inline date uses this pattern: an uneditable `<time>` tag wrapped with `&#65279;` no-break spaces on both sides [7]. That guarantees the user can click around the date and get a caret either before or after the `<time>` element, even though the date itself isn't directly editable. ProseMirror's own NodeView implementations often employ similar tactics for atomic inline nodes (like embedded widgets or tags).

Side Effects: The big consideration here is that **the annotated text can no longer be edited character-by-character** by the user. If your use case is something like a **mention or special token that should be kept intact**, this is fine (users weren't supposed to edit inside it anyway). But if these annotations are simply highlights or comments on text that users still need to freely edit, making them non-editable is not appropriate. You would be preventing the user from inserting or deleting letters inside the annotated range – effectively treating it as one unit that can only be removed or moved as a whole.

Another challenge is the complexity of contenteditable "islands." Creating a non-editable parent with editable children (or vice versa) can confuse browsers. It's known that one **should not create overlapping editable/non-editable regions**, as it can break selection updates [9] [8]. In our case, we place distinct editable spans around the non-editable one, which is a bit cleaner than an editable island inside a non-editable, but it still must be handled carefully. For example, ensure that the buffers are truly zero-width or otherwise have no effect on copy-paste or text output, aside from caret placement. ProseMirror's developers have noted that when using contenteditable=false inline nodes, you sometimes need to implement custom arrow-key or deletion handling, since the browser might skip or not skip the node in ways you have to manage in code [10] [11]. In fact, one workaround to unify cross-browser behavior was to programmatically add zero-width spaces around a contenteditable=false node in the DOM so that arrow key navigation and mouse clicks behave consistently [12] [13]. This underscores that while the **non-editable span + ZW spaces technique does solve the immediate cursor visibility issue in Safari**, it can introduce complexity in the editor logic.

Compatibility: Firefox and Chrome generally handle atomic inline nodes and adjacent whitespace similarly, especially if using ProseMirror/TipTap's built-in gap cursor (which displays a caret-like gap at boundaries of such nodes). Firefox actually tends to handle caret around non-editable nodes more gracefully out of the box. The presence of the buffer spans means even browsers that normally might not allow a click directly at that boundary will have an element to focus. So cross-browser, the approach works, but the **editing experience changes** (no internal editing of the annotation). If that aligns with the product requirements (e.g., annotations are meant to be immutable once applied), this strategy is effective. If not, it's likely too heavy-handed.

5. WebKit-Specific CSS Tweaks (`-webkit-user-modify`, `caret-color`, etc.)

Safari/WebKit supports some non-standard CSS that can influence contenteditable behavior. One such property is `-webkit-user-modify`. Setting an element to `-webkit-user-modify: read-write-plaintext-only` essentially tells Safari to treat the content as plain text (disallowing any rich text formatting within that element) [14]. This mode can have a side effect of simplifying how the browser handles the caret and selection. By applying this CSS to the annotation spans, we aim to coax Safari into handling the span's text more like a normal text node. In theory, this could **allow the cursor to appear at**

the click point because Safari isn't trying to maintain some complex styled span selection – it's treating it as plain text editing. This property has been used to solve various Safari contenteditable oddities (for example, it prevents Safari from preserving styles on paste and can avoid certain selection jumps [14]).

To use this approach, one would add something like:

```
.annotation-span {
  -webkit-user-modify: read-write-plaintext-only;
  caret-color: auto;
  user-select: text;
}
```

The `caret-color` property here is to ensure the caret remains visible. In some cases, if the annotation has a background or color, Safari's caret might default to white or transparent. For example, if you had a dark highlight on the span, you might explicitly set `caret-color: black` (or just use `auto` which lets the browser decide a proper contrast). The `user-select: text` simply reinforces that the text should be selectable in case the `user-modify` or other styles interfere (usually `read-write-plaintext-only` already implies text selection is allowed, but it doesn't hurt to be explicit).

Effectiveness: This approach is somewhat speculative but rooted in known WebKit behaviors. By forcing plaintext editing mode, you remove Safari's tendency to create or manipulate `<span>` tags internally when editing. It may also break some of Safari's hesitance around placing cursors in styled spans. However, **there isn't widespread documentation of using `user-modify` specifically to fix the mid-span click issue**. It's a newer idea drawn from similar problems. It won't hurt Firefox or Chrome (they simply ignore the unknown `-webkit-user-modify` property, and `caret-color`/`user-select` are standard). So it's safe to try. In fact, some editors (like Obsidian's live preview and CodeMirror) apply `-webkit-user-modify: read-write-plaintext-only` to their entire editable area to tame Safari's behavior [15]. This indicates it's a reliable way to get Safari into a more predictable state for text editing.

Side Effects: The primary side effect is that **Safari will only allow plain text operations in that span**. If your annotations are just text with a colored background or underline, that's fine – users weren't going to add sub-formatting inside it anyway. But `read-write-plaintext-only` means if a user tried to bold part of that annotated text or insert a line break, Safari wouldn't do it (it treats it as a plain text field). This generally aligns with how one expects an annotation span to behave (they usually don't want arbitrary formatting inside). Another side effect is on paste behavior: Safari will strip any formatting from content pasted into that span (again, likely fine or even desirable).

This property also has some quirks on older Safari versions (and it's non-standard), but since we're targeting Safari specifically, that's acceptable. The inclusion of `caret-color` is solely to address any **caret visibility issues** – for instance, ensuring the blinking cursor is obvious against any custom background the annotation might have. It doesn't influence placement, just appearance.

Summary: Using `-webkit-user-modify: read-write-plaintext-only` can be seen as a **Safari-specific refinement** that potentially fixes the cursor placement by simplifying Safari's editing model for those spans. It should be combined with the standard `user-select: text` (to ensure text selection

works normally) and a defined caret color. While not as commonly cited as the inline-block solution, this approach is low-risk to try and can complement other fixes. If Safari was misbehaving due to treating the annotated span as a complex rich element, this forces it into a plain text mode and may thereby resolve the issue without additional dummy characters or margins.

Comparison of Approaches

To recap the outcomes in Safari and elsewhere:

- **Inline-Block (Approach 1):** Proven to make Safari place the caret at click positions inside the span [1]. Needs a small CSS tweak to avoid tall cursor (e.g. adjust line-height). No negative impact on Firefox (Firefox never had the bug and still works normally). Chrome benefits similarly. Overall keeps the content model the same (text remains editable). This is a **widely adopted fix** for cursor placement issues in contenteditables.

- **Margin on spans (Approach 2):** Helps specifically with **caret between adjacent annotations** [5]. It doesn't directly fix clicking *inside* a single span, but it's a good complementary fix if your content can have back-to-back annotated spans. Minimal visual impact (1px gap) and no effect on internal span editing. Safe across browsers. Usually used alongside another fix.

- **Zero-Width Space inside (Approach 3):** Can force Safari to acknowledge an internal break in the text, thus enabling mid-span cursor. However, it **introduces hidden characters** into your text that can be accidentally removed [8]. It's a bit of a hack and can complicate text handling (especially in an editor like ProseMirror which tries to manage document text). Not needed if a CSS solution works. Could be a quick fix for specific edge cases (like ensuring there's always a trailing invisible char so you can click at end of span, etc., which Safari then might position before that char – effectively looking like mid-span). Use with caution.

- **Non-Editable Span with Buffers (Approach 4): Guarantees cursor placement** (since the span itself isn't focusable, the caret goes to neighbors) and is a solid approach when annotations represent protected tokens (e.g., @mentions). It aligns with how some rich editors handle special entities [7]. But it fundamentally changes editing: users can't modify the span text itself without special handling. It may overkill the original problem if your goal was just to highlight text that users can edit. Also, implementing it can be tricky (must manage the buffer elements and possible selection issues). Best suited when you intentionally want the annotated text to be immutable and separate from normal typing.

- **WebKit Plaintext Mode (Approach 5):** Attempts to solve the Safari issue *within Safari's rendering logic* via CSS. It doesn't add extra DOM elements or characters, and doesn't restrict editing in non-WebKit browsers (they ignore it). This is a comparatively clean solution if it achieves the desired effect. It has less community reportage specifically for cursor fixes, but it is known to stabilize Safari's contenteditable behavior [14]. The risk is low – if it doesn't fix the cursor, you can remove it; if it does, it's a very clean fix. Ensure to test that it doesn't interfere with any needed rich-text behavior inside the span (likely not, if your annotations are simple).

Recommendation

The most effective strategy (as of our evaluation) is to **use a CSS-based solution combining inline-block and a small margin**, which addresses Safari's cursor placement peculiarity without altering the document content or general editing flow. In practice, setting annotation spans to `display:inline-block` resolves the primary issue – Safari will allow cursor placement at any click point inside the span [1] [2]. To cover edge cases (like consecutive annotation spans or clicks at very edges), adding `margin-right:1px` to the span ensures there's always a spot for the cursor between spans [5]. This combo has been validated by multiple developers in the ProseMirror/TipTap community as a reliable fix for Chrome/Safari, and it does **not break functionality in Firefox** (Firefox already handled it correctly). The only adjustment you might need is to control the caret's appearance if you notice it too tall; this can be done by CSS (for example, set the span's `line-height` equal to its text size, or use `vertical-align: middle` for the inline-block).

Alternate approaches like injecting zero-width characters or making the span non-editable are generally **less desirable unless necessary**. They solve the cursor issue but at the cost of adding non-visible text nodes [8] or restricting user editing. Those approaches might be justified for specific cases (e.g., an app where annotated text must stay unmodified by users – then approach 4 is ideal, as used in Confluence [7]). Otherwise, avoiding extra DOM clutter is better for maintainability.

Using `-webkit-user-modify: plaintext-only` (with `caret-color` and `user-select`) is an elegant Safari-targeted fix to try if for some reason inline-block is not feasible for your styling needs. It keeps the span inline as normal but leverages Safari's own contenteditable mode to improve caret behavior. While not as time-tested for this particular cursor issue, it aligns with best practices for Safari contenteditable and **can be combined with the inline-block solution** if needed (they don't conflict: you can have an inline-block span that is also `-webkit-user-modify:read-write-plaintext-only`). Combining them might give the best of both – ensuring Safari handles the span as plain text and places the caret correctly, even as Firefox/Chrome just see a normal inline-block.

In summary, **start with the simple CSS fixes**: make annotation spans `inline-block` and add a tiny `margin-right`. This has a high success rate in Safari [1] [5] and no significant downsides in other browsers. Verify that the caret appears at the click point in Safari (it should). If the caret height looks odd, adjust the CSS as mentioned (this issue is purely visual [4]). Only if issues persist should you consider the heavier approaches (like zero-width spaces or non-editable spans). By using these refined strategies, you can achieve **consistent, correct cursor behavior across Safari, Firefox, and other browsers**, ensuring a smooth editing experience in your TipTap/ProseMirror editor.

Sources: Inline cursor fixes from ProseMirror discuss forum [1] [5]; Safari contenteditable hacks [8] [14]; real-world examples (Atlassian Confluence, Obsidian editor) of zero-width space and WebKit fixes [7] [15]; and StackOverflow reports of caret behavior [4].

---

[1] [2] [3] [5] Cursor appears inside inline node when at end of preceding text node - discuss.ProseMirror
https://discuss.prosemirror.net/t/cursor-appears-inside-inline-node-when-at-end-of-preceding-text-node/2538

[4] javascript - After adding span tag in contenteditable cursor move to end in IOS safari - Stack Overflow
https://stackoverflow.com/questions/58776408/after-adding-span-tag-in-contenteditable-cursor-move-to-end-in-ios-safari

6 7 Discussion: Inline nodes with content - Show - discuss.ProseMirror
https://discuss.prosemirror.net/t/discussion-inline-nodes-with-content/496

8 9 Non editable span in contenteditable | Lulu's blog
https://lucidar.me/en/rich-content-editor/non-editable-span-in-contenteditable/

10 11 Cursor issues with marks rendered as not editable - discuss.ProseMirror
https://discuss.prosemirror.net/t/cursor-issues-with-marks-rendered-as-not-editable/6360

12 13 Understanding Cursor Position Behavior in Custom NodeView with Undo/Redo in ProseMirror - discuss.ProseMirror
https://discuss.prosemirror.net/t/understanding-cursor-position-behavior-in-custom-nodeview-with-undo-redo-in-prosemirror/6053

14 javascript - Stop pasting html style in a contenteditable div only paste the plain text - Stack Overflow
https://stackoverflow.com/questions/58980235/stop-pasting-html-style-in-a-contenteditable-div-only-paste-the-plain-text

15 Obsidian | loikein's wiki
https://wiki.loikein.one/computer/software/multi/obsidian/