Why the Sanitizer API is just `setHTML()`

Sun 07 December 2025

Sanitizing HTML is the practice of taking a piece of HTML and removing some unwanted elements and attributes. Most often this is done to allow user-generated content with HTML but without causing XSS bugs. When imported from a library, a sanitizer typically looks like this:

```
const clean = DOMPurify.sanitize(input);
context.innerHTML = clean;
```

However, the API that we are building doesn't look like this at all. The core feature of the Sanitizer API is actually just `Element.setHTML(input)`.

This blog post will explain why.

To do so, we have to study the two lines of code from the DOMPurity example above. They result in the following steps:

1. Take an input string (and optionally a list of allowed elements as parameter).
2. Parse the input into an HTML fragment (no context element given).
3. Traverse the HTML fragment and remove elements as configured.
4. Serialize the remaining fragment into a string.
5. Parse the sanitized string (again), this time with `context` as context node into a fragment.
6. Insert the new fragment below `context` in the DOM tree.

Quick exercise for the reader: Can you spot where line 1 (`DOMPurify.sanitize()`) stops and line 2 (the `innerHTML` assignment) starts?

▶ Solution

This is pretty similar to the Sanitizer that I wanted to build into the browser:

```
const mySanitizer = new Sanitizer(/* config */);
//XXX This never shipped.
context.innerHTML = Sanitizer.sanitize(input);
```

But that is *NOT* the Sanitizer we ended up with.

And the reason is essentially Mutated XSS (mXSS). To quickly recap, the idea behind mXSS is that HTML parsing is not stable and a line of HTML being parsed and serialized and parsed again may turn

into something rather different. (See this description of mXSS bugs collected by SonarSource if you need a refresher.)

Another key point with mXSS is that HTML parsing can be quite *context-sensitive*: How an input string will be interpreted depends on the current node it is being inserted into.

Now let's go back to the algorithm steps 1-6. Did you notice that step 2 and 5 both perform HTML parsing? DOMPurify and most other sanitizers do this *without* any supplied context element. Typically, they parse into a new document and only return the content of the resulting `<body>`. The second parse step (step 5), however, *does* include a context element.

This means that we are parsing the input subtly different each time. We accidentally built a weird machine that will turn HTML into mXSS.

A better HTML sanitizer therefore needs to do away with all of that. How about the following:

- Use the right context when parsing HTML input.
- Remove the need for parsing twice.

Starting from an API design with a constructor like `new Sanitizer()`, it felt pretty hard to think of a context-sensitive method. I wanted something like `Sanitizer.sanitize(input, context)`. But how would we actually ensure that the return value can not be used another, potentially wrong `context`?

What we settled on was an API that has no return value:

```
context.setHTML(input, {sanitizer: ... } );
```

The internal algorithm is now the following:

1. Parse the input (with the right context element) into a document fragment
2. Traverse the resulting fragment and sanitize. (Using safe defaults or a user-specified configuration).
3. Replace the child nodes below `context` with the sanitized up fragment.

No superfluous parsing. No ambiguous contexts. Just setting HTML.

As a nice side-effect, you can replace existing code in the style of `ctx.innerHTML = input` with `context.setHTML(input)` and it should just work the same.

Except that there's no XSS.

To learn more about the Sanitizer API, please continue on MDN, in the Sanitizer Playground, or the Specification).

If you find a mistake in this article, you can submit a pull request on GitHub.

Recent posts

1. Why the Sanitizer API is just `setHTML()` *(Sun 07 December 2025)*
2. The C3PO Bug in Lego Star Wars: The Complete Saga *(Sat 06 December 2025)*
3. With Carrots & Sticks - Can the browser handle web security? *(Tue 08 April 2025)*
4. Home assistant can not be secured for internet access *(Sun 15 December 2024)*
5. Modern solutions against cross-site attacks *(Tue 26 November 2024)*