Element: setHTML() method

Limited availability

Experimental: This is an experimental technology
Check the Browser compatibility table carefully before using this in production.

The **`setHTML()`** method of the `Element` interface provides an XSS-safe method to parse and sanitize a string of HTML into a `DocumentFragment`, and then insert it into the DOM as a subtree of the element.

Syntax

JS

```
setHTML(input)
setHTML(input, options)
```

Parameters

`input`

A string defining HTML to be sanitized and injected into the element.

`options` Optional

An options object with the following optional parameters:

> `sanitizer`
>
> A `Sanitizer` or `SanitizerConfig` object which defines what elements of the input will be allowed or removed, or the string `"default"` for the default configuration. Note that generally a `"Sanitizer"` is expected to be more efficient than a `SanitizerConfig` if the configuration is to reused. If not specified, the default sanitizer configuration is used.

Return value

None (`undefined`).

Exceptions

`TypeError`

This is thrown if `options.sanitizer` is passed a:

- non-normalized `SanitizerConfig` (one that includes both "allowed" and "removed" configuration settings).
- string that does not have the value `"default"`.
- value that is not a `Sanitizer`, `SanitizerConfig`, or string.

Description

The **`setHTML()`** method provides an XSS-safe method to parse and sanitize a string of HTML into a `DocumentFragment`, and then insert it into the DOM as a subtree of the element.

`setHTML()` drops any elements in the HTML input string that are invalid in the context of the current element, such as a `<col>` element outside of a `<table>`. It then removes any HTML entities that aren't allowed by the sanitizer configuration, and further removes any XSS-unsafe elements or attributes — whether or not they are allowed by the sanitizer configuration.

If no sanitizer configuration is specified in the `options.sanitizer` parameter, `setHTML()` is used with the default `Sanitizer` configuration. This configuration allows all elements and attributes that are considered XSS-safe, thereby disallowing entities that are considered unsafe. A custom sanitizer or sanitizer configuration can be specified to choose which elements, attributes, and comments are allowed or removed. Note that even if unsafe options are allowed by the sanitizer configuration, they will still be removed when using this method (which implicitly calls `Sanitizer.removeUnsafe()`).

`setHTML()` should be used instead of `Element.innerHTML` for inserting untrusted strings of HTML into an element. It should also be used instead of `Element.setHTMLUnsafe()`, unless there is a specific need to allow unsafe elements and attributes.

Note that since this method always sanitizes input strings of XSS-unsafe entities, it is not secured or validated using the Trusted Types API.

Examples

Basic usage

This example shows some of the ways you can use `setHTML()` to sanitize and inject a string of HTML.

JS

```
// Define unsanitized string of HTML
const unsanitizedString = "abc <script>alert(1)<" + "/script> def";
// Get the target Element with id "target"
const target = document.getElementById("target");

// setHTML() with default sanitizer
target.setHTML(unsanitizedString);

// Define custom Sanitizer and use in setHTML()
// This allows only elements: div, p, button (script is unsafe and will be removed)
const sanitizer1 = new Sanitizer({
  elements: ["div", "p", "button", "script"],
```

```
});
target.setHTML(unsanitizedString, { sanitizer: sanitizer1 });

// Define custom SanitizerConfig within setHTML()
// This removes elements div, p, button, script, and any other unsafe elements/attributes
target.setHTML(unsanitizedString, {
  sanitizer: { removeElements: ["div", "p", "button", "script"] },
});
```

`setHTML()` live example

This example provides a "live" demonstration of the method when called with different sanitizers. The code defines buttons that you can click to sanitize and inject a string of HTML using a default and a custom sanitizer, respectively. The original string and sanitized HTML are logged so you can inspect the results in each case.

HTML

The HTML defines two `<button>` elements for applying different sanitizers, another button to reset the example, and a `<div>` element to inject the string into.

HTML

```
<button id="buttonDefault" type="button">Default</button>
<button id="buttonAllowScript" type="button">allowScript</button>

<button id="reload" type="button">Reload</button>
<div id="target">Original content of target element</div>
```

JAVASCRIPT

First we define the string to sanitize, which will be the same for all cases. This contains the `<script>` element and the `onclick` handler, both of which are considered XSS-unsafe. We also define the handler for the reload button.

JS

```
// Define unsafe string of HTML
const unsanitizedString = `
  <div>
    <p>This is a paragraph. <button onclick="alert('You clicked the button!')">Click me</button></p>
    <script src="path/to/a/module.js" type="module"><script>
  </div>
`;

const reload = document.querySelector("#reload");
reload.addEventListener("click", () => document.location.reload());
```

Next we define the click handler for the button that sets the HTML with the default sanitizer. This should strip out all unsafe entities before inserting the string of HTML. Note that you can see exactly which elements are removed in the `Sanitizer()` constructor examples.

JS

```
const defaultSanitizerButton = document.querySelector("#buttonDefault");
defaultSanitizerButton.addEventListener("click", () => {
  // Set the content of the element using the default sanitizer
  target.setHTML(unsanitizedString);

  // Log HTML before sanitization and after being injected
  logElement.textContent =
    "Default sanitizer: remove script element and onclick attribute\n\n";
  log(`\nunsanitized: ${unsanitizedString}`);
  log(`\nsanitized: ${target.innerHTML}`);
});
```

The next click handler sets the target HTML using a custom sanitizer that allows only `<div>`, `<p>`, and `<script>` elements. Note that because we're using the `setHTML` method, `<script>` will also be removed!

JS

```
const allowScriptButton = document.querySelector("#buttonAllowScript");
allowScriptButton.addEventListener("click", () => {
  // Set the content of the element using a custom sanitizer
  const sanitizer1 = new Sanitizer({
    elements: ["div", "p", "script"],
  });
  target.setHTML(unsanitizedString, { sanitizer: sanitizer1 });

  // Log HTML before sanitization and after being injected
  logElement.textContent =
    "Sanitizer: {elements: ['div', 'p', 'script']}\n Script removed even though allowed\n";
  log(`\nunsanitized: ${unsanitizedString}`);
  log(`\nsanitized: ${target.innerHTML}`);
});
```

RESULTS

Click the "Default" and "allowScript" buttons to see the effects of the default and custom sanitizer, respectively. Note that in both cases the `<script>` element and `onclick` handler are removed, even if explicitly allowed by the sanitizer.

Specifications

Specification
HTML Sanitizer API # dom-element-sethtml

Browser compatibility

Report problems with this compatibility data • View data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android	Safari on iOS	Samsung Internet	WebView Android	WebView on iOS
`setHTML`	No *	No *	138	No *	No	No *	No	No *	No	No *	No *	No

No support Experimental. Expect behavior to change in the future. * See implementation notes.

User must explicitly enable this feature.

See also

- `Element.setHTMLUnsafe()`
- `ShadowRoot.setHTML()` and `ShadowRoot.setHTMLUnsafe()`
- `Document.parseHTML()` and `Document.parseHTMLUnsafe()`
- HTML Sanitizer API