Element: setHTML() method

- Syntax
  - Parameters
  - Return value
  - Exceptions
- Description
  - Re-parsing and mutated XSS (mXSS)
- Examples
  - Basic usage
  - setHTML() live example
    - HTML
    - JavaScript
    - Results
- Specifications
- Browser compatibility
- See also

{{APIRef("HTML Sanitizer API")}}

The `setHTML()` method of the {{domxref("Element")}} interface provides an XSS-safe method to parse and sanitize a string of HTML and insert it into the DOM as a subtree of the element.

The method removes any elements and attributes that are considered XSS-unsafe, even if allowed by a passed sanitizer. Notably, the following elements are always removed: {{HTMLElement("script")}}, {{HTMLElement("frame")}}, {{HTMLElement("iframe")}}, {{HTMLElement("embed")}}, {{HTMLElement("object")}}, {{SVGElement("use")}}, and event handler attributes.

It is recommended (if supported) as a drop-in replacement for {{domxref("Element.innerHTML")}} when setting a user-provided string of HTML.

Syntax

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

Parameters

- `input`
  - : A string defining HTML to be sanitized and injected into the element.
- `options` {{optional_inline}}
  - : An options object with the following optional parameters:
    - `sanitizer`
      - : A {{domxref("Sanitizer")}} or {{domxref("SanitizerConfig")}} object which defines what elements of the input will be allowed or removed, or the string `"default"` for the default configuration. The method will remove any XSS-unsafe elements and attributes, even if allowed by the sanitizer. If not specified, the default sanitizer configuration is used. Note that if you're using the same configuration multiple times, it's expected to be more efficient to use a `Sanitizer` and modify it when you need to.

Return value

None (`undefined`).

Exceptions

- {{jsxref("TypeError")}}
  - : This is thrown if `options.sanitizer` is passed a:
    - {{domxref("SanitizerConfig")}} that isn't valid. For example, a configuration that includes both "allowed" and "removed" configuration settings.
    - string that does not have the value `"default"`.
    - value that is not a {{domxref("Sanitizer")}}, {{domxref("SanitizerConfig")}}, or string.

Description

The `setHTML()` method provides an XSS-safe method to parse and sanitize a string of HTML into a {{domxref("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 {{htmlelement("col")}} element outside of a {{htmlelement("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.

If no sanitizer is specified in the `options.sanitizer` parameter, `setHTML()` is used with the default sanitizer configuration. This configuration is suitable for the majority of use cases as it prevents XSS attacks, as well as other attacks like clickjacking or spoofing.

A custom `Sanitizer` or `SanitizerConfig` 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, they will still be removed when using this method (it removes the same elements as a sanitizer on which {{domxref('Sanitizer.removeUnsafe()')}} has been called).

`setHTML()` should be used instead of {{domxref("Element.innerHTML")}} for inserting untrusted strings of HTML into an element. It should also be used instead of {{domxref("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.

Re-parsing and mutated XSS (mXSS)

Even after sanitizing HTML input with `setHTML()`, it is still not safe to serialize the HTML and re-parse it using `innerHTML`. For example, the following code is unsafe.

```
div.setHTML(unsafeString); // Safe
const serializedHTML = div.innerHTML; // No longer sanitized!
otherElement.innerHTML = serializedHTML;
```

The reason for this is that sanitization is context-aware. When you call `setHTML()` on a particular element, the unsafe elements and attributes for that context are removed. If you serialize the HTML and use it directly in another element, it may still contain elements that are unsafe in that element.

This would be safe (if pointless):

```
div.setHTML(unsafeString); // Safe
const serializedHTML = div.innerHTML; // Serialized as a plain string
```

```
otherDiv.setHTML(serializedHTML); // Safe — re-sanitized by setHTML()
```

There is a class of attacks that take advantage of this flaw, referred to as mutation XSS. The simple rule to avoid this problem is to only ever inject HTML strings using safe methods such as `setHTML()`.

Examples

Basic usage

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

```
// 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 {{htmlelement("button")}} elements for applying different sanitizers, another button to reset the example, and a {{htmlelement("div")}} element to inject the string into.

```
<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>
```

```
<pre id="log"></pre>
```

```
#log {
  height: 320px;
  overflow: scroll;
  padding: 0.5rem;
  border: 1px solid black;
  margin: 5px;
}
```

JavaScript

```
const logElement = document.querySelector("#log");
function log(text) {
  logElement.textContent += text;
}
```

```
if ("Sanitizer" in window) {
```

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

```
// Define unsafe string of HTML
const unsanitizedString = `
  <div>
    <p>Paragraph to inject into shadow DOM.
      <button onclick="alert('You clicked the button!')">Click me</button>
    </p>
    <script src="path/to/a/module.js" type="module"><\/script>
    <p data-id="123">Para with <code>data-</code> attribute</p>
  </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.

```
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, onclick attribute, data- attribute\n\n";
  log(`\nunsanitized: ${unsanitizedString}`);
  log(`\n\nsanitized: ${target.innerHTML}`);
});
```

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

```
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(`\n\nsanitized: ${target.innerHTML}`);
});
```

```
} else {
  log("The HTML Sanitizer API is NOT supported in this browser.");
  // Provide fallback or alternative behavior
}
```

Results

Click the "Default" and "allowScript" buttons to see the effects of the default and custom sanitizer, respectively.

Note that because we are using a safe sanitization method, in both cases the `<script>` element and `onclick` handler are removed, even if explicitly allowed by the sanitizer. However while the `data-` attribute is removed with the default sanitizer, it is allowed when we pass a sanitizer.

{{EmbedLiveSample("setHTML() live example","100","450px")}}

Specifications

{{Specifications}}

Browser compatibility

{{Compat}}

See also

- {{domxref("Element.setHTMLUnsafe()")}}
- {{domxref("ShadowRoot.setHTML()")}} and {{domxref("ShadowRoot.setHTMLUnsafe()")}}

- {{domxref("Document.parseHTML_static", "Document.parseHTML()")}} and {{domxref("Document.parseHTMLUnsafe_static", "Document.parseHTMLUnsafe()")}}
- HTML Sanitizer API

□□□ #2
□ □□□ □□□ 2026-10-31 03:27:53 UTC
□ □□□ □□□ 2026-10-31 03:27:53 UTC