Jake Archibald wrote... who?

Making XML human-readable without XSLT

Posted 02 September 2025

Ok. This one is niche. But heh, you might enjoy a dive into this esoteric corner of the web – I certainly did.

XSLT and browser support

I want to get to the technical-fun bits quickly, so here's a quick rundown:

- XSLT is an XML language for transforming XML, including transforming it into non-XML formats, such as HTML.
- Browsers currently support a ~25 year old version of XSLT natively (examples coming up later).
- This feature is barely used. Usage is lower than features that were subsequently removed from browsers, such as mutation events and Flash.
- The feature is often the source of significant security issues.

This leaves browsers with a choice: Take development time away from features that have significant usage and developer demand, and instead use those resources to improve (probably rewrite) XSLT processing in browsers. Or, remove XSLT from browser engines.

Currently, Chrome, Safari, and Firefox support removing XSLT.

If you want to know more of the process & history here, I recommend Eric Meyer's blog post.

Right, let's take a look at XSLT and the alternatives.

How XSLT works

Take some XML like this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<?xml-stylesheet type="text/xsl" href="books.xsl"?>
<authors>
  <author>
    <name>Douglas Adams</name>
    <book>
      <title>The Hitchhiker's Guide to the Galaxy</title>
      <genre>Science Fiction Comedy</genre>
      <cover>https://covers.openlibrary.org/b/isbn/0330258648-L.jpg</cover>
    </book>
    <book>
      …
    </book>
    …
  </author>
  <author>
    <name>George Orwell</name>
    <book>
      …
    </book>
  </author>
  …
</authors>
```

If you visit that document, you'll (for now, at least), see a grid of books, with formatting and images, and that's all down to:

```
<?xml-stylesheet type="text/xsl" href="books.xsl"?>
```

…which tells the browser to transform the XML using XSLT.

Here's the full source if you're interested, but here's the snippet that renders each book:

```
<xsl:template name="book">
```

```
  <div class="book">
    <img class="cover" src="{cover}" alt="" />
    <div class="book-info">
      <div>
        <span class="title"><xsl:value-of select="title" /></span>
        -
        <span class="author"><xsl:value-of select="../name" /></span>
      </div>
      <div>
        <span class="genre"><xsl:value-of select="genre" /></span>
      </div>
    </div>
  </div>
</xsl:template>
```

This template maps a `<book>` into a `<div class="book">`, although it reaches up into the parent `<author>` to get the author name.

I used to write XSLT as part of my first job ~20 years ago. Making this demo felt *deeply weird*.

Anyway, assuming the above will stop working someday, what are the alternatives?

Styling XML

In some cases, you might get away with just styling XML, which you can do using:

```
<?xml-stylesheet type="text/css" href="styles.css"?>
```

Now you can style your XML document using regular CSS. Just make sure your document is served with `Content-Type: text/xml` (rather than, e.g. a more specific RSS content type), otherwise the browser won't use the styles.

What you can do here is pretty limited. In the XSLT example, I take the text content of the `<cover>` element, and make it the `src` of an image. I also repeat the author's `<name>` for each book. Neither of these is possible with CSS alone.

Also, the document will have the semantics of the XML document, which likely means "no semantics", so the result will have poor accessibility. For example, even if your XML contains `<h1>`, unless you've told it to be an HTML element, the browser won't recognise it as a heading.

That said, if you just want to catch users that mistakenly land on some XML, you can get away with something like this:

```
:root {
  & > * {
    display: none;
  }

  &::before {
    content: 'Hi there! This is my RSS feed. '
      "It isn't meant to be human-readable…";
  }
}
```

Which is what I've done with my RSS feed.

For anything more complex, you need to do some kind of transformation of the source data.

Wait, do you really want to do this client-side?

In almost all cases, the best thing to do is convert your data to HTML on the server, or as part of a build process. When you do that:

- The HTML form of your data is more likely to be seen by crawlers and scrapers.
- The browser can render the page in a streaming way, meaning a faster experience for users.
- You can use whatever tools you like to do the transformation – you aren't limited to things the browser supports natively.

This might mean you end up with two published files, one for humans (as HTML), and one for machines (as XML or whatever). This is fine. My blog posts and my RSS feed are different resources, even though they have data in common. All my posts have this in the head:

```
<link
  rel="alternate"
  type="application/rss+xml"
  title="Jake Archibald's Blog"
  href="https://jakearchibald.com/posts.rss"
/>
```

…meaning feed readers can find the RSS feed from any post.

But if you really need to, here's how to do the transformation on the client, without XSLT:

Transforming XML client-side without XSLT

If you just want to make some existing XSLT work in browsers that don't support it, check out Mason's XSLT polyfill. But if you don't need to use XSLT, JavaScript is, in my opinion, a better option.

Here's a demo that produces the same result as the XSLT demo, but this time using JS. Here's how it works…

In the XML, instead of including XSLT, I include a script as the first child of the root:

```
<authors>
  <script
    xmlns="http://www.w3.org/1999/xhtml"
    src="script.js"
    defer=""
  />
  …
```

The `xmlns` attribute is important, as it tells the browser this is an HTML script element, rather than some other kind of XML element.

Technically, this is changing the data structure. Validating parsers would see a `<script>` they weren't expecting, and throw an error. However, most parsers are not validating, as they want to allow for extensions to whatever format they're reading.

Then, in that script:

```
// Get the XML data
const data = document.documentElement;
// Create an HTML root element
const htmlEl = document.createElementNS(
  'http://www.w3.org/1999/xhtml',
  'html',
);
// Swap the two over
data.replaceWith(htmlEl);
```

Now all I had to do was populate that `html` element. I created a simple templating function to handle escaping. For example, here's the snippet that renders each book:

```
const bookHTML = (bookEl) => html`
```

```
  <div class="book">
    <img
      class="cover"
      src="${bookEl.querySelector(':scope > cover').textContent}"
      alt=""
    />
    <div class="book-info">
      <div>
        <span class="title">
          ${bookEl.querySelector(':scope > title').textContent}
        </span>
        -
        <span class="author">
          ${bookEl.parentNode.querySelector(':scope > name').textContent}
        </span>
      </div>
      <div>
        <span class="genre">
          ${bookEl.querySelector(':scope > genre').textContent}
        </span>
      </div>
    </div>
  </div>
`;
```

It's very similar to the XSLT equivalent, *and* it can be debugged using regular JavaScript tooling.

Ensure you're creating *HTML* elements

One gotcha when creating HTML in an XML document is it's easy to accidentally create XML elements rather than HTML elements. If you want proper semantics, accessibility, and behaviour, you want real HTML elements.

For example:

```
// In an XML document this will not create an HTML element:
const xmlElement = document.createElement('h1');
// Instead, you need to use:
const htmlElement = document.createElementNS(
  'http://www.w3.org/1999/xhtml',
  'h1',
);
```

If you're using a framework to create the elements, check that it's creating real HTML elements – `el.namespaceURI` should be `"http://www.w3.org/1999/xhtml"`.

The `xmlns` attribute we used earlier only works via the parser, so this *doesn't* create an HTML element in an XML document:

```
const h1 = document.createElement('h1');
h1.setAttribute('xmlns', 'http://www.w3.org/1999/xhtml');
```

However, `innerHTML` and `setHTMLUnsafe` will assume the content is HTML, which is what I used:

```
htmlEl.setHTMLUnsafe(html`
  <head>
    <title>Some books</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
  </head>
  <body>
    <h1>Some books</h1>
    <ul class="book-list">
      ${[...data.querySelectorAll('book')].map(
        (book) => html`<li>${bookHTML(book)}</li>`,
      )}
    </ul>
  </body>
`);
```

Here's the full working demo, and the source. And that's it! Wow you made it to the end. Huh. I didn't think anyone would.

View this page on GitHub

Comments powered by Disqus