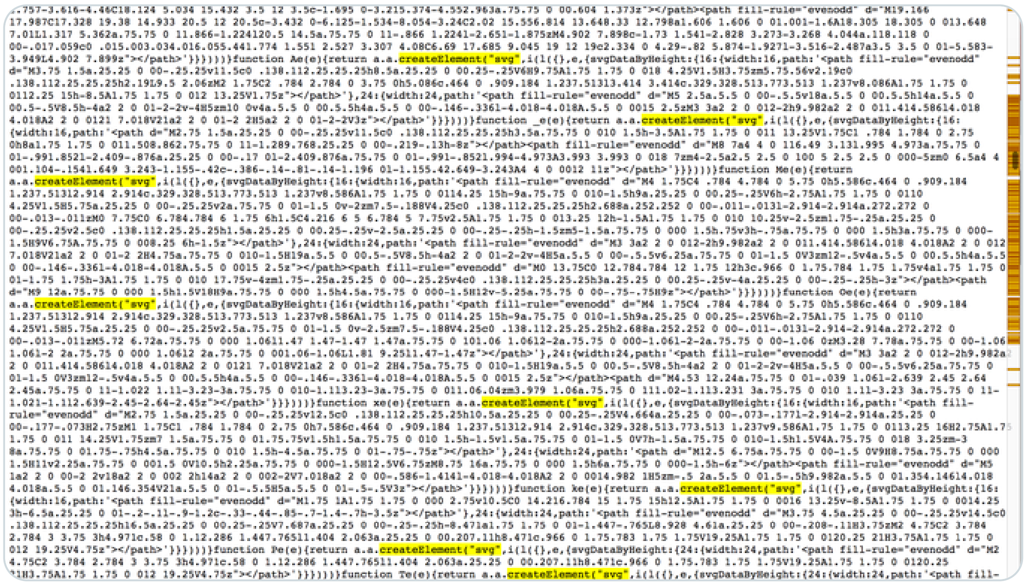
Breaking Up with SVG-in-JS in 2023

Jun 30, 2023

In December last year, “Why We’re Breaking Up with CSS-in-JS” made rounds and describes why you no longer want to have CSS inside of your JS bundles. However, CSS isn’t the only thing that lands in JS bundles this day - SVGs do too, as Jason Miller, the author of Preact shows:

Jason Miller 🦊⚛
@_developit · Follow

Please don't import SVGs as JSX. It's the most expensive form of sprite sheet: costs a minimum of 3x more than other techniques, and hurts both runtime (rendering) performance and memory usage.
This bundle from a popular site is almost 50% SVG icons (250kb), and most are unused.



```
..757-3.616-4.46C18.124 5.034 15.432 3.5 12 3.5c-1.695 0-3.215.374-4.552.963a.75.75 0 00.604 1.373z"></path><path fill-rule="evenodd" d="M19.166
17.987C17.328 19.38 14.933 20.5 12 20.5c-3.432 0-6.125-1.534-8.054-3.24C2.02 15.556.814 13.648.33 12.798a1.606 1.606 0 01.001-1.6A18.305 18.305 0 013.648
7.01L1.317 5.362a.75.75 0 11.866-1.224120.5 14.5a.75.75 0 11-.866 1.2241-2.651-1.875zM4.902 7.898c-1.73 1.541-2.828 3.273-3.268 4.044a.118.118 0
00-.017.059c0 .015.003.034.016.055.441.774 1.551 2.527 3.307 4.08C6.69 17.685 9.045 19 12 19c2.334 0 4.29-.82 5.874-1.9271-3.516-2.487a3.5 3.5 0 01-5.583-
3.949L4.902 7.899z"></path>'}}})))}function Ae(e){return a.a.createElement("svg",i(l({},e,{svgDataByHeight:{16:{width:16,path:'<path fill-rule="evenodd"
d="M3.75 1.5a.25.25 0 00-.25.25v11.5c0 .138.112.25.25.25h8.5a.25.25 0 00.25-.25V6H9.75A1.75 1.75 0 018 4.25V1.5H3.75zm5.75.56v2.19c0
.138.112.25.25.25h2.19L9.5 2.06zM2 1.75C2 .784 2.784 0 3.75 0h5.086c.464 0 .909.184 1.237.51313.414 3.414c.329.328.513.773.513 1.237v8.086A1.75 1.75 0
0112.25 15h-8.5A1.75 1.75 0 012 13.25V1.75z"></path>'},24:{width:24,path:'<path fill-rule="evenodd" d="M5 2.5a.5.5 0 00-.5.5v18a.5.5 0 00.5.5h14a.5.5 0
00.5-.5V8.5h-4a2 2 0 01-2-2v-4H5zm10 0v4a.5.5 0 00.5.5h4a.5.5 0 00-.146-.3361-4.018-4.018A.5.5 0 0015 2.5zM3 3a2 2 0 012-2h9.982a2 2 0 011.414.58614.018
4.018A2 2 0 0121 7.018V21a2 2 0 01-2 2H5a2 2 0 01-2-2V3z"></path>'}}})))}function _e(e){return a.a.createElement("svg",i(l({},e,{svgDataByHeight:{16:
{width:16,path:'<path d="M2.75 1.5a.25.25 0 00-.25.25v11.5c0 .138.112.25.25.25h3.5a.75.75 0 010 1.5h-3.5A1.75 1.75 0 011 13.25V1.75C1 .784 1.784 0 2.75
0h8a1.75 1.75 0 011.508.862.75.75 0 11-1.289.768.25.25 0 00-.219-.13h-8z"></path><path fill-rule="evenodd" d="M8 7a4 4 0 116.49 3.131.995 4.973a.75.75 0
01-.991.8521-2.409-.876a.25.25 0 00-.17 01-2.409.876a.75.75 0 01-.991-.8521.994-4.973A3.993 3.993 0 018 7zm4-2.5a2.5 2.5 0 100 5 2.5 2.5 0 000-5zm0 6.5a4 4
001.104-.1541.649 3.243-1.155-.42c-.386-.14-.81-.14-1.196 01-1.155.42.649-3.243A4 4 0 0012 11z"></path>'}}})))}function Me(e){return
a.a.createElement("svg",i(l({},e,{svgDataByHeight:{16:{width:16,path:'<path fill-rule="evenodd" d="M4 1.75C4 .784 4.784 0 5.75 0h5.586c.464 0 .909.184
1.237.51312.914 2.914c.329.328.513.773.513 1.237v8.586A1.75 1.75 0 0114.25 15h-9a.75.75 0 010-1.5h9a.25.25 0 00.25-.25V6h-2.75A1.75 1.75 0 0110
4.25V1.5H5.75a.25.25 0 00-.25.25v2a.75.75 0 01-1.5 0v-2zm7.5-.188V4.25c0 .138.112.25.25.25h2.688a.252.252 0 00-.011-.0131-2.914-2.914a.272.272 0
00-.013-.011zM0 7.75C0 6.784.784 6 1.75 6h1.5C4.216 6 5 6.784 5 7.75v2.5A1.75 1.75 0 013.25 12h-1.5A1.75 1.75 0 010 10.25v-2.5zm1.75-.25a.25.25 0
00-.25.25v2.5c0 .138.112.25.25.25h1.5a.25.25 0 00.25-.25v-2.5a.25.25 0 00-.25-.25h-1.5zm5-1.5a.75.75 0 000 1.5h.75v3h-.75a.75.75 0 000 1.5h3a.75.75 0 000-
1.5H9V6.75A.75.75 0 008.25 6h-1.5z"></path>'},24:{width:24,path:'<path fill-rule="evenodd" d="M3 3a2 2 0 012-2h9.982a2 2 0 011.414.58614.018 4.018A2 2 0 012
7.018V21a2 2 0 01-2 2H4.75a.75.75 0 010-1.5H19a.5.5 0 00.5-.5V8.5h-4a2 2 0 01-2-2v-4H5a.5.5 0 00-.5.5v6.25a.75.75 0 01-1.5 0V3zm12-.5v4a.5.5 0 00.5.5h4a.5.5
0 00-.146-.3361-4.018-4.018A.5.5 0 0015 2.5z"></path><path fill-rule="evenodd" d="M0 13.75C0 12.784.784 12 1.75 12h3c.966 0 1.75.784 1.75 1.75v4a1.75 1.75 0
01-1.75 1.75h-3A1.75 1.75 0 010 17.75v-4zm1.75-.25a.25.25 0 00-.25.25v4c0 .138.112.25.25.25h3a.25.25 0 00.25-.25v-4a.25.25 0 00-.25-.25h-3z"></path><path
d="M9 12a.75.75 0 000 1.5h1.5V18H9a.75.75 0 000 1.5h4.5a.75.75 0 000-1.5H12v-5.25a.75.75 0 00-.75-.75H9z"></path>'}}})))}function Oe(e){return
a.a.createElement("svg",i(l({},e,{svgDataByHeight:{16:{width:16,path:'<path fill-rule="evenodd" d="M4 1.75C4 .784 4.784 0 5.75 0h5.586c.464 0 .909.184
1.237.51312.914 2.914c.329.328.513.773.513 1.237v8.586A1.75 1.75 0 0114.25 15h-9a.75.75 0 010-1.5h9a.25.25 0 00.25-.25V6h-2.75A1.75 1.75 0 0110
4.25V1.5H5.75a.25.25 0 00-.25.25v2.5a.75.75 0 01-1.5 0v-2.5zm7.5-.188V4.25c0 .138.112.25.25.25h2.688a.252.252 0 00-.011-.0131-2.914-2.914a.272.272 0
00-.013-.011zM5.72 6.72a.75.75 0 000 1.0611.47 1.47-1.47 1.47a.75.75 0 101.06 1.0612-2a.75.75 0 000-1.061-2-2a.75.75 0 00-1.06 0zM3.28 7.78a.75.75 0 00-1.06
1.061-2 2a.75.75 0 000 1.0612 2a.75.75 0 001.06-1.06L1.81 9.2511.47-1.47z"></path>'},24:{width:24,path:'<path fill-rule="evenodd" d="M3 3a2 2 0 012-2h9.982a2
2 0 011.414.58614.018 4.018A2 2 0 0121 7.018V21a2 2 0 01-2 2H4.75a.75.75 0 010-1.5H19a.5.5 0 00.5-.5V8.5h-4a2 2 0 01-2-2v-4H5a.5.5 0 00-.5.5v6.25a.75.75 0
01-1.5 0V3zm12-.5v4a.5.5 0 00.5.5h4a.5.5 0 00-.146-.3361-4.018-4.018A.5.5 0 0015 2.5z"></path><path d="M4.53 12.24a.75.75 0 01-.039 1.061-2.639 2.45 2.64
2.45a.75.75 0 11-1.022 1.11-3.23-3a.75.75 0 010-1.113.23-3a.75.75 0 011.06.04zm3.979 1.06a.75.75 0 111.02-1.113.231 3a.75.75 0 010 1.11-3.23 3a.75.75 0 11-
1.021-1.112.639-2.45-2.64-2.45z"></path>'}}})))}function xe(e){return a.a.createElement("svg",i(l({},e,{svgDataByHeight:{16:{width:16,path:'<path fill-
rule="evenodd" d="M2.75 1.5a.25.25 0 00-.25.25v12.5c0 .138.112.25.25.25h10.5a.25.25 0 00.25-.25V4.664a.25.25 0 00-.073-.1771-2.914-2.914a.25.25 0
00-.177-.073H2.75zM1 1.75C1 .784 1.784 0 2.75 0h7.586c.464 0 .909.184 1.237.51312.914 2.914c.329.328.513.773.513 1.237v9.586A1.75 1.75 0 0113.25 16H2.75A1.75
1.75 0 011 14.25V1.75zm7 1.5a.75.75 0 01.75.75v1.5h1.5a.75.75 0 010 1.5h-1.5v1.5a.75.75 0 01-1.5 0V7h-1.5a.75.75 0 010-1.5h1.5V4A.75.75 0 018 3.25zm-3
8a.75.75 0 01.75-.75h4.5a.75.75 0 010 1.5h-4.5a.75.75 0 01-.75-.75z"></path>'},24:{width:24,path:'<path d="M12.5 6.75a.75.75 0 00-1.5 0V9H8.75a.75.75 0 000
1.5H11v2.25a.75.75 0 001.5 0V10.5h2.25a.75.75 0 000-1.5H12.5V6.75zM8.75 16a.75.75 0 000 1.5h6a.75.75 0 000-1.5h-6z"></path><path fill-rule="evenodd" d="M5
1a2 2 0 00-2 2v18a2 2 0 002 2h14a2 2 0 002-2V7.018a2 2 0 00-.586-1.4141-4.018-4.018A2 2 0 0014.982 1H5zm-.5 2a.5.5 0 01.5-.5h9.982a.5.5 0 01.354.14614.018
4.018a.5.5 0 01.146.354V21a.5.5 0 01-.5.5H5a.5.5 0 01-.5-.5V3z"></path>'}}})))}function ke(e){return a.a.createElement("svg",i(l({},e,{svgDataByHeight:{16:
{width:16,path:'<path fill-rule="evenodd" d="M1.75 1A1.75 1.75 0 000 2.75v10.5C0 14.216.784 15 1.75 15h12.5A1.75 1.75 0 0016 13.25v-8.5A1.75 1.75 0 0014.25
3h-6.5a.25.25 0 01-.2-.11-.9-1.2c-.33-.44-.85-.7-1.4-.7h-3.5z"></path>'},24:{width:24,path:'<path fill-rule="evenodd" d="M3.75 4.5a.25.25 0 00-.25.25v14.5c0
.138.112.25.25.25h16.5a.25.25 0 00.25-.25V7.687a.25.25 0 00-.25-.25h-8.471a1.75 1.75 0 01-1.447-.765L8.928 4.61a.25.25 0 00-.208-.11H3.75zM2 4.75C2 3.784
2.784 3 3.75 3h4.971c.58 0 1.12.286 1.447.76511.404 2.063a.25.25 0 00.207.11h8.471c.966 0 1.75.783 1.75 1.75V19.25A1.75 1.75 0 0120.25 21H3.75A1.75 1.75 0
012 19.25V4.75z"></path>'}}})))}function Pe(e){return a.a.createElement("svg",i(l({},e,{svgDataByHeight:{24:{width:24,path:'<path fill-rule="evenodd" d="M2
4.75C2 3.784 2.784 3 3.75 3h4.971c.58 0 1.12.286 1.447.76511.404 2.063a.25.25 0 00.207.11h8.471c.966 0 1.75.783 1.75 1.75V19.25A1.75 1.75 0 0120.25
21H3.75A1.75 1.75 0 012 19.25V4.75z"></path>'}}})))}function Te(e){return a.a.createElement("svg",i(l({},e,{svgDataByHeight:{24:{width:24,path:'<path fill-
```

1.6K Reply Copy link

Read 75 replies

SVGs in JS have a cost and SVGs do not belong into your JS bundle. It’s about time to bring back SVG-in-HTML.

Let’s take a look at better techniques for using SVG in JSX, while keeping our JS bundle small and performant.

Table of contents

▶ Open Table of contents

How does `<svg>` end up in JavaScript?

First of all, let’s make sure we know how the SVGs end up inside the JavaScript source-code. Usually, this is done as part of writing JSX:

```
<!-- HeartIcon.svg -->
<svg viewBox="0 0 300 300">
  <g><path d="M0 200 v-200 h200 a100,100 90 0,1 0,200 a100,100 90 0,1 -200,0z" /></g>
</svg>
```

```
// App.jsx
import { HeartIcon } from "./HeartIcon.svg";

const App = () => <HeartIcon fill="red" />
```

To make the **`.svg`** file import work, bundlers need to be told what to do with files that are not JavaScript (or TypeScript). A Webpack loader like svgr is commonly used – it transforms the **`.svg`** file into a React component. It allows you to conveniently add attributes (like **`fill="red"`**) to the SVG tag:

```
// HeartIcon.svg after svgr (before JSX transformation) - ⚠️ don't copy this.
export default props => (
  <svg viewBox="0 0 300 300" {...props}>
    <g><path d="M0 200 v-200 h200 a100,100 90 0,1 0,200 a100,100 90 0,1 -200,0z" /></g>
  </svg>
);
```

⚠ Never manually write React components that return SVG tags, it is an anti-pattern, with the reasons written in the intro. Full SVG content belongs into **`.svg`** files only.

Note the **{...props}** spread and that the SVG file content has been inlined.
After rendering, the HTML output might look like this:

```
<svg viewBox="0 0 300 300" fill="red"><!-- the difference is only the `fill` attribute -->
  <g><path d="M0 200 v-200 h200 a100,100 90 0,1 0,200 a100,100 90 0,1 -200,0z" /></g>
</svg>
```

Granted, this is very convenient and easy to use, but the ease of use comes with a drawback your users have to pay…

Performance Deep Dive: Why SVG-in-JS is an anti-pattern

… so, why would you not want to have SVG code inside your JS bundle?

Parsing & Compilation

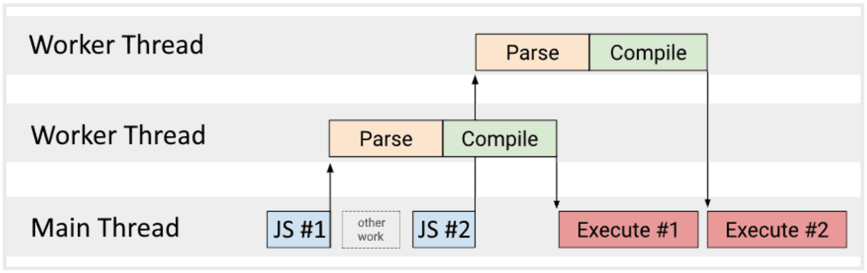
JavaScript parsing & compilation is not free – the more you have inside your bundle, the longer JavaScript engines will take to go through the source code.

On a fast M2 laptop, the difference might not be obvious, but as Alex Russell from the Microsoft Edge team reports year-by-year, there is a performance equality gap. As he puts it, a Samsung Galaxy A50 and the Nokia G11 are the best devices you can buy to understand world-wide p75 users. Web development should be inclusive for everyone, and not only for wealthy regions.

> *"Byte-for-byte, JavaScript is **more expensive** for the browser to process **than** the **equivalently** sized **image** or Web Font"*

— Tom Dale, *web.dev*

SVGs are not JavaScript, they are HTML-like XML tags that describe images. You certainly don't want images inside your JS. By moving SVGs out of the JS bundle, you are moving them out of the parsing & compilation step. In the following, you'll see why this is beneficial.



Visualization of JavaScript download & execution in Chromium. Parsing & Compilation does not block the main thread. V8.dev

Parsing & Compilation is the step right before the execution. So whenever your JavaScript is downloaded and ready to run, the **time** it takes **to parse** and the time it takes **to compile**, is the **time where the user is waiting for the interactivity**.
For React, you also need hydration on top. Download + Parsing + Compilation + Hydration is the Time-To-Intereactive there. The bigger the component tree, the more needs to be hydrated. The article "JavaScript Start-up Optimization" from Addy Osmani has really gone in-depth on this topic.

On a side-note, compression can also positively impact the processing time, but do not forget that JS engines work with the uncompressed source code. For that, you could move the parsing & compilation to an earlier time by using `<link rel="modulepreload">` , or by intercepting with a ServiceWorker. In that case, parsing & compilation will be done right after downloading, instead of right before execution. Managing the timing can lead to better results, but does not fix the root cause.

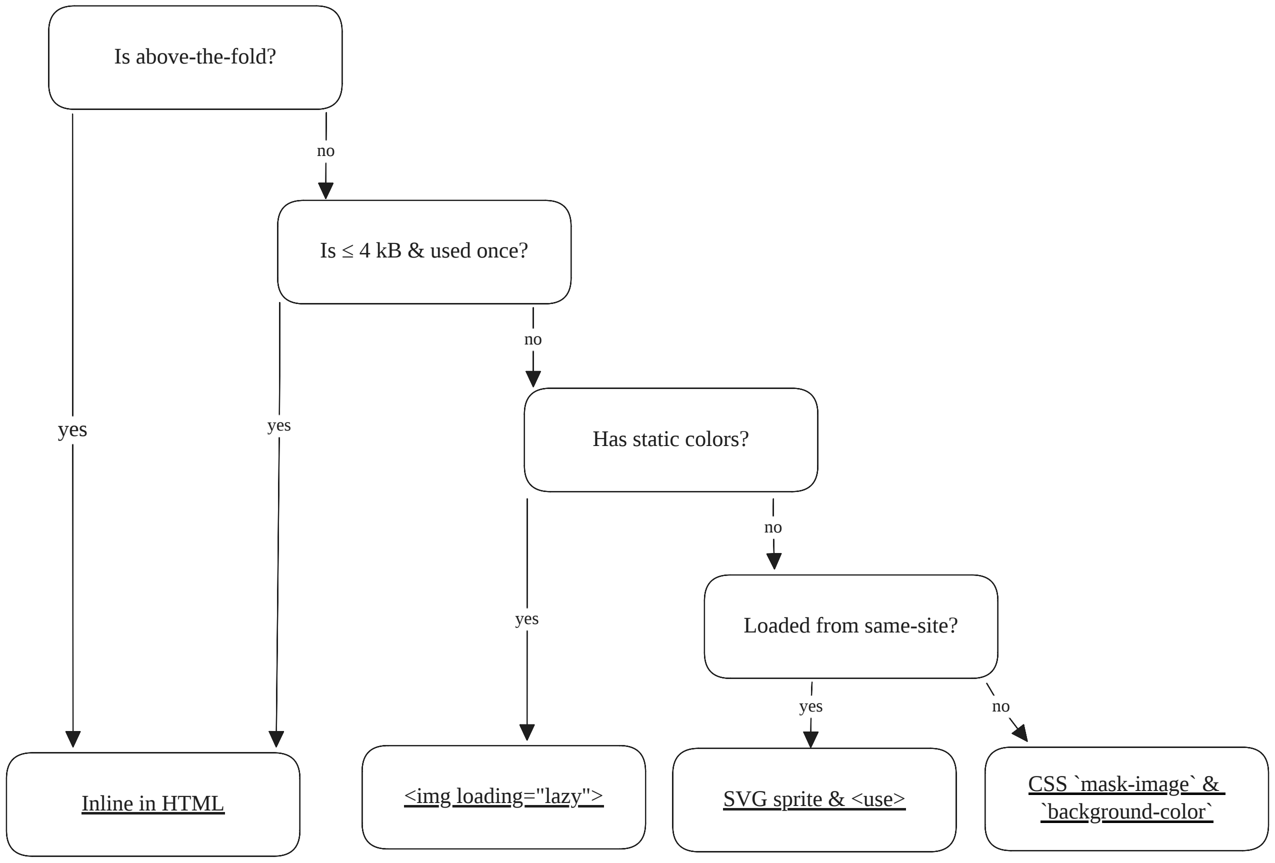
Memory Usage

Anything that is parsed needs to be kept around, which is for the duration of the page in the JavaScript memory heap, and most likely inside of any of the various memory caches browsers have. A Galaxy A50 has 4 GiB of RAM, and your website is not the only application running on that device, so there is not much space left. Be gentle to your users.

Best practices for removing SVGs from the JS bundle

Before we start, the first step is finding out if you have SVG in your JavaScript bundle. This can be done by either inspecting the source code and searching for "svg", or by using Lighthouse's in-built bundle viewer. You can access it with the button in the "Performance" section of a report.

If you have a non-trivial amount inside your bundle, there are some options that help you get rid of them. The following diagram might help you make a decision in most cases:



Each of those options is explained in the following.

Using `<img>` to load the SVG

In order to use SVGs inside of **`<img>`** tags, you have to tell your bundler/framework to externalize them (or in other words: to create a static URL). For Webpack, this can be done by updating the Webpack config to set all **`.svg`** files to the type **`asset/resource`**[1]:

```
const config = {
  // … other parts of the Webpack config …
  module: {
    rules: [
      {
        test: /\.svg/,
        type: "asset/resource",
      },
    ],
  },
};
```

Other frameworks, such as Astro, might do this automatically. Make sure your infrastructure applies Brotli/Gzip compression, as the **`svg`** file format can be compressed like JS files.

Referencing the SVG then becomes as easy as regular (PNG/JPG/…) images:

```
import HeartIcon from "./HeartIcon.svg";

const App = () => <img src={HeartIcon} loading="lazy" />;
```

💡 Perf tips:

- With **`<img>`** you can use attributes like **`loading="lazy"`** for native lazy-loading or **`importance="high"`** for changing the fetch priority⚡
- For complex SVG animations on DPR > 1x screens, **`<img>`** might consume less CPU compared to an inline SVG[2]

⚠ Caveats of **`<img>`**:

- Usage of the CSS value **`currentcolor`** and CSS custom props (**`--variable`**) is harder, as they do not inherit values from the current page (as the SVG file is treated as an external resource and not as part of the DOM)[3]
- Chromium: SVG animations run capped to 60 Hz[4] and use more CPU on DPR = 1x screens[2]
- **`<a>`** tags *inside* the SVG can't be clicked

I would **recommend** to use single out-of-viewport SVGs like this. However, if you need to style them or have a lot of them, you should probably pick one of the other options.

SVG sprites – using `<use>`

If we want to use **`fill`** and other (custom) CSS properties, or **`currentcolor`** as value, we need to use the **`<use>`** tag that allows us to load SVGs. Together with the same Webpack rule as above, we can reference the SVG like this:

```
import HeartIcon from "./HeartIcon.svg";

const App = () => <svg><use href={`${HeartIcon}#heart`} /></svg>;
```

If you look carefully, you can see we reference an ID, which is needed for **`<use>`**. Let's update our HeartIcon SVG:

```
<svg viewBox="0 0 300 300" id="heart">...</svg>
```

If you have many SVGs on your site, you can put them all into one file. **SVG sprites** are built using the `<symbol>` tag. You need to give them an ID, so you can `<use>` them (literally). Styling, `currentcolor` et. all works like before:

```
<!-- icons.svg -->
<svg>
  <!-- 1: add a `<defs>` tag -->
  <defs>
    <!-- 2: wrap in `<symbol>` and give it an ID (and other attributes such as `viewBox`) -->
    <symbol id="icon1">
      <!-- 3: paste the content of the SVG inside of the `<symbol>` -->
      ...
    </symbol>
    <symbol id="icon2">...</symbol>
  </defs>
</svg>
```

The sprite file will only be loaded once and will be cached. We can now reference the SVGs in the following way:

```
<svg><use href="icons.svg#icon1" /></svg>
<svg><use href="icons.svg#icon2" /></svg>
```

Writing your own SVG sprite takes some time if done manually, but fear not, I’ve collected open-source solutions which automate this in the ➡ tools chapter.

⚠ Caveats of `<use>`:

- `<mask>` and `<clipPath>` do **not** work when externally loading SVGs[5]. Solved by ➡ inlining.
- SVGs cannot be loaded from a CDN when using `<use>`, see ➡ CORS chapter.

Remove even more JS: CSS & `currentcolor` for attributes like `fill`, `stroke`, `width`, `height` etc.

While you could write code like this:

```
// 😕 not optimal
const Icon = (favColor, width) => (
  <svg><use href={`${HeartIcon}#heart`} fill={favColor ? favColor : "red"} width={width} />
</svg>
);
const App = () => (
  <><Icon favColor="#FFFF00" /><Icon width={300} /></>
);
```

I do not recommend doing so. The logic ends up in the JavaScript bundle again and needs to be executed by the JavaScript engine. Similar to avoiding CSS-in-JS, we want to have class names instead, so we can use CSS to style the SVG:

```
// ✅ better
const Icon = (className) => (
  // add a class and let any consumer handle the details via CSS ⬇
  <svg><use href={`${HeartIcon}#heart`} className={`heart ${className}`} /></svg>
);
const YellowHeart = () => <Icon className="yellow" />;
const BigHeart = () => <Icon className="big" />;
const App = () => <><YellowHeart /><BigHeart /></>;
```

You can now use `currentcolor` to make the SVGs inherit the color from the CSS attribute `color`:

```
/* 👍 good */
.heart { fill: currentcolor } /* ⬅ apply the current `color` */

/* ⬇ those classes might even come from your design system and don't need to be SVG specific
*/
.big { width: 300px }
.yellow { color: #FFFF00 }
```

or even better if you have access to the SVG:

```
<!-- HeartIcon.svg -->
<svg viewBox="0 0 300 300" id="heart">
  <g>
    <path
      d="M0 200 v-200 h200 a100,100 90 0,1 0,200 a100,100 90 0,1 -200,0z"
      fill="currentcolor"
    /><!-- ⬆ add `currentcolor` to `fill`/`stroke` -->
  </g>
</svg>
```

```
/* 🏆 optimal */
.heart { /* `fill` is no longer needed in the CSS */ }
```

Any CSS attribute applied to the `<use>` tag, applies the styling to the `<svg>` / `<symbol>` tag of the referenced SVG automatically. So you can easily add `stroke` to `.heart` while not touching the color of the `<path>` element.

⚠ Be careful: `width` and `height` on `<use>` require the original `<svg>` to have a `viewBox` attribute (or `<view>`).

Server Components (React)

Very recent, but also a viable solution that does not require too many changes, would be adopting the upcoming Server Components from React, which can be used in e.g. NextJS 13.4. They are especially helpful when you need to change behaviour of components at runtime and allow you to write JSX that is only executed on the server. This way, the SVGs are not part of the JavaScript bundle shipped to browsers anymore. In order to keep them on the server only, it is as simple as *not* adding `'use client'` to the file.

💡 Make sure to read the inlining chapter, as making every SVG a Server Component inlines them into the HTML response and has its own set of drawbacks.

In the case of CORS: CSS `mask-image`

> *“SVG `<use>` elements don’t currently have any way to ask for cross-origin permissions. **They just don’t work cross-origin, at all.**”*

— *O’Reilly Media book by Amelia Bellamy-Royds, Kurt Cagle, and Dudley Storey*

In order to use `<use>`, you **must** load the (sprite) SVG from the same domain. If you have to use a CDN, you’ll run into the quoted limitation.

You now have two options:

- A way to avoid this issue is to follow the method described in chapter `<img>` – but this breaks **`currentcolor`**
- If you need to apply only one color, use CSS **`mask-image`** + **`background-color`** instead:

```css
.heart {
  mask-image: url("somecdn.com/HeartIcon.svg#heart");
  /* ⬇ 'color' the SVG */
  background-color: currentcolor;
}
```

However, using this approach has the same drawback as CSS background images used on LCP elements:
Browsers need to download and execute the CSS first before they can discover and download the SVG, which increases the time until the SVG shows up meaningfully. This can be mitigated by using `<link rel="preload" as="image">`. Additionally, all caveats of `<img>` apply to the SVG itself as well (a mask is also not part of the DOM).
One advantage is, the browser won't download the mask image, if the element is hidden (with **`display: none`**).

Performance vs. Time-To-Load: To inline or not to inline?

Inlining allows us to save one HTTP request, so SVGs are displayed immediately. The downside is, which is the same as for critical CSS, the bytes are downloaded on every non-browser-cached page. Additionally, inline SVGs are DOM elements and thus will increase the amount of calculations browsers need to do[6]. So while we decrease the amount the JavaScript engine has to do, we do not want to meaningfully slow-down the time-to-download of the HTML response or blow up the DOM size. If you're using SVG animations, depending on the DPR of various devices, inlined **`<svg>`** elements might hurt performance[2].

Thus, we can derive a few rules for what to inline:

1. Logos have the highest priority, a logo makes your users recognize your brand. Some websites even measure the time to load the logo (even if not the LCP element). While the LCP element is an important factor for SEO (as it is a Core Web Vital), visual completeness implies "the website is usable now" for most users. It also prevents a visual "flicker" or maybe even CLS.

2. Next, icons in the viewport, e.g. a search or hamburger icon. Those are more likely to be touched by users, so if they load late, it impacts the experience of your users. By now you might see parallels to critical CSS: Everything above-the-fold is more important, anything below not so much.

3. The rest. Do not inline them and if possible lazy-load.

💡 As a rule of thumb, I'd recommend a budget similar to critical CSS. Astro uses **4 kB** as a threshold for inlining files[7]. In general, anything inlined (CSS, SVGs, JS, content) should be kept below **14 kB** (after compression)[8].

If you happen to exceed the budget, benchmark first and if needed, use a technique that does not inline and try boosting the resource priority with `<link rel="preload" as="image">`.

If you decide to inline and want to maximize caching benefits, you could inline icons for 1st-time-visitors, prefetch (`<link rel="prefetch">`) the sprite SVG in the background and for subsequent visits, only load the sprite SVG (e.g. by checking a cookie value on the server).

Inlining SVGs without polluting the JS bundle

Now that we know what to inline, let's see how to do it without adding the SVG back into our JS bundle again. In the following, we inline our Heart icon right after `<body>`, so any SVG referencing the sprite will show up right from the start. You could use the same technique for less important sprites, by placing the sprite output right before `</body>`.

```
import fs from "node:fs";

const svgIcons = await fs.readFile("path/to/icons.svg"); // ← load SVG file content

// … router might come from HTTP frameworks like fastify or express …
app.get("/", function () {
  const reactOutput = renderToString(App);
  return `<!doctype html><head><title>SVG-in-HTML</title></head>
     <body>
      <!-- ⬇ output the SVG sprite to be re-used and make it invisible -->
      <div style="display:none">${svgIcons}</div>

      ${reactOutput}
```

```
    </body>`;
});
```

Now we need to make sure any instance where the SVG should be used does not link to the file, but rather only to the ID:

```
<!-- Note the missing file path in the `href` attribute. -->
<svg><use href="#heart"></svg>
```

Be careful though, IDs now are global and not only inside of the SVG file.

Voilà, SVG-in-HTML. This could be expanded to extract all the IDs used per page/output, so it only spills out the SVGs that are actually used instead of all of them. **`used-styles`** is a good example of how to do this for critical CSS.

Wrap Up

By using the described techniques, you can make your JavaScript bundle smaller and more performant, which helps slow and old devices, and creates a more inclusive internet.

Aside from that and as always for web-performance related topics, there might be other lower-hanging-fruits, or more important things to optimize. So before you jump right into optimizing your SVGs, keep in mind, SVG-in-JS might not be your biggest culprit. Measure, then optimize.
Tip: A Ctrl+F for `<svg>` might be enough to give you an idea.

Other things to make your JS bundle smaller: ship Preact instead of React, ship Redaxios instead of Axios, look through Luke Edward's module collection that replaces bigger modules (e.g. uuid, clsx), replace CSS-in-JS with either a solution that has no runtime overhead like ecsstatic/kuma UI/Panda, or with e.g. CSS modules. There are also some Webpack plugins which replace bigger modules with smaller ones. Bundlephobia is your friend for finding smaller modules.

Tools / Snippets

Ben Adam has written a similar piece, where he shows a snippet on how to manage SVG sprites in React.

Epic Stack + SVG sprites - "using rmx-cli to automate the sprite generation"

JetBrains SVG sprite loader - “Webpack loader for creating SVG sprites.”

Icon-pipeline - ”🚚 SVG icon pipeline - Optimize icons & build SVG sprites”

SVGomg - “SVG Optimizer’s Missing GUI”

Footnotes & Comments

Thank you Barry Pollard and Kevin Farrugia for the invaluable feedback for this post.

1. Webpack Docs. As an alternative, you could do `new URL('path/to/svg.svg', import.meta.url)` for single SVGs. ↩

2. Inline SVGs run on the compositor on DPR = 1x screens, which usually leads to higher performance, as mentioned in this Chromium bug. For devices with a screen over 1x DPR, it is the opposite, `<svg>` consumes more CPU than `<img>`. You can try yourself: `<img>` Codepen and the inline `<svg>` Codepen. Use DevTools -> Ctrl + Shift + P -> Performance monitor. ↩ ↩[2] ↩[3]

3. Alfredo Lopez made me aware of a clever trick, setup by one of his colleagues: You could use query parameters (e.g. `HeartIcon.svg?color=green`) to dynamically embed `<style>:root { --color: green }</style>` into the SVG, so that CSS custom props can be used. Once a blog post about this technique is up, I’ll link it here. ↩

4. Chromium comment ↩

5. StackOverflow Reference ↩

6. web.dev article regarding DOM size ↩

7. Astro docs ↩

8. web.dev article regarding critical CSS ↩

Loading comments. If you see this for longer, please refresh the page.

#performance #images #svg #webpack #jsx #javascript

⬆ Back to top · 📖 Read other posts