



You don't need JavaScript for that

by [Kilian Valkhof \(https://kilianvalkhof.com\)](https://kilianvalkhof.com), published on Dec 02, 2023

Please don't feel antagonised by the title of this article. I don't hate JavaScript, I love it. I write bucketloads of it every single day. But I also love CSS, and I even love JSX HTML. The reason I love all three of these technologies is something called:

The rule of least power

It's one of the core principles of web development and it means that you should **Choose the least powerful language suitable for a given purpose.**

On the web this means preferring HTML over CSS, and then CSS over JS. JS is the most versatile language out of the three because you're the one describing how the browser should act, but it can also break, it can fail to load and it takes extra resources to download, parse and run. It is also very easy to

exclude keyboard users and people using assistive technologies with it.

In contrast to JS, which is imperative, HTML and CSS are declarative. You tell the browser *what* to do, now *how* to do it. That means the browser gets to choose how to do it, and it can do it in the most efficient way possible.

Because HTML and CSS features are handled by the browser they can be more performant, more native, more adaptable to user preferences and in general, more accessible. That doesn't mean it will always be (especially when it comes to accessibility) but when the browser does the heavy lifting for you, your end users will generally have a better experience.

But I need JS for that!

You might be thinking “All the things I use JS for, I *need* JS for”. That might be true, but it's good to know that both browser makers and specification writers have been porting a lot of functionality over to CSS and HTML that up to a few years ago needed JS. And that's what this article is about.

The tricky thing with the web is that once you learn how to build something there is never a reason to have to learn it again. That's the contract we have: the web is backwards compatible. (Very few exceptions apply, but the first web page still runs fine in all modern browsers.)

That also means that the solution you learned once becomes part of your toolbox, and you can keep re-implementing it and everytime it will still work. So the examples I'm going to give

below are cool (that's why I'm listing them) but what I want you to take away from this article is that just because you *know* something needs JavaScript, doesn't mean it still does. You can make better websites if you test those assumptions every now and then.

Custom Switches

We'll start this article off with something we've all had to implement at some point, custom switches. Instead of using a regular checkbox, the design calls for a nice looking switch. Instead of reaching for a JS solution with divs, onclick handlers and internal state, we're going to make use of a regular checkbox and the `:checked` pseudo class. Here's the HTML we're going to use:

```
<label>
  <input type="checkbox" />
  My awesome feature
</label>
```

There's a label element, and inside it a checkbox. The nice thing about this is that the browser is already doing things for us. Because the input is inside the label, the browser has associated them and now we can click anywhere on the label to toggle the checkbox, no onclick handler in sight. The browser gives us this for free. Feature-wise, we're done.

My awesome feature

A checkbox

Of course, designers may not like the way this looks and we want to create a great looking custom switch. So let's add a bunch of CSS:

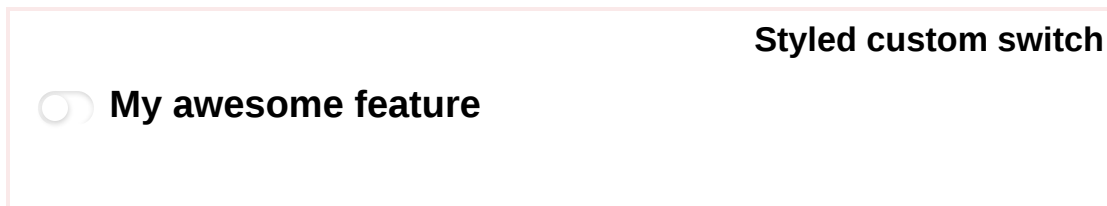
```
input {
  appearance: none;
  position: relative;
  display: inline-block;
  background: lightgrey;
  height: 1.65rem;
  width: 2.75rem;
  vertical-align: middle;
  border-radius: 2rem;
  box-shadow: 0px 1px 3px #0003 inset;
  transition: 0.25s linear background;
}

input::before {
  content: "";
  display: block;
  width: 1.25rem;
  height: 1.25rem;
  background: #fff;
  border-radius: 1.2rem;
  position: absolute;
  top: 0.2rem;
  left: 0.2rem;
  box-shadow: 0px 1px 3px #0003;
  transition: 0.25s linear transform;
  transform: translateX(0rem);
}
```

All the specifics of the styling here don't matter as much, but I want you to take a look at that first rule: `appearance: none`.

Form elements, along with images, are something called "replaced content". That means they're not really part of your HTML, but supplied by the browser. When the browser renders your HTML and finds replaced content, it leaves a box for it, and then replaces that box with the actual content. This is why, for example, images and form elements can't have pseudo-elements: they get replaced when the browser replaces the entire element.

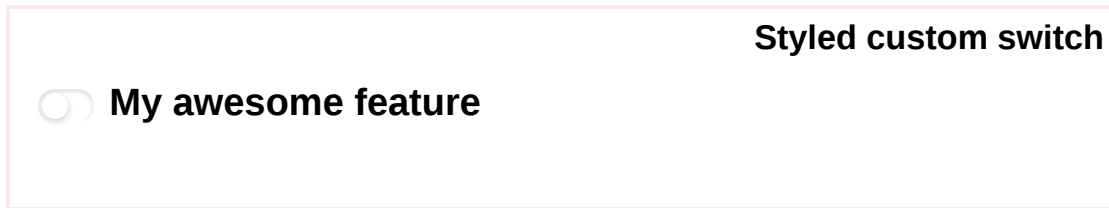
appearance is a way of telling the browser to stop doing that. It tells the browser: "Thanks, but I want to style my own form control". And that then allows us to use the `::before` pseudo-element. The input itself is now the background of our switch, and the `::before` pseudo-element is the little dot inside of it that does the toggling.



Clicking this still checks and unchecks the checkbox, but because we replaced the element we need to do the work of making that visible ourselves. That's where the `:checked` pseudo-class comes in:

```
:checked {  
  background: green;  
}  
:checked::before {  
  transform: translateX(1rem);  
}
```

When you click the checkbox, that `:checked` pseudo-class starts to match and that causes the styling to update.



So we have a great looking custom switch using native HTML elements and a bit of CSS, but we're not done yet. While for mouse users it's really clear which form control they're interacting with (since they're pointing at it and clicking), for people using a keyboard that's not so easy.

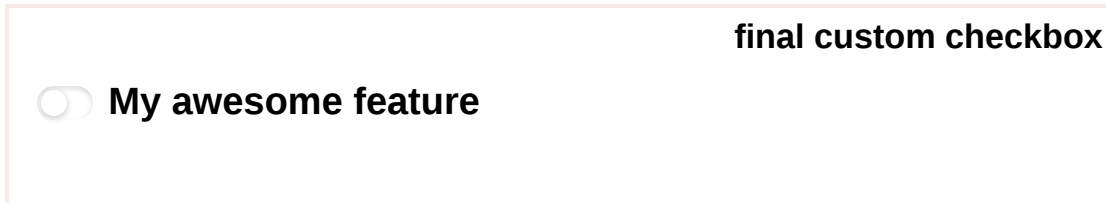
I'm sure you're familiar with this bit of CSS. To get rid of that ugly, dotted, boxy outline.

```
input:focus {  
  outline: none;  
}
```

If you're reading this, know that's not a good idea. But how do we make it look, well, nicer? Here too browsers have updated to make things better for us. The `outline` now follows the border-radius of an element, and we can also offset it away, or inside of, the element:

```
input:focus-visible {  
  outline: 2px solid dodgerblue;  
  outline-offset: 2px;  
}
```

Now, when a user interacts with an element using the keyboard (you can try pressing the spacebar after clicking it, or tabbing to it), `:focus-visible` will match (it won't when using a mouse) and they get a good looking, blue outline slightly around the element.



Lastly, I want you to replace that `outline: none` with something else:

```
input:focus {  
  outline-color: transparent;  
}
```

This will have the same result: Instead of the outline not being visible because it's hidden, it's not visible because it's transparent. For users that have high contrast mode (also called forced colors) turned on however, that outline becomes visible again because in high contrast mode, that transparent color gets replaced with a color the user chose, helping them see what they're interacting with even if they use a mouse.

This article isn't long enough to also go into what forced-colors does but if you want to learn more check out my article [forced colors explained \(https://polypane.app/forced-colors/\)](https://polypane.app/forced-colors/).

Datalist, a native autosuggest

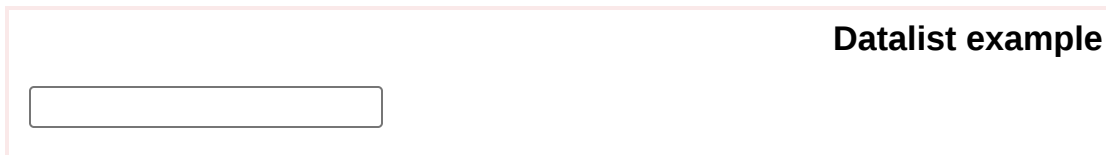
Instead of installing `$your-framework-autosuggest`, try out datalist in your next project. Datalist is the browsers built-in way of

showing a list of options as a user types into an input.

```
<input list="frameworks" />

<datalist id="frameworks">
  <option>Bootstrap</option>
  <option>Tailwind CSS</option>
  <option>Foundation</option>
  <option>Bulma</option>
  <option>Skeleton</option>
</datalist>
```

To use it, you add a datalist element with an ID and a set of options to your HTML. Don't worry, the element won't be visible. Then you use the `list` attribute on an input to associate the two.



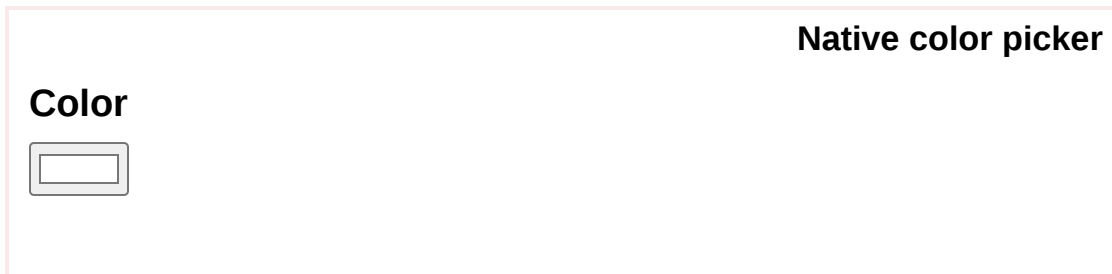
As a user now types into the input, the browser will show the datalist as a dropdown, automatically filtering the options as the user types. Because it's a regular input though, users still have the option to type in their own value. Lastly, they can see all of the options by selecting the input and using the arrow keys to navigate the list, or clicking the dropdown icon the browser adds.

A color picker that does more

There are a ton of good looking color pickers out there, with nice canvas UIs and sliders that are built with 100s of lines of JavaScript. But did you know that you can also use a native color picker?


```
<label> <input type="color" /> Color </label>
```

This single line of HTML also give you a color picker with a nice UI, already saving you a bunch of JS. But because we're letting the browser handle it, we actually get more functionality for free. In Chromium browsers, that native color picker also lets you pick a color, not just from your own site but from anywhere on the screen. Pretty neat!



A quick note here is that even though browsers show a nice color picker, your users might not all be able to use it. So offering a different way of picking a color (like a regular text input) is still a good idea.

Accordions

Accordions are a great way of making a page with a lot of content more structured and uncluttered by keeping content out of the way until a user needs it. And browsers give them to you for free with the `details` and `summary` elements:

```
<details>
  <summary>My accordion</summary>
  <p>My accordion content</p>
</details>
```

Accordion

▶ My accordion

By default everything inside a `details` element is hidden except for the `summary`. Then when a user clicks the `summary` element the browser will show the rest of the content.

What you'll often see is that one of the accordion items is already open, and the rest are closed. You can do that with the `open` attribute:

```
<details open>
  <summary>My accordion</summary>
  <p>My accordion content</p>
</details>
```

Accordion that starts open

▼ My accordion


My accordion content

If you come from the React world, you might look at this code and think “Well that's great, now it has the `open` prop and isn't going to close anymore” but luckily, that's not the case. That `open` attribute is only the starting state, and will update when a user interacts with the accordion.

When it comes to styling, the `details` element also has you covered. That little triangle (that your designer will want to replace the instant they see it) is a `::marker` pseudo-element that you can style:

```
summary::marker {
  font-size: 1.5em;
  content: "📧";
}
[open] summary::marker {
  font-size: 1.5em;
  content: "📧";
}
```

Accordion with styled markers

 My accordion

Keep in mind that changing the content can affect how assistive technologies announce your accordion. Read Manuel's article on that here: [details/summary_inconsistencies](https://www.matuzo.at/blog/2023/details-summary) (<https://www.matuzo.at/blog/2023/details-summary>).

The marker pseudo-element can't be styled as extensively as other elements (many CSS properties do not work on it, like positioning it in a completely different place), but you can replace its content, for example with emoji, or set a background color or image and change its font size.

With the `open` attribute you can easily give it different styling from the closed state.

Lastly, we want to do something about that summary element. It's clickable, but unlike a link it doesn't get a pointer cursor, and unlike a button it, well, doesn't look like a button. So I think we should add a hover and focus state to it and help our visitors realise that it's clickable:

```
summary: hover,  
summary: focus {  
  cursor: pointer;  
  background: deeppink;  
}
```

Accordion with indication on hover

▶ My accordion

I'm sidestepping the “only links should have pointer cursors” discussion here, the main point I'm making is that you need to do *something*.

Dialog modals

Sometimes you need to inform the user about something, or ask them something or get them to confirm something. In JavaScript, that's what `alert()`, `prompt()` and `confirm()` do. But they have a pretty big downside: they lock the main thread, meaning your page can't do anything else. They're also browser-native, so you can't style them to work with your design.

Building your own dialog is also asking for trouble: you need to keep the focus inside the dialog for accessibility, announce it's modal-ness, make sure users can't exit it accidentally, and you'll have to fight with whatever chat widget occupied the z-index of 2147483647 (if you know you know).

So that's why browsers now come with a native dialog element:

```
<dialog>
  <form method="dialog">
    <h3>This is a pretty dialog</h3>
    <button type="submit">Close</button>
  </form>
</dialog>
```

This element isn't shown by default and, for now, this is where I'm going to cheat a little and use JavaScript:

```
document.querySelector("button").addEventListener("click",
  document.querySelector("dialog").showModal();
});
```

A native dialog

Open dialog

Now there's changes in the works that will let you open dialogs without JavaScript, but they're not fully specced yet, let alone implemented. So for now, we need to use JavaScript to open the dialog. But that's it, the rest is all native HTML and CSS.

The dialog element has a `showModal()` function that it exposes and with it, you open the dialog. This dialog is opened on something called the `top-layer`, which is a new concept in browsers. For a primer, check out the explainer on MDN: [Top layer \(https://developer.mozilla.org/en-US/docs/Glossary/Top_layer\)](https://developer.mozilla.org/en-US/docs/Glossary/Top_layer).

The top layer is a new layer that's separate from your HTML, and you can "promote" elements to it. That means that elements on the top layer will always be above everything else, regardless of the z-index of an element and stacking context nesting.

Now that it's open though, you might notice that the browser doesn't give you any UI. The dialog is pretty much a div (not a button!) and it's up to you to provide the UI for closing. That's what the form in the code above does. You might've noticed it has a method of "dialog". When this form gets submitted, the browser takes that as a signal to close the dialog again.

With that, you can also create confirmation dialogs by providing two buttons, each with each own values:

```
<dialog>
  <form method="dialog">
    <p>Tabs or spaces?</p>
    <button type="submit" value="wrong">Tabs</button>
    <button type="submit" value="correct">Spaces</button>
  </form>
</dialog>
```

The button that a user clicked can be found by listening to the `close` event on the dialog and reading it's 'returnValue' property:

```
dialog.addEventListener("close", function () {
  console.log(dialog.returnValue);
});
```

Dialog with multiple buttons

Open dialog

If you have any other form data in there you can also read that as `formData` (<https://developer.mozilla.org/en-US/docs/Web/API/FormData>).

Because the dialog is essentially a `div` as far as styling it concerned, you can style it however you want. The browser will automatically place it in the middle of the screen for you, but everything else is up to you.

Dialog also comes with a new pseudo-element called `::backdrop`. That's the layer that sits between the dialog and the rest of the page, and you can style it to e.g. dim the rest of the page or otherwise direct a users attention to the dialog. For example, you can overlay a white layer and blur the page:

```
dialog::backdrop {  
  background: #fff5;  
  backdrop-filter: blur(4px);  
}
```

Dialog with styled backdrop

Open dialog

Just like the dialog element itself, the backdrop is positioned by the browser, so you won't need to worry about scrolling, fixed

elements and browser resizing. It's all handled for you by the browser.

In closing

I hope you found a few things in this article that made you realise you can use a little bit less javascript in your next project.

Whenever you change a known battle-tested implementation to something new it's good to test it, especially when it comes to accessibility, to make sure that you're not excluding anyone.

There are dozens more examples I could've added into this article, here are just some you can look into:

- Native smooth scrolling with `scroll-behavior: smooth` (but only when `prefers-reduced-motion: no preference` matches!),
- Native carousels with `scroll-snap`,
- "In-view" elements with `position: sticky`
- ...Not to name the whole concept of container queries.

And if we look into the future, we're getting even more cool things:

- Scroll driven animations
- Masonry layouts without `masonry.js` but with `grid-template-rows: masonry`
- A fully stylable `select` with the new `selectlist` element (where you can style each part of a select without destroying all the native functionality it comes with)
- The `:has()` selector that's going to eliminate a whole class of JS selection

This article is an adaption of a conference talk I gave that goes into more detail on these and other topics, and you can watch it here: [Stop Using JavaScript for That: Moving Features from JS to CSS and HTML. \(https://www.youtube.com/watch?v=ZTMUJu26b7Q\)](https://www.youtube.com/watch?v=ZTMUJu26b7Q).

So let me re-iterate the main point of this article:

Just because you *know* something needs JavaScript, doesn't mean it still does. You can make better websites if you test those assumptions every now and then.

About Kilian Valkhof

Web developer and creator of Polypane.app, the browser for developers.

Blog: [kilianvalkhof.com \(https://kilianvalkhof.com\)](https://kilianvalkhof.com).

Mastodon: [@kilian \(https://mastodon.social/@kilian\)](https://mastodon.social/@kilian).

X: [@kilianvalkhof \(https://twitter.com/kilianvalkhof\)](https://twitter.com/kilianvalkhof).

Polypane: [Polypane.app \(https://polypane.app\)](https://polypane.app).

More articles

built with [eleventy \(https://www.11ty.dev/\)](https://www.11ty.dev/) and  in Vienna by [@mmatuzo \(https://www.matuzo.at\)](https://www.matuzo.at).

[About](https://github.com/matuzo/HTMHell) [Contribution](https://github.com/matuzo/HTMHell) [Github \(https://github.com/matuzo/HTMHell\)](https://github.com/matuzo/HTMHell) [Feed](#)