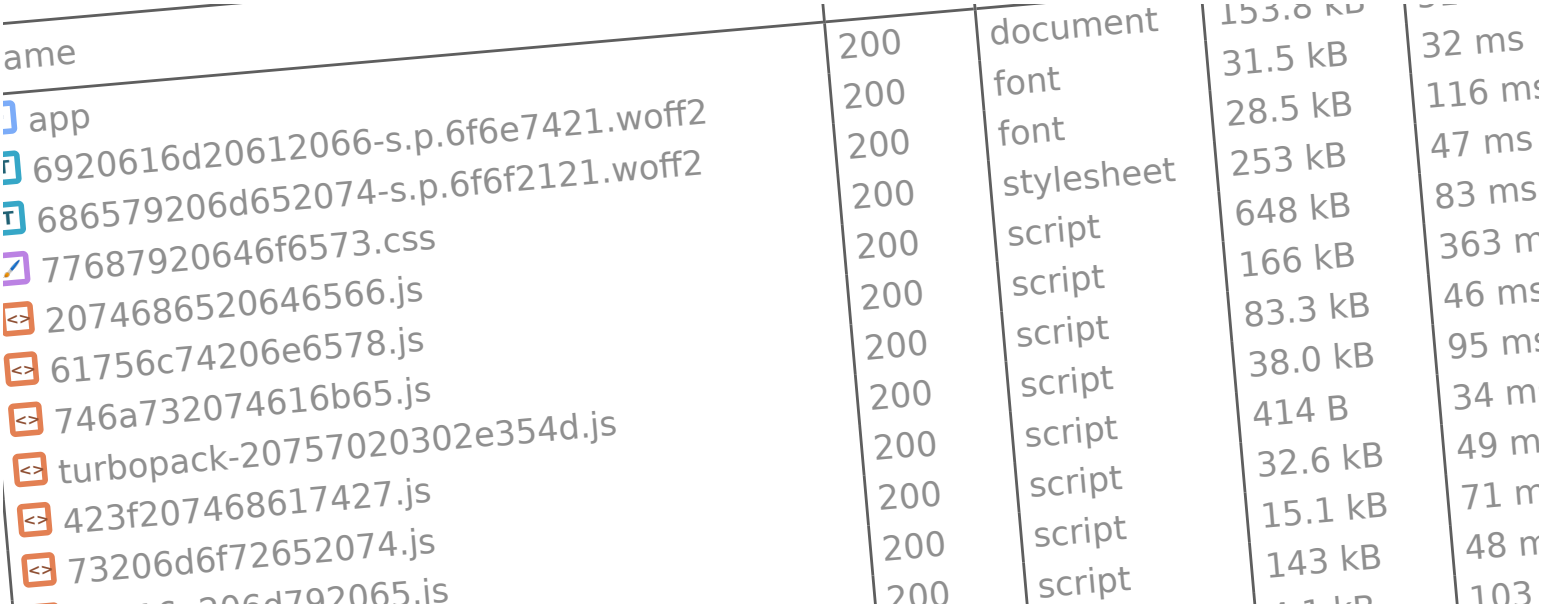
You no longer need JavaScript

2025-08-28 ¦ css

So much of the web these days is ruined by the bloat that is modern JavaScript frameworks. React apps that take several seconds to load. NextJS sites that throw random hydration errors. The *node_modules* folder that takes up gigabytes on your hard drive.

It's awful. And you don't need it.



The intro paragraph of this post is tongue-in-cheek. It's there to get you to read the rest of the post. I suspect the megabytes of tracking scripts intertwined with bad code is far more likely to be the real culprit behind all the terrible sites out there. Web frameworks have their time and place. And despite my personal distaste for them, I know they are used by many teams to build awesome well-optimized apps.

Despite that, I think there's some beauty in leaving it all behind. Not just the frameworks, but JavaScript altogether. Not every site needs JavaScript. Perhaps your e-commerce site needs it for its complex carts and data visualization dashboards, but is it really a necessity for most of what's out there?

It's actually pretty incredible what HTML and CSS alone can achieve.

So, what do you say?

My goal with this article is to share my perspectives on the web, as well as introduce many aspects of modern HTML/CSS you may not be familiar with. I'm not trying to make you give up JavaScript, I'm just trying to show you everything that's possible, leaving it up to you to pick what works best for whatever you're working on.

I think there's a lot most web developers don't know about CSS.

And I think JS is often used where better alternatives exist.

So, let me show you what's out there.

"But CSS sucks"

I believe a lot of the negativity towards CSS stems from not really knowing how to use it. Many developers kind of just skip learning the CSS fundamentals in favor of the more interesting Java- and TypeScript, and then go on to complain

about a styling language they don't understand.

I suspect this is due to many treating CSS as this silly third wheel for adding borders and box-shadows to a webapp. It's undervalued and often compared to glorified crayons, rather than what it really is - a powerful domain-specific programming language.

It's telling when to this day the only CSS joke in the webdev circles is centering a div.

Yes, the syntax isn't the prettiest, but is it *really* that hard?

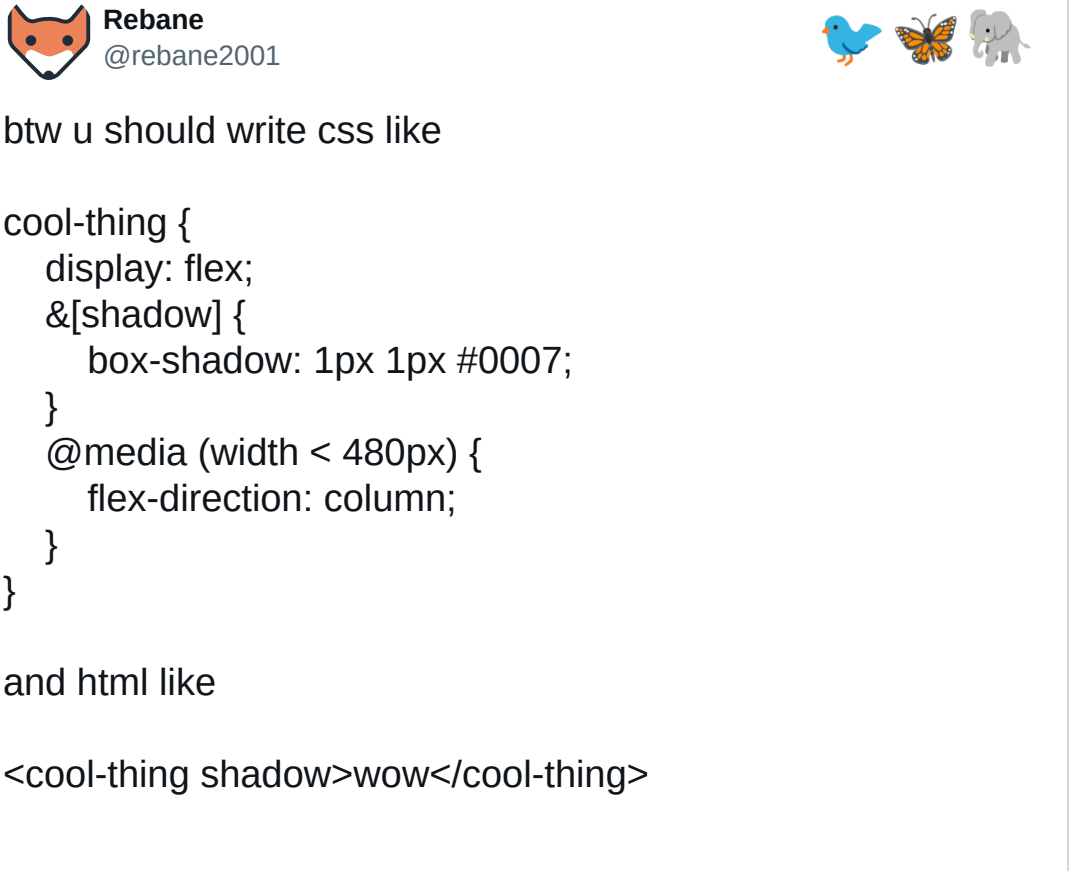
Besides, your devtools probably[1] come with a fun little gadget that lets you fiddle with the flexbox by just clicking around. You don't even need to remember the syntax.

I don't think CSS is fundamentally any more difficult than JS, but if you skip the basics on one and only focus on the other, it's no surprise it feels that way.

"But it's painful to write"

Another source of disdain for CSS is how awful it has been to write in the past. This is very much true, and is probably why things like Sass and Tailwind[2] exist.

But that's the thing, it *used* to be bad.



because it's allowed & modern & neat!

11:58 AM · Apr 8, 2025

1.5K

(yes! the code above is standards compliant[3])

In the past few years, CSS has received a ton of awesome quality-of-life additions, making it nice to do stuff that has historically required preprocessors or JavaScript.

Nesting is definitely one of my favorite additions!

In the past, you've had to write code that looks like this:

```
:root {
  --like-color: #24A4F3;
  --like-color-hover: #54B8F5;
  --like-color-active: #0A6BA8;
}

.post {
  display: block;
  background: #EEE;
  color: #111;
}

.post .avatar {
  width: 48px;
  height: 48px;
}

.post > .buttons {
  display: flex;
}

.post > .buttons .label {
  font-size: 24px;
  padding: 8px;
}

.post > .buttons .like {
  cursor: pointer;
  color: var(--like-color);
}

.post > .buttons .like:hover {
  color: var(--like-color-hover);
}

.post > .buttons .like:active {
  color: var(--like-color-active);
}

@media screen (max-width: 800px) {
  .post > .buttons .label {
    font-size: 16px;
    padding: 4px;
  }
}

@media (prefers-color-scheme: dark) {
  .post {
    background: #222;
    color: #FFF;
```

```
  }
}
```

And yeah, that's pretty awful to work with. For anything that involves multiple chained selectors, you kind of have to keep a mental map of how every parent selector relates to its children, and the more CSS you add the harder it gets.

But let's try it with nesting:

```
:root {
  --like-color: #24A4F3;
  --like-color-hover: hsl(from var(--like-color) h s calc(l + 10));
  --like-color-active: hsl(from var(--like-color) h s calc(l - 20));
}

.post {
  display: block;
  background: #EEE;
  color: #111;
  @media (prefers-color-scheme: dark) {
    background: #222;
    color: #FFF;
  }
  .avatar {
    width: 48px;
    height: 48px;
  }
  & > .buttons {
    display: flex;
    .label {
      font-size: 24px;
      padding: 8px;
      @media (width <= 800px) {
        font-size: 16px;
        padding: 4px;
      }
    }
    .like {
      cursor: pointer;
      color: var(--like-color);
      &:hover { color: var(--like-color-hover); }
      &:active { color: var(--like-color-active); }
    }
  }
}
```

That is way nicer to read[4]! All the relevant parts are right next to each other, so it's a lot easier to understand what's going on. Seeing the `&:hover` and `&:active` right next to the `.like` button is especially nice imo.

And since you can sort of see the structure - the parent selectors "guarding" the child ones - it also makes it a lot easier to get away with short and simple class names (or even referring to elements themselves).

You may have noticed that I'm also making use of relative colors in the second example. I think the MDN article has a lot of awesome examples, but the jist of it is that you can take an existing color, modify it in many different ways across multiple color spaces, and mix it with other colors using color-mix().

```
/* remove blue from a color */
rgb(from #123456 r g 0);
/* make a color transparent */
rgb(from #123456 r g b / 0.5);
/* make a color lighter */
hsl(from #123456 h s calc(l + 10));
/* change the hue in oklch color space */
oklch(from #123456 l c calc(h + 10));
```

```
/* mix two colors in oklab color space */
color-mix(in oklab, #8CFFDB, #04593B 25%);
```

These snippets are really useful for when you want something to be just ever so slightly darker or brighter, such as a button hover effect or a matching border color, and they're way nicer to use than doing all those color conversions in JavaScript. If you're feeling particularly adventurous, you could even go ahead and generate your entire color scheme in just CSS.

(yes! the color picker above is written in just css) *view-source*

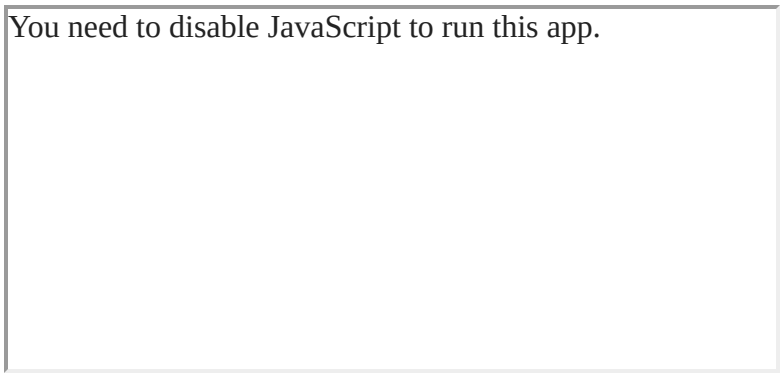
There are so many cool new CSS features that make writing it just that little bit nicer. Things like letting you use `(width <= 768px)` instead of `(max-width: 768px)` in your *@media* query, the lh unit that matches the line-height, the scrollbar-gutter property that solves the little scrollbar-related layout shifts, or the ability to finally center stuff vertically without flex/grid.

And all of this is brought together by the cherry on top that is Baseline. It's a guarantee that a specific feature works in every major browser[5], and it also lets you know since when - **newly available** features work in all the latest browsers, and **widely available** ones work in browsers up to 2.5 years old. Nesting, for example, has been fully supported in all browsers since December 2023, and thus will become *widely available* in June 2026. You can find the Baseline symbols in various places, such as the MDN docs[6].

These are just a few examples of what makes modern CSS so much nicer to write than what we had even just 5 years ago. It almost feels like comparing ES3[7] to ECMAScript 2025 - and I wouldn't blame your grudge if the former is what you're used to.

Why bother?

Okay, so CSS has more quality-of-life stuff than before. Still, why would one choose to use it over something else? Doesn't JavaScript already let us do everything just fine?



I think my reasons for using CSS fall into two main categories - because some users don't want to use JavaScript, and because doing things in CSS can be genuinely better.

My blog, for example, focuses on infosec topics. Many security researchers (myself included) use a hardened browser configuration to protect themselves, which often means disabling JavaScript by default. I think it's nice that they can fully experience my blog without changing their security settings or running a separate, sandboxed browser.

The same goes for privacy-conscious users, and it makes sense! As an experiment, I opened up a local Estonian news site in a web browser with JavaScript enabled. Can you guess how many js files it fetched? *(answer in footnote[8])* That's crazy! You do not want that running on your computer.

But surely, you are not *one of the evil devs* who loads a double-digit number of analytics scripts on your site - is there still any reason to reach for CSS?

Well, I think a lot of things are just plain nicer to make in HTML/CSS, both from the developer and end-user perspectives, be it for ease of use, accessibility, or performance.

Hover effects for your buttons? Toast animations? Input validation? All of these things *just work* in CSS, and you won't have to reinvent the wheel, or throw kilobytes of someone else's code at it. There will always be some cases where you do need that extra flexibility JavaScript often provides, but if you don't need that, and doing it in CSS is easier, then why not save yourself the trouble?

And the performance of CSS is so much better! Every JavaScript interaction has to go through an event loop that wastes CPU cycles, eats some battery, and adds that tiny bit of stutter to everything.

Sure, in the grand scale of things it isn't *that* bad, APIs like requestAnimationFrame are really good at keeping things smooth. But CSS animations run in the separate compositor thread, and aren't affected by stutters and blocking in the event loop.

It makes quite a difference on low-end devices, but feels nice even on high-end ones. CSS animations on my 240hz monitor look amazing[9] - JS can look pretty good too, but it has that tiny bit of stutter to it that keeps it from being perfect, especially if you plan on running other heavy code at the same time.

It also means you won't have to worry as much about optimization, as the browser takes care of a lot more of the rendering side of things, and often runs your stuff on the GPU if possible.

Pro tip! Wanna trigger animations from JS anyways? Use the modern Web Animations API to easily play the smooth CSS animations from JS.

Transitioning

Speaking of which, I think it's time I start showing you practical examples, and a good place to *start* showing the *styles* is well, *@starting-style*.

In the past it has been pretty annoying to add start animations (such as fade-ins) to elements. You've had to either set up an entire CSS animation with a separate *@keyframes* block to go with it, or do a transition using JavaScript where you first add an element to the page, then wait a frame, and then add a class to the element.

```css
.toast {
  transition: opacity 1s, translate 1s;
  opacity: 1;
  translate: 0 0;
  @starting-style {
    opacity: 0;
    translate: 0 10px;
  }
}
```

But this has all changed thanks to the new *@starting-style* at-rule!

Pretty much all you have to do is set your properties as usual, add the initial transition states to *@starting-style*, and add those properties to a transition. It's pretty simple and it kind of *just works* without having to trigger the animation in any way.

Lunalover

Another good example of where CSS shines is theming. Many sites *need* separate light and dark modes, and modern CSS makes dealing with that pretty easy.

```css
:root {
  color-scheme: light dark;
  --text: light-dark(#000, #FFF);
  --bg: light-dark(#EEE, #242936);
}
```

By setting the color-scheme property to `light dark`, you are telling the browser to automatically pick the theme according to the user preference, and you can then make use of that by setting color values with the light-dark() function.

Not only does it set your own colors, but also those of the native components, such as the default buttons, form elements, and scrollbars. It kind of just makes stuff work by default, and that's nice!

```
:root {
  color-scheme: light dark;
  &:has(#theme-light:checked) {
    color-scheme: light;
  }
  &:has(#theme-dark:checked) {
    color-scheme: dark;
  }
}
```

You can then add some way of overriding the *color-scheme* property to let the user pick a theme different from their system setting. Here I am using radio buttons to accomplish that.

Pro tip! CSS can't save the theme preference, but you can still do progressive enhancement. Make the themes work CSS-only, and then add the saving/loading of preference as an optional extra in JavaScript or server-side code.

Lyres and accordions

"But those don't look like radio buttons" I hear you cry.

Input elements such as radio buttons and checkboxes are a great foundation to build other stuff on top of - the example above consists of labels for the buttons and invisible radio buttons that can be checked for with the *:checked* pseudo-class.

```
<radio-picker aria-label="Radio buttons example" role="radiogroup">
  <label><input type="radio" name="demo" id="veni" checked>veni</label>
  <label><input type="radio" name="demo" id="vidi">vidi</label>
  <label><input type="radio" name="demo" id="vici">vici</label>
</radio-picker>
<style>
  radio-picker {
    display: flex;
    label {
      &:has(input:checked) {
        box-shadow: inset 0px 0px 8px 0px #888;
      }
      &:has(input:focus-visible) {
        outline: 2px solid #000;
      }
      box-shadow: inset 0px 0px 1.2px 0px #000;
      padding: 10px;
      cursor: pointer;
      background: #0002;
      &:hover { background: #0004; }
      &:active { background: #0006; }
    }
    input {
      /* To allow screen reader to still access these. */
      opacity: 0;
      position: absolute;
      pointer-events: none;
    }
  }
</style>
```

This is how I made the theme selector from the previous example. I've made the radio buttons half-visible in the demo for clarity, but with the `opacity: 0` they would not actually be visible.

There's a whole lot going on here, so let's break it down.

```
<radio-picker aria-label="Radio buttons example" role="radiogroup">
```

We start off with the *radio-picker* element - I just made it up, you can use a div instead if you'd prefer. We give it an aria-label to give the group an accessible name, and the aria role of *radiogroup* to make it work as a group for the radio buttons.

You could also use the *fieldset* element instead of doing the aria roles if that'd fit your use case better.

```
<label><input type="radio" name="demo" id="veni" checked>veni</label>
<label><input type="radio" name="demo" id="vidi">vidi</label>
<label><input type="radio" name="demo" id="vici">vici</label>
```

Next, we add the radio buttons with their respective labels - usually you'd have to use the *for* attribute on labels to define which element they're referring to, but since we have the *input* inside the *label* we don't have to do that.

All the `type="radio"` inputs should also have a *name* value set to the same thing so that they are grouped together (you still need[10] the radiogroup though). And then you can give them values or ids however you want.

```
label {
  &:has(input:checked) {
    box-shadow: inset 0px 0px 8px 0px #888;
  }
  &:has(input:focus-visible) {
    outline: 2px solid #000;
  }
  box-shadow: inset 0px 0px 1.2px 0px #000;
  padding: 10px;
  cursor: pointer;
  background: #0002;
  &:hover { background: #0004; }
  &:active { background: #0006; }
}
```

We then style the labels as we wish - the *:hover* and *:active* pseudo-classes can be used to make the buttons more fun to click, the *:has(input:checked)* selector can be used to define the style of the selected button, and the *:has(input:focus-visible)* selector can be used to add an outline when someone tabs over to the button.

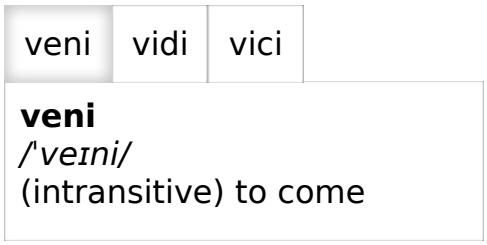
The difference between *:focus* and *:focus-visible* is that the former shows up even if you use your mouse, while the latter only shows up when you use keyboard navigation, so it's often visually more clean to use the latter.

```
input {
  opacity: 0;
  position: absolute;
  pointer-events: none;
}
```

And last, we make the radio button input *exist* while not being visible. This is a bit hacky, but it's how you can keep this control accessible to keyboard navigation and screen readers.

And that's how we get the cool-looking radio buttons!

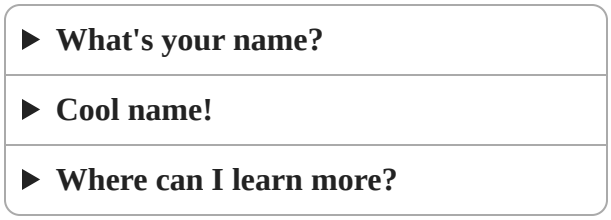
```
<radio-tabs>
  <div tabindex=0 id="tab-veni">veni...</div>
  <div tabindex=0 id="tab-vidi">vidi...</div>
  <div tabindex=0 id="tab-vici">vici...</div>
</radio-tabs>
<style>
  body:has(#veni:not(:checked)) #tab-veni,
  body:has(#vidi:not(:checked)) #tab-vidi,
  body:has(#vici:not(:checked)) #tab-vici {
    display: none;
  }
</style>
```



We can now use them in the CSS however we want by just seeing if they're *:checked*. Here I made tabs with separate divs for the content by using a *:has* selector on a parent element to find out which radio button is currently selected.

The *:has* selector has to be on a parent element that contains both the radio button and the target element - you can simply use *html* or *body* if you want it to work across the entire page. You should **never** use something like `:has(…)` by itself as it'll run the selector for every element of the page, which can cause performance issues (`body:has(…)` is okay).

```
<div>
  <details name="deets">
    <summary>What's your name?</summary>
    My name is Lyra Rebane.
  </details>
  <details name="deets">
    ...
  </details>
</div>
<style>
  div {
    border: 1px solid #AAA;
    border-radius: 8px;
    /* based on the MDN example */
    summary {
      font-weight: bold;
      margin: -0.5em -0.5em 0;
      padding: 0.5em;
      cursor: pointer;
    }
    details {
      &:last-child { border: none }
      border-bottom: 1px solid #aaa;
      padding: 0.5em 0.5em 0;
      &[open] {
        padding: 0.5em;
        summary {
          border-bottom: 1px solid #aaa;
          margin-bottom: 0.5em;
        }
      }
    }
  }
</style>
```



Finally, before we move on, I want to give you a quick introduction to the details element. It's great for if you want an accordion-style menu, such as for a FAQ section. The details open and close independently of each other, but you can set their name attribute to the same value to have only one open at a time.

Using them is pretty easy, put your content and a *summary* tag inside a *details* tag, and put the title inside the *summary* tag. The example above is a bit more convoluted for the visual flair, but all you *really* need is the html part of it.

The details elements are pretty stylable! You can add animations depending on the *[open]* state, and you can also get rid of the arrow by setting `list-style: none` on the *summary*.

Also, ctrl+f works with it, which is a big win in my book!

Validation

And lastly, I want to show you the power of input validation in HTML and CSS.

```
<label for="usrname">Username</label>
<input type="text" id="usrname" pattern="\w{3,16}" required>
<small>3-16 letters, only alphanum and _.</small>
<style>
 input:valid {
   border: 1px solid green;
 }
 input:invalid {
   border: 1px solid red;
 }
</style>
```

Username

3-16 letters, only alphanum and _.

This is a simple example of how you can validate an input field with a regex pattern. If you set a *pattern* attribute like above, a form that contains the input cannot be submitted unless the field matches the pattern. If you're submitting something like an e-mail address, a phone number, or a url, it might make sense to use the respective input types instead of writing your own regex.

Now, where CSS comes in is styling the input to show whether its value is valid. In the example above, I'm using *:valid* and *:invalid* to set a border color, but that comes with the downside of *always* having your input marked, even if the user hasn't entered anything yet.

```
input {
  border: none;
  border-radius: 2px;
  outline: 1px solid #000;
  &:focus { outline-width: 2px; }
  &:user-valid { outline-color: green; }
  &:user-invalid { outline-color: red; }
}
```

Username

3-16 letters, only alphanum and _.

An easy win here is to instead use *:user-valid* and *:user-invalid* - these pseudo-classes only become active once you've interacted with input field. I also made this example use an outline instead of a border, which I think looks a lot nicer.

It may sometimes even make sense to use a combination of *:valid* and *:user-invalid*.

And of course, you can use the *:has* selector to style other elements depending on the input too!

Password

The password must:
- be 8-16 characters
- contain at least i roman numeral
- not end with a letter

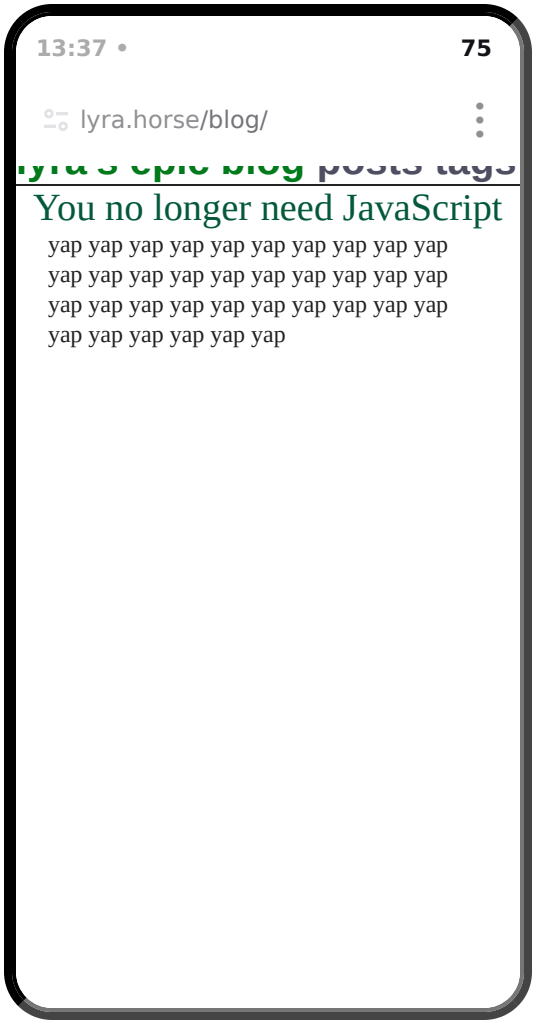
This one's just for fun ^_-!

I do want to mention that for some stuff, such as date pickers (mm/dd/yyyy) or datalists (pony), there are built-in elements that do the job, but you may find them limited in one way or the other. If you're making an input like that with specific requirements, you may still need to dip your feet in a bit of JavaScript.

Do not the vw/vh

This section is kind of random but I wanted to include it here because I think a lot of people are messing this one up and I want more people to know how to do this stuff right.

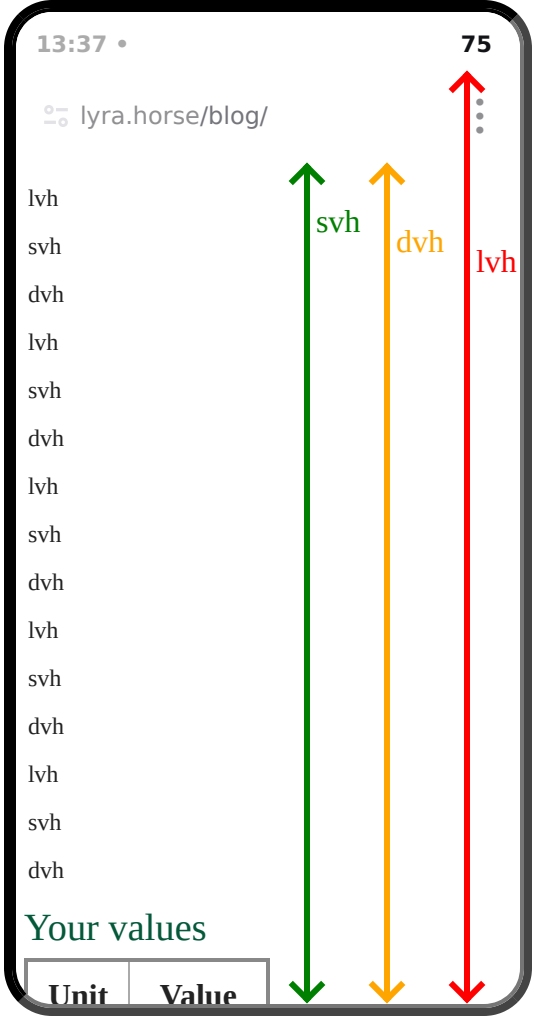
So CSS has vw/vh units that correspond to 1% of the viewport width and height respectively, which makes perfect sense for desktop browsers.



Where it becomes a bit more nuanced is on mobile devices. For example, mobile versions of both Firefox and Chrome will hide the URL bar when scrolling down on a page.

This causes the vw/vh units to be a bit ambigous - do they represent the *entire* available screen, only the area that's visible with the URL bar, or something in between?

If it's the first option, you might end up with buttons or links off-screen[11]! If it's the second, you may end up with a background div that doesn't cover the entire background.



The solution to this is to use the new responsive viewport units: **lvh**, **svh**, and **dvh**.

lvh stands for *largest* viewport height, and thus is useful for things like backgrounds that you'd want to cover the entire screen with, and wouldn't care about getting cut off.

svh stands for *smallest* viewport height, and should be used for things that must always fit on the screen, such as buttons and links.

And **dvh** stands for *dynamic* viewport height - this one will update to whatever the current viewport height is. It might seem like the obvious choice, but it should not be used for elements you don't want resizing or moving around as the user scrolls the page, as it could become quite annoying and possibly even laggy otherwise.

Of course, the respective **lvw**, **svw**, and **dvw** units exist too :).

Keyboard cat

By default, the viewport units do not account for the keyboard overlaying the page.

There are two ways to deal with that: the *interactive-widget* attribute, and the *VirtualKeyboard* API.

The former option is widely supported across browsers, works without JS, and goes in the meta viewport tag. It makes it so that opening the keyboard will change *all* of the viewport units.

```
<meta name="viewport" content="width=device-width, interactive-widget=resizes-content">
```

The latter option is currently only supported in Chromium-based browsers, and requires a single line of JavaScript to use:

```
navigator.virtualKeyboard.overlaysContent = true;
```

The advantage of the second option is that it allows you to use environment variables in CSS to get the position and size of the keyboard, which is pretty cool.

```
floating-button {
  margin-bottom: env(keyboard-inset-height, 0px);
}
```

But considering the fact that it doesn't work cross-browser, I'd avoid it.

CSS wishlist

Alright, so this is a little different from the rest of the post, but I wanted to bring up some things that I wish were in CSS. I haven't fully fleshed out all of them, so some definitely wouldn't fit the spec as-is, but maybe they can inspire some other stuff at least.

They are just fun ideas, don't take them too seriously.

Reusable blocks

I wish it was possible to put classes in other classes in CSS, so that you could write something like:

```
.border {
  border: 2px solid;
  border-radius: 4px;
}

.button {
  @apply border;
}

.card {
  @apply border;
}
```

This is something that Tailwind already has, and that makes me jealous.

Combined @media selectors

We can currently do nested *@media* queries, and also multiple selectors at the same time:

```
div {
  &.foo, &.bar {
    color: red;
    padding: 8px;
    font-size: 2em;
  }
  @media (width < 480px) {
    color: red;
    padding: 8px;
    font-size: 2em;
  }
}
```

But we cannot combine the two into a single selector:

```
div {
  @media (width < 480px), &.foo {
    color: red;
    padding: 8px;
    font-size: 2em;
  }
}
```

Which means if you want to do that you'll inevitably have to repeat code or do some silly variable hacks, neither of which is ideal.

n-th child variable

For many of the CSS crimes I like to commit, I often end up writing code like:

```
div {
  span:nth-child(1) { --nth: 1; }
  span:nth-child(2) { --nth: 2; }
  span:nth-child(3) { --nth: 3; }
  span:nth-child(4) { --nth: 4; }
  span:nth-child(5) { --nth: 5; }
  ...
  span {
    top: calc(--nth * 24px);
    color: hsl(calc(var(--nth) * 90deg) 100 90);
  }
}
```

And I think it would be a lot nicer if we could instead just do:

```
div {
  span {
    --nth: nth-child();
    top: calc(--nth * 24px);
    color: hsl(calc(var(--nth) * 90deg) 100 90);
  }
}
```

n-th letter targeting

CSS has the ability to style the ::first-letter of text. It'd be cool if were was also a **::nth-letter(…)** selector, similar to :nth-child. I suspect the reason this isn't a thing is because the *::first-letter* selector is a pseudo-element, which would be a bit tricky to implement with the nth-letter idea.

```
/* not a real feature */
p::nth-letter(2) {
  color: red;
}
```

hi there~

Blackle suggested that combining the nth-child() variable with :nth-letter targeting would also be fun for certain effects, such as putting the value in the sin() function to create wavy text.

```
div {
  /* not a real feature */
  --nth: nth-child(nth-letter);
  will-change: transform;
  translate: 0 calc(sin(var(--nth) * 0.35 - var(--wave) * 3) * 5px);
  color: color-mix(in oklch, #58C8F2, #EDA4B2 calc(sin(var(--nth) * 0.5 - var(--wave)) * 50% + 50
}
```

(hover to play animation)

Unit removal

I wish you could easily remove units from values, for example by dividing them.

```
div {
  /* Turns into: 816 (no unit) */
  --screen-width: calc(100vw / 1px);
  color: hsl(var(--screen-width) 100, 50);
}
```

This would allow you to use the size of the viewport or container as a numeric variable for things other than length. For example, the color picker from earlier uses it to convert the location of the color picker dot to a number to be used in a color value instead.

Uh, but wait? Does that mean this feature already exists?

Yeah, lol! We already have the ability to get unitless values in CSS, but it involves doing hacky stuff such as `tan(atan2(var(--vw), 1px))` with a custom *@property*. It'd be nice to have this as just a division, for example.

Oh, and good news, this one we might actually be getting soon!

Also if you do something like `calc(1px + sqrt(1px * 1px))` your browser will crash[12].

A better image function

The **image()** function exists, but no browsers implement it. It's similar to just using *url()*, but adds some really cool features such as a fallback color, and image fragments to crop a smaller section out of a bigger image (think spritesheets).

We can already do both fallbacks and spritesheets with the various background properties, but it'd be nice to have this pretty syntax. I'd honestly love this syntax even more for <img> tags than CSS.

style tags in body

I make heavy use of <style> tags in <body> for my projects. On my blog, for example, I write the relevant CSS close to their graphics so that you can start reading the blog before the entire page (or the entire CSS) has finished loading[13]. And it works great!

But what's unfortunate is that despite browsers supporting this, and major sites using this, it's not officially spec-compliant. I suspect it's in the spec to avoid the FOUC footgun, but there are so many reasons you would want/need style in body that I don't think it justifies it.

I think an HTML validator should warn for this, but not error.

The art

I want to end this article by saying that to me, web development is an art, and thus, CSS is too. I often have a hard time relating to people who do webdev solely to earn money or build a startup - web development is very different when you're on a team and are given tasks from above instead of having free will over what you create for fun.

It's probably most apparent with things like AI[14], that for me take all the fun and creativity out of my work. But it also applies to build chain tooling such as linters and minifiers - the way I write my code is part of the art, and I don't want a tool to erase that. I don't even use an IDE[15].

Among the practical reasons for sticking to CSS listed throughout this post, there's a secret extra reason I like to do everything in CSS, and that's expression and art. Art isn't always practical, and using CSS isn't either. But it's how I like to express myself, and it's why I do what I do.

I tried to keep this post approachable and practical for all web developers. But there is so much more to CSS that I'd like to talk about, so expect another post about the stuff that isn't practical, and is instead just cool as fuck. I think *CSS is a programming language*, and I made a game to prove it.

But that's a topic for another time.

afterword

it's been almost a year since my last post, but i hope it's been worth the wait ^_^

as usual, this post is a self-contained html file with no javascript, images, or other external resources - everything on the page is handwritten html/css, weighing in at around 49kB gzipped. it was really fun creating all the little interactive widgets and visuals this time around, i think i've improved in css a lot since the last time i posted.

this entire post turned out to be a bit of a fun mess (as did i!), it's almost like a chaotic gradient of tone throughout, i hope it was still interesting and enjoyable to read though.

i have a few new posts in the works: in addition to the second css one mentioned earlier, i also have one about a new web vulnerability subclass i discovered, and one about a trans topic. i'm not sure when these posts will come out, but we'll see! make sure to add me to your rss reader if that sounds fun.

i'll also be giving a talk at bsides tallinn in september! i'm hoping to also do css-related talks at the next ccc and disobey, but we'll have to see whether i get accepted and have the travel budget for those.

thank you so much for reading <3

Discuss this post on: twitter, mastodon, lobsters

---

1. Chrome's DevTools come with the cool flexbox widget. Firefox's however don't seem to for some reason? I find that weird because Firefox does have really good tools for flexbox and grid development, so this seems like an odd omission. ↩

2. While I think what I said is true, Tailwind does have more to its existence, the core of which can be found in this post by its creator. ↩

3. You are allowed to just make up elements as long as their names contain a hyphen. Apart from the 8 existing tags listed at the link, no HTML tags contain a hyphen and none ever will. The spec even has `<math-α>` and `<emotion-😍>` as examples of allowed names. You are allowed to make up attributes on an autonomous custom element, but for other elements (built-in or extended) you should only make up `data-*` attributes. I make heavy use of this on my blog to make writing HTML and CSS nicer and avoid meaningless div-soup. ↩

4. Still not nice to read for you? I'm personally not a fan of BEM, but I'd definitely recommend reading up on it too if you just don't vibe with the way I'm writing my examples. Also, my example intentionally shows off a lot of the syntax at once, but in the real world it might make sense to structure things a little differently. ↩

5. Baseline browsers are Safari (macOS/iOS), Chrome (desktop/Android), Edge (desktop), and Firefox (desktop/Android). ↩

6. The MDN docs of course also list detailed browser compatibility, but the Baseline symbols are nice for just getting a quick "yeah, we can use it and it'll work for everyone" type overview. ↩

7. ES3 (1999) is the last “classic” version of JavaScript. In 2009 we got the first major revision known as ES5, and a few years later we kicked off the yearly spec updates with ES2015. Also ES4 was abandoned which makes me feel sad :c. ↩

8. 93 files!! Seems like they’re 1/3 functionality, 1/3 ads, and 1/3 analytics. The site works just fine with JavaScript disabled - only stuff like the comments section and ads won’t load. It’s no longer a laggy mess either for some reason. ↩

9. I think the x3ctf challenges page looks really smooth on my computer - the marquee text animation and clicking on the challenges is buttery. And it also runs pretty well on the low-end hardware I have. Note that some browser performance recording tools can act a bit weird with CSS animations, so make sure your tools are working as expected before using them. Unrelated, but I made some other cool x3ctf web stuff too - check out the archive. ↩

10. There’s a bug in Chrome that *requires* you to use a fieldset/radiogroup for the radio button index to work correctly in screenreaders. Eg if you have 3 radio buttons with the same *name*, selecting one of them should read *“radio button 1 of 3”*, which is what Firefox does, but in Chrome it will instead read it as *“radio button 4 of 9”* or whatever if you don’t have a fieldset/radiogroup because it kind of just combines all the radio buttons on the page into a single index. ↩

11. A certain HR platform I have to use puts its action buttons at the very bottom of a 100vh container, leading to them not being visible/interactable on my phone - not a headache you want to go through when requesting sick days. It’s a good example of how just using the wrong unit can cause a pretty bad real world accessibility problem. ↩

12. Well, probably not. This is a bug I found while writing this post that only affects Chrome, and it’ll probably get fixed before it even manages to hit stable. *Update:* I took so long to get this blog post out that it has been fixed now. During the writing of this blog post I found another bug in Chrome though, which is pretty funny. *Update 2:* I found yet another Chrome bug while writing this post, this one is kinda weird, you should read it. ↩

13. This matters for people on slow connections, such as bad mobile data, satellite internet, tor, or iodine. While my blog posts are very small in size, the CSS alone can take up more than the first 14kB of a TCP round trip, so with blocking CSS in the head you might have to wait a few extra seconds (or minutes, in the case of iodine) just to start reading the first paragraph. Now, that 14kB number isn’t completely accurate in the modern world, but testing on my own server (HTTP/2, TLS 1.3), around ~16kB of the compressed html reaches the browser in the first batch of http data. ↩

14. By this I mean tools such as Copilot, Cursor, chatbots etc. I understand there is a huge difference between full-on *vibe coding* and just using the tab key, but I **do not** want to use or interact with any of those tools. Please respect that. ↩

15. I write all my code (and blogposts) in Sublime Text, which to me is just a glorified version of Notepad. The features over Notepad it gives me are syntax highlighting, multiple cursors, keyboard shortcuts, and a better visual design. It doesn’t do that much, and yet, it’s perfect. It’s so good I paid for it. ↩
