```
    --max-content: 1024px;
    --max-hero: calc(var(--max-content) - 16px);
}

.hero {
    max-width: var(--max-hero);
    &__image {
        object-fit: cover;
        &::before {
            content: "&copy;";
        }
        @media (max-width: var(--grid-breakpoint--sm))
            @include semitransparent(0.5);
        }
    }
}
```

Edit Manage Stats

Ingo Steinke
Posted on Mar 30 • Updated on May 11

10

Vanilla+PostCSS as an Alternative to SCSS

#css #webdev #programming #tutorial

Sass and SCSS have been popular tools to enhance the official CSS syntax, much like TypeScript or CoffeeScript have added features to JavaScript missing in the "vanilla" core language specification.

Would you still use CoffeeScript in 2023?

As a front-end web developer, do you still use CoffeeScript or jQuery? Unlikely, as TypeScript, ES/TC39 and Babel (and the retirement of Internet Explorer thanks to @codepo8 and his EDGE team) have helped to transform JavaScript into some kind of a modern programming language.

Vanilla CSS + PostCSS

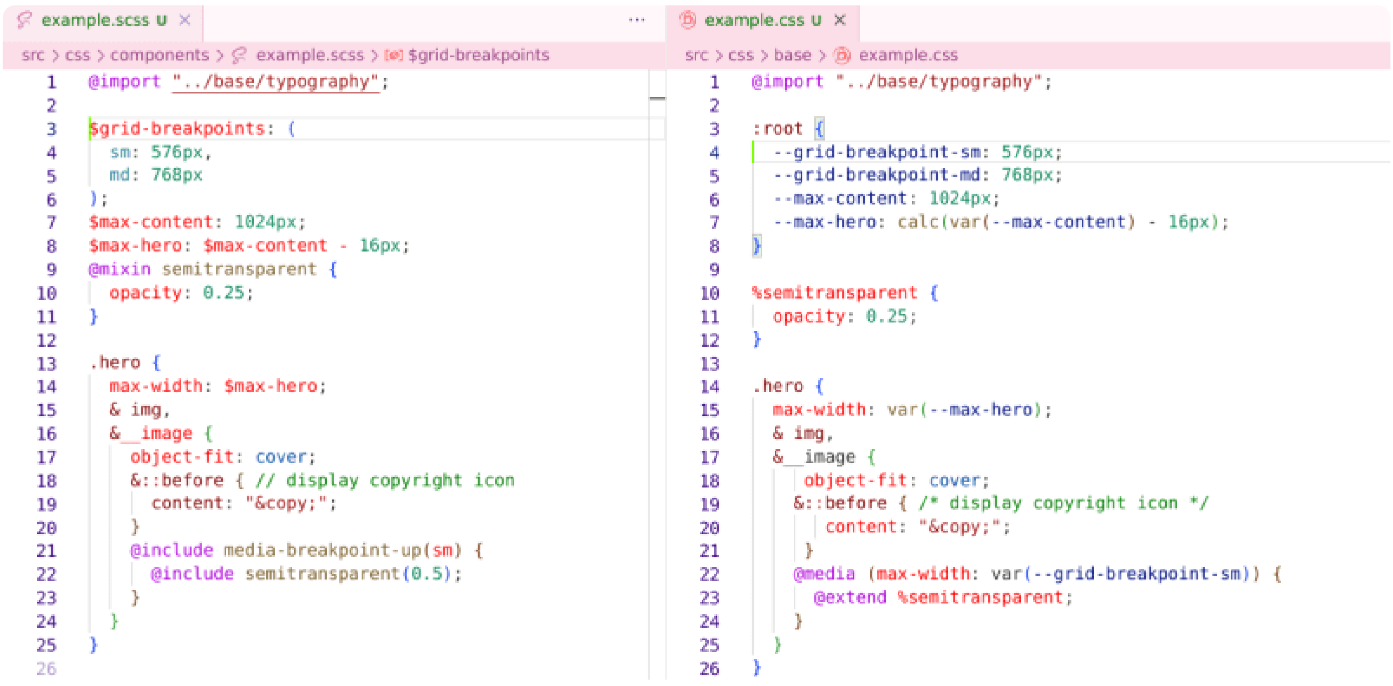
Vanilla CSS has taken a similar path with ambitious working drafts, better browser support, and PostCSS to fill the gap for user agents lagging behind. So why is Sass/SCSS still so popular? Maybe we go so used to it that we might have forgotten what problems it was meant to solve in the first place.

Cases for Using Sass in 2023

While we may disagree whether Sass is still relevant today, Mayank's case for using Sass in 2022 sums up the many use cases for Sass/SCSS including a timeline from 2006 (Sass) to "2022+" (nesting). Nesting CSS used to one of the few good reasons left to choose Sass, SCSS (or less) in a new web project.

Making Code Easier to Read and Maintain

Nesting, variables, extends / mixins, and imports all help us to make our code more readable and easier to maintain. But we can achieve that with or without SCSS, if you don't mind the occasional extra `var(--)` or `calc()`. That's valid CSS without any parsing, so we could even copy and paste between our editor and our browser and reduce the mental load of learning an additional syntax.


example.scss U
src > css > components > example.scss > $grid-breakpoints

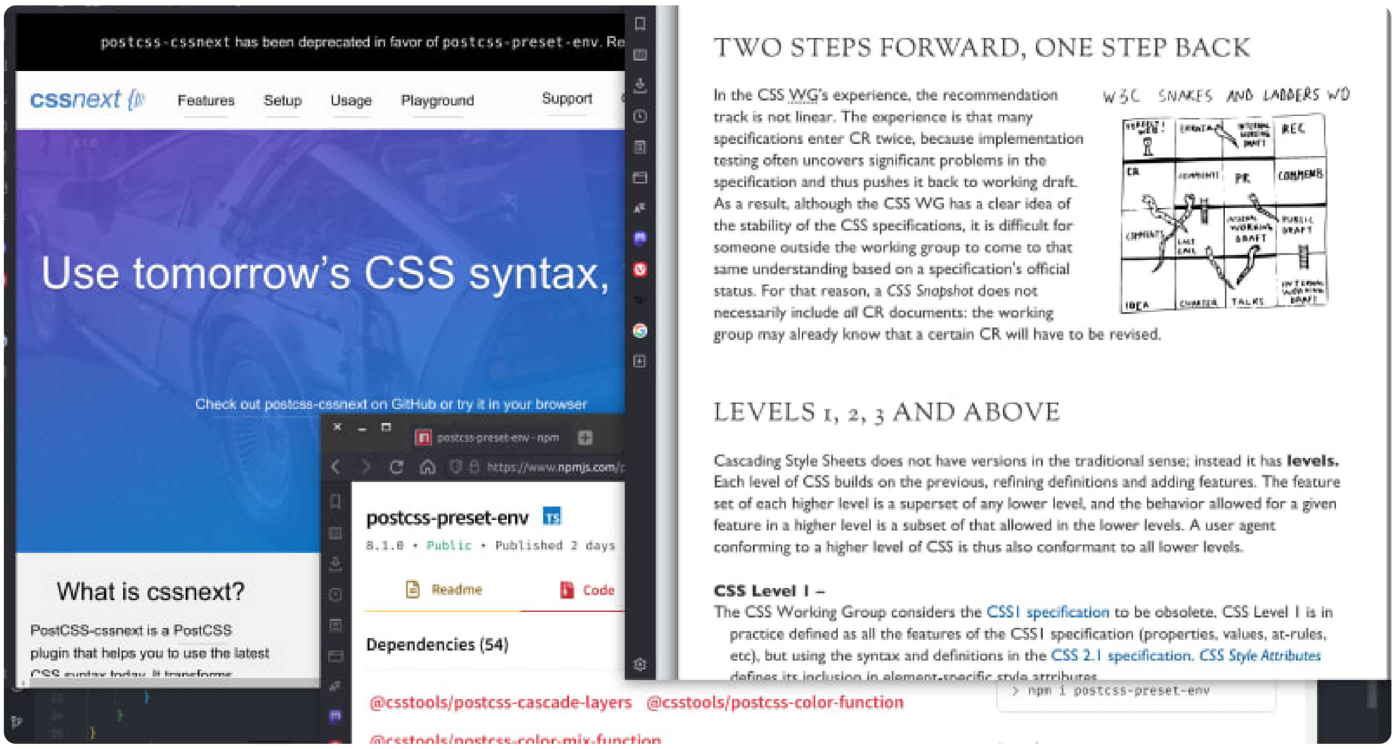
```scss
@import "../base/typography";

$grid-breakpoints: (
  sm: 576px,
  md: 768px
);
$max-content: 1024px;
$max-hero: $max-content - 16px;
@mixin semitransparent {
  opacity: 0.25;
}

.hero {
  max-width: $max-hero;
  & img,
  &__image {
    object-fit: cover;
    &::before { // display copyright icon
      content: "&copy;";
    }
    @include media-breakpoint-up(sm) {
      @include semitransparent(0.5);
    }
  }
}
```

example.css U
src > css > base > example.css

```css
@import "../base/typography";

:root {
  --grid-breakpoint-sm: 576px;
  --grid-breakpoint-md: 768px;
  --max-content: 1024px;
  --max-hero: calc(var(--max-content) - 16px);
}

%semitransparent {
  opacity: 0.25;
}

.hero {
  max-width: var(--max-hero);
  & img,
  &__image {
    object-fit: cover;
    &::before { /* display copyright icon */
      content: "&copy;";
    }
    @media (max-width: var(--grid-breakpoint-sm)) {
      @extend %semitransparent;
    }
  }
}
```

Nesting, Variables, Extends, and Imports

Some features like calculation have long been possible in native CSS, and some other improvements like scoped modules, parent selectors, or container queries, aren't related to Sass, but still worth mentioning for a modern / future CSS setup recommendation using PostCSS.


postcss-cssnext has been deprecated in favor of postcss-preset-env. Re

Use tomorrow's CSS syntax,

Check out postcss-cssnext on GitHub or try it in your browser

What is cssnext?

PostCSS-cssnext is a PostCSS plugin that helps you to use the latest CSS syntax today. It transforms

postcss-preset-env

8.1.0 • Public • Published 2 days

Dependencies (54)

@csstools/postcss-cascade-layers @csstools/postcss-color-function

@csstools/postcss-color-mix-function

```
> npm i postcss-preset-env
```

TWO STEPS FORWARD, ONE STEP BACK

In the CSS WG's experience, the recommendation track is not linear. The experience is that many specifications enter CR twice, because implementation testing often uncovers significant problems in the specification and thus pushes it back to working draft. As a result, although the CSS WG has a clear idea of the stability of the CSS specifications, it is difficult for someone outside the working group to come to that same understanding based on a specification's official status. For that reason, a CSS *Snapshot* does not necessarily include *all* CR documents: the working group may already know that a certain CR will have to be revised.

LEVELS 1, 2, 3 AND ABOVE

Cascading Style Sheets does not have versions in the traditional sense; instead it has **levels.** Each level of CSS builds on the previous, refining definitions and adding features. The feature set of each higher level is a superset of any lower level, and the behavior allowed for a given feature in a higher level is a subset of that allowed in the lower levels. A user agent conforming to a higher level of CSS is thus also conformant to all lower levels.

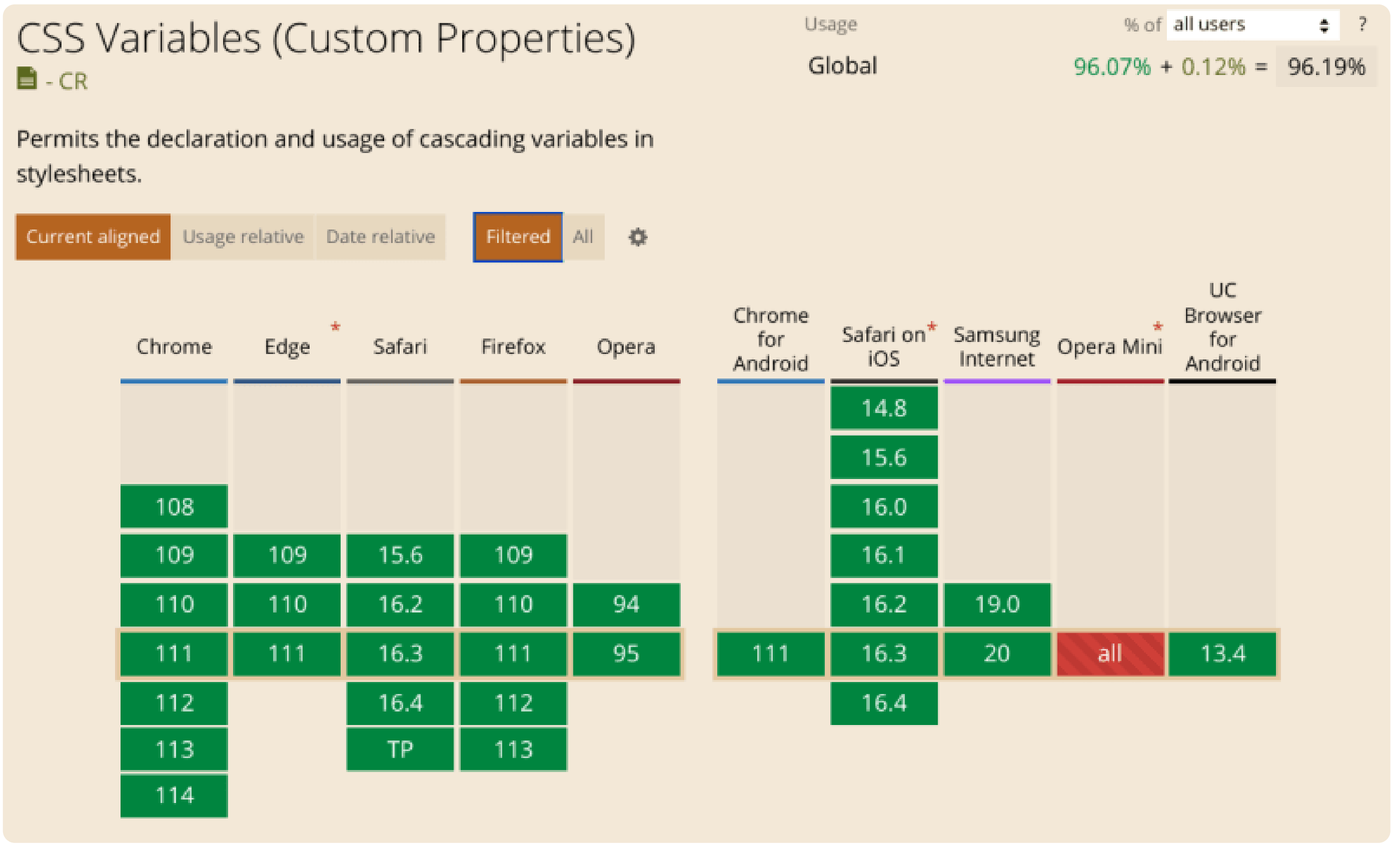
CSS Level 1 –
The CSS Working Group considers the CSS1 specification to be obsolete. CSS Level 1 is in practice defined as all the features of the CSS1 specification (properties, values, at-rules, etc), but using the syntax and definitions in the CSS 2.1 specification. *CSS Style Attributes* defines its inclusion in element-specific style attributes.

Future CSS Configuration beyond CSS next

Switching from a ready-made tool like Sass or a recommendation package like cssnext (deprecated since 2019) or PostCSS Preset Env (archived in 2022), to the modular PostCSS Preset Env plugin set we can choose a helpful and convenient set of future CSS features beyond the current stable client CSS.

SCSS Variables vs. CSS Custom Properties

Can we use custom properties? Yes, we can!


CSS Variables (Custom Properties)

- CR

Usage % of all users ?

Global 96.07% + 0.12% = 96.19%

Permits the declaration and usage of cascading variables in stylesheets.

Chrome	Edge *	Safari	Firefox	Opera	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	UC Browser for Android
						14.8			
						15.6			
108						16.0			
109	109	15.6	109			16.1			
110	110	16.2	110	94		16.2	19.0		
111	111	16.3	111	95	111	16.3	20	all	13.4
112		16.4	112			16.4			
113		TP	113						
114									

Unless we can't live without hierarchical objects to structure our design tokens (or use map values to iterate over an abundant number of adaptive breakpoints), native custom properties can replace SCSS variables even without any transpilation.

The native CSS syntax is a bit more verbose though, as we need to use the built-in `var()` function to access custom properties:

```
// SCSS
$red = red;
color: $red;


/* CSS */
--red = red;
color: var(--red);
```

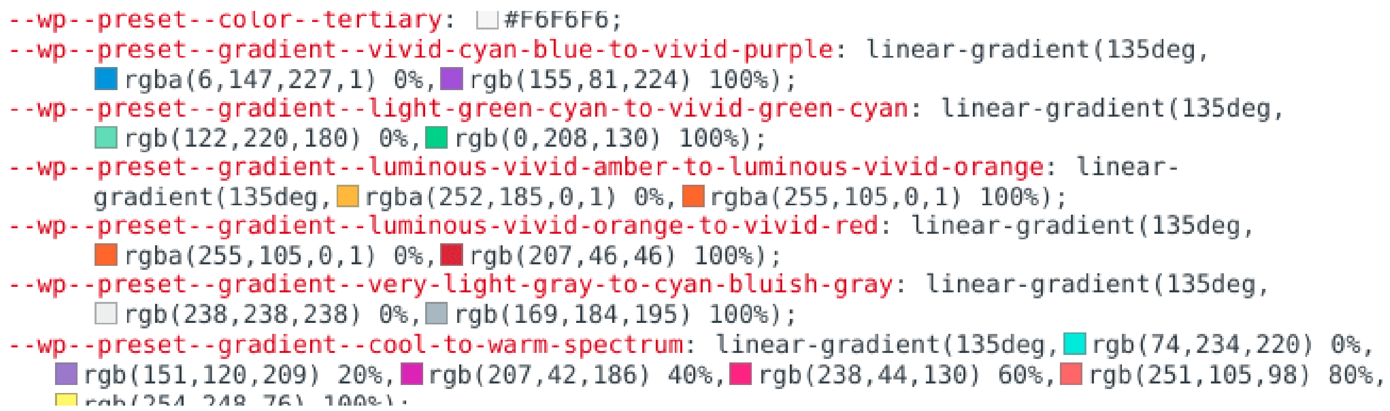
But to be absolutely sure to ship 100% backwards compatible code, we can use, postcss-custom-properties a PostCSS) plugin to export static values just like SCSS variables are converted when generating the final CSS code.

`postcss.config.js`:

```
const postcssCustomProperties = require(
  'postcss-custom-properties'
);
```

```
module.exports = {
    plugins: [
        postcssCustomProperties({
            preserve: false,
        }),
    ]
}
```

Custom properties are expected to be in kebab-case with hyphens, but without dots or hierarchy, at least according to stylelint's recommended custom-property-pattern. Variable objects like SCSS `$grid-breakpoint.md` could become `--grid-breakpoint--md`, which is good enough in my opinion. Even a complex prefixed design system like the WordPress theme engine is still readable. They even export those values to the client, so maybe I shouldn't worry about converting my `var()` and `calc()` statements anymore as well? Here is an example of a very long property name found in the default WordPress presets palette: `--wp--preset--gradient--very-light-gray-to-cyan-bluish-gray`.



```
--wp--preset--color--tertiary: ⬜#F6F6F6;
--wp--preset--gradient--vivid-cyan-blue-to-vivid-purple: linear-gradient(135deg,
      🟦rgba(6,147,227,1) 0%,🟪rgb(155,81,224) 100%);
--wp--preset--gradient--light-green-cyan-to-vivid-green-cyan: linear-gradient(135deg,
      🟩rgb(122,220,180) 0%,🟩rgb(0,208,130) 100%);
--wp--preset--gradient--luminous-vivid-amber-to-luminous-vivid-orange: linear-
      gradient(135deg,🟨rgba(252,185,0,1) 0%,🟧rgba(255,105,0,1) 100%);
--wp--preset--gradient--luminous-vivid-orange-to-vivid-red: linear-gradient(135deg,
      🟧rgba(255,105,0,1) 0%,🟥rgb(207,46,46) 100%);
--wp--preset--gradient--very-light-gray-to-cyan-bluish-gray: linear-gradient(135deg,
      ⬜rgb(238,238,238) 0%,⬜rgb(169,184,195) 100%);
--wp--preset--gradient--cool-to-warm-spectrum: linear-gradient(135deg,🟦rgb(74,234,220) 0%,
   🟪rgb(151,120,209) 20%,🟪rgb(207,42,186) 40%,🟥rgb(238,44,130) 60%,🟥rgb(251,105,98) 80%,
   🟨rgb(254,248,76) 100%);
```

`calc()` and other built-in Functions

When Sass was released as "syntactically awesome style sheets" in 2006, CSS still lacked many of the features making it so versatile and useful today, like the `calc()` function, which has become so common and well supported that I nearly forgot to mention calculation as an (obsolete) Sass/SCSS feature at all.

Today, Sass calculations are nothing but syntactical sugar allowing us to omit an explicit built-in `calc()` function of native CSS.

Future-Proof CSS?

While the SCSS version looks a little bit more compact and elegant, but the native CSS version will work in nearly 100% of your customers's browsers even if you shipped it without any PostCSS postprocessing. It will most probably also work when you'll open the project for bugfixing 5 years later without worrying about that forgotten legacy syntax of Sass, CoffeeScript, or jQuery.

Importing or Inlining CSS Files

We can write `@import` in modern CSS much like in SCSS, but CSS will keep distinct files.

The plugin postcss-import provides an easy way to inline all import rules into a single file.

As it is recommended to be run before any other PostCSS plugins, we can insert it before `postcssCustomProperties` in our `postcss.config.js`:

```
const postcssCustomProperties = require('postcss-custom-properties');
module.exports = {
  plugins: [
    require('postcss-import'),
    postcssCustomProperties({
      preserve: false,
    }),
  ]
}
```

CSS Modules

CSS modules are not to be confused with mixins, as they serve the opposite purpose. While mixins are components or functions to be reused globally, modules are style sheets with a local scope used in a similar way as styled components in React.

Using CSS modules is a strategy to ensure that they don't interfere with other CSS, so we could safely use modular projects without cryptic hashes as class names or even IDs. But that's what happens when using postcss-modules, or at least the module classes get prefixed with random hash.

This is not that much different — but less human-readable — than using prefixes matching a namespace assumed to be unique by convention, like an npm module or a WordPress theme or plugin.

Modules are not expected to become part of the official CSS specification in the near future, but I wanted to mention them here mostly to disambiguate modules and mixins.

Will there ever be a CSS Modules Module?

To add to our possible confusion, "module" has another meaning, as the World Wide Web Consortium (W3C) organizes language specifications and drafts on a modular level, so there are, for example,

- the CSS Nesting Module,
- the CSS Color Module Level 3,
- the CSS Backgrounds and Borders Module Level 4,

and there will probably also be a "CSS Modules Module" in the future.

Extends, Mixins, and User-Defined Custom Functions

We might think of mixins and SCSS functions as user-defined custom CSS functions functions. SCSS functions can take and process optional parameters and define complex output rules in their `@return` statements.

Composes are a more minimal way to avoid redundant code that exceeds the power of custom properties when it can't be fit into a single line of shorthand properties. Although they're not exactly upcoming native CSS without a working draft, there have been similar implementations like using classes from another CSS module. We can use postcss-composes although the plugin has not been updated for several years.

PostCSS Extend Rule (deprecated?)

Maybe closer to possible future standards (or maybe not, as I heard it became deprecated and the PostCSS plugin is "mostly unmaintained" since 2022), the PostCSS Extend Rule lets you use the `@extend` at-rule and functional selectors in CSS, following the speculative CSS Extend Rules Specification.

We have to tell stylelint about it:

```
"rules": {
"at-rule-no-unknown": [true,{"ignoreAtRules": ["extend"]}],
```

And we have to make sure that it runs before the nesting plugin, otherwise it will not work at all due to an issue when using both plugins together.

PostCSS' `@extend` implementation has several limitations, but the simple use case is straightforward.

```
.font-xl {
  font-size: 48px;
  line-height: 1.5;
}

.hero__heading--xl {
  @extend .font-xl;
}
```

Update May 2023: @extend seems to be deprecated, and I didn't manage to use it anymore. @mixin (postcss-mixins seems to be a good alternative, see my comment and code example below.

PostCSS Mixins

`@mixin` might seem a little unintuitive and overengineered at first sight, at least for my simple use case, but we can use it like this:

```
@define-mixin heading--h2 $className {
  $(className) {
    font-family: var(--font-family-heading);
    font-weight: 400;
    font-size: 2rem;
    line-height: 140%;
    font-style: normal;
    @mixin-content;
  }
}

@mixin heading--h2 .intro__keytext__headline {
  margin-bottom: 1rem; /* this is the @mixin-content */
}
```

I installed `npm install --save-dev postcss-mixins`,
added `postcss-mixins` as a PostCSS plugin in `postcss.config.js` between `postcss-import` and `postcss-custom-properties` and told stylelint about the new at-rules:

```
"at-rule-no-unknown": [true,{
  "ignoreAtRules": [
  "define-mixin",
  "mixin",
  "mixin-content"
]
```

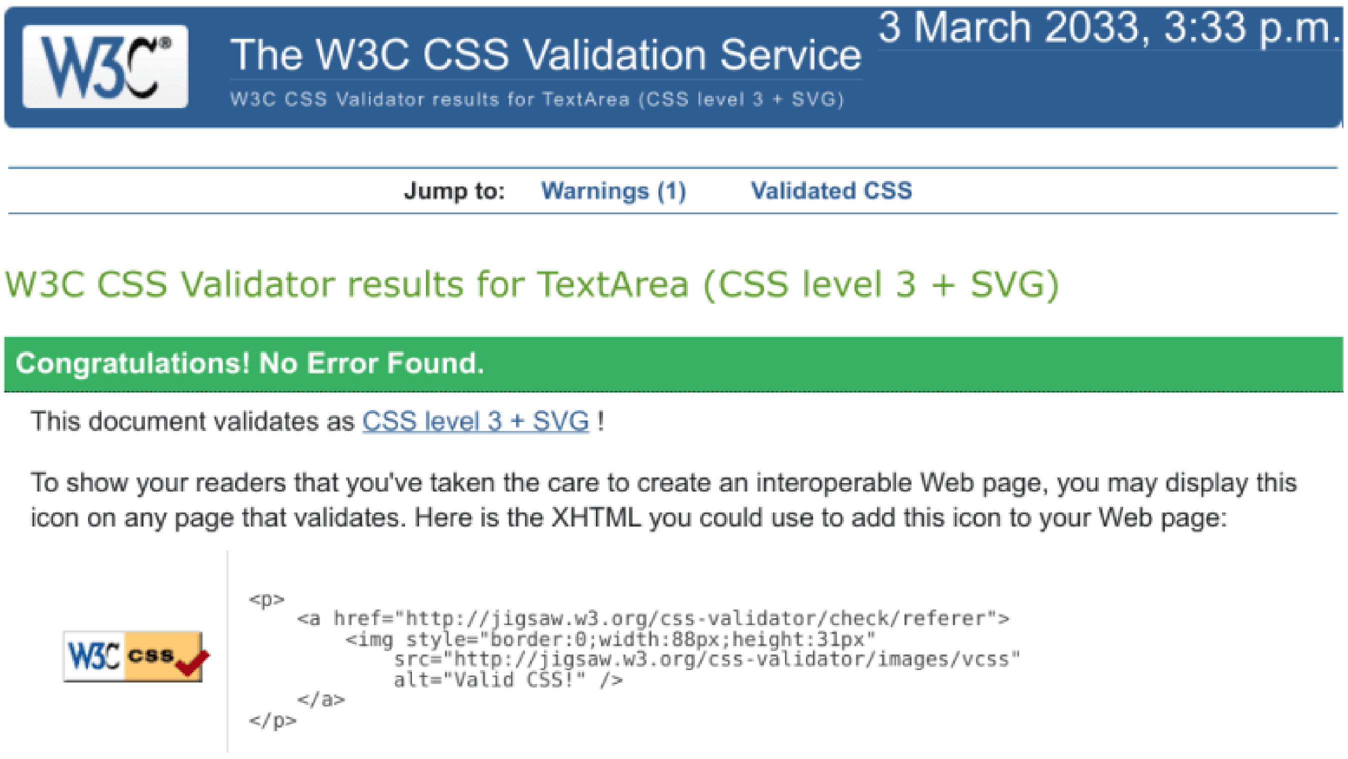
User Defined Functions

User defined functions might become part of CSS one day, but currently there isn't even a working draft. So I would contradict my claim at least in this aspect, when I told you to use the PostCSS define-function plugin which mimics a subset of SCSS's function syntax.

Another, completely different approach, would be using JavaScript functions with the postcss-functions plugin.

Future-Proof CSS causing Breaking Changes in SCSS?

When choosing CSS custom properties (`--` and `var(--`) over the (slightly more compact, consistent, and elegant) `$` made our code more valid and future-proof, that was due to the fact that we didn't actually use future CSS yet. Custom properties are approved, shipped, and supported by every major mainstream browser, so we can be 99% sure the same syntax will still be valid and supported 10 years later.


3 March 2033, 3:33 p.m.

The W3C CSS Validation Service

W3C CSS Validator results for TextArea (CSS level 3 + SVG)

Jump to: Warnings (1) Validated CSS

W3C CSS Validator results for TextArea (CSS level 3 + SVG)

Congratulations! No Error Found.

This document validates as CSS level 3 + SVG !

To show your readers that you've taken the care to create an interoperable Web page, you may display this icon on any page that validates. Here is the XHTML you could use to add this icon to your Web page:

```
<p>
    <a href="http://jigsaw.w3.org/css-validator/check/referer">
        <img style="border:0;width:88px;height:31px"
            src="http://jigsaw.w3.org/css-validator/images/vcss"
            alt="Valid CSS!" />
    </a>
</p>
```

SCSS vs. CSS Nesting

Depending on our coding preference, we might even (try to) write nesting code that is valid SCSS and future CSS at the same time, simply by never omitting a seemingly optional `&` (ampersand) sign before nested declaration. Is it naive to think that's a sustainable solution?

```
.hero {
  ::before {} // valid only in SCSS
  &::before {} // valid SCSS and possibly valid future CSS
```

We might worry and argue that native nesting (CSS Nesting Level 1) is still a draft and that it behaves differently from SCSS.

There have been proposals to native CSS nesting that deviate beyond the "safe" SCSS syntax as shown above, requiring additional curly braces or a `@nest` keyword, none of which have made it into the latest CSS nesting working draft in spring 2023.

If we use (and don't overuse) nesting in a conservative way, I wouldn't expect critical problems when maintaining my own projects, but as I hinted before, that would be too idealistic.

Upcoming (Breaking) Changes to the SCSS Nesting Syntax

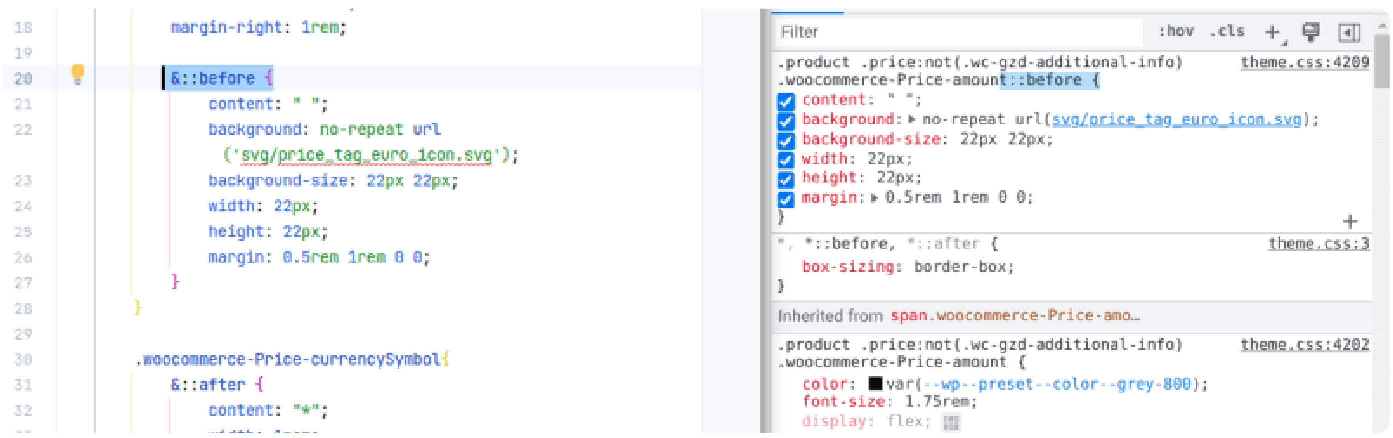
Yesterday, Sass wrote about the issues and incompatibilities and possible plans to support native CSS syntax nesting in the future, indicating unintended problems of native nesting due to its current implementation using an implicit `:is` wrapper:

> This changes the specificity: :is() always has the specificity of its most specific selector. [...]
> In the long term, once :is() is supported by 98% of the global browser market share, we'll start transitioning Sass to emit :is() when resolving Sass nesting. This will make Sass behave like CSS in the first two behavioral incompatibilities. We will consider this a breaking change, and release it as part of a major version release to avoid unexpectedly breaking existing stylesheets. We'll do our best to make this transition as smooth as possible using the Sass Migrator.
>
> We will not drop our current behavior for &-suffix unless we can come up with a comparably ergonomic way to represent it that's more compatible with CSS. This behavior is too important to existing Sass users, and the benefit of the plain CSS version is not strong enough to override that.

Source: https://sass-lang.com/blog/sass-and-native-nesting

Nesting can make CSS Harder to Debug

But I have been quite reluctant to adopt SCSS's nesting feature in general anyway.



```
    margin-right: 1rem;

    &::before {
        content: " ";
        background: no-repeat url
          ('svg/price_tag_euro_1con.svg');
        background-size: 22px 22px;
        width: 22px;
        height: 22px;
        margin: 0.5rem 1rem 0 0;
    }
}

.woocommerce-Price-currencySymbol{
    &::after {
        content: "*";
```

```
.product .price:not(.wc-gzd-additional-info)              theme.css:4209
.woocommerce-Price-amount::before {
  content: " ";
  background: ▸ no-repeat url(svg/price_tag_euro_icon.svg);
  background-size: 22px 22px;
  width: 22px;
  height: 22px;
  margin: ▸ 0.5rem 1rem 0 0;
}
*, *::before, *::after {                                     theme.css:3
  box-sizing: border-box;
}
Inherited from span.woocommerce-Price-amo…
.product .price:not(.wc-gzd-additional-info)              theme.css:4202
.woocommerce-Price-amount {
  color: ■var(--wp--preset--color--grey-800);
  font-size: 1.75rem;
  display: flex;
```

One reason against nested code is that it gets even harder to find the source code for an element that you inspect in your browser's developer tools, although source maps should solve this problem.

Early Adopters risk Technical Debt

If we are unlucky, we will swap one non-standard CSS syntax (SCSS) for another (preliminary future CSS draft about to change). That's our risk as early adopters and it's not that hard to refactor our code if we have to. Maybe we don't, as long as our PostCSS module outputs valid CSS to our clients.

Support Nesting in our CSS Code

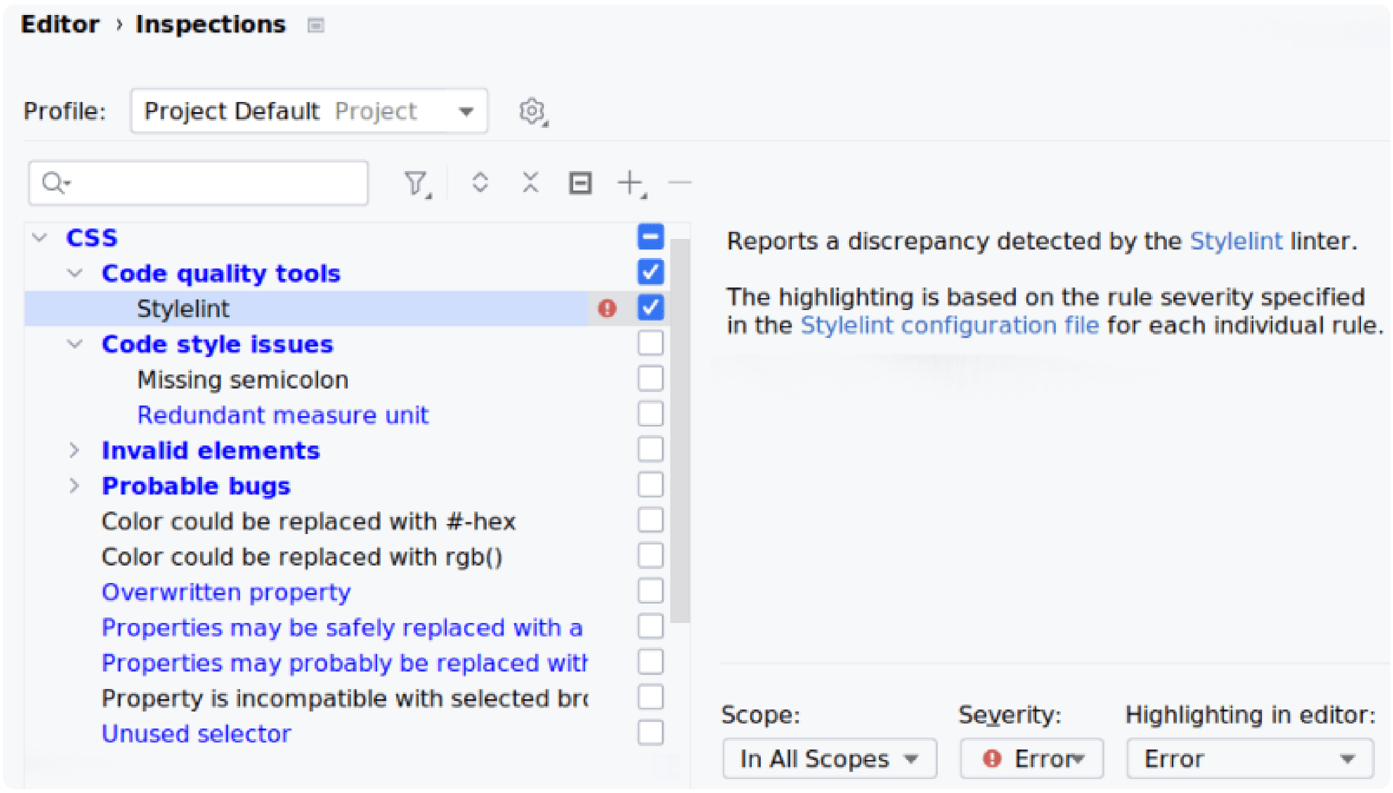
As we don't want to ship experimental future code to our clients, we need to configure PostCSS to convert nested code back to valid CSS.

Also we have to configure stylelint by adding stylelint-use-nesting and extend our configuration.

```
{
  "plugins": [
    "stylelint-use-nesting"
  ],
  "rules": {
    "csstools/use-nesting": "always" || "ignore"
  }
}
```

The `use-nesting` rule has two primary options: "always" means that we must use nesting wherever possible, so I prefer "ignore" which simply does not complain about nesting syntax, but without enforcing it.

If we still see errors and warnings in our editor, they might come from additional tools, like the built-in WebStorm inspections. Like SonarLint, we might just turn them off completely, and leave linting and annotating to our tools (stylelint and eslint) with their explicit project configuration that should work for everyone no matter which tool or platform they prefer.



Editor › Inspections

Profile: Project Default Project

- CSS ☐
  - Code quality tools ✓
    - Stylelint ✓
  - Code style issues ☐
    - Missing semicolon ☐
    - Redundant measure unit ☐
  - Invalid elements ☐
  - Probable bugs ☐
  - Color could be replaced with #-hex ☐
  - Color could be replaced with rgb() ☐
  - Overwritten property ☐
  - Properties may be safely replaced with a ☐
  - Properties may probably be replaced witl ☐
  - Property is incompatible with selected brc ☐
  - Unused selector ☐

Reports a discrepancy detected by the Stylelint linter.

The highlighting is based on the rule severity specified in the Stylelint configuration file for each individual rule.

Scope: In All Scopes
Severity: Error
Highlighting in editor: Error

Untranspilable CSS Features

All of the new "future" features until this very point have been nothing but syntactic sugar to improve our codebase's developer experience.

It does not matter if a feature is supported by all client browsers, as long as there is a way to generate CSS code that every popular device should understand at least in a basic way.

But there are some modern CSS features that can't be safely used, no matter with or without SCSS or PostCSS.

Container queries and parent selectors have been two breakthrough achievements of CSS in 2022. There is a cq-prolyfill for container queries, but it seems that there is no simple polyfill or transpilation code that can be generated automatically.

So it's up to us as developers to provide fallback code ourselves or if we only use it for progressive enhancement so that our site does not break on older devices.

Conclusion and Code Example

We can replace most SCSS by native CSS and an appropriate PostCSS and stylelint configuration, but early adoption of CSS drafts like native nesting might introduce technical debt into our project.

Here is a minimal configuration focused on the main features: custom properties, extends, import, and nesting.

Node modules:

```
npm install --save-dev
```

- `postcss`
- `postcss-cli`
- `postcss-custom-properties`
- `postcss-extend-rule`
- `postcss-import`
- `postcss-nesting`
- `stylelint`
- `stylelint-config-standard`
- `stylelint-use-nesting`

`postcss.config.js`:

```
const postcssCustomProperties = require(
  'postcss-custom-properties'
);
module.exports = {
  plugins: [
    require('postcss-import'),
    postcssCustomProperties({
      preserve: false,
    }),
    require('postcss-extend-rule'),
    require('postcss-nesting'),
  ],
}
```

`.stylelintrc`:

```
{
  "extends": "stylelint-config-standard",
    "plugins": [
      "stylelint-use-nesting"
    ],
    "rules": {
    "at-rule-no-unknown": [true,{
      "ignoreAtRules": ["extend"]
    }],
    "csstools/use-nesting": "ignore"
  }
}
```

What's next in CSS? (9 Part Series)

Top comments (6)

⍼Wii 🏳️‍⚧️ • Mar 30

Yet another CSS article that totally misses `@scope` ... I wish people would be more hyped about it, because it really deserves more attention.

Ingo Steinke • Mar 31

Scoping sounds promising. Unlike CSS modules, this doesn't work on a file basis, and it even allows nesting. But, unlike parent selectors or an `nth-of-class` / `nth-of-type` that's no feature that I have been missing so far. But that's also true for container queries and nesting. So I am open for a positive surprise...

⍼Wii 🏳️‍⚧️ • Mar 31

Ah, you're linking to an old spec there; that one had been introduced years ago, and dropped because at the time there was no real interest in it, and the way it was specified wasn't as flexible as it is now.

The current one can be found in *CSS Cascading and Inheritance Level 6* and looks a lot different (and is a lot more versatile, due mainly to the facts that it now supports lower boundaries and is an at-rule instead of an HTML attribute).

The spec also adds the concept of scope proximity, so when two scoped selectors of the same specificity conflict on one element, the one where the upper boundary of the scope is closest to the styled element will take precedence. There was a discussion about making scope proximity stronger than selector specificity, but it was ultimately settled on weak proximity as a default.

Ingo Steinke • Mar 31

Thanks for mentioning, I will surely have a look.

Ingo Steinke • May 11 • Edited

To extend my experience with `@extend`, well I didn't manage to make it run at all. I tred `@mixin` instead, which I found unintuitive and overengineered at first sight, at least for my simple use case, at least it works!

If we just want to reuse some code, we might do it like this:

```
@define-mixin heading--h2 $className {
  $(className) {
    font-family: var(--font-family-heading);
    font-weight: 400;
    font-size: 2rem;
    line-height: 140%;
    font-style: normal;
    @mixin-content;
  }
}

@mixin heading--h2 .intro__keytext__headline {
  margin-bottom: 1rem; /* this is the @mixin-content */
}
```

I installed `npm install --save-dev postcss-mixins`, added `postcss-mixins` as a PostCSS plugin in `postcss.config.js` between `postcss-import` and `postcss-custom-properties` and told stylelint about the new at-rules:

```
"at-rule-no-unknown": [true,{
  "ignoreAtRules": [
    "define-mixin",
    "mixin",
    "mixin-content"
  ]
```

What I didn't manage to achieve (yet) is make PhpStorm (2023.1) recognize my `stylelintrc.json` (although I added the `.json` suffix so it matches their list of default configuration filenames, so I have to live with some false positive warnings in my editor (or prefix every such line with `/*noinspection ALL */` but even that would not silence "unexpected token" warnings when using `$(`. Another reason to put all mixins in one file so we could disable IDE inspections for that single one, like listing the filename in an `.ignore` file or using other custom options. ¯\_(ツ)_/¯

TL;DR

postcss-mixins worked for me, but postcss-extend-rule didn't!

Ingo Steinke • Mar 31 • Edited

Nesting syntax still evolves! CSS developers may have found a way to "relax the syntax to be more Sass-like (i.e. so that the `&` is never required)" as Lea Verou tweeted:

> **Lea Verou**
> @leaverou
>
> For those following the development of the CSS Nesting syntax, amazing news today from the Chrome team: It looks like we may actually be able to relax the syntax to be more Sass-like (i.e. so that the & is never required)! What a great start to the day!
> github.com/w3c/csswg-draf...
>
> 13:41 PM - 30 Mar 2023