Drupal.org

SASS Audit on 1x branch

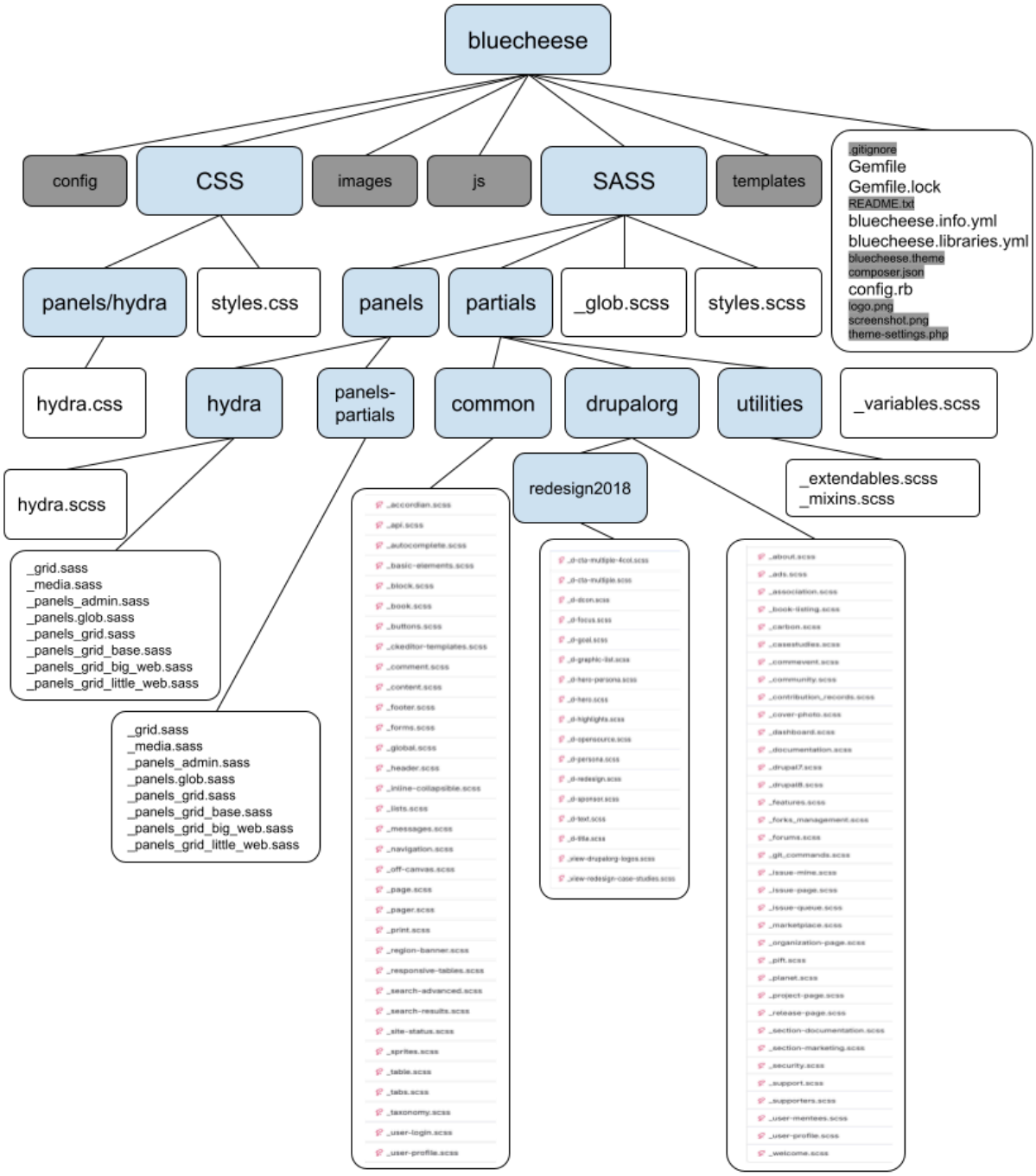
Presented By

Palantir.net, Inc.
www.palantir.net

Table of Contents

Table of Contents
Folder Structure
    File Breakdowns
        CSS
        SASS
Sass Features in Use
    Browser Support
    Features
    Is Everything Portable?
        Features
            Extend
            Mixins
            Variables
            Glob
        Browsers
        Compile Time
        Compiling v Compressing v Minifying - What's the difference?
Devil's Advocate
Key Recommendations Moving Forward
Conclusion

Folder Structure



Key:
Folder
File
Inconsequential to this audit

File Breakdowns

Based on the file structure, below are a few key files and their functionality. Panels are a predecessor to Layout Builder, and the latter is the recommended tool for Drupal site versions 8+. Panels will not be used in the new iteration of the theme.

CSS

File	Purpose
bluecheese.libraries.yml	Identifies location of primary CSS file (/css/styles.css)
/css/styles.css	Lays out styling for the site
/css/panels/hydra/hydra.css	Lays out styling for the hydra panel

SASS

File	Purpose
Gemfile	Defines the packages (Gems) needed from Ruby (package manager)
Gemfile.lock	Full breakdown of packages and dependencies
/sass/_glob.scss	Import list for external libraries; import variables, abstractions, base styles, and components for globbing
/sass/styles.scss	Compass imports; style imports for partials folder
/sass/panels/hydra/hydra.scss	Lays out styling for the hydra panel
/sass/panels/panels-partials/*	Breakout styles to pull into panels
/sass/partials/_variables.scss	Defines all variables used in the Sass
/sass/partials/common/*	Styling of specific elements of the site
/sass/partials/drupalorg/*	Styling for select pages on the drupal.org site
/sass/partials/utilities/*	All extendables and mixins

Sass Features in Use

Browser Support

Browser support takes into consideration the requirements below from drupal.org docs:

- The latest release of each of the latest two supported major versions of:
    - Desktop browsers:
        - Google Chrome
        - Firefox
        - Safari
        - Microsoft Edge

        - Opera
    - Mobile browsers:
        - Safari for iOS
- The latest supported release of the latest major version of:
    - Desktop browsers:
        - Firefox ESR
    - Mobile browsers:
        - Chrome for Android
        - Chrome for iOS
        - Opera Mini (except for 'extreme data savings' mode)
        - Samsung Internet

Features

Sass Feature	CSS3 Equivalent	Browser Support
Extend	Re-use class selectors	OK
Mixins/Includes	Rewrite with variables and classes Native mixins (still conceptual only)	OK, kinda Not yet
Variables	CSS Variables (Custom Properties)	No Opera Mini, but Opera Mobile OK
Import/Partials	CSS at-rule: @import	OK
Nesting	Already built in/unnecessary CSS Nesting	OK No Opera Mini, but Opera Mobile OK
Glob	Several options, inc PHP glob	OK
Grid	CSS Grid Dynamic viewport units	OK or unknown No Opera Mini, but Opera Mobile OK/unknown

Is Everything Portable?

Based on the research done in the table above, not all Sass features have a one-to-one equivalent in CSS yet (not with full browser compatibility, anyway). That said, CSS has native workarounds for many of these as well as newer options with widespread browser compatibility.

Features

The tricky features that don't seem to have a solid equivalent are Extend, Mixins, Variables, and Glob.

Extend

The Sass site itself shows how using CSS selectors can accomplish the same thing that Sass @extend functionality accomplishes. The limitations listed on the site reference cluttered HTML,

user error, and “non-semantic style concerns,” but they also reference the BEM methodology, which I expand upon below, as a possible solution.

SCSS Sass

```scss
.error {
  border: 1px #f00;
  background-color: #fdd;

  &--serious {
    @extend .error;
    border-width: 3px;
  }
}
```

CSS

```css
.error, .error--serious {
  border: 1px #f00;
  background-color: #fdd;
}
.error--serious {
  border-width: 3px;
}
```

Mixins

One possible way to get around the Mixins (aside from the methods listed above) would be the ::part() pseudo-selector, which has unknown compatibility with Opera Mini. This could be combined with the BEM syntax (expanded on below), or be a tradeoff.

Variables

Another option for writing variables might work via the BEM convention, which breaks classes into Blocks, Elements, and Modifiers, similar to how an Object is broken down in OOP. BEM is not a library and doesn’t involve any dependency, rather it is a guide for reducing CSS and maintaining reusable and readable CSS.

Glob

Although not a 1-for-1 tradeoff, the CSS @import statement already allows users to import styles from other stylesheets.

Browsers

Opera Mini seems to be the holdout on most features, but Opera Mobile is compatible with every feature lacking Opera Mini support. Opera Mini’s global usage statistic seemed low based on caniuse data, but Opera has been expanding their global support in recent years, with campaigns like the Africa First strategy garnering millions of users in Kenya and more than a billion users worldwide.

One option that would allow us to both have CSS variables and stay compatible with Opera Mini would be to create a shim or polyfill. This would increase the workload to transition to CSS, as developers would need to learn potentially unfamiliar technologies to produce this code, but it would either create or correct the API to work around the incompatible code. Unless we decide this should be looked into further, I will assume the setup time required overrides any benefit.

Compile Time

Compile time is tricky, as we cannot pin down an exact metric, but we know that compile time will grow as the code grows. As Eren mentions in their article about SASS vs CSS,

> *As someone who has been using SASS for years, I believe that as of 2024, the benefits of SASS, including the hassles of installation, usage, and compilation, no*

> *longer justify its use. Particularly as projects grow, the compilation times become significantly burdensome.*

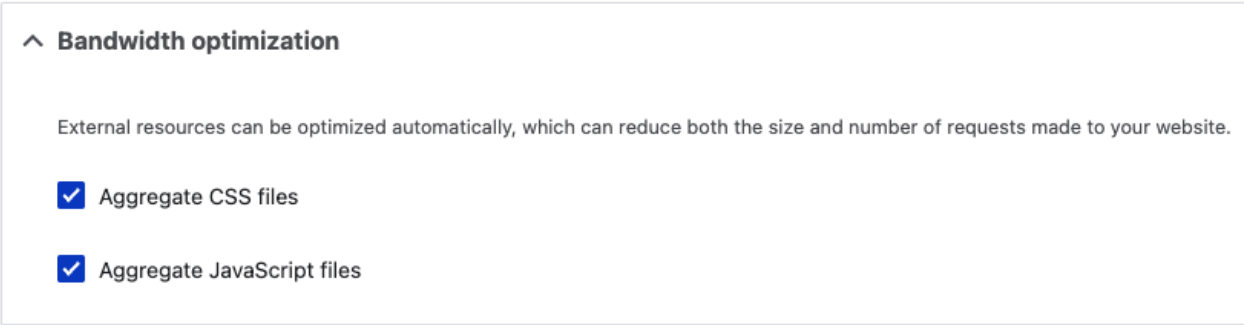
This isn't to say CSS doesn't also have build times that can affect performance. On the contrary, as Odili Charles Opute mentions in their article about minifiers, without any additional compression or minification tools, "CSS is a render-blocking resource on the web." As the linked article in Opute's quote mentions, slowing down web performance by not compressing CSS can affect SEO, reducing site traffic. Therefore using a tool to either compress or minify the CSS will be important. Some free recommendations from Opute's article include:

- Minify
- CSS Minifier
- CSS Minify
- Dan's Tools CSS Minify

Compiling v Compressing v Minifying - What's the difference?

- **Minification** in CSS is the process of removing all unnecessary whitespace so that files are smaller in size, but can still be understood by the browser. (from Salma Alam-Naylor)
- **Compression** makes the file smaller through the use of a compression algorithm (such as gzip) and does not actually alter the file's contents. (from cloudflare)
- Sass works in such a way that when you write your styles in a .scss file, it gets **compiled** into a regular CSS file. The CSS code is then loaded into the browser. (from Israel Mitolu)

Choosing either minification or compression for the CSS would be a great move, but notice how the SASS needs to be converted to CSS before it can be processed in the browser - that alone would be a huge reason for sticking to CSS. Think of the time loss as the project grows! Fortunately the amazing power of Drupal handles aggregating the CSS and JS files for us. By checking the aggregation boxes at *Admin > Configuration > Development > Performance*, Drupal will automatically combine multiple CSS files into a single file.



This is, unfortunately, the extent of the actions until the Advanced Aggregation module is updated for 10.1+ (or another module steps in to serve as a minifier/compressor), but this is still helpful in reducing file size. We can also incorporate some Front-End Developer tools to assist development workflow and keep checking back on incorporating a CSS minifier into core. As Nathaniel Catchpole wrote regarding Drupal 10 upgrades, "In Drupal 10.1, completely rewritten aggregation generation logic" resolves some of the issues previously experienced in aggregation and nearly eliminates the need for 3rd-party JavaScript minification.

Devil's Advocate

We've mostly targeted the benefits of CSS over SASS, but that's not to say it's the best choice. As Rob O'Leary talks about in their article about not following the trend from SASS to CSS, choosing the latter has major drawbacks and costs when switching from legacy code or an existing site. Although we will be working with an existing theme, we will not be simply updating the existing code; we have to revamp the styling entirely to reflect the new brand standards. With the overhaul required in this project, we can almost presume we are styling from scratch rather than simply refactoring existing SASS. Really the only counterpoint to making the switch that O'Leary suggests that doesn't pertain to changing existing code is single line comments. Since multiline comments are still a thing in CSS, this seems like more of a minor annoyance than an actual issue.

Key Recommendations Moving Forward

Based on the advent and iteration on things like CSS Modules, along with persistent emergence and widespread adoption of new CSS features, I believe that CSS3+ will allow users to harness capabilities for which we formerly relied on preprocessors to accomplish. Based on the incompatibility with variables and Opera Mini, however, the argument for transitioning at this time could fall short. Currently CSS variables have >97% browser support, which is indicative of future compatibility, we just can't know when it will get there.

All said, moving from Sass to Vanilla CSS *could* be a huge undertaking for the Drupal theme. If, however, we are creating a component library and building these components from the ground up, then including a stylesheet/BEM Block for each component wouldn't be unreasonable. We couldn't move the existing stylesheets, but we could use them for reference as we build out the components. Plus, moving them directly wouldn't have been possible anyway since the current bluecheese theme is not based on Single Directory Components and needs updating to new brand standards.

Although writing vanilla CSS can get messy and burdensome, following naming conventions and structure standards can keep styling sleek and readable. We as a society are on a tipping point of CSS overpowering its preprocessors, but every tipping point means the ball could roll back just as easily.

Conclusion

At this time, moving to a preprocessor-free CSS structure seems possible and safe to try, with the biggest hurdle being the learning curve for proper structuring and methodology.