PostCSS Deep Dive: Building a PostCSS workflow

For a while, SASS was all we needed. I was OK with it only being available as a Ruby Gem (the original implementation was written in Ruby), then I was happy when LibSASS came around (written in C) and might have even been OK with the Dart implementation becoming the reference implementation and the SASS team deprecating the Ruby and C implementations, eventually stopping work on them altogether.

But one thing that SASS has never done is give you the possibility of testing new CSS features like Babel does for Javascript.

I first head of PostCSS when Autoprefixer became a processor for PostCSS rather than a standalone tool and the tooling used PostCSS to run.

I've always wanted to move my worflow to PostCSS and now think it's time to do it. Other than the program-like functions (`@if`/`@else`, `for` and others) there is little that we can't do with a different preprocessor and, eventually, with CSS alone.

So in this post I will evaluate what it would take to move to PostCSS based on a given set of requirements.

Feature set

Here's the feature set I want to implement if possible

- A PostCSS equivalent to Babel's `preset-env`
- Color manipulation functions equivalent to SASS @lighten and @darken
- Nesting
    - Nested selectors
    - Relationship notation
- Variables
    - Houdini-style CSS Custom Properties using @property

There is also a nice to have set of features:

- function-like conditionals and iteraators, similar to what we can do in SASS
- Mixins

Implementation

The first thing to do is decide what features we want to implement.

I realized that, rather than implement every single feature from the start, it would be better to start with the most important ones and work my way down. Based on the feature lists that I want to implement the initial list of features looks like this:

Autoprefixer : Adds vendor prefixes to properties that require them based on a specified list of browsers and versions : The plugin uses browserslist to create and access the list, and caniuse to provide a list of browsers that support a given feature : Because `preset-env` also supports Autoprefixer, I don't know if I want to leave it as a separate package

postcss-nested : nested rules and at-rules. It provides a close aproximation to what SASS does with nested selectors and relationship notation

postcss-sorting : Sorts the rules inside selectors for you so you don't have to do it yourself manually

postcss-easy-import : Inline the contents of imported files reducing the number of request and potentially improving performance

Implementation: Gulp task V1

The first pass of the Gulp-based workflow is to implement the basic set of features listed in the previous section.

The code below assumes that all the plugins are installed.

The first step, as usual, is to set up the modules that we want to use. Gulp and sourcemaps are at the top and then we import the PostCSS plugins.

```
const gulp = require('gulp');
const sourcemaps = require('gulp-sourcemaps');
```

```
const postcss = require('gulp-postcss');
const autoprefixer = require('autoprefixer');
const importer = require('postcss-easy-import');
const nesting = require('postcss-nesting');
const simpleVars = require('postcss-simple-vars');
const sorting = require('postcss-sorting');
```

The next step is to create the list of processors we will use and any necessary configuration.

The importer plugin uses the `glob` attribute to indicate that we're allowing globs to be used in import statements.

The next plugin is Autoprefixer. We define the browserslist attribute in package.json to specify the browsers we want to support. Even though my primary usage is in CSS, I'm following the guidance from Publish, ship, and install modern JavaScript for faster applications so that Javascript tools that use Browserslist will work properly.

The next plugin is the sorting plugin. Here we have to be specific about the grouping of different rules, how we want the properties sorted and what to do with properties that are unspecified.

```
const processors = [
  importer({
    glob: true
  }),
  simpleVars,
  nesting,
  autoprefixer,
  sorting({
    order: [
      'custom-properties',
      'dollar-variables',
      'declarations',
      'at-rules',
      'rules',
```

```
    'custom-properties'order': 'alphabetical',
    'unspecified-properties-position': 'bottom',
  }),
];
```

That is the build of the work. The rest is just defining a Gulp task that uses PostCSS, the plugins we just specified and sourcemaps to build the production CSS.

```
gulp.task('css', () => {
  return gulp.src('src/css/main.css')
    .pipe(sourcemaps.init())
    .pipe(postcss(processors))
    .pipe(sourcemaps.write('./'))
    .pipe(gulp.dest('build/static/css'));
});
```

Incorporating into a Fractal design system

To run this task in a Fractal-based design system, we need to complete the following steps:

1. Make sure you have a Gulp-based build system, if not you need to se one up
2. Copy the code we discussed earlier and modify it to suite your needs
3. Install the required plugins

Once these steps are completed you can run `gulp -T` to get a list of the tasks. Gulp parses the `gulpfile.js` file before rendering the list of tasks so it will report if there are any syntax errors.

After testing for syntax, you can run `gulp css` to build the CSS. This is where the PostCSS plugin erros are likely to happen. Repeat as necessary until there are no errors and the task completes.

Conclusions

Now that we have working code we can look at the next steps and get fancier with what we do with PostCSS.

The next post will look at these additional features, how necessary they are and how to incorporate them into the existing PostCSS Workflow