PostCSS deep dive

PostCSS is an interesting tool and ecosystem. It is a CSS processor written in Javascript.

What first brought my attention to PostCSS is Autoprefixer, my go to tool for automating adding prefixes to my CSS. A few years ago they moved from a separate project into a PostCSS plugin.

However, it wasn't until recently that I started looking at the original purpose of PostCSS, to be able to write future CSS for today's browsers.

The goals of the research seeks to answer the following questions:

- Can we replace SCSS with CSS using PostCSS plugins?
- How can we leverage new CSS features in our existing stylesheets?
- Is it worth using Javascript to do this?

For this experiment I will use the following plugins:

- css-preset-env
- Autoprefixer
- CSS Nano

I will also use Gulp as my build system. It's what I'm already using and I won't consider any solution that doesn't allow me to leverage it.

Getting things started

The first step is to configure the repository, if you haven't done so already, using these commands:

```
git init #1
npx license $(npm get i$(npm get init.license)npm get "$(npm get init.auth
npx gitignore node #3
npm init -y #4
```

1. Initializes an empty Git repository
2. Uses the license NPM package through NPX to create an MIT LICENSE file

(MIT is the default license for my projects)
3. Uses the gitignore NPM package to create a Node-based exclusion file
4. Creates an empty package.json file accepting all default values

Installing the packages

Once we have configured the project we can install the packages we will use:

```
npm install --save gulp \
  postcss \
  autoprefixer \
  postcss-preset-env \
  cssnano
```

If you are integrating this experiment into an existing project then whatever packages are already installed they will be updated.

Once we install the packages we need to create a `gulpfile.js` file where we'll write the tasks to test the plugins.

The Gulpfile

We first require Gulp, gulp-postcss (to make Gulp and PostCSS work together), and the PostCSS plugins that we want to use.

This is a very small sample of the PostCSS plugins available.

```
const gulp = require('gulp');
const postcss = require('gulp-postcss');
const postcssPresetEnv = require('postcss-preset-env');
const sorting = require('postcss-sorting');
const autoprefixer = require('autoprefixer');
const nano = require('cssnano');
```

The following sections represent different portions of a single Gulp task. It has been broken down for easy of explaining.

First we run the code through postcss-preset-env, a CSS version of Babel, to convert the CSS we enter into CSS that is usable by our target browsers.

It uses the lst of features identified at stage 3 from cssdb and additionally it will transform nested rules.

It is tempting to use stage 2 features from the database but until a feature reaches stage 3 in the database it is still subject to change and these changes can be substantial.

```
gulp.task('css', () => {
  return gulp.src('./src/*.css').'./src/*.css'tcss([
      postcssPresetEnv({
        stage: 3,
        features: {
          'nesting-rules': true
        }
      }),
```

Once we have CSS that will work, we run it through postcss-sorting to group content inside rules according to a set criteria.

The plugins:

- Sorts rules and at-rules content.
- Sorts properties.
- Sorts at-rules by different options.
- Groups properties, custom properties, nested rules, nested at-rules.

I do this rather than sort them alphabetically because it's easier to reason what selectors do when I can see all the rules for specific functionality grouped together.

```
      sorting({
        'order': [
          'custom-properties',
          'at-rules',
          'declarations',
          'rules'
```

```
        ],
        "properties-order": [
          "font",
          "color",
          "margin",
          "padding",
          "border",
          "background"
        ],
        'unspecified-properties-position': 'bottom'
      }),
```

The final steps are to run the stylesheet through Autoprefixer and CSS Nano

Autoprefixer saves us from having to manually enter prefixes for the rules that needs it, based on the browsers we support.

CSS Nano is a CSS minimizer. It uses multiple strategies to make the resulting stylesheets as small as possible.

```
      autoprefixer(/* no options needed*/),
      nano(/* no options needed*/)
    ])
  )
```

It does an awesome job but it may go too far. I'll explain why I think so in the evaluation at the end of the post.

The final step in the task is to pipe the stream to its final destination. Now we're done with the styling.

```
  .pipe(
    gulp.dest('./dist/css')
  )
});
```

Modifying `package.json`

PostCSS and its ecosystem rely on browserslist to provide a list of supported browsers.

The best way to do it is to integrate it on your project's `package.json` file. To do so add the following block to your `package.json` file.

```
"browserslist": [
  "last 3 versions",
  "> 0.2%",
  "not IE 11",
  "not IE_Mob 11",
  "not dead"
],
```

Doublechecking your work

To make sure that your list of supported browsers works as intended you can use tools like browserl.ist to validate that it works and that it returns the browsers you want.

In theory, running `npx browserslist` would also validate the `browserslist` section ofoo package.json but I've been unable to make it work.

Evaluation: Yes, no, maybe?

PostCSS eliminates the need for node-sass and the dependency on specific versions of Node for specific versions of SASS.

However there are way too many tools that do essentially the same thing differently based on the creator's preference. That gets confusing as to what plugin to use to accomplish a goal.

Of the plugins I selected to test they all work fine with some exceptions where, in my opinion, the tool is too good for what it tries to do.

If I expand a minimized stylesheet I would like to know that the original used a

`calc` value instead of a fixed one.

For example, the following code:

```
.box {
    width: calc(2 * 100px);
}
```

Gets transformed into the code below, accoording to the CSS Nano documentation.

```
.box {
    width: 200px;
}
```

It is impossible to tell what code resulted on the code shown in the docs. This doesn't matter if you have the source stylesheets but it's important if you're trying to reason through someone's code (and I still think this is an important thing to do).

As long as we're ok with that kind of aggressive minimization then the tool doooes what it's meant to and we're good to go.

If we're not OK with those minimizations we need to ofigure out how to remove them or find another minimizer that we can configure to our liking.