PostCSS

Okay, so what is PostCSS?

From the official website:

“PostCSS is a tool for transforming CSS with JS plugins. These plugins can support variables and mixins, transpile future CSS syntax, inline images, and more.”

TL;DR

It's effectively like Babel but for CSS.

There are effectively a few steps that it goes through…

1. Parse the CSS file and turn it into an abstract syntax tree (AST),
2. maybe do stuff,
3. turn it back into CSS.

Sounds exciting right?

Getting it set up with like Gulp or something is pretty straight-forward.

```
const gulp = require('gulp');
const postcss = require('gulp-postcss');
const autoprefixer = require('autoprefixer');

gulp.task('css', () => {
  return gulp.src('src/style.css')
    .pipe(postcss()) // This is where the 🎩 happens.
    .pipe(gulp.dest('dest/style.css'));
});
```

Here is what it might look like in Webpack…

```
module.exports = {
  module: {
    loaders: [
      {
        test:   /\.css$/,
        loader: "style-loader!css-loader!postcss-loader"
      }
    ]
  }
}
```

Or, if you wanted to do it programmatically…

```
const postcss = require('postcss');

const mySuperAwesomeProcessor = postcss()

const output = mySuperAwesomeProcessor.process(someCSS).css;
```

So, let's take a look at what happens if we put in the following CSS:

```
body {
  background-color: rebeccapurple;
}
```

PostCSS would spit out:

```
body {
  background-color: rebeccapurple;
}
```

Pretty epic, right?

```css
body {
  background-color: rebeccapurple;
}
```

1. Parse the CSS file and turn it into an abstract syntax tree (AST),
2. maybe do stuff,
3. turn it back into CSS.

Somewhere along the middle, you actually get something along the lines of this:

```
{
  "type": "rule",
  "nodes": [
    /* We'll take a look at this in second… */
  ],
  "source": {
    "start": {
      "line": 1,
      "column": 1
    },
    "input": {
      "css": "body {\n  background-color: rebeccapurple;\n}",
      "id": "<input css 8>"
    },
    "end": {
      "line": 3,
      "column": 1
    }
  },
  "selector": "body"
}
```

Inside of that nodes property:

```json
{
  "type": "decl",
  "source": {
    "start": {
      "line": 2,
      "column": 3
    },
    "input": {
      "css": "body {\n  background-color: rebeccapurple;\n}",
    },
    "end": {
      "line": 2,
      "column": 34
    }
  },
  "prop": "background-color",
  "value": "rebeccapurple"
}
```

- Each CSS rule is a branch on the tree
- Knows about what selector the rule is for and it's exact location.
- Each declaration is a child of that node.
- It knows the properties and values and most of the good stuff above.

Right now, we're not doing anything with that AST. I'm going to round back on how to work with and manipulate the tree, but let's take about this at a higher level for a bit.

High-Level Stuff

- You can use PostCSS with vanilla CSS or Sass.
- PostCSS has a modular, “use what you need” nature.
- It looks at your CSS and just processes it. There is no need to write special functions or anything like that.
- It's super fast.

Why should I use this?

If you're using something like `autoprefixer`, then you already are using it.

Immediate implementation: if you want a new functionality, you don’t have to wait for Sass to be updated; you can make it on your own for find a plugin.

Let's take a look at some plugins in the wild

- Official repository
- cssnext
- postcss.parts
- Rucksack

LINTING

One of the big problems of working with CSS is that it doesn’t throw errors. The browser just does the best it can to make everything work.

It would be nice to have some tools that make sure we we’re writing the best possible code.

(Some examples)

Uses for us at SendGrid (Part I)

- As a support for Sass and other CSS that we’re writing
- Linting our CSS to make sure we’re conforming with best practices
- Refactoring existing code and shifting it towards new standards

Uses for us at Sendgrid (Part II)

- MCAM: Make it easier when users change stuff in the Design Mode.
- MCAM: Programmatically work with code so that their emails either look great or at least don’t conflict with our UI.
- (A lot of these are approaches that we can take with HTML as well.)

A real world example...

```
(css, selector = '.scoped') => {
  const ast = postcss.parse(css);

  ast.walkRules((rule) => {
    rule.selector = `${selector} ${rule.selector}`;
  });

  return ast.toString();
};
```

ASTs aren't just for CSS

As I mention earlier, they're what power Babel to transform ES6 (and beyond) code to ES5 syntax. (Not to mention JSX.)

There are also libraries that can parse HTML in a similar way.

Some other interesting uses of ASTs:

- inline-css
- Markdown HTML AST
- htmlparser2