PostCSS is an interesting project. In a nutshell, It takes CSS and turns it into an Abstract Syntax Tree, a form of data that JavaScript can manipulate. JavaScript-based plugins for PostCSS then performs different code manipulations. **PostCSS itself doesn't change your CSS**, it allows plugins to perform transformations they've been designed to make.

There are essentially no limitations on the kind of manipulation PostCSS plugins can apply to CSS. If you can think of it, you can probably write a PostCSS plugin to make it happen.

It's important to also know what PostCSS is not. This material is adapted from Envato-tuts+ PostCSS Deep Dive: What You Need to Know

PostCSS is Not a Pre-processor

Yes, you can absolutely use it as a preprocessor, but you can also use PostCSS without any preprocessor functionality. I only use Autoprefixer and, some times, CSS Nano. Neither of these tools is a pre-processor.

PostCSS is Not a Post-processor

Post processing is typically seen as taking a finished stylesheet comprising valid/standard CSS syntax and processing it, to do things like adding vendor prefixes. However PostCSS can do more than just post process the file; it's just limited by the plugins you use and create.

PostCSS is Not "Future Syntax"

There are some excellent and very well known PostCSS plugins which allow you to write in future syntax, i.e. using CSS that will be available in the future but is not yet widely supported. However PostCSS is not inherently about supporting future syntax.

Using future syntax is your choice and not a requirement. Because I come from SCSS I do all weird development there and use PostCSS in a much more limited capability. If I so choose I can turn to PostCSS and use future looking features without being afraid that my target browsers will not support it.

PostCSS is Not a Clean Up / Optimization Tool

The success of the Autoprefixer plugin has lead to the common perception of PostCSS as something you run on your completed CSS to clean it up and optimize

it for speed and cross browser compatibility.

Yes, there are many fantastic plugins that offer great clean up and optimization processes, but these are just a few of the available plugins.

Why I picked PostCSS and what we'll do with it

I initially decided not to use PostCSS until I discovered that Autoprefixer and CSSNano, some of my favorite tools, are actually PostCSS plugins. That made me research PostCSS itself and see what it's all about. What I found out is a basic tool and a rich plugin ecosystem that can do a lot of the things you may want to do with your CSS from adding vendor prefixes based on what you expect your users to have to analyzing your code for compliance with a given methodology like BEM.

I also like how PostCSS advocates for the single responsibility principle as outlined by Martin:

> The single responsibility principle is a computer programming principle that states that every module or class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.
>
> Wikipedia Entry: Single responsibility principle

Basically each PostCSS plugin should handle only one task and do it well. We should not create classes that do more than one thing and we shouldn't duplicate functionality that is already available through another PostCSS plugin.

In this post we'll explore how to build a PostCSS workflow using Gulp, how to build a plugin and how would you add plugins to the PostCSS workflow we created.

Running PostCSS

I work primarily in a Gulp environment so I built this task to work with PostCSS plugins and Autoprefixer in particular. Assuming you haven't done before, install Gulp globally

```
npm i -g gulp
```

And then install the plugins we need: `gulp`, `postcss` and `autoprefixer` into the project you're working it. the `-D` flag will save the plugins as development dependency.

```
npm i -D gulp gulp-postcss autoprefixer
```

The task itself is currently made of two parts:

- The list of processors to use
- The task itself

The task pipes the input through sourcemaps, then it runs postCSS and Autoprefixer that has already been configured with what versions of browsers to prefix for. It then writes the sourcemap and the output to the destination directory.

```
gulp.task("processCSS", () => {
  // What processors/plugins to use with PostCSS
  const PROCESSORS = [
    autoprefixer({browsers: ['last 3 versions']})
  ];
  return gulp
    .src("src/css'last 3 versions'/**/*.css")
    .pipe($.sourcemaps.init())
    .pipe(postcss(PROCESSORS))
    .pipe($.sourcemaps.write("."))
    .pipe(gulp.dest("src/css"))
    .pipe($.size({
      pretty: true,
      title:
        "processCSS"
    }));
});
```

If we run this task last in our CSS handling process we can change the destination

from `src/css` to `dest/css` but the process where this task was first used there was an additional compression process beyond what SASS gave me; I wasn't using CSSNano so I had to keep the files in the source directory to do further processing. We'll revisit this when we discuss other plugins we can use.

Adding a second plugin

Even through the CSS for this task is compressed using SASS compressed format. I want more compression so we'll use CSS Nano to do further compression.

To use it we first need to install the plugin

```
npm i -D cssnano
```

Next we need to modify our build script to require CSS Nano:

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

And, finally, we need to modify the task to incorporate CSS Nano. We do this by adding CSS Nano to our PROCESSORS array. The modified task now looks like this:

```
gulp.task("processCSS", () => {
  // What processors/plugins to use with PostCSS
  const PROCESSORS = [
    autoprefixer({browsers: ['last 3 versions']}),
    ccssnano()
  ];
  return gulp
    .src("src/css'last 3 versions'/**/*.css")
    .pipe($.sourcemaps.init())
    .pipe(postcss(PROCESSORS))
    .pipe($.sourcemaps.write("."))
    .pipe(gulp.dest("src/css"))
    .pipe($.size({
      pretty: true,
      title:
```

```
        "processCSS"
    }));
});
```

We can add further processors following the same formula: install and require the plugin, add the plugin (and any configuration) to the PROCESSORS array and test to make sure that it does what you want it to.

Building a plugin

> The code for this section originally appeared in Tuts+ PostCSS Deep Dive: Create Your Own Plugin

What I find the most intriguing about PostCSS is the API and how easy it makes it for developers to create plugins to address specific needs.

What the CSS code will look like

Let's assume, for example, that we have a set of fonts that Marketing has decided we should use in our content. Rather than type the full string of all the fonts in the stack you can do something like this instead:

```
html {
  font-family: stack("Arial");
  weight: normal;
  style: normal;
}
```

And the resulting CSS will appear like this:

```
html {
  font-family: 'Arial, "Helvetica Neue", Helvetica, sans-serif';
  weight: normal;
  style: normal;
```

```
}
```

Configuring the project

To initialize the plugin project we have to create a folder and initialize the package with NPM and accept the defaults automatically. We do this with the following commands:

```
mkdir local-stacks # Creates the directory
cd local-stacks # Changes to the directory we just created
npm init --yes # Inits NPM accepting all defaults automatically
```

Now we must create the file we'll use as our plugin's entry point, `index.js`. We can create this with:

```
touch index.js
```

Or create the file in your text editor. I normally use Visual Studio Code.

Writing the code

To get the plugin going we need to install and require two plugins: The PostCSS core engine (`postcss`) and Underscore (`underscore`) that we will use to merge local and plugin configurations. I am not using ES6 module import syntax (although it would make the code simpler) because I want to use the module with older versions of Node.

We then define an array of the font stacks that we want to use. The name we want to use for the stack is the key and the stack itself is the value for the key.

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

'underscore'// Font stacks from http://www.cssfontstack.com/
const fontstacks_config = {
```

```
  'Arial': 'Arial, 'Helvetica Neue', Helvetica, sans-serif',
  'Times New Roman': 'TimesNewRoman, 'Times New Roman', Times, Baskervill
  'Lucida Grande':, 'Lucida Sans Unicode', 'Lucida Sans', Geneva, Verdana
}
```

toTitleCase will convert the string passed to it so that the first letter of each word is capitalized. The regular expression that we use to capture the string to title case is a little complicated (it was for me when I first saw it) so I've unpacked it below:

- \w matches any word character (equal to `[a-zA-Z0-9_]`)
- \S* matches any non-whitespace character (equal to `[^\r\n\t\f ]`)
- * Quantifier — Matches between zero and unlimited times, as many times as possible, giving back as needed (greedy)
- g modifier - Return all matches (don't return after first match)

```
// Credit for this function to http://bit.ly/1hJj9jb in SO
function toTitleCase(str) {
  return str.replace(/\w\S*/g, function(txt){return txt.charAt(0).toUpperC
    txt.substr(1).toLowerCase();});
}
```

The module we're exporting is the actual plugin. We give it a name, `local-stacks`, and we define it as function. In the function:

- We walk through all the rules in the stylesheet using `walkRules`, part of the PostCSS API
- For each rule we walk through all the declaractions using `walkDecls`, also part of the PostCSS API
- We test if there is a fontstack call in the declaration. If there is one we:
    1. Get the name of the fontstack requested by matching the value inside the parenthesis and then replacing any quotation marks
    2. Title case the resulting string in case the user didn't
    3. Look the name of the font stack in the `fontstack_config` object
    4. Capture any value that was in the string before the fonstack call
    5. Create a new string with both the first font and the value of our font stack
    6. Return the new value as the value of our declaration

```
module.exports = postcss.plugin('local-stacks', function (options) {

    return function (css) {

      options = options || {};

    fontstacks_config = _.extend(fontstacks_config, options.fontstacks);

    css.walkRules(function (rule) {

      rule.walkDecls(function (decl, i) {
        var value = decl.value
        if (value.indexOf( 'fontstack(' ) !== -1) {

          var fontstack_requested = value.match(/\(([^)]+)\)/)[1]
            .replace(/[/\(([^)]+)\)/"']/g, "");

          fontstack_requested = toTitleCase(fontstack_requested);

          var fontstack = fontstacks_config[fontstack_requested];

          var first_font = value.substr(0, value.indexOf('fontstack('));

          var new_value = first_font + fontstack;

          decl.value = new_value;

          }
        });
      });
    });
  }
});
```

Next Steps and Closing

This is a very simple plugin as far as plugins go. You can look a Autoprefixer and CSS Nano for more complex examples and ideas of what you can do with PostCSS.

If you're interested in exploring the API, it is fully documented at http://api.postcss.org/.

> An important reminder, **you don't have to reinvent the wheel**. There is a large plugin ecosystem available at www.postcss.parts. We can use these plugins to get the results we want.
>
> This makes writing your own plugin a fun but not always necessary exercise.

Once you've scoped your CSS project you can decide how much of PostCSS you need and how your needs can be translated into existing plugins and custom code.

Links, Resources and Ideas

- Tuts+ Post CSS Deep Dive
  - Roll Your Own Preprocessor
  - PostCSS Shortcuts and Shorthand
  - PostCCSS Miscellaneous Goodies
  - Create Your Own Plugin
- It's time for everyone to learn about PostCSS
- Some Things You May Think About PostCSS And You Might Be Wrong
- A look into writing future CSS with PostCSS and cssnext
- Breaking up with Sass: it's not you, it's me
- Extending Sass with PostCSS
- How to Build Your Own CSS Preprocessor With PostCSS
- I'm Excited About PostCSS
- Improving the Quality of Your CSS with PostCSS
- PostCSS - Future of CSS after preprocessors
- https://ashleynolan.co.uk/blog/postcss-a-reviewPostCSS – Sass Killer or Preprocessing Pretender?
- Musings from Someone Discovering PostCSS
- So you want to make a PostCSS plugin
- That postcss. Its so hot right now
- Introduction to Postcss
- 7 Postcss Pluguins to Ease You Into Postcss
- So you want to make a PostCSS plugin
- Awesome PostCSS