Putting the SASS/LESS before CSS

CSS Pre-Processors What? Why? How?

Jon Hilton

CSS Pre-processors

What are they?

CSS pre-processors run as part of your development setup. They take the CSS you write and transform it for the browser.

Just as you'll tend to use a Javascript compiler (e.g. Babel or Typescript) to make your JS compatible with most browsers, you can do the same with CSS.

This opens up the door to all sorts of time-saving shortcuts and tricks you can pull off with the help of a pre-processor to take everything you write and transform it back into regular, boring (often perplexing) CSS.

LESS and SASS are the most CSS pre-processors and the ones we'll focus on now.

Here are a few reasons to use a CSS Pre-processor...

Nesting

If you've done any work with CSS you've likely run into things like this...

```
.menu li { color: #666; }
.menu li.important { color: rgb(255, 21, 21); }

.menu a { color: #666; }
.menu a:hover { color: #ccc; }
```

You want to target your elements (`.menu li`, or `.menu a`) but need slightly different rules for more specific variations (e.g. `.important` or `:hover`).

In "vanilla CSS" this leads to duplication. Each style has to be repeated for every variation you need to target.

If this were a real-world application we would no doubt end up with a lot of lines starting with `.menu`.

CSS pre-processors use nesting to let you do this instead...

SASS/LESS

```
.menu  {
    li {
        color: #666;
        .important { color: rgb(255, 21, 21); }
    }
    a {
        color: #666;
        &:hover {
            color: #ccc;
        }
    }
}
```

This results in exactly the same CSS we had before, but makes it easier to see the hierarchy and removes the duplication...

That funny looking `&:hover` is worth breaking down.

`&` is a shorthand way of referring to the parent element; `a` in this case.

We want the hover style to apply to the `a` tag hence the `&` reference.

Just to emphasise the point. We could do this...

```
.menu {
    p:hover { color: #ccc; }
}
```

or this...

```
.menu {
    p {
        &:hover { color: #ccc; }
    }
}
```

Both are equivalent and would result in this css.

```
.menu p:hover {
  color: #ccc;
}
```

Variables

With traditional CSS you'll often use things like colors in more than one place.

This is fine until you have to change it! Then you end up having to update your colors in multiple places (possibly across different files).

```
#main .tagline { color: rgb(255, 21, 21); }
.highlight { color: rgb(255, 21, 21); }
span.error { color: rgb(255, 21, 21); }
```

All the same shade of red...

With the CSS Pre-processors you can declare variables for things like colors (font-sizes etc.) and reference them wherever you like.

SASS

```
$primaryColor: rgb(255, 21, 21);

#main .tagline { color: $primaryColor; }
.highlight { color: $primaryColor; }
span.error { color: $primaryColor; }
```

LESS

```
@primaryColor: rgb(255, 21, 21);

#main .tagline { color: @primaryColor; }
.highlight { color: @primaryColor; }
span.error { color: @primaryColor; }
```

Mixins

You can use Mixins to take a group of CSS declarations and reuse them throughout your CSS (LESS/SASS) files.

Let's say you want to use some newer CSS features (like flexboxes). Depending on which browsers you're targeting you may need to specify vendor prefixes to make everything work.

> This happens when features are in "draft" form in the official specs but not yet finalised.
>
> The browser vendors (MS, Google, Mozilla etc) often implement support for these features anyway, but you have to employ vendor prefixes to use them (until the specs are finalised and published).

Here's an example (taken from the handy ShouldIPrefix.com).

```
.page-wrap {
    display: -webkit-box;  /* OLD - iOS 6-, Safari 3.1-6, BB7 */
    display: -ms-flexbox;  /* TWEENER - IE 10 */
    display: -webkit-flex; /* NEW - Safari 6.1+. iOS 7.1+, BB10 */
    display: flex;         /* NEW, Spec - Firefox, Chrome, Opera */
  }
```

Now I don't know about you, but typing this out every time I want to use a flexbox seems a bit... much!

SASS Mixins

With **SASS** we can create a mixin like this...

```
@mixin flex {
    display: -webkit-box;  /* OLD - iOS 6-, Safari 3.1-6, BB7 */
    display: -ms-flexbox;  /* TWEENER - IE 10 */
    display: -webkit-flex; /* NEW - Safari 6.1+. iOS 7.1+, BB10 */
    display: flex;         /* NEW, Spec - Firefox, Chrome, Opera */
}
```

And use it wherever we like...

```
.page-wrap {
    @include flex;
}
```

... saving us a lot of typing (and making it easy to change the CSS should we need to).

LESS Mixins

Less does a similar thing, but with different syntax.

```
.flex {
    display: -webkit-box;  /* OLD - iOS 6-, Safari 3.1-6, BB7 */
    display: -ms-flexbox;  /* TWEENER - IE 10 */
    display: -webkit-flex; /* NEW - Safari 6.1+. iOS 7.1+, BB10 */
    display: flex;         /* NEW, Spec - Firefox, Chrome, Opera */
}

.page-wrap {
    .flex;
}
```

You can take this even further and pass arguments to mixins.

SASS Mixins with arguments

```
@mixin round-borders($radius) {
    -moz-border-radius: $radius;
    -webkit-border-radius: $radius;
    border-radius: $radius;
}

.button {
    @include round-borders(2px);
}
```

LESS Mixins with arguments

```
.round-borders(@radius) {
    -moz-border-radius: @radius;
    -webkit-border-radius: @radius;
    border-radius: @radius;
}

.button {
    .round-borders(2px);
}
```

Both will result in this CSS...

```
.button {
  -moz-border-radius: 2px;
  -webkit-border-radius: 2px;
  border-radius: 2px;
}
```

Conditionals

Sometimes you're going to want to re-use your CSS but have it behave differently depending on where it's used.

Both SASS and LESS support conditional logic for this very purpose.

SASS conditionals

```
@if $primary {
    color: $primaryColor;
}
```

Combining conditionals, mixins and variables we can do funky things like this!

```
$primaryColor: rgb(255, 21, 21);

@mixin button($primary) {
    @if $primary {
        color: $primaryColor;
    }
    border: 1px solid grey;
}

.action {
    @include button(true); // primary color (and our border)
}

.anotherAction {
    @include button(false); // no color specified
}
```

Our button's color will be set to `primaryColor` if we pass `true`. If not, the color property won't be set at all.

If we make `$primary` optional then we can safely omit it and nothing will blow up.

```
@mixin button($primary: false) {
    // rest of mixin code
}

.anotherAction {
    @include button(); // no value for $primary, defaults to false
}
```

Here we default to `false` until told otherwise!

LESS Conditionals

Less takes a different approach using "Guards".

This can be a little less intuitive if you're used to conditional statements...

```
@primaryColor: rgb(255, 21, 21);

.button(@primary) when (@primary) {
    color: @primaryColor;
}

.button(@primary) {
    border: 1px solid grey;
}

.action {
    .button(true); // primary color (and our border)
}

.anotherAction {
    .button(false); // no color specified (but still our border)
}
```

Here we'll only get a button set to the primary color if we pass `true` when we reference the mixin...

`.button(true)`

We get the border in both cases.

To make the `@mode` argument optional we can tweak it thus...

```
@primaryColor: rgb(255, 21, 21);

.button(@primary) when (@primary) {
    border: 1px solid grey;
    color: @primaryColor;
}

.button(@primary: false) when not (@primary) {
    border: 1px solid grey;
}

.action {
    .button(true);
}

.anotherAction {
    .button();  // no value specified for @primary, defaults to false
}
```

We specified a default value of `false` for `@primary` using the syntax `@primary: false`;

Imports

And finally...

We've all been there. You go to tweak some CSS for an application, go to `site.css`, scroll to the bottom and add your style there.

Then you cross your fingers, hope for the best and get on with your day!

Thankfully SASS and LESS have a solution to this "one CSS file to rule them all" problem.

You can split your CSS up into different files, organise them into folders, store them next to the relevant HTML etc.

You can then import your various files whenever you need them.

SASS

```
@import "mixins/mixin.scss";
```

LESS

```
@import "mixins/mixin.less";
```

In both cases, you can refactor your mixins out to the respective file (with the correct extension), then import them where you need them.

Try it yourself: Use LESS (the easy way with Parcel)

If you've read "front-end in four hours" you'll have seen an easy way to get going with Typescript using Parcel.

We can very easily use LESS with Parcel too...

- Open up a folder in Visual Studio Code, or your editor of choice (this will be the root folder for your test application)
- Open up a terminal pointing to this folder (Visual Studio Code has its own; View > Terminal)
- Type this command (if you haven't already on your machine)...

```
npm install -g parcel-bundler
```

- Then this one...

```
npm init -y
```

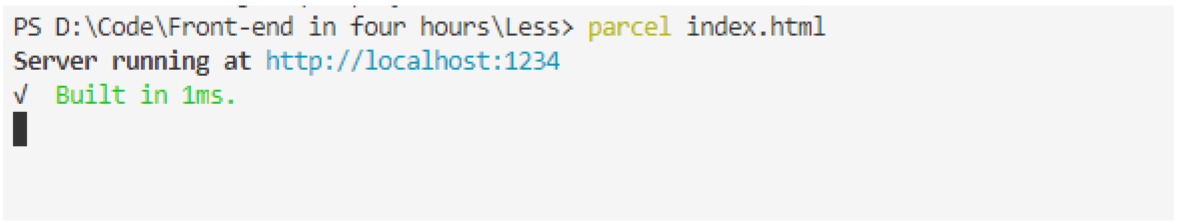
- Create a new file in this folder and call it `index.html`
- Type this code in `index.html`

```
<!DOCTYPE html>
<html>
<head>
    <link href="/app.less" />
</head>
<body>
    Hello LESS
</body>
</html>
```

- Now create an `app.less` file in the `src` folder.
- Run parcel (specifying `index.html` as the entry point for the application)

```
parcel index.html
```

All being well you'll see something like this...



```
PS D:\Code\Front-end in four hours\Less> parcel index.html
Server running at http://localhost:1234
√  Built in 1ms.
```

If you get this error "index.html: Cannot read property 'walk' of null", it's probably because your HTML is malformed. This can happen when you copy and paste from this guide.

Type the code out instead and you should be good to go.

Now try making a change to `app.less`

Parcel will spot that you're trying to use less and will download/install the relevant NPM packages.

From here on, whenever you make changes to your less files, parcel will pre-process theme and turn them into regular CSS.

Check out the `dist` folder. You'll find the bundled .css file in there which contains the regular CSS for your LESS code.

Try it yourself: Use SASS (the easy way with Parcel)

We can very easily use SASS with Parcel too...

- Open up a folder in Visual Studio Code, or your editor of choice (this will be the root folder for your test application)
- Open up a terminal pointing to this folder (Visual Studio Code has its own; View > Terminal)
- Type this command (if you haven't already on your machine)...

```
npm install -g parcel-bundler
```

- Then this one...

```
npm init -y
```

- Create a new file in this folder and call it `index.html`
- Type this code in `index.html`

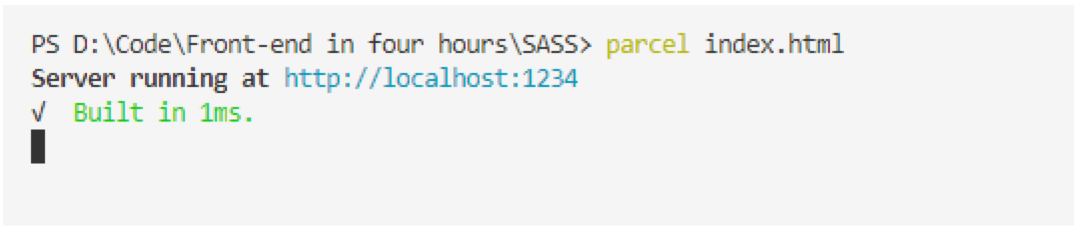
```
<!DOCTYPE html>
<html>
<head>
    <link href="/app.scss" />
</head>
<body>
    Hello SASS
</body>


</html>
```

- Now create an `app.scss` file in the `src` folder
- Run parcel, specifying the entry point for the app (`index.html`).

```
parcel index.html
```

You should see something like this...



```
PS D:\Code\Front-end in four hours\SASS> parcel index.html
Server running at http://localhost:1234
√  Built in 1ms.
```

If you get this error "index.html: Cannot read property 'walk' of null", it's probably because your HTML is malformed. This can happen when you copy and paste from this guide.

Type the code out instead and you should be good to go.

Now add some css (or indeed SASS) to `app.scss` and save it

Parcel will spot that you're trying to use SASS and will download/install the relevant NPM package for you.

It will then pre-process the .scss file and publish the resulting css to the `dist` folder.

What next?

We've just scratched the surface of what Less and SASS can do. Spin up a test app (Parcel is probably the easiest way to get up and running) and give them a whirl.

One last thing before we go, both SASS and LESS can do loops and maths, which raises some interesting possibilities...

SASS Loops and Maths

```
@for $i from 1 through 12 {
    .col-#{$i} {
        width: (100%/12)*$i
    }
}
```

LESS Loops and Maths

```
.make-columns(@i:1) when (@i =< 12) {
  .column-@{i} {
    width: (100%/12)* @i;
  }
  .make-columns(@i + 1);
}

.make-columns();
```

Both examples generate CSS classes for `col-1` all the way up to and including `col-12` (with the relevant fraction of 100%).

This is basically a simplistic version of something like Bootstrap's grid system. You'd probably need a `float:left` in there somewhere and these days everyone seems to be talking about Flexbox as a better alternative...

But you get the idea.

Here's the generated CSS in both cases.

```
.col-1 {
  width: 8.3333333333%;
}

.col-2 {
  width: 16.6666666667%;
}

.col-3 {
  width: 25%;
}

.col-4 {
  width: 33.3333333333%;
}

.col-5 {
  width: 41.6666666667%;
}

.col-6 {
  width: 50%;
}

.col-7 {
  width: 58.3333333333%;
}

.col-8 {
  width: 66.6666666667%;
}

.col-9 {
  width: 75%;
}

.col-10 {
  width: 83.3333333333%;
}

.col-11 {
  width: 91.6666666667%;
}

.col-12 {
  width: 100%;
}
```

Pretty neat huh?! :-)