Making your life easier with Sass

Preface

When I entered my current project one of the first things I noticed was it used stylesheet languages for its styles. “Yay!”, I thought. But then I opened up one of those files and found nothing but plain vanilla CSS inside. It was the same for most of them. No nesting, no variables, no mixins, no inheritance, no operators… Well, you get the idea. I won’t judge anyone and I’d rather believe those who wrote them didn’t use them just because they didn’t know how to. So I thought it would be interesting to write a short article explaining the basics of them.

I could be explaining LESS instead of SASS but as the differences between them syntax-wise are minimal and I personally prefer the later I’ll go with it. Bear with me ;)

PS: All the code I’ll be showing here can be found on my demos git repo.

Variables and import

One of the most basic features of SASS is the use of variables. Be it strings, numbers or lists you can use them.

“Variables in CSS? Is this magic?”

This allows easier understanding of what we’re using and reusability between CSS files. As an example, let’s say we have a page with one header and one footer. Each one will have its own CSS file but they both share the same font color. Let’s see how that would look in plain old CSS:

header.css

header {

color: #39ACE5;

}

footer.css

footer {

color: #39ACE5;

}

“Looks fine to me”.

But it can be better. Let’s look at how it would look with variables:

header.scss

@import 'colors.scss';

header {

color: $cyan;

}

footer.scss

@import 'colors.scss';

footer {

color: $cyan;

}

colors.scss

$cyan: #39ACE5;

“What’s this? We’ve got even more code and files now!”.

Indeed we do, but there’s a reason behind it. Let’s say you change the website’s background color and now your font color is too dark to be read. It may be easy to change it in the example, having only two files that use this same color, but it can be tedious in real projects where you may be referencing a color in dozens of CSS files. By extracting a color to a variable you don’t just make it more readable (let’s face it, it’s not easy to understand #39ACE5 equals the cyan color). In case of need to change it, you’d only need to change wherever it was defined (in just one place) and then re-compile your SCSS files into CSS ones. Pretty nifty, huh?

Operators and loops

We have variables so it only makes sense to have operators as well. And they can store lists so Sass obviously gives supports for looping.

Let’s say we want to incorporate the column system bootstrap provides, but we don’t want the other features.

“I do dislike how it messes up the default HTML components”.

We could write our own column system with operators and loops. If we compiled the class below:

columns.scss

.row-fluid {

.column {

display: inline-block;

}

}

@for $i from 1 to 13 {

.col-#{$i} {

@extend .column;

width: $i / 12 \* 100%;

}

}

…we’d end up generating this monster:

columns.css

.row-fluid .column,

.row-fluid .col-1,

.row-fluid .col-2,

.row-fluid .col-3,

.row-fluid .col-4,

.row-fluid .col-5,

.row-fluid .col-6,

.row-fluid .col-7,

.row-fluid .col-8,

.row-fluid .col-9,

.row-fluid .col-10,

.row-fluid .col-11,

.row-fluid .col-12

{ display: inline-block; }

.col-1 { width: 8.33333%; }

.col-2 { width: 16.66667%; }

.col-3 { width: 25%; }

.col-4 { width: 33.33333%; }

.col-5 { width: 41.66667%; }

.col-6 { width: 50%; }

.col-7 { width: 58.33333%; }

.col-8 { width: 66.66667%; }

.col-9 { width: 75%; }

.col-10 { width: 83.33333%; }

.col-11 { width: 91.66667%; }

.col-12 { width: 100%; }

“Now THAT’s time saving”.

Mixins

We talked about *variables*, but SASS has functions as well. These are called mixins.

Let’s say we need to use the filter CSS rule. But it relies on different prefixes depending on the browser.

“I hate browser dependent rules”.

We can easily avoid repeating ourselves by the use of mixins. Let’s @include the browserPrefixed mixin and use it for writing it:

mixins.scss

$browser-prefixes: ("moz", "webkit", "ms", "o");

@mixin browserPrefixed($property, $value: ()) {

#{$property}: $value;

@each $prefix in $browser-prefixes {

#{'-' + $prefix + '-' + $property}: $value;

}

}

filters.scss

@import 'mixins.scss';

.blurred {

@include browserPrefixed(filter, blur(2px));

}

As you can see, we can use as well lists and arrays with SASS. Something one who only knew CSS would have probably never even dreamed of.

“Is this what heaven looks like?”.

The resulting compiled CSS file would look like this:

filters.css

.blurred {

filter: blur(2px);

-moz-filter: blur(2px);

-webkit-filter: blur(2px);

-ms-filter: blur(2px);

-o-filter: blur(2px);

}

How do you like it, writing browser dependant rules with one single line? I myself find it heaven-like.

Inheritance

Let’s say we want to style feedback messages. They all will share all their styling, except from the background-color that will depend on the source and criticality of the message and success and error messages will additionally use bold text. We’d end up with something like this:

messages.css

.msg-success, .msg-info, .msg-warning, .msg-error {

border-radius: 8px;

padding: 12px 16px;

color: white;

text-align: center;

border: 4px solid white;

}

.msg-success, .msg-error {

font-weight: bold;

}

.msg-success {

background-color: #5cb85c;

}

.msg-info {

background-color: #5bc0de;

}

.msg-warning {

background-color: #f0ad4e;

}

.msg-error {

background-color: #d9534f;

}

Not bad, but let’s add variables and inheritance for a better readability:

messages.scss

@import 'bootstrap-colors.scss';

.feedback-message {

border-radius: 8px;

padding: 12px 16px;

color: white;

text-align: center;

border: 4px solid white;

}

.important-message {

font-weight: bold;

}

.msg-success {

@extend .feedback-message;

@extend .important-message;

background-color: $brand-success;

}

.msg-info {

@extend .feedback-message;

background-color: $brand-info;

}

.msg-warning {

@extend .feedback-message;

background-color: $brand-warning;

}

.msg-error {

@extend .feedback-message;

@extend .important-message;

background-color: $brand-danger;

}

Easier to comprehend and way less cluttering. And, more importantly, we know at the class level what other classes it extends, instead of needing to look for wherever the never-ending list of classes that share some style rules is defined.

“That’s a good point”.

Nesting

Another cool feature is nesting. As usual, let’s go with an example.

Imagine you have the following structure on HTML:

<div class="minion-table-wrapper">

<table>

<tr>

<th>

<th>

<th>

<tr>

<td>

<td>

<td>

[...]

And you want to apply rules without adding new classes; just relying on the wrapper class. With plain CSS, we could end up with something like this:

table.css

.minion-table-wrapper table {

margin: auto;

}

.minion-table-wrapper table tr:nth-child(even) {

background: #CB99C9;

}

.minion-table-wrapper table tr:nth-child(odd) {

background: #B19CD9;

}

.minion-table-wrapper table tr th {

background: #CCC;

}

.minion-table-wrapper table tr td {

text-align: left;

}

.minion-table-wrapper table tr td:first-child {

background: #CCC;

text-align: center;

}

.minion-table-wrapper table tr th, .minion-table-wrapper table tr td {

min-width: 32px;

padding: 0 4px;

}

.minion-table-wrapper table tr th:first-child, .minion-table-wrapper table tr td:first-child {

text-align: center;

}

As you can see, it’s not easy to tell apart the actual structure of the components. Let’s write the same code but with Sass and nesting:

table.scss

@import 'colors.scss';

.minion-table-wrapper {

table {

margin: auto;

tr {

&:nth-child(even) {

background: $pastel-violet;

}

&:nth-child(odd) {

background: $pastel-purple;

}

th {

background: $light-gray;

}

td {

text-align: left;

&:first-child {

background: $light-gray;

text-align: center;

}

}

th, td {

min-width: 32px;

padding: 0 4px;

&:first-child {

text-align: center;

}

}

}

}

}

As you can tell, the actual structure of the page elements is now quite obvious and we remove the need to prefix it to every rule we write.

“It’s like HTML within CSS”.

Final words

To sum up, Sass is just plain awesome. If you give it a chance, it will for sure make your lives quite easier.

Well, that was it. I hope I made my explanations and examples clear and interesting enough (“I’d like to think I did”).

My hope is I imbued this article with at least a bit of my love for Sass and some of that gets to you.