Home / Articles /

Approaches to Media Queries in Sass

Author
Eduardo Bouças

Last Updated
Apr 23, 2020

Using media queries in CSS as part of responsive websites is bread and butter stuff to todays front-end developer. Using preprocessors to make them more comfortable to write and easier to maintain has become common practice as well.

I spent a few months experimenting with a dozen different approaches to media queries in Sass and actually used a few in production. All of them eventually failed to cater for everything I needed to do in an elegant way. So I took what I liked about each of them and created a solution that covered all scenarios I came across.

(#why-use-a-preprocessor-at-all) Why use a preprocessor at all?

That's a fair question. After all, what's the point of doing all this if one can simply write media queries using pure CSS? Tidiness and maintainability.

The most common use for media queries is the transformation of a layout based on the browser's viewport width. You can make a layout adapt in such a way that multiple devices with different screen sizes can enjoy an optimal experience. As a consequence, the expressions used to define the media queries will make reference to the typical screen width of those devices.

So if your code contains 5 media queries that target tablet devices with a width of *768px*, you will hardcode that number 5 times, which is something ugly that my OCD would never forgive. First of all, I want my code to be easy to read to the point that anyone understands instantly that a media query is targeting tablet devices just by looking at it – I reckon the word *tablet* would do that better than *768px*.

Also, what if that reference width changes in the future? I hate the idea of replacing it in 5 instances around the code, especially when it’s scattered around multiple files.

A first step would be to store that breakpoint in a variable and use it to construct the media query.

```scss
/* Using plain CSS */
@media (min-width: 768px) {

}

/* Using SCSS variables to store breakpoints */
$breakpoint-tablet: 768px;
@media (min-width: $breakpoint-tablet) {

}
```

Another reason to write media queries with a preprocessor like Sass is that it can sometimes provide some precious help with the syntax, in particular when writing an expression with a logical *or* (represented with a comma in CSS).

For example, if you want to target retina devices (https://css-tricks.com/snippets/css/retina-display-media-query/) , the pure CSS syntax starts getting a bit verbose:

```scss
/* Plain CSS */
@media (min-width: 768px) and
       (-webkit-min-device-pixel-ratio: 2),
       (min-width: 768px) and
       (min-resolution: 192dpi) {
}

/* Using variables? */
@media (min-width: $bp-tablet) and ($retina) { // or #{$retina}

}
```

It does look nicer, but unfortunately it won't work as expected.

(#a-problem-with-logic) A problem with logic

Because of the way the CSS "or" operator works, I wouldn't be able to mix the *retina* conditions with other expressions since `a (b or c)` would be compiled into `(a or b) c` and not `a b or a c`.

scss

```
$retina: "(-webkit-min-device-pixel-ratio: 2), (min-resolution: 192dpi)";

// This will generate unwanted results!
@media (min-width: 480px) and #{$retina} {
  body {
    background-color: red;
  }
}
```

css

```
/* Not the logic we're looking for */
@media (min-width: 480px) and (-webkit-min-device-pixel-ratio: 2), (min-re
  body {
    background-color: red;
  }
}
```

I realized I needed something more powerful, like a mixin or a function, to address this. I tried a few solutions.

(#dmitry-sheikos-technique) Dmitry Sheiko's technique

One I tried was Dmitry Sheiko's technique (https://codepen.io/dsheiko/pen/KeLGy), which had a nice syntax and includes Chris' retina declaration.

scss

```scss
// Predefined Break-points
$mediaMaxWidth: 1260px;
$mediaBp1Width: 960px;
$mediaMinWidth: 480px;

@function translate-media-condition($c) {
  $condMap: (
    "screen": "only screen",
    "print": "only print",
    "retina": "(-webkit-min-device-pixel-ratio: 1.5), (min--moz-device-pix
    ">maxWidth": "(min-width: #{$mediaMaxWidth + 1})",
    "<maxWidth": "(max-width: #{$mediaMaxWidth})",
    ">bp1Width": "(min-width: #{$mediaBp1Width + 1})",
    "<bp1Width": "(max-width: #{$mediaBp1Width})",
    ">minWidth": "(min-width: #{$mediaMinWidth + 1})",
    "<minWidth": "(max-width: #{$mediaMinWidth})"
  );
  @return map-get( $condMap, $c );
}

// The mdia mixin
@mixin media($args...) {
  $query: "";
  @each $arg in $args {
    $op: "";
    @if ( $query != "" ) {
      $op: " and ";
    }
    $query: $query + $op + translate-media-condition($arg);
  }
  @media #{$query}  { @content; }
}
```

But the problem with logical disjunction was still there.

scss

```
.section {
  @include media("retina", "<minWidth") {
    color: white;
  };
}
```

css

```
/* Not the logic we're looking for */
@media (-webkit-min-device-pixel-ratio: 1.5), (min--moz-device-pixel-ratic
  .section {
    background: blue;
    color: white;
  }
}
```

(#landon-schropps-technique) Landon Schropp's technique

Landon Schropp's (https://davidwalsh.name/write-media-queries-sass) was my next stop. Landon creates simple named mixins that do specific jobs. Like:

```scss
$tablet-width: 768px;
$desktop-width: 1024px;

@mixin tablet {
  @media (min-width: #{$tablet-width}) and (max-width: #{$desktop-width -
    @content;
  }
}

@mixin desktop {
  @media (min-width: #{$desktop-width}) {
    @content;
  }
}
```

He has a single-responsibility retina version as well.

But another problem hit me when I was styling an element that required additional rules on intermediate breakpoints. I didn't want to pollute my list of global breakpoints with case-specific values just so I could still use the mixin, but I definitely didn't want to forgo the mixin and go back to using plain CSS and hardcoding things every time I had to use custom values.

```scss
/* I didn't want to sometimes have this */
@include tablet {

}

/* And other times this */
@media (min-width: 768px) and (max-width: 950px) {
}
```

(#breakpoint-technique) Breakpoint technique

Breakpoint-sass (http://breakpoint-sass.com/) was next on my list, as it supports both variables and custom values in its syntax (and, as a bonus, it’s really clever with pixel ratio media queries (https://github.com/at-import/breakpoint/wiki/Advanced-Media-Queries#resolution-media-queries)).

I could write something like:

```scss
$breakpoint-tablet: 768px;

@include breakpoint(453px $breakpoint-tablet) {

}

@include breakpoint($breakpoint-tablet 850px) {

}

/* Compiles to: */
@media (min-width: 453px) and (max-width: 768px) {

}

@media (min-width: 768px) and (max-width: 850px) {

}
```

Things were looking better, but I personally think that Breakpoint-sass' syntax feels less natural than Dmitry's. You can give it a number and it assumes it's a *min-width* value, or a number and a string and it assumes a property/value pair, to name just a few of the combinations it supports.

That's fine and I'm sure it works great once you're used to it, but I hadn't given up on finding a syntax that was both simple and as close as possible to the way I orally describe what a media query must target.

Also, if you look at the example above you'll see that a device with a width of exactly *768px* will trigger both media queries, which may not be exactly what we want. So I added the ability to write inclusive and exclusive breakpoints to my list of requirements.

(#my-eduardo-boucass-technique) My (Eduardo Bouças's) technique

This is my take (https://github.com/eduardoboucas/include-media) on it.

(#clean-syntax-dynamic-declaration) Clean syntax, dynamic declaration

I'm a fan of Dmitry's syntax, so my solution was inspired by it. However, I'd like some more flexibility in the way breakpoints are created. Instead of hardcoding the names of the breakpoints in the mixin, I used a multidimensional map to declare and label them.

```scss
$breakpoints: (phone: 640px,
               tablet: 768px,
               desktop: 1024px) !default;

@include media(">phone", "<tablet") {
}
```

```
@include media(">tablet", "<950px") {
}
```

The mixin comes with a set of default breakpoints, which you can override anywhere in the code by re-declaring the variable `$breakpoints`.

(#inclusive-and-exclusive-breakpoints) Inclusive and exclusive breakpoints

I wanted to have a finer control over the intervals in the expressions, so I included support for the *less-than-or-equal-to* and *greater-than-or-equal-to* operators. This way I can use the same breakpoint declaration in two mutually exclusive media queries.

scss

```
@include media(">=phone", "<tablet") {

}

@include media(">=tablet", "<=950px") {

}

/* Compiles to */
@media (min-width: 640px) and (max-width: 767px) {

}

@media (min-width: 768px) and (max-width: 950px) {

}
```

(#infer-media-types-and-handle-logic-disjunction) Infer media types and handle logic disjunction

Similarly to the breakpoints, there's a list for media types and other static expressions declared by default (which you can override by setting the variable `$media-expressions`). This adds support for optional media types, such as *screen* or *handheld*, but it's also capable of correctly handling expressions with logical disjunctions, such as the retina media query we saw before. The disjunctions are declared as nested lists of strings.

```scss
$media-expressions: (screen: "screen",
                    handheld: "handheld",
                    retina2x:
                    ("(-webkit-min-device-pixel-ratio: 2)",
                    "(min-resolution: 192dpi)")) !default;

@include media("screen", ">=tablet") {

}

@include media(">tablet", "<=desktop", "retina2x") {

}

/* Compiles to */
@media screen and (min-width: 768px) {

}

@media (min-width: 769px) and
       (max-width: 1024px) and
       (-webkit-min-device-pixel-ratio: 2),
       (min-width: 769px) and
       (max-width: 1024px) and
       (min-resolution: 192dpi) {

}
```

There's no rocket science under the hood, but the full implementation of the mixin isn't something I could show in just a few lines of code. Instead of boring you with huge code snippets and neverending comments, I included a Pen with everything working and I'll briefly describe the process it goes through to construct the media queries.

Embedded Pen Here

(#how-it-works) How it works

1. The mixin receives multiple arguments as strings and starts by going through each one to figure out if it represents a breakpoint, a custom width, or one of the static media expressions.
2. If an operator is found, it is extracted and any matching breakpoint will be returned, or else we assume it's a custom value and cast it to a number (using SassyCast (https://github.com/HugoGiraudel/SassyCast)).
3. If it's a static media expression, it checks for any `or` operators and generates all the combinations necessary to represent the disjunction.
4. The process is repeated for all the arguments and the results will by glued together by the `and` connector to form the media query expression.

If you'd like to look at the complete Sass for it, it's here (https://raw.githubusercontent.com/eduardoboucas/include-media/master/dist/_include-media.scss) . It's called include-media (https://github.com/eduardoboucas/include-media) on GitHub.

(#final-thoughts) Final thoughts

- I'm a big fan of this technique (https://css-tricks.com/making-sass-talk-to-javascript-with-json/) to make Sass talk to JavaScript. Because we declare breakpoints as a multidimensional list with their names as keys, exporting them in bulk to JavaScript becomes really straightforward and can be done automatically with just a few lines of code.
- I'm not trying to put down other people's solutions and I'm definitely not saying this one is better. I mentioned them to show some of the obstacles I found along the way to my ideal solution, as well as some great things they introduced that inspired my own solution.
- You might have some concerns about the length and complexity of this implementation. While I understand, the idea behind it is that you download one single file, `@import` it into your project and start using it without having to touch the source code. Ping me on Twitter (https://twitter.com/eduardoboucas) though if you have any questions.
- You can get it from GitHub (https://github.com/eduardoboucas/include-media) and you are very welcome to contribute with issues/code/love. I'm sure there's still a lot we can do to make it better.

(#update) Update!

Eduardo made a website for his approach: @include-media (https://eduardoboucas.github.io/include-media/) .