Today we're going to talk about Sass performance. This is a tricky subject and one that is often very specific to each individual project, its dependencies, the local or production environment and the scale of the project.

However, performance is an important subject so it's something I wanted to try and tackle.

Improving Compiler Performance

Having a solid workflow when writing code is really important to make sure you can work as quickly as you need to.

Sass needs to be compiled into CSS before it can be read by the browser and this compile step can sometimes become a bit of a bottle neck and start slowing down your development.

On a recent client project (which I inherited from an agency in pretty terrible shape) compile time for the styles was frequently taking over 14 seconds. On a similar scale project of my own, styles take less than 1 second to compile. So why such a difference?

The Client Project (which shall remain nameless) has the following setup:

- Ruby Sass
- Compass mixin library
- 2 compiled stylesheets
- 67 partials
- Grunt task for compiling

My Personal Project (The Food Rush food tech magazine) has the following setup:

- Node Sass
- Autoprefixer
- 1 compiled stylesheet
- 42 partials
- Gulp task for compiling

While the client project has 25 more partials and 1 more compiled stylesheet, I believe the major bottleneck in this compile process is the Compass library and the use of Ruby Sass.

Compass is a library full of all sorts of handy mixins and functions for working with Sass. The majority of which provides CSS3 mixins for handling vendor prefixes across different browsers. Compass modules can be loaded on a case by case basis or you can include the whole library (as has been done on this project) by importing it in your compiled stylesheets:

{% highlight scss %}
@import 'compass';
{% endhighlight %}

Having analysed the whole repository for this client project, the only Compass mixins being used are for:

- transitions
- transform
- opacity
- box-sizing

These are widely supported without any vendor prefixes. If prefixes were needed to support a particular range of browsers, it wouldn't be hard to add these manually, create a custom mixin as needed or use an automated tool for prefixing like Autoprefixer.

The second change to improve the performance of compiling this project would be to switch from using Ruby Sass to Libsass (often done through the node-sass wrapper as part of a Gulp task).

Libsass is a C/C++ port of the original Ruby Sass engine and runs incredibly fast. If you refer back to the episode about Grunt and Gulp, you'll see the difference in compile time for exactly the same code: Ruby Sass with Grunt compiled in **1.519 seconds** and Node Sass with Gulp compiled in **10 mili-seconds**. Check out this video for practical step by step instructions to use Grunt or Gulp to compile your Sass (I'd recommend Gulp, but that's just my personal preference these days).

For more details and some cold hard numbers about compile times of various versions of Sass and other preprocessors, take a look at this post on Opinionated Programmer.

It's worth mentioning that Libsass doesn't support every single Sass feature that's in Ruby Sass. But in my experience I haven't found this to be an issue and have yet to run into a Ruby Sass feature that's not supported in Libsass that's been a blocker on any project so far.

So to wrap up this part of the discussion, there are really two takeaways:

To improve the performance of your compiler, **reduce the amount of plugins and/or libraries** used in a project and **try Libsass** for super-fast compile times.

Optimising CSS for Performance

Moving away from thinking about performance in terms of speed and efficientcy in our workflow, let's turn our attention to writing Sass that compiles to compact, clean and clear CSS.

I've spent many hours writing CSS and have tried a number of modular coding techniques, naming conventions, methodologies like OOCSS and SMACSS and experimented with both `@extend` and `@mixin` a lot.

It would probably take hours to go into fine detail about what I've learned on the topic so I've tried to distill it down into 5 tips for writing Sass that outputs CSS without too much bloat.

Tip 1: Read your compiled CSS

When optimising anything, it's important to know what you're optimising. Being aware of the CSS that's being generated from your Sass is sometimes all you need to know whether your generated code is compact and lightweight or bloated and a bit of a mess.

If you're working on a large project, reading through the whole compiled stylesheet might be impossible so why not just experiment with some snippets of code on Sassmeister or Codepen to check the compiled output.

Tip 2: Avoid over qualified selectors and nesting

Specificity is something to keep in mind when writing Sass or CSS. The more specific selectors you write, the more specific selectors you'll need to override previously declared styles. This can lead to many more lines of code and can make life difficult for yourself and/or other members of your team.

Avoid over-qualified selectors like `a.button` or `div.main` as this limits the reuse of these styles. If you think you need to modify existing styles for specific elements, adding a modifier class is a better approach than differentiating by element type which can be very limiting.

It's almost impossible to read a blog post or watch a video about Sass without hearing the warnings about "don't nest too deeply" but it's worth reiterating: don't nest too deeply.

Nesting selectors can lead you into a false sense of security as you're only typing something very short but they can compile into something very long and complex.

I once inherited a project which had a `_homepage.scss` file containing over 1000 lines of Sass. The first line of code was `body.homepage {` and every other line of code was nested inside that. This means that these 1000 lines of code would only ever apply to a page who's `body` element had a class of `homepage`. This is the opposite of modular and reusabel code and was a nightmare to maintain.

Tip 3: Be wary of mixins (but don't worry about them too much)

We've looked at mixins in previous videos and I cover them in detail in my Up and Running with Sass ebook.

Mixins are used to generate similar patterns of code throughout a codebase. They often get a bad reputation because they generate multiple lines of (almost) identical code.

My advice is to be aware of the mixins you use to ensure they don't generate unnecessary duplication.

However, don't worry about this too much as this kind of duplication can be heavily compressed with gzip and the apparent "bloated CSS" can actually end up very compact.

Tip 4: Avoid extend

Yet another favourite buzz-phrase among Sass users is "don't use extend". We covered `@extend` in Episode 5 and talked about it's pros and cons at length.

Without revisting the exact same argument, to imrpove the performance of your compile times and your generated CSS I would avoid using `@extend` for anything other than utility classes.

Tip 5: Consider using multiple compiled stylesheets

Finally, a tip that may seem a little contradictory of the ususal advice about the importance of concattenating multiple files into one to reduce http requests.

In some cases it may be better for performance to have multiple compiled stylesheets that are loaded on different pages of a large site or application.

Let me explain.

Imagine you have a site with a load of content and informational pages and an online shop. The shop has a number of pages and templates that show off products, product detail pages and all the various steps for adding things to a basket, checkout process, thank you pages and all that jazz.

These pages would no doubt have a significant amount of custom components and styles that aren't used throughout the site.

These pages are probably only seen by a small percentage of the site's visitors.

Conventional wisdom says combine scripts and stylesheets into as few files as possible to reduce http requests. However, if your single stylesheet is full of code that only gets used by a small percentage of your users, would it not make more sense to seperate it out into two smaller stylesheets?

This also has caching benefits too. With one massive stylesheet, a change in the CSS to any distinct part of the website will break the cache on the single file meaning all users need to re-download it. If a small change is made to the ecommerce styles then the main stylesheet will remain cached and only customers to the shop will need to re-download the updated stylesheet.

Bonus Performance Tips

Don't forget that making CSS perform better (either for developer happiness or for users) is kind of a micro optimisation.

Get the low hanging fruit out of the way first: turn on gzip, compress images, leverage caching, compress and minify assets and consider using CDNs.