Writing efficient CSS selectors

csswizardry.com/2011/09/writing-efficient-css-selectors/

Efficient CSS is not a new topic, nor one that I really need to cover, but it's something I'm really interested in and have been keeping an eye on more and more since working at Sky.

A lot of people forget, or simply don't realise, that CSS can be both performant and non-performant. This can be easily forgiven however when you realise just how little you can, err, realise, non-performant CSS.

These rules only *really* apply to high performance websites where speed is a feature, and 1000s of DOM elements can appear on any given page. But best practice is best practice, and it doesn't matter whether you're building the next Facebook, or a site for the local decorator, it's always good to know…

CSS selectors

CSS selectors will not be new to most of us, the more basic selectors are type (e.g. `div`), ID (e.g. `#header`) and class (e.g. `.tweet`) respectively.

More uncommon ones include basic pseudo-classes (e.g. `:hover`) and more complex CSS3 and 'regex' selectors, such as `:first-child` or `[class^="grid-"]`.

Selectors have an inherent efficiency, and to quote Steve Souders, the order of more to less efficient CSS selectors goes thus:

1. ID, e.g. `#header`
2. Class, e.g. `.promo`
3. Type, e.g. `div`
4. Adjacent sibling, e.g. `h2 + p`
5. Child, e.g. `li > ul`
6. Descendant, e.g. `ul a`
7. Universal, i.e. `*`
8. Attribute, e.g. `[type="text"]`
9. Pseudo-classes/-elements, e.g. `a:hover`

Quoted from Even Faster Websites by Steve Souders

It is important to note that, although an ID is technically faster and more performant, it is barely so. Using Steve Souders' CSS Test Creator we can see that an ID selector and a class selector show very little difference in reflow speed.

In Firefox 6 on a Windows machine I get an average reflow figure of 10.9 for a simple class selector. An ID selector gave a mean of 12.5, so this actually reflowed slower than a class.

The difference in speed between an ID and a class is almost totally irrelevant.

A test selecting on a type (<a>), rather than a class or ID, gave a much slower reflow.

A test on a heavily overqualified descendant selector gave a figure of around 440!

From this we can see that the difference between IDs/classes and types/descendants is fairly huge… The difference between themselves is slight.

N.B. These numbers can vary massively between machine and browser. I *strongly* encourage you to run/play with your own.

Combining selectors

You can have standalone selectors such as `#nav`, which will select any element with an ID of 'nav', or you can have

combined selectors such as #nav a, which will match any anchors within any element with an ID of ‘nav’.

Now, we read these left-to-right. We see that we’re looking out for #nav and then any a elements inside there. Browsers read these differently; **browsers read selectors right-to-left**.

Where we see a #nav with an a in it, browsers see an a in a #nav. This subtle difference has a *huge* impact on selector performance, and is a very valuable thing to learn.

For an in-depth reason as to why they do this see this discussion on Stack Overflow.

It’s more efficient for a browser to start at the right-most element (the one it *knows* it wants to style) and work its way back *up* the DOM tree than it is to start high up the DOM tree and take a journey *down* that might not even end up at the right-most selector--also known as the *key* selector.

This has a very significant impact on the performance of CSS selectors...

The *key* selector

The key selector, as discussed, is the right-most part of a larger CSS selector. This is what the browser looks for first.

Remember back up there we discussed which types of selector are the most performant? Well whichever one of those is the key selector will affect the selector’s performance; when writing efficient CSS it is this key selector that holds the, well, key, to performant matching.

A key selector like this:

```
#content .intro{}
```

Is probably quite performant as classes are an inherently performant selector. The browser will look for all instances of .intro (of which there aren’t likely to be many) and then go looking up the DOM tree to see if the matched key selector lives in an element with an ID of ‘content’.

However, the following selector is not very performant at all:

```
#content *{}
```

What this does is looks at *every single* element on the page (that’s *every* single one) and then looks to see if any of those live in the #content parent. This is a very un-performant selector as the key selector is a very expensive one.

Using this knowledge we can make better decisions as to our classing and selecting of elements.

Let’s say you have a massive page, it’s enormous and you’re a big, big site. On that page are hundreds or even thousands of <a>s. There is also a small section of social media links in a <ul> with an ID #social; let’s say there is a Twitter, a Facebook, a Dribbble and a Google+ link. We have four social media links on this page and hundreds of other anchors besides.

This selector therefore is unreasonably expensive and not very performant:

```
#social a{}
```

What will happen here is the browser will assess all the thousands of links on that page before settling on the four inside of the #social section. Our key selector matches far too many other elements that we aren’t interested in.

To remedy this we can add a more specific and explicit selector of .social-link to each of the <a>s in the social area. But this goes against what we know; we know not to put unnecessary classes on elements when we can use (c)leaner markup.

This is why I find performance so interesting; it’s a weird balance between web standards best practices and sheer

speed.

Whereas we would normally have:

```
<ul id="social">
 <li><a href="#" class="twitter">Twitter</a></li>
 <li><a href="#" class="facebook">Facebook</a></li>
 <li><a href="#" class="dribble">Dribbble</a></li>
 <li><a href="#" class="gplus">Google+</a></li>
</ul>
```

with this CSS:

```
#social a{}
```

We'd now have:

```
<ul id="social">
 <li><a href="#" class="social-link twitter">Twitter</a></li>
 <li><a href="#" class="social-link facebook">Facebook</a></li>
 <li><a href="#" class="social-link dribble">Dribbble</a></li>
 <li><a href="#" class="social-link gplus">Google+</a></li>
</ul>
```

with this CSS:

```
#social .social-link{}
```

This new key selector will match far fewer elements and means that the browser can find them and style them faster and can move on to the next thing.

And, we can actually get this selector down further to .social-link{} by not overqualifying it; read on to the next section for that...

So, to recap, your key selector is the one which determines just how much work the browser will have to do, so **this is the one to keep an eye on**.

Overqualifying selectors

Okay so now we know what a key selector is, and that that is where most of the work comes from, we can look to optimise further. The best thing about having nice explicit key selectors is that you can often avoid overqualifying selectors. An overqualified selector might look like:

html body .wrapper #content a{}

There is just too much going on here, and at least three of these selectors are totally unnecessary. That could, at the very most, be this:

```
#content a{}
```

So what?

Well the first one means that the browser has to look for all a elements, then check that they're in an element with an ID of 'content', then so on and so on right the way up to the html. This is causing the browser way too many checks that we really don't need. Knowing this, we can get more realistic examples like this:

```
#nav li a{}
```

Down to just:

```
#nav a{}
```

We know that if the a is inside an li it *has* to be inside the #nav so we can instantly drop the li from selector. Then, as the nav has an ID we know that only one exists in the page, so the element it is applied to is wholly irrelevant; we can also drop the ul.

Overqualified selectors make the browser work harder than it needs to and uses up its time; make your selectors leaner and more performant by cutting the unnecessary bits out.

Is all this really necessary?

The short answer is; *probably not*.

The longer answer is; *it depends on the site you're building*. If you're working on your next portfolio then go for clean code over CSS selector performance, because you really aren't likely to notice it.

If you're building the next Amazon, where microseconds in page speeds *do* make a difference then maybe, but even then maybe not.

Browsers will only ever get better at CSS parsing speeds, even mobile ones. You are very unlikely to ever notice slow CSS selectors on a websites **but**…

But

It is still happening, browsers still are having to do all the work we've talked about, no matter how quick they get. Even if you don't need or even want to implement any of this it is something that is definitely worth knowing. Bear in mind that selectors can be expensive and that you should avoid the more glaring ones where possible. That means if you find yourself writing something like:

```
div:nth-of-type(3) ul:last-child li:nth-of-type(odd) *{ font-weight:bold }
```

Then you're probably doing it wrong.

Now, I'm still kind of new to the world of selector efficiency myself so if I've missed anything, or you have anything to add, please pop it in the comments!

More on CSS selector efficiency

I cannot recommend the website and books of Steve Souders enough. That's pretty much all the *further reading* recommendation you'll need. The guy knows his stuff!