CSS Optimization Basics

Jens Oliver Meiert

CSS Optimization Basics

Jens Oliver Meiert

This book is for sale at http://leanpub.com/css-optimization-basics

This version was published on 2018-04-08

ISBN 978-0-9911480-5-9

Tweet This Book!

Please help Jens Oliver Meiert by spreading the word about this book on Twitter!

The suggested hashtag for this book is #cssoptim.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#cssoptim

Contents

With special thanks to Tony Ruscoe[1] for reviewing the first draft of this book, and Markus Käding for help with the book cover.

[1] http://ruscoe.net/

Introduction

We live in exciting times. Indeed, politically, but particularly, and particularly in the context of this book, technologically. The times are exciting because there has never been a greater wealth of web technologies, tools, and resources to build ever more appealing and performant web sites as well as web apps.

The times are exciting, too, because they are challenging. They are challenging because the wealth of technologies, tools, and resources has led to other types of wealth: of complexity, and of confusion.

The confusion is to be thought of in a sense of focus, precisely: lack of focus, for with all the options we have it has become less apparent what we best pay attention to, whether that's what we should do in the first place, or how we should work with what we have at our disposal. That relates to the sites and apps we build, to the frameworks and libraries we use, and, and here we get back to why you're holding this book in your hands, to the code we write, including: the CSS we write.

In this book, then, we'll look at ways to improve the CSS we write. We'll go over why that generally matters, and why each method matters. We'll also talk about what doesn't matter so much. For example, processors. Much of that will have to do with quality as well as craft. Both I deem important, no less of an idea of web development would I wish to promote. How to best *automate* such optimization is a great question, but first we need to understand what to optimize for exactly—and though it's encouraged to automate as much as possible, whether optimization is to be done manually or automatically is not a concern of this book.

Why Optimization Is Important

Sometimes it appears difficult to answer legitimate questions. Why is optimization important? Why is breathing useful? Why should we eat?

Optimization is important because quality is important; and quality (and quality control) is important because, as I've established in *The Little Book of Website Quality Control*[2], "without it we have no robust way of determining whether what we do and produce is any good. Quality control, therefore, is a key differentiator between professional and amateur work. Consistent quality is the mark of the professional. Quality control, finally, saves time and money and sometimes nerves, particularly in the long run."

That spontaneous definition, aiming at ways to check our work, doesn't yet make the beneficiary clear: the user or customer. It's the end user who benefits and who we want to benefit from great quality work.

[2] http://www.oreilly.com/web-platform/free/the-little-book-of-website-quality-control.csp

As such, optimization, code optimization, CSS optimization all represent a striving to deliver better quality work to the benefit of our users, and to our benefit of becoming greater professionals.

Optimization has a touch of perfectionism, but it's truly a means, and not an end; with optimization we acknowledge that our initial work can be improved.

What's Not Covered by Optimization

In this book, then, I won't cover *all* things relating to CSS development these days. The reason for this is simple: Not all of this relates to *optimization*. And so one thing I won't cover here are CSS preprocessors. Personally, I've sided with Roger Johansson[3] and outlined reasons for not using preprocessors[4], but the little there is to optimize through them is already going to be covered in this book in other ways: for example, through repeat emphasis on coding and formatting standards.

On top of that comes that I regard it more useful to stay close to core technologies, and not get bogged down by abstractions, abstractions like preprocessors. In the short run we may benefit from knowing well how to handle abstractions; but in the long run we get much greater rewards from knowing the underlying systems and technologies. And therefore we'll focus on standard CSS rather than non-standard Sass, Less, Stylus, and whatnot.

The same applies to other forms of abstraction, and so we'll look and point at CSS tools, but not discuss optimization of use of these tools, their workflows, nor how to improve those tools themselves.

*
**

The code in this book follows Google's HTML/CSS style guide[5]. (For an introduction to coding guidelines and notes on the Google style guide, see *The Little Book of HTML/CSS Coding Guidelines*[6].)

Before we begin, this is a self-published book made with little extra help. If you find mistakes in it, you have all reason to blame me, the author—but also a great opportunity to help make the book a little better as a contributor (who are all mentioned unless they choose not to be named). Please file an issue[7] for every problem you find, or submit a pull request[8] for specific fixes and suggestions. If you bought the book through Leanpub[9] you should get updates containing your and other people's fixes automatically. (Otherwise there's always the original source[10].)

Thank you for your support.

[3] http://www.456bereastreet.com/archive/201603/why_i_dont_use_css_preprocessors/
[4] https://meiert.com/en/blog/no-css-preprocessors/
[5] https://google.github.io/styleguide/htmlcssguide.html
[6] http://www.oreilly.com/web-platform/free/little-book-html-css-coding-guidelines.csp
[7] https://github.com/j9t/css-optimization-basics/issues/new
[8] https://github.com/j9t/css-optimization-basics/pulls
[9] https://leanpub.com/css-optimization-basics
[10] https://github.com/j9t/css-optimization-basics

Development Mindsets

We all know the adage, "Give a man a fish, and you feed him for a day; teach a man to fish, and you feed him for a lifetime." To me, we feed developers for a lifetime by teaching and cherishing certain *mindsets*. They aren't enough, *per se*, but I view them as the foundation that decides where we're headed with our work and our careers; they're not all related to CSS optimization—that would be odd for us as web developers dealing with many different technologies—, but they're so useful, let's look at the key ones.

Do One Thing Really, Really Well

Out of the Google book[11], one wonderful idea is to do things really, really well. But I used the plural here, and Google uses singular for a reason, and for us here as web developers this thing should be *web development*.

To do one thing really well means to really kneel into web development, to pierce through it, to fully understand it, to practice it, to master it, to do it really well. Optimizing CSS is part of writing CSS well is part of doing web development well, and that's why this mindset is important for us. The goal to do web development well is an idea that serves us in our work.

Know Your Needs

An idea occasionally forgotten, and not strictly followed, is the one of knowing our needs—of knowing our needs *well*. It's actually the idea of knowing our *goals*, or having goals in the first place—because our goals likewise determine the specific needs we have.

Knowing our goals and needs is important because evidently, it's difficult to meet a goal when we don't know it, and to satisfy a need that we're ignorant of. In the web development world, there are many goals and needs we may have: from most important goals like how to maintain our projects (from a destructive "fire and forget"[12] to a very efficient process of iteration) to clearly quantified performance, and from basal needs like frameworks that cover our design ideas but don't mean bloat[13], to decisive ones like automation tools.

[11] https://www.google.com/about/philosophy.html

[12] https://meiert.com/en/blog/fire-and-forget/

[13] http://www.oreilly.com/web-platform/free/book-of-html-css-frameworks.csp

Stay in the Present

Without becoming overly self-helpish or even philosophical, another useful mindset for us is to also stay in the present, to stay in the now. It means both a rigid eye to get rid of, that is, archive everything—concepts, documentation, patterns, code snippets, libraries, &c.—that we don't need and use anymore ("Know Your Needs"), and not bother prematurely coding (or planning to code) something that we don't know to come *with certainty*.

This mindset matters because it keeps our code base and our documentation clean, and hence significantly contributes to our focus ("Do One Thing Really, Really Well", "Keep It Simple").

Keep It Simple

The idea to keep it simple has a lazy touch, and yet it requires high skill. Keeping it simple really means *focus*, to know what's important, and *through that* be efficient. Keeping it simple does not mean to simply do less; it means to *do what matters*.

As such, keeping things simple requires that we thoroughly understand our field—the core technologies, first and foremost, but everything in between and to the sides as well. And it means true economy of motion, because we don't want to engage in something that doesn't need to be done.

As a mindset, keeping it simple is truly grandiose, and the interested reader may not want to pick up a book on minimalism but rather one about focus—my recommendation is Gary Keller's *The One Thing*[14].

Automate

Lastly, a most powerful mindset as loosely as we're working with them here, is the one to automate. Every step of our work should, if possible, be automated. Sometimes that will be obvious—every time after we made CSS changes we run some optimization script—, sometimes that will require a bit of listening—a team member mentions how he manually updates documentation each time someone reports downtime issues—, and at other times it will be obscure—no one even knew that there was the option to automate visual design regression testing.

Whenever we repeat something in our work, we should look into whether that repeat work can be handled through a script or tool—no matter how "hacky" the solution may be. (Personally, I work with a recurring reminder to review my work and check for automation options.) And then it requires attention, for automation is powerful. And still it doesn't "take work away"—but liberates us to do the more important things.

[14] https://www.the1thing.com/

Operational Optimization

In this first section, on operational optimization, we're going to talk about the options we have to write higher quality CSS while we're right at it—in operation. (The second part will deal with options to optimize for production, that is, for release and live use.)

Just as it didn't seem useful to cover preprocessors so to get a clear look at CSS itself, we're not going to discuss documentation, as with in-file comments, nor guidelines that we can enforce in an automated fashion. Documentation is an entirely different topic that has more to do with maintenance than with optimization, and coding guidelines, as important as they are and as many of them we can set up, are often a matter of preference that doesn't necessarily have a bearing on quality (also see: *The Little Book of HTML/CSS Coding Guidelines*[15]). We're already applying two of our mantras: keeping it simple, and automating.

Understandability

In operation, we need to make sure our work is understandable. This does not only refer to multi-person environments in which we're not alone working on a project; with consistency, for example, there's at least individual consistency, which means to write code consistent with ourselves.

This understandability, as we established for documentation and comments, can in parts be achieved automatically. But some pieces, and we'll right start with one (consistent declaration sorting), are so important and useful to learn how to do manually, that we'll still go over them. And so we'll first look at consistency and simplicity as the two pillars for more understandable, for simplistically optimized CSS code.

Consistency

Consistency in the case of code means to write and format things the same way every time. With individual or "level 1" consistency[16], this means to be consistent with how we ourselves write code. With collective and "level 2" consistency, we strive to stay consistent within the realm that we work in, as when we touch third party code and stick to their code style. And then there's institutional or "level 3" consistency, referring to being consistent with coding standards put up by our organization.

Consistency is a *foundational* part of optimization; it's the first step of optimization. Without any sort of consistency, any other optimization attempts are a lot harder.

[15] http://www.oreilly.com/web-platform/free/little-book-html-css-coding-guidelines.csp
[16] https://meiert.com/en/blog/consistency-levels/

```
.feeds ul,
.posts ul ,
.events ul {
  list-style: none;
  margin: 0
}

.feeds li,
.posts li,
.events li {
  border-top: 1px solid #eeeeee;
  padding: 0.7667em 0;
}

.feeds li:first-child,
.posts li:first-child,
.events li:first-child {
  border-top: 0;
  padding-top: 0;
}

.feeds li:last-child,
.posts li:last-child,
.events li:last-child {
  padding-bottom: 0;
}

.authors > ul > li > a {
  margin-bottom: 1em;
  display: inline-block;
}

.authors ul {
  list-style: none;
  margin:0;
}

.authors li {
  padding: .7667em 0;
  border-top: 1px solid #eee;
}

```

```
.authors li:first-child {
   border-top: 0;
   padding-top: 0;
}

.authors li:last-child {
  padding-bottom: 0;
}
```

Example: Random sample code.

```
.feeds ul,
.posts ul ,
.events ul,
.authors ul {
  list-style: none;
  margin: 0;
}

.feeds li,
.posts li,
.events li,
.authors li {
  border-top: 1px solid #eee;
  padding: .77em 0;
}

.feeds li:first-child,
.posts li:first-child,
.events li:first-child,
.authors li:first-child {
  border-top: 0;
  padding-top: 0;
}

.feeds li:last-child,
.posts li:last-child,
.events li:last-child,
.authors li:last-child {
  padding-bottom: 0;
}
```

```
31
32  .authors > ul > li > a {
33    display: inline-block;
34    margin-bottom: 1em;
35  }
```

Example: Adjusted.

Consistency is, in theory, reasonably easy to achieve. We establish coding guidelines[17], we use (or build) tools to help follow and test for the guidelines, and then we enforce the guidelines. This goes as far as that many guidelines can be enforced right after writing and editing our CSS, and then again for production, where we may apply slightly different rules particularly geared towards production.

We'll cover this last step in the chapter "Production Optimization" and go over some tools under "Tools and Resources." We'll spare ourselves from going over often subjective coding guidelines and how to automate their implementation and enforcement. What we'll do is cover select aspects of consistency that are of particular import to CSS optimization. One is automatable; the other isn't: declaration sorting and selector sorting.

Declaration Sorting

This is trivial and at the same time automatable: *Sort declarations alphabetically.* As Google advocates—disclaimer: I had been involved in setting up respective guidelines—, the only exception are vendor-specific extensions (self-destructing declarations that start with hyphens) which are located right before respective declaration to complement.

```
1   .example {
2     background: none;
3     border: 1em dotted #069;
4     color: #096;
5     display: block;
6     -moz-filter: blur(33.35px) sepia(0.34);
7     -webkit-filter: blur(33.35px) sepia(0.34);
8     filter: blur(33.35px) sepia(0.34);
9     float: none;
10    font-size: 1em;
11    font-style: italic;
12    height: 100px;
13    margin-top: 1em;
14    max-width: calc(100vw - 10em);
15    outline: 2em solid #609;
```

[17] http://www.oreilly.com/web-platform/free/little-book-html-css-coding-guidelines.csp

```
  overflow: auto;
  padding: 1em;
  position: relative;
  text-align: center;
  top: 1em;
  white-space: pre-wrap;
  width: 100%;
  z-index: 1;
}
```

Example: Quick, where do we add `transform: rotateY(10deg);` so that someone else can spot it as quickly?

This is trivial and automatable but still, in my eyes, one of the key optimization methods. That is so because an almost failsafe, easily repeatable, soon habitual, quickly communicable, and quite universal method to structure our code is something that has tremendous value. A simple and robust sorting scheme like the alphabetical ordering of declarations will at once make our code more understandable and help anyone touching it navigate in it.

I intentionally raise alphabetical sorting to "optimization status" because of its many benefits; our code will be better once we structure it, and we'll write better code once we can do it ourselves, and not require some processor to do that for us. (Not all CSS we'll touch will have a script behind it that sorts for us, so we strongly benefit from internalizing this way of sorting.)

Sort declarations alphabetically.[18]

Selector Sorting

Selector sorting, then, is the antithesis to declaration sorting because it's far less trivial to standardize and likewise difficult to automate. (On a complementary investigation it seems the author's own draft[19] is the only discoverable attempt for an order.)

Yet selector sorting is the next impactful method at our disposal to make our style sheets consistent and, in a way, "optimize them for further optimization." When it comes to maintenance, for example, a defined and followed selector order is key to successfully DRYing up (from "Don't Repeat Yourself," to keep the cost of maintenance low) our style sheet declarations, because it's what ever helps us to avoid an extra round of DRYing *selector groups* and spares us from great additional testing complexity.

[18] https://meiert.com/en/blog/on-declaration-sorting/
[19] https://meiert.com/en/blog/how-to-order-css-selectors/

```
html {
  font: 87.5%/1.5 'helvetica neue', helvetica, sans-serif;
  max-width: 600px;
  padding: 1em;
}

footer {
  border-top: 1px solid #eee;
  margin: 2.5em 0 0;
  padding: 3px 0 0;
}

h1 {
  font-weight: 300;
  margin: 0;
}

ul {
  list-style: none;
  margin-left: 0;
  padding-left: 0;
}

a {
  color: #e30613;
  text-decoration: none;
}

a:focus,
a:hover,
a:active {
  text-decoration: underline;
}

strong,
em {
  font-style: normal;
  font-weight: 600;
}

.info,
#intro,
```

```
#error {
  border-radius: 5px;
  padding: 5px;
}

.info {
  background: #fff3ca;
  margin: 1em -5px;
}
```

Example: High level block elements, at some point text level elements, then classes and IDs, &c.[20]

Although it seems that many generations of web developers and teams of web developers have survived without a firm idea of how to arrange selectors, only when we have some sense of order can we truly get to consistent style sheets (granted that this is our goal), and, more importantly, do we ever have a chance of eliminating possible extra work through haphazard by-chance ordering. As we'll see with the avoidance of repetition, that extra work is otherwise actually awaiting us.

Accordingly, the lack of a consensus on selector sorting, and particularly the lack of *options* for selector orders should give us much to think, and something important to work on community-wise. If you're in a position to do so, consider what we do have and help us come up with additional options and, perhaps, standards.

Simplicity

Making code simpler is our next optimization goal. Alas, that goal of code simplicity is often the goal of code minimalism, and simplicity and minimalism don't necessarily correlate—they don't correlate in our coming case of shorthands, indeed not, but they do when we speak about character minimization as part of production optimization.

And yet, the goal of simplicity, just seen as one, is important: Optimizing for simplicity appears to make code more understandable and our work easier, and it seems to challenge ourselves as craftsmen to use the least amount of code possible.

ID and Class Naming

In the times of a still surprisingly alive OOCSS and BEM and Atomic CSS, ID and class naming has drowned as a developer topic, sinking to a second order afterthought not paid much attention to. And yet when we consider that not all web projects are super-complex mega sites with internationally distributed teams of dozens of developers and third party agencies in automated testing and deployment environments so to warrant presentational naming schemes because our

[20] https://meiert.com/en/blog/how-to-order-css-selectors/

guideline and communication processes are so difficult to implement that no hand knows what the other hand does—indeed—, we don't need nor at all benefit from naming schemes that sacrifice understandability and maintainability for brief presentational class soups well matching the 90's (Atomic CSS). The times of knowing how to name IDs and classes, and to actually use *both* are not over yet—IDs aren't to be shunned (just as little as universal selectors, `!important`, and shorthands), as has all been attempted in the past.

The rules for ID and class naming[21] are simple and lasting:

- keep use of IDs and classes to a *minimum*;
- if needed, use *functional* ID and class names;
- otherwise use *generic* ID and class names; and
- use names that are *as short as possible but as long as necessary*.

This doesn't aim to rule out other naming schemes, but it's the starting point for every web developer. It's useful to learn and apply these rules first before switching to a scheme that violates them, because only when we have the experience of developing and maintaining web projects will be able to tell whether the violation is smart and justified, or haphazard and premature.

Even though the topic of ID and class naming is a beginner topic it's also much neglected, and that's why it's so useful to be discussed here: An otherwise optimized style sheet that uses poor ID and class names is still a poor, that is, poorly crafted style sheet.

Shorthands

Shorthands—CSS declarations that combine other declarations, like `font`, `border`, and `animation`—are a very useful part of CSS because they help to make CSS more minimal-simple. They allow us to write less code though not necessarily more understandable code—a situation that perhaps made one camp declare that CSS shorthands were an anti-pattern[22], and the other say they weren't[23].

Modifying what I put to protocol as a part of that other camp, they're both.

Shorthands make code less understandable in complex projects, but make code more minimal. What tips the simplicity scale in my eyes is the fact that shorthands *always* make our CSS more minimally simple, whereas *only in large and complex projects*, they make it less understandably simple because they imply too much: Everything a shorthand declaration doesn't say still means something, because the values that aren't set are really set to their initial values. And that can—in larger projects—lead to problems.

[21] https://meiert.com/en/blog/best-practice-ids-and-classes/
[22] https://csswizardry.com/2016/12/css-shorthand-syntax-considered-an-anti-pattern/
[23] https://meiert.com/en/blog/css-shorthands/

```
html {
  background: #fff;
  font: 100%/1.88 palatino, lora, georgia, cambria, serif;
}

body {
  margin: auto;
  padding: 1.25em 1em;
}

#announcement {
  background: #ff0;
  margin: 1em auto 1.5em;
  padding: .5em;
}
```

Example: In this shorthands-only piece from an actual website[24], the shorthands simplify the code and only help.

For their value in making code more compact, their positive effect on small projects, and the few tools to automate shorthands, we benefit from using shorthands in our code, and with that optimizing for them.

Performance

Performance is one of the most obvious goals to optimize for. The faster, the better, because we know that the user experience improves the faster everyone gets what they want[25], and that with more speed, conversions increase as well[26].

And yet what we can roughly say is that improving rendering performance is not nearly as effective and important as is improving loading performance. That is, not omitting optional tags for the reason that the browser would otherwise need to "put them back" is not nearly as helpful for performance as is compressing images. These calculations are generally done so quickly that respective issues don't matter (even though personally, as I do when coding without optional tags and without unneeded quotes around attribute values, we may ignore such non-needs).

The cases that we'll look at now, still in our section on "operational" optimization, will exemplify a bit of this situation. Performance is not nearly as clear cut as it's sometimes presented.

[24] https://mirrors.meiert.org/coderesponsibly.org/
[25] https://www.nngroup.com/articles/website-response-times/
[26] https://blog.kissmetrics.com/speed-is-a-killer/

Irrelevant: Selector Performance

Selector performance is *not* something to optimize for. Selector performance is irrelevant. The price we pay for optimizing for selector performance is, indeed, terrible: We micro-manage our work for gains that aren't even noticeable.

In 2009, Steve Souders contributed perhaps the first important insights into the topic, and one of his main articles[27] on the subject should have made more of us think. First, in his tests, even style sheets with 18,000 rules (!) were rendered within 600 ms (!). Then, Steve explained in quite clear terms, his hypothesis: "For most web sites, the possible performance gains from optimizing CSS selectors will be small, and are not worth the costs."

Perhaps people misunderstood when Steve went on to write, "There are some types of CSS rules and interactions with JavaScript that can make a page noticeably slower. This is where the focus should be."—What we saw, namely, was an undue focus[28] on selector performance and an outright childish ban of selectors, notably the universal selector.

(Years later, in additional investigations, we even find problems with the methodology, leading to important observations like this one, by Ben Frain[29]: "It is practically impossible to predict the final performance impact of a given selector by just examining the selectors.")

The first problem with optimizing for selectors is that the gains are negligible. Yes, some selectors are slower than others, and from the way they work this is probably going to be the case until the world ends. Alas, nothing practically relevant follows from this observation: The effects are not felt by us nor by our end users. We do not ship websites that use 25,000 rules of the type of `:after` (and nothing else) to bring pages to a crawl; a) they don't come to a crawl, b) who writes code like that has other issues.

The second problem with optimizing for selectors is that we over-optimize and micro-manage, and actually slow down our workflow writing worse CSS. Putting up rules in place for how *selectors* should look like is adding a cognitive restriction that only hinders a team of developers, and the outcome is not any more elegant code in terms of code minimalism (though those of us who connect verbosity with understandability may have a point).

Where we are right now, selector performance is still a box better left closed.

Irrelevant: Inline CSS

Similarly, though practically speaking much more legitimate, inline CSS—applying CSS directly through `style` attributes—should also be avoided. This recommendation comes again from the idea of keeping the big picture[30] in mind, by acknowledging that yes, inline CSS can be useful for

[27] http://www.stevesouders.com/blog/2009/03/10/performance-impact-of-css-selectors/
[28] https://meiert.com/en/blog/performance-of-css-selectors/
[29] https://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/
[30] https://meiert.com/en/blog/big-picture-thinking/

performance reasons (saving one or more style sheet HTTP requests, applying faster selectors), but no, it violates separation of concerns and makes maintenance harder. Too hard.

It is important to me to drive this point home, and to risk making this book a lot more complicated: We are of no good use as web developers when we are blind to the consequences of our work. An accessibility expert, a JavaScript expert, a performance expert are all still rather poor web developers if they don't understand *and factor in* what their expert knowledge means for other areas of the field.

```
<div style="display:none" jsl="$t t-aTz9-_sUcEc;$x 0;" class="r-ild1JbEZKhjg"></\
div><div id="duf3-46" data-jiis="up" data-async-type="duffy3" data-async-context\
-required="type,open,feature_id,async_id,entry_point,authority,card_id,header,su\
ggestions,surface,suggestions_types,suggestions_subtypes" class="y yp"></div><a \
class="duf3 _sWr" href="#" id="sbfblt" data-async-trigger="duf3-46" jsaction="as\
ync.u">Report foo</a></div></form></div><div class="sfbgx"></div><div id="gac_sc\
ont"></div><div class="spch s2fp-h" style="display:none" id="spch"><div class="s\
pchc" id="spchc"><div class="_o3"><div class="_AM"><span class="_CMb" id="spchl"\
></span><span class="button" id="spchb"><div class="_wPb"><span class="_AUb"></s\
pan><div class="_Fjd"><span class="_oXb"></span><span class="_dWb"></span></div>\
</div></span></div><div class="_gjb"><span class="spcht" id="spchi" style="color\
:#777"></span><span class="spcht" id="spchf" style="color:#000"></span></div><di\
v class="foo-logo"></div></div><div class="_ypc"><div class="_zpc"></div></div><\
/div><div class="close-button" id="spchx">&times;</div></div><div style="display\
:none" jsl="$t t-orNZyHXTT74;$x 0;" class="r-iCneKvRyCT78"></div><div class="con\
tent" id="main"><span class="ctr-p" id="body"><center><div style="height:233px;m\
argin-top:89px" id="lga"><div style="padding-top:109px"><style>#hplogo{backgroun\
d:url(/images/branding/foologo/2x/foologo_color_272x92dp.png) no-repeat}@media (\
-webkit-max-device-pixel-ratio:1),(max-resolution:96dpi){#hplogo{background:url(\
/images/branding/foologo/1x/foologo_color_272x92dp.png) no-repeat}}</style><div \
style="background-size:272px 92px;height:92px;width:272px" title="foo" align="le\
ft" id="hplogo" onload="window.lol&&lol()">
```

Example: Can these styles lie?

As such, yes, inline CSS is good for performance, and as such we should consider it a CSS optimization measure. But unless, perhaps, we deal with truly unique styling on ever unique single pages, a categorical *"no"* to use inline CSS. Our vision of web development, to write the leanest possible HTML for maximum freedom of movement in terms of updates and maintenance, forbids this. We don't want to touch HTML for CSS updates—and clearly, inline CSS prevents that.

With improvements to our content management, build, and deployment processes, HTML maintenance has become a lot easier and cheaper. We've gotten to a point where we can speak of a new paradigm[31] for web development, where the first paradigm, absolute separation of concerns, has been weakened decisively. That is development-practically speaking a helpful and development-theoretically speaking a fascinating development. For the time being I still recommend to follow the first paradigm and separate concerns (like structure, presentation, behavior).

Possibly Relevant: Declaration Prudence

For declarations we face a similar situation as with selectors in that avoiding some of them seems to cause more harm and be more cognitively demanding than not doing anything (or to continue doing what we used to do).

With declarations, we are reasonably certain that declarations like `box-shadow`, `filter`, `opacity`, and `transform`, plus some particular values, are *significantly* slower and hence more expensive than others. So much more expensive, indeed, that the effects can be perceivable even with few declarations, hence making this a much more important point on our optimization agenda than selectors.

However, the problem is that data are hard to come by. Ben Frain has contributed some data on selector and declaration performance[32], and so has "kangax,"[33] alas the data are not robust enough or not available anymore (I could not reach kangax for clarification).

Until we do more research and can properly, reliably document all the declarations and value ranges that are particularly slow, there's no point in panicking and no point in blindly guessing what declarations should be avoided.

There may not even be a point taking any action at all, because at the end of the day, we'll still need to look at the cases in question: How important is respective styling, and what's the performance impact of its replacement? In this case, then, it seems advisable to me to flag declarations as a possible optimization source, but not make it a priority—until we have better data.

Rule Hygiene

Where we can finally start optimizing for performance is with always removing anything in our CSS that isn't needed anymore: rule hygiene. This sort of hygiene is simple in theory—just get rid of what isn't needed—but tricky in practice.

The trickiness consists in the fact that it's often hard, if not impossible, to tell what rules truly aren't needed. There are tools for this purpose, but it's difficult to be certain of where styles are used, and

[31] https://meiert.com/en/blog/two-paradigms/

[32] https://benfrain.com/css-performance-revisited-selectors-bloat-expensive-styles/

[33] http://perfectionkills.com/profiling-css-for-fun-and-profit-optimization-notes/

often not feasible on the really large websites—on sites so large that they have similarly looking marketing sites run by third parties that secretly hot-link the mothership's main style sheets. On top of that, it's difficult to automate the procedure.

Generally speaking though, what helps this hygiene is the following:

- Discipline to remove page elements
- Diligence to clean style sheets
- Section and general code comments
- Descriptive, at least direct selectors (`footer` instead of something like `body > :last-child`)
- Tools
  - Chrome Developer Tools[34] browser extension audit tab
  - Dust-Me Selectors[35] Firefox browser extension
  - Unused CSS[36] crawler
  - PurifyCSS[37] script
  - UnCSS script[38] and online tool[39]

The Unused CSS crawler goes furthest where we actually want to go, to have something that tells us with considerable certainty what is used and what's not. (It has become one of my favorite tools for CSS rule hygiene optimization.)

Images have always been a matter of optimization on the Web. At first they hadn't been recognized as something to optimize for and rather, as the infamous "spacer GIFs," served the systemic *deterioration* of websites by being used endlessly to build entire layouts, then layout tables (though in a way, they only served optimization of said layouts). But then our attention was on

- what formats to use for quality and compression ("GIF or JPG?");
- how to compress images;
- how to limit the number of images (and the number of HTTP requests, through sprites and such);
- how not to use images (data URIs);
- and again, what formats to use ("…or SVG?").

These topics, without the redundancy of the formats and compression questions, are exactly the ones we should still focus on today. But although I've debated to make image optimization a part of this book, it's not *CSS* optimization—it's image optimization. And Addy Osmani has just written such a book[40].

[34] https://developer.chrome.com/devtools
[35] https://addons.mozilla.org/en-US/firefox/addon/dust-me-selectors/
[36] https://unused-css.com/
[37] https://github.com/purifycss/purifycss
[38] https://github.com/giakki/uncss
[39] https://uncss-online.com/
[40] https://images.guide/

Quality

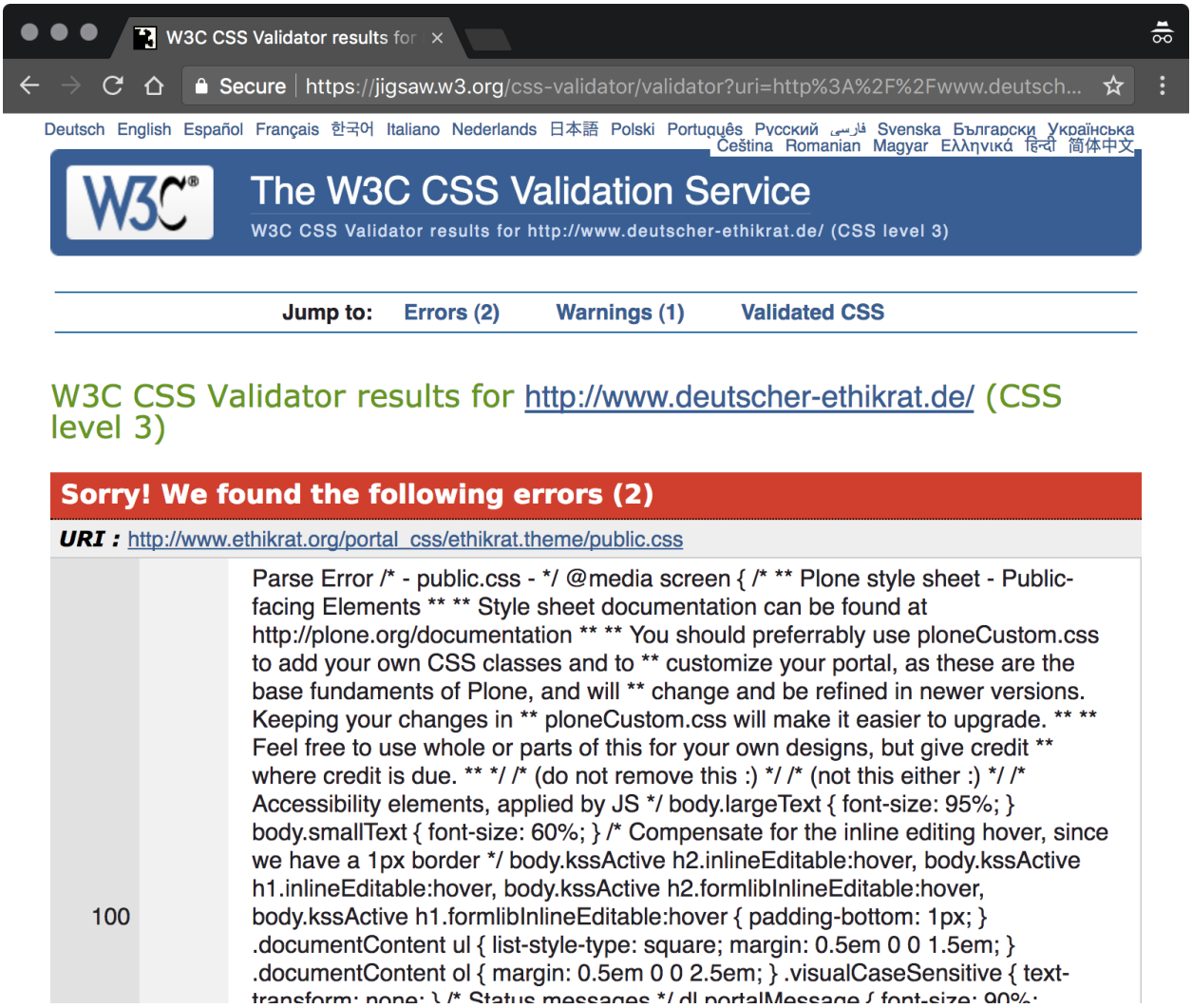
On the surface, optimization seems to have an obvious goal of increasing quality, but, and my own knowledge of the philosophy of language leaves me here, optimization seems to have value just as a process, on its own. One can optimize something, for something, and something can have a quality of something, but not of something else.

Generall speaking, we cover the general optimization and the general quality of CSS; all that we've established about big picture thinking supports this notion. And yet quality should be called out explicitly, as our general goal. And for that goal, there's one particular aspect to pay attention to.

Validation

How can we tell that what we're coding works? We test. How can we tell it's error-free? We validate. Yes, in a way testing already exposes errors in our code, and on the other end we depend on validators to actually be up-to-date on the latest and greatest. And still, validation is a crucial part in making sure that our code is syntactically correct and uses selectors and declarations (CSS) as well as a structure and elements and attributes (HTML) that actually exist.

There is a gray area when respective working groups just came up with new things (that is, just extended the specifications), and the validator teams couldn't catch up yet, but that gray area must not prevent us from validating. That would be like only using cellphones whose batteries are fully charged, or only wearing clothes are brand new.



"We found the following errors."

Example: Issues.[41]

Through validation—whether using the W3C CSS validator[42] or some other tool[43]—we optimize our style sheets because we can correct or remove code that doesn't work. We also benefit through, and I've once called these the actually two great things[44] about validating, getting a better technical understanding (validation issues can be quite informative and instructive) and with that becoming better professionals (everyone can write poor and invalid code—we're experts because we can and we do write valid code).

[41] https://jigsaw.w3.org/css-validator/validator?uri=http%3A%2F%2Fwww.deutscher-ethikrat.de%2F&profile=css3&usermedium=all&warning=1&vextwarning=&lang=en

[42] https://jigsaw.w3.org/css-validator/

[43] https://uitest.com/en/analysis/#validation

[44] https://meiert.com/en/blog/about-validation/

Maintainability

Optimizing for maintainability means to make sure that everything is done to make maintenance as easy as possible. It starts with making sure to avoid *unnecessary* work, and surprisingly, this begins with the mundanest of things—proper style sheet naming. We won't discuss such simple things here, but let's all keep an eye on the senseless activity of changing style sheet names and such. When we're doing that, we're not taking care of a maintenance task—we, excuse the language, just f'ed up one of the simplest things there is in web development. (Defensive caching assumed, let's just use functional or generic names. Personally, I've never in my 20 years career changed a style sheet's name once it was just called "default.css"—even on heavily frequented Google sites.)

Irrelevant: `!important`

Similar to performance, there's some trash to bring out here. Contrary to some developers' views, there's nothing harmful, and nothing dangerous, and neither anything unmaintainable about using `!important`. `!important`, a legitimate piece of the cascade[45], is a tool.

As with every tool, one can use it wrong. Using a knife to eat a soup is frustrating. Using a hammer to solder circuit boards is very difficult. Using `!important` to fix every layout problem is like using pesticides against bugs around crops: It's tough on the environment.

Alas, now, that doesn't mean one shouldn't use knifes, hammers, or `!important`. What it means is to learn when it's best and most effectively used.

In essence, let's not refrain from using `!important`. Let's not shy away from using it for quick *testing and debugging*. Let's well use it for declarations, then, that come early in our style sheets but that are rather unspecific, and hence face a risk (or probably just suffered from) being overridden in some way. *That's not bad development practice. That makes sense.*

`!important` is fine.

Separation of Concerns

Separation of concerns is one key for maintainability. It's defined as follows in Wikipedia[46]:

> In computer science, separation of concerns (SoC) is a design principle for separating a computer program into distinct sections, such that each section addresses a separate concern. A concern is a set of information that affects the code of a computer program. A concern can be as general as the details of the hardware the code is being optimized for, or as specific as the name of a class to instantiate. A program that embodies SoC well is called a modular program. Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface.

[45] https://www.w3.org/TR/CSS22/cascade.html#cascade
[46] https://en.wikipedia.org/wiki/Separation_of_concerns

The first and most important thing to connect with separation of concerns is to strictly separate structure (HTML), presentation (CSS), and behavior (JavaScript, a term I still use over ECMAScript).

Although macro-separation, it makes sure that we develop using the appropriate technology (in the 90's and 2000's we've seen what solving design issues in HTML leads to) and know where to look when there are issues (we don't want to fix document-structural issues in JavaScript).

For CSS that means to look beyond and make sure to collect everything that's related to the appearance of our site or app and have it in our style sheets. In a way separation of concerns is the basis for all optimization, because only that way do we know that our CSS is actually all handled by style sheets (and not, as it happens, in HTML templates or JavaScript files).

On a micro and intra-style sheet level, there are now two options for separation of concerns.

One is to separate all styles by modules (functionality), or perhaps, blocks, or elements (BEM), and to keep them separate. (This goes in the direction of the single responsibility principle[47].)

Two is not to do anything on top—as we don't have to do anything more if we're sorting selectors consistently (such sorting will *inherently* lead to a modular order within our style sheets), and especially not *if* we DRY our style sheets entirely. As we'll see, we can DRY up individual CSS modules, but we can also just DRY them up on the whole, something for which module separation is rather inconvenient. (This is another example for where our work requires some feeling for balance and priority—we may well decide for a different course, especially when our projects are of large size.)

Be cautious around OOCSS[48] (object-oriented CSS), BEM[49] (Block Element Modifier), Atomic CSS[50], and similar conventions. Look very closely how they help, and how they hinder you.

In a basic sense, what they do is address some of the problems of *complex* web development, but they also curb freedom and possibilities. In some cases (Atomic CSS), they make only for fancy beginner-style code.

When we're asked to avoid the descendant selector (OOCSS), then great, no worries about inheritance (that is the TSA writing CSS right there)—but also no incredible elegance through contextual styling. When we shall avoid nested selectors (BEM), but "in this case [Christmas example] they are reasonable," then we are not better off than before. When we must memorize more than 40 new classes (Atomic CSS) only to write the most presentational markup (which, by the way, is the opposite of separation of concerns), then we negate quite *all* the advantages of CSS and separation through a single convention.

[47] https://en.wikipedia.org/wiki/Single_responsibility_principle
[48] https://www.slideshare.net/stubbornella/object-oriented-css
[49] http://getbem.com/
[50] https://acss.io/

Don’t Repeat Yourself

The next maintainability pillar to optimize for is to avoid repetition. When we think about selectors in terms of *grouped* selectors (that is, `section` to be different from `section, div`), and when we assume declarations to be written so consistently as to not inadvertently repeat through different spellings (compare `outline: 0` and `outline: none`), there really is only one way to avoid repetition: through avoiding repeat *declarations*.

Most of the time that makes our style sheets more compact and also more maintainable, because we limit the number of places where a particular style is defined.

Using Declarations Just Once

“Using declarations just once” is an old technique. It means what is says: To focus on using each declaration—if coding guidelines have assured us to always write them the same way—just a single time.

It’s a tangible optimization step in that we’re still encouraged to write CSS the natural way[51], to then go over what we’ve written to make sure we leave no repetition behind. That process is roughly as follows[52]:

- Write CSS, the natural way.
- Decide on DRY boundaries: section (functionally separate CSS parts) or file/`@media` level?
- Make sure to format code consistently, as `background: none;`, `background:none;`, or `background-image: none;` could all mean the same but make our task of finding duplicates unnecessarily complicated.
- Search for duplicate declarations:
  - For new style sheets: after initial setup is done.
  - For new features and bug fixes: after respective work is done.
  - Tip: If version control highlighting for file changes is not enough, temporarily indent changed declarations to only check for their repetition.
- Dissolve duplicate declarations:
  - Check each declaration (in new style sheets) or each changed declaration for re-occurrence within the set boundary (when limiting de-duplication to sections, take care to limit search scope to these sections).
  - For each duplicate declaration (the actual work):
    * Determine which respective rule should come first in the style sheet (for this one has to have an unwritten or written[53] standard for how to order selectors).

[51] https://meiert.com/en/blog/css-dry-and-optimization/
[52] https://meiert.com/en/blog/dry-css/
[53] https://meiert.com/en/blog/how-to-order-css-selectors/

        * If this first rule contains additional declarations, i.e. declarations that we haven't checked yet or that aren't duplicates, copy the entire rule and paste it after the original; keep the discovered duplicate in the first rule and remove the other declarations, and vice-versa in the second rule, so that that rule is like the old rule just without the declaration we found to be used more than once.
        * Copy the selectors of the *other* rules that contain respective duplicate declaration to the rule that comes first.
        * Make sure to remove the duplicate declarations whose selectors have just been copied up in the style sheet, and to remove the entire rule if the rule only consisted of the now moved duplicate declaration.
        * (Repeat.)
    - Make sure to check the correct *order of the selectors* for the rules that now handle the formerly duplicate declarations.
    - Make sure to check for the correct *placement* of the rules that now combine formerly duplicate declarations.
- (Repeat.)
- Test.

This may seem intimidating, but is rather precise; and it's much work only when we just wrote and optimize an entire style sheet. For small updates, for example, the process is considerably easier and much more grateful.

What we receive at the end is a style sheet that is, in many cases[54], lighter than what we started with; but as file size is not always a factor for performance (mainly thanks to compression), the main benefit is with improved maintainability: We end with style sheets that are more compact, easier to understand, and easier to manage. So much, indeed, that this optimization step may mean that we don't need something like variables, because when unlike the average website[55] we don't repeat each declaration almost four times, we don't face the issues of such repetition.

Try this process on for size, and apply it to CSS sections first (that is, avoid declaration repetition within functional blocks, as with page styles, navigation styles, login styles, however the blocks are divided). This way it's easier to get acquainted with the process, get a moderate result, and also find a way to feed back to the community your very own ideas on DRY CSS.

*
**

We're concluding the section on operational optimization. What we've covered are all things we need to focus on while we're working on style sheets. Most of that could not be automated (with the exception of unintermittent controls that inform us if, say, our style sheet contained too much repetition, or if we were about to submit something invalid). But even if it could all be automated, I

[54] https://meiert.com/en/blog/70-percent-css-repetition/#toc-example
[55] https://meiert.com/en/blog/70-percent-css-repetition/

believe it makes sense for us to internalize and live what we've just discussed, for it to improve our code and us as professionals.

That exception there in parentheses, about unintermittent controls, looks innocent but somewhat hints at the future of professional development: automated live feedback. Likely an area for modern editors, this could mean to immediately provide notifications on unused code, inconsistencies, redundancies, validation issues, &c. The way seems long at this point—we'll need great UI and AI features to make it work effectively, from easy ways to mark false positives to train the software false negatives—but it's what's rather glaringly missing from the current way of development, that only checks code once we checked it into a repository or deployed it to a staging or production environment.

Production Optimization

For production, we usually take care of all the things that are good for machines. I've organized this section so that it also contains the things that can be done *by* machines (tools)—without, again with an eye on us as professionals, including anything that, if we were to miss it, would make us produce significantly worse code.

It turns out, then, that we do again benefit from trying to follow some of the (human-friendly) ideas ourselves. For example, we can well try to write minimal CSS (like omitting leading `0`'s), or to only link one style sheet (to keep the number of requests low and to avoid future HTML changes), without leaving this to tools.

Performance

Performance is one of the primary optimization goals, and again *rendering* performance is not the subject here for its often marginal achievements and yet grave practical impediments.

As long as our code is consistent, the steps suggested here can all be taken care of right before production, automatically, and they don't have to find their way back into the operational repository. The exception is what we'll start with, because it can first be covered by our code style guidelines and all get handled manually.

Character Minimization

To make our style sheets more compact, that is, make them easier to read and smaller in size, we remove all unneeded characters. That doesn't refer to whitespace and comments yet—let's make this a separate step, although both can be covered by the same tool—, but to removing any CSS that isn't strictly needed. This entails:

- semicolons after the last declaration of each rule;
- leading `0`'s (`margin: .1em`, not `margin: 0.1em`);
- unneeded trailing `0`'s (`line-height: 1`, not `line-height: 1.0`);
- color notations to use 3-digit hex, where possible (`#fff`, not `#ffffff` or `white`);
- value shortening (`border: 0` instead of `border: none` but also (these steps aren't trivial!) `font-family: arial, sans-serif` instead of `font-family: arial, helvetica, sans-serif`—the fallback font is good practice but Helvetica, here, would never be used).

As noted in the introductory section, these items can well be covered by coding guidelines but they're also easy enough to automate and run through just before pushing style sheets to production.

Code Minimization

After we've minimized our style sheet code without impairing understandability, the next steps means to remove all the characters that aren't needed for the style sheet to work—this makes it production-ready.

Note again that this is only presented as a separate step here to *illustrate* that we can remove certain characters manually—perhaps our style guide requires us to, where possible, use 3-digit hex color values—and still are able to work with our style sheets. As a true manual task it would make style sheet maintenance rather cumbersome if not practically impossible.

What's done here are typically two things:

1) remove all comments, 2) remove all (non-required) whitespace characters.

```
1  html {
2    font: 87.5%/1.5 'helvetica neue', helvetica, sans-serif;
3    max-width: 600px;
4    padding: 1em;
5  }
6
7  footer {
8    border-top: 1px solid #eee;
9    margin: 2.5em 0 0;
10   padding: 3px 0 0;
11 }
12
13 footer small,
14 label {
15   display: block;
16 }
```

Example: Random non-minified CSS snippet.

```
1  html{font:87.5%/1.5 'helvetica neue',helvetica,sans-serif;max-width:600px;paddin\
2  g:1em}footer{border-top:1px solid #eee;margin:2.5em 0 0;padding:3px 0 0}footer s\
3  mall,label{display:block}
```

Example: Random now minified CSS snippet.

As mentioned in the beginning of this book, we benefit from automating our work—as this optimization step is ugly to do and ugly to work with we actually have something here that is almost *always* automated.

For both character and code optimization there are several tools and scripts on the market, for example:

- minifier.org[56] based on Matthias Mullie's Minify[57]
- YUI CSS Compressor[58] based on Yahoo's YUI CSS Compressor PHP port[59]
- CSS Minifier[60] by Andrew Chilton

File Normalization

Although it makes it more difficult to avoid declaration repetition (see "Using Declarations Just Once"), we can work with as many CSS or preprocessor files as we want. However, at the end, for production, we should make sure to combine them to a single file to load. That's important for more effective compression and even more so for fewer HTTP requests. (HTTP/2 alleviates[61] here but for this first edit, I wish to stick to advising for fewer requests.)

That's simplified but the basic idea behind this is that HTTP request overhead makes for 500–700 bytes (based on work from Steve Souders and Kyle Simpson) costing "about 100 ms" (Kyle Simpson).

This data has led others to recommend that files smaller than 1 KB should be inlined, and to avoid inlining of files that are bigger than 4 KB (Guy Podjarny).

We may say that most sites that use several style sheets (to then be combined to a single one for production) do work with files that are larger than 1 KB. As such it's useful to have these as individual files (and not inline), but for each style sheet we get rid of and merge we save roughly half a kilobyte and 10 ms. This leads to the general recommendation to merge all styles into one file and link that from our documents.

As we can tell, there must be something else here. What about tiny sites, and what about overall file size and caching?

Tiny sites, particularly one-pagers with just a few rules, may be under the threshold. If the site is and will be a one-pager forever we have a stronger incentive to inline all code, both styles and scripts. But that also depends on the page itself and our ideas about true separation of concerns. If the page resources are so few and so small that the extra CSS request is not felt anyway, then it may well be fine to stick with separation. Likewise if we strongly believe in separation of concerns, we might just always go for that separation (and what may sound a little rigid here I consider useful for its simplicity).

[56] http://www.minifier.org/
[57] https://github.com/matthiasmullie/minify
[58] https://hell.meiert.org/aux/compress/css/gui/
[59] https://github.com/tubalmartin/YUI-CSS-compressor-PHP-port
[60] https://cssminifier.com/
[61] https://http2.github.io/faq/#what-are-the-key-differences-to-http1x

File size and caching are normally the issues that are of more concern now. The concern boils down to the following two questions:

1) If all our styles are in one style sheet, but a user visits just one page, how can we justify pushing all the unused styles onto them?

(Per RocketFuel and Kissmetrics, the average bounce rate on the Web is around 50%—just to provide some number, as bounce rates actually vary heavily per field as well as per type of content.)

2) Same scenario, if all our styles are in one style sheet, what is a good balance to make sure that the style sheets get cached but we can swiftly roll out updates?

I set out to go over both questions in detail but I believe it to be more useful to leave them to everyone to decide on their own. This should be useful, too, to remind ourselves that even though technical questions often beg quite definite answers, there are questions that don't lead to a strict "right" or "wrong"—the answers often still depend on our priorities.

What's there to consider for our decision?

- For performance in terms of efficiency and file size, each page should only come with the styles it uses.
- For performance in terms of caching, each style sheet should be cacheable.
- For performance in terms of efficiency and file size if a user visits many pages (low bounce rate with visitors covering a high percentage of needed styles), a single style sheet is fastest.
- For HTML maintainability, style sheets should not be versioned manually (we don't want to update HTML code every time we update CSS code).
- For CSS maintainability, style sheets should not be broken up manually (we don‘t want to merge style sheets every time we update CSS code).
- For general maintainability, code should be kept simple.

et cetera.

I'll still offer a view. There are a few reasons for only providing the styles that are used on each page. This approach should be taken if 1) the project is large (perhaps >10,000 pages, or >10,000 declarations) *and* 2) it can be automated.

- index.css (or)
- contact.css (or)
- search.css (or)
- …

Example: Page-oriented, automated, cacheable style sheets carrying everything each page needs. (Bonus exercise not covered here: Dynamically load only the styles needed on subsequent page visits.)

In other cases it seems most effective to go for a single style sheet, whether to begin with (easiest to set up) or for production (in which case the merging of files should be automated, which can happen through something as simple as PHP-importing sub style sheets into a default.css—processed as PHP but returned as `text/css`, of course).

- default.css

Example: Just one style sheet.

And there are compromises (aka exceptions) to be used sparingly and wisely, like the approach taken with Google's Go framework[62]: Provide a small core style sheet sufficient for basic pages, a bigger one for more complex pages, and individual style sheets for custom sub sites and pages.

- go.css (or)
- go-x.css (or)
- default.css (importing either go.css or go-x.css)

Example: The approach taken by the Go framework.

At the end of the day there is no single answer for optimal CSS file management. Complexity is an important aspect, automation is an important approach, simplicity is an important criterion.

Output Control

As the final step as per this basic treatise, it's important to check what we're finally shipping. Did all our other optimization steps work? Have we missed anything?

Reviews and Sanity Checks

For the finishing optimization step we want to regularly employ code reviews and tests, and to run *both* manual and automated checks.

The reason to do automated checks is efficiency, as we don't want to spend any human time on constantly validating or otherwise confirming the quality of our style sheets.

The reason to do manual checks is not to miss anything glaring, issues that fly under the radar because we missed how, say, a preprocessor may have had a flag that left in comments, or that otherwise generates production output that isn't desirable.

[62] https://meiert.com/en/blog/google-web-frameworks/

- Automated tests: daily and on deployment
- Manual tests: weekly

Example: CSS routines.

The manual checks can be swift; gloss over the output, perhaps re-formatted to be legible again (tools like CSSTidy[63] allow to throw a CSS URL at them to be "uncompressed").

The automated checks ask for time to be set up properly and depend on one's priorities and needs; one popular way is to use Git hooks[64] to validate and lint on commit, as through scripts like Wouter Sioen's pre-commit[65].

These reviews wrap a nice tie around much of what we've discussed so far: We enforce our desire for quality (the reason why we optimize in the first place) through our development mindsets (like doing our work really well and automating it so to focus on the things that matter).

*
**

This concludes the overview on CSS optimization basics. We've covered the most important aspects, and more than basics; and I don't say that so to inflate the idea of "basics" but because pretty much everything else now depends on development paradigms, priorities, and the big picture.

That, then, I'm willing to put to the test: Please share your thoughts on what should also be *required* optimization steps. Share them privately, by contacting me[66], or share them publicly, perhaps tagging them so that others can find your views. Although my own shortcut was `#csso` I'll propose the more verbose and more clear `#cssoptim`.

Other than that, it's my wish that everyone, the aspiring CSS developer as well as the senior CSS wizard, could take something out of this book. Its value is surely related to how much you can take with you. To compound that value we'll finish with an overview and a selection of tools and references.

[63] https://hell.meiert.org/aux/optimize/css/
[64] https://git-scm.com/book/en/v2/Customizing-Git-Git-Hooks
[65] https://github.com/WouterSioen/pre-commit
[66] https://meiert.com/en/contact/

Overview

CSS optimization is important because we, especially as individuals, don't always write perfectly efficient and maintainable and understandable code.

Writing such code, quality code, however, is what makes us professionals.

CSS optimization does not entail "everything" that can be done in CSS or through tools; it entails what makes it more efficient and maintainable and understandable while achieving a good balance.

As professionals, we benefit from doing one (or two, or three, but not eighty-five) things really, really well; as web developers, it's not a bad idea if CSS is one of these things.

It's useful when we know our needs.

It's practical when we develop for the present.

It makes our work easier when we keep it simple.

There's nothing wrong with doing less work—by automating it.

There's nothing wrong, either, with not doing work that is automated.

(There's much wrong with not doing needed work that we haven't yet automated, however.)

Code should be understandable, and for that it's useful when code is consistent and simple.

Consistent CSS is footed on alphabetical declaration sorting (everything else is much too complicated) and on a robust order for selector sorting.

Simple CSS may avoid IDs and classes altogether and uses functional or generic names.

Simple CSS also likes shorthands (yes).

Speed is important.

In CSS, selector performance doesn't matter, and inline CSS is a crime in many countries.

(Web development is not software development.)

Some declarations may be slower than others but there's rarely reason for panic.

Rule hygiene may be most important for CSS performance.

The most basic way way to determine quality is through validation, and high error counts unequal high quality scores.

Every piece of code gets at least touched *twice*.

Web Design is a process.

For these last two reasons, maintainability and maintenance are critical.

`!important` is not an obstacle to maintenance; it's a tool.

One of the most important principles is separation of concerns, and contents, structure, presentation, behavior are all different concerns.

In code, we don't want to repeat ourselves.

Accordingly, we do want to limit the repetition of declarations.

Optimization for production is an important second step, because in production machines use our code, whereas in operation, humans use it.

When we cannot production-optimize automatically, we should do as much as we can manually.

For production, we remove all unneeded characters. All of them.

We typically want to reference one style sheet that includes all the styles, though there are cases of high complexity in which a different approach can lead to better performance.

(We cannot be absolved from making decisions about our particular projects.)

(Exceptions prove the rule.)

We regularly review our CSS, both pre-production and in production.

We are professionals and we aim for quality work.

As professionals we are aware of our responsibilities, we hold ourselves accountable, and we set high ethical standards for ourselves.

And, as professionals, we lead by example.

Tools and References

The following lists a small selection of useful *CSS* tools, websites, and books. They are geared towards more experienced developers, and they are because the novice developer may learn more and make bigger jumps learning here than staying with beginner materials.

Tools

A few handy CSS tools, both for manual and automated testing.

- CSS Analyser online tool[67], to validate CSS, units of measurement, and to check for sufficient color contrast
- CSS Minifier online tool[68], to minify CSS

[67] http://juicystudio.com/services/csstest.php
[68] https://cssminifier.com/

- CSS Stats script[69] and online tool[70], to analyze and show data about style sheets
- CSS Validation Service script[71] and online tool[72], to validate CSS
- CSSCheck online tool[73], to validate CSS
- CSSJanus Left-to-Right-to-Left-ifyer script[74] and online tool[75], to convert CSS directionality (LTR/RTL)
- CSSLint script[76] and online tool[77], to identify CSS code issues
- CSSTidy script[78] and online tool[79], to reformat and optimize CSS
- Minify script[80] and online tool[81], to minify CSS
- pre-commit script[82], to validate syntax errors in CSS (through CSS Lint) and SCSS (through SCSS Lint)
- purifycss script[83], to remove unused CSS
- UnCSS script[84] and online tool[85], to remove unused CSS
- Unused CSS online tool[86], to remove unused CSS
- YUI CSS Compressor script[87] and online tool[88], to format and compress (minify) CSS

Websites

A selection of CSS experts and resources for web development.

- A List Apart[89]
- Andrew, Rachel[90]
- Atkins Jr., Tab[91]

[69] https://github.com/cssstats/cssstats
[70] http://cssstats.com/
[71] https://github.com/w3c/css-validator
[72] https://jigsaw.w3.org/css-validator/
[73] http://www.htmlhelp.com/tools/csscheck/
[74] https://code.google.com/archive/p/cssjanus/
[75] https://cssjanus.appspot.com/
[76] https://github.com/CSSLint/csslint
[77] http://csslint.net/
[78] http://csstidy.sourceforge.net/
[79] https://hell.meiert.org/aux/optimize/css/
[80] https://github.com/matthiasmullie/minify
[81] http://www.minifier.org/
[82] https://github.com/WouterSioen/pre-commit
[83] https://github.com/purifycss/purifycss
[84] https://github.com/giakki/uncss
[85] https://uncss-online.com/
[86] https://unused-css.com/
[87] https://github.com/tubalmartin/YUI-CSS-compressor-PHP-port
[88] https://hell.meiert.org/aux/compress/css/gui/
[89] https://alistapart.com/
[90] https://rachelandrew.co.uk/
[91] http://www.xanthir.com/blog/

- Can I Use[92]
- CSS-Tricks[93]
- Frain, Ben[94]
- Heilmann, Christian[95]
- Kinlan, Paul[96]
- Meiert, Jens Oliver[97]
- Meyer, Eric[98]
- Roberts, Harry[99]
- Smashing Magazine[100]
- W3C (Cascading Style Sheets)[101]
- Weyl, Estelle[102]

Books

An even smaller selection of books (my recommendation: the specs[103]).

- *CSS Secrets: Better Solutions to Everyday Web Design Problems* (Lea Verou)[104]
- *CSS: The Definitive Guide: Visual Presentation for the Web* (Eric Meyer and Estelle Weyl)[105]
- *CSS: The Missing Manual: The Missing Manual* (David Sawyer McFarland)[106]
- *The Little Book of HTML/CSS Frameworks* (Jens Oliver Meiert)[107]

---

[92] https://caniuse.com/
[93] https://css-tricks.com/
[94] https://benfrain.com/
[95] https://christianheilmann.com/
[96] https://paul.kinlan.me/
[97] https://meiert.com/en/blog/categories/development/
[98] https://meyerweb.com/
[99] https://csswizardry.com/
[100] https://www.smashingmagazine.com/
[101] https://www.w3.org/Style/CSS/
[102] http://www.standardista.com/
[103] https://www.w3.org/TR/
[104] https://www.amazon.com/dp/B0131MQ1NS/?tag=j9t-21-20
[105] https://www.amazon.com/dp/1449393195/?tag=j9t-21-20
[106] https://www.amazon.com/dp/B0026OR2QI/?tag=j9t-21-20
[107] http://www.oreilly.com/web-platform/free/book-of-html-css-frameworks.csp

About the Author

Jens Oliver Meiert is a German author and developer who has been a technical lead at companies like GMX[108] and Google[109]. He has written a few simple, non-bestselling books about a variety of topics (last: *The Little Book of Website Quality Control*[110]). Online, he lives at meiert.com[111]. Please say hello.

[108] https://gmx.de/
[109] https://www.google.com/
[110] http://www.oreilly.com/web-platform/free/the-little-book-of-website-quality-control.csp
[111] https://meiert.com/en/