# CSS
# Best Practices Guide

Version: 2013.11.18

The following best practice guidance is for general Cascading Style Sheets (CSS) development. Details on each checklist item are discussed later in the document.

## Checklist

### Development-time

- ❑ Use CSS pre-processors for manageable styling
- ❑ Group similar CSS rules into discrete files and reference them using `@import` statements
- ❑ Use CSS3 instead of images where possible
- ❑ Use CSS3 advanced styling in moderation
- ❑ Remove unused CSS rules
- ❑ Keep selectors simple
- ❑ Avoid universal selectors
- ❑ Review rules for possible CSS Optimizations for Compositing Hardware
- ❑ Avoid dynamic CSS language compilation at runtime
- ❑ Use a “reset” CSS base line for consistent presentation
- ❑ Use percentages for layouts, not pixels
- ❑ Use relative font sizes, not fixed pixel sizes
- ❑ Keep your CSS well formatted

- ❑ Always use relative based path names for resources [ ]
- ❑ Understand the costs associated with CSS animations [ ]

### Build-time

- ❑ Inline @include CSS references into a single CSS file to reduce number of resources loaded. [ ]

## Discussion

Most best practices regarding CSS revolve around CSS3 enhancements. Under most circumstances, you'll want to exploit these newer features over older workarounds previously provided by JavaScript toolkits.

CSS3 Benefits include:

- Improved performance
- Less images to download/process
- Proper separation of concerns (CSS for styling, JavaScript for logic)

There are many possible CSS development-time maintenance optimizations and runtime performance optimizations that should be reviewed periodically throughout the projects lifetime.

#### Use Preprocessors for manageable styles

There are many preprocessors available for CSS. The most common are Sass, Less, and Stylus. All provide support for variables, macros, logic, browser compatibility, and other enhancements above and beyond standard CSS. Preprocessors can greatly simplify the organization and maintainability of your CSS. It is recommended hat you should adopt a preprocess that suits your style and needs.

If you do use a pre-processor, then it should be processed at build time, not at runtime.

Each of the preprocessors have extensions that add extra functionality like hiding vendors prefixes, generating gradients, assisting with image sprites and more.

References:

➠ How to choose the right CSS preprocessor

- Sass, Compass extension, Compass-Placeholders extension
- Less
- Stylus, nib extension

### Using @import <file.css> in your master CSS

As a developer, you want to break your CSS rules into contextually grouped files. Maintenance becomes an issue when you have too many rules in a single file. By breaking them up, you gain clarity, and reduce the chances of duplicate and/or conflicting rules.

In your master CSS file, you reference these external CSS files through the `@import` statement.

As a deployment manager, you want to inline all external CSS into a master document.

Example: `@import buttons.css`

### Use CSS3 instead of images

Things like gradients, shadows, and rounded corners are faster through CSS than using discrete images.

### Use CSS3 advanced styling in moderation

Rules such as box-shadow are expensive to process and can slow down animations on low-end hardware. Apparently, border-radius is another very expensive rule during repaints, and we all use that one. Rotation of many elements also incurs high repaint times.

References:

- Perfection Kills - Profiling CSS

### Remove unused CSS rules

Unused CSS rules add code bloat, complicates rendering, and app maintenance.

References:

- Google's Make the Web Faster

### Keep selectors simple

CSS parsers work from right to left, which is counter-intuitive. You want selectors to go from coarse to fine, not fine to coarse grained.

References:

- Google's Make the Web - Removed unused CSS Faster
- Google Optimize browser rendering (Need Examples)

### Avoid universal selectors

Selectors such as "*" indicate any node. You can read that as "ALL NODES". Therefore, you should avoid selectors like "`button > *`", as all nodes are selected first and then narrowed down to those with button parents. Using attribute only selectors also cause universal selection (eg `[type="text"]` is much slower than `input[type="text"]` )

### Review rules for possible CSS Optimizations for Compositing Hardware

- CSS Optimizations for Compositing Hardware

### Avoid dynamic CSS language compilation at runtime

When using 'less' or other CSS compilers, they should be run at build/deploy time, not at runtime.

Anti-pattern:

The following lines force the styles.less to be transformed into a .css file at runtime. This should be done in a build step, so that only the styles.css is included.

```
<link href="css/styles.less" rel="stylesheet/less" type="text/css">
<script src="./js/less.js"></script>
```

### Use a "reset" CSS base line for consistent presentation

Use a reset CSS base line like Normalize CSS to ensure consistent styling across all browsers. Most theme-able toolkits also provide normalization.

Your CSS should contain a rule that applies "`display:block`" to the HTML5 elements you use. Example:

```
header,hgroup,nav,section,article,aside,footer {
  display:block;
}
```

### Use percentages for layouts, not pixels

This is key to responsive design. When paired with media queries, serves as a foundation for "Mobile First" design. Example:

```
titleSection { height: 10%; }
```

### Use relative font sizes, not fixed pixel sizes

Helps in responsive and accessibility

- Good: `h1 { text-size: 1.25em; }`
- Bad: `h1 { text-size: 24px; }`

References:

➠ Which CSS Measurement to Use When?

### Keep your CSS well formatted

Well formatted CSS makes maintenance easier. Development and build time linting tools are strongly encouraged.

References:

➠ Procssor - Advance CSS prettifier

➠ CSS Lint

➠ Grunt CSS Lint

### Always use relative based path names for resources

All resources loaded through CSS should be relatively referenced. This includes `@import`, and image `url()` directives. Never use full or root based paths for local references. This rule does not apply to remotely accessed resources.

### Understand the costs associated with CSS animations

Animating object on the screen through CSS can greatly enhance an apps visual appeal. But not all animations are created equal. Be aware the different CSS rules impact different browser reactions.

Layouts - These are the most expensive (read “slow) changes to a page as generally the entire may need to be recalculated and redrawn. Animating these properties should generally be avoided if possible. The rules that cause a layout change include:

> border, border-width, bottom, clear, display, float, font-family, font-size, font-weight, height, left, line-height, margin, min-height, overflow, overflow-y, padding, position, right, text-align, top, vertical-align, white-space, width

Paints - Rules that impact painting are also generally slow to perform, but since they are localized to the immediate containing elements, they are not are bad as layout changes. The rules that cause a repaint include:

> background,background-image, background-position, background-repeat, background-size, border-radius, border-style, box-shadow, color, outline, outline-color, outline-style, outline-width, text-decoration, visibility

Composites - These changes come relatively cheaply and can take advantage of the browsers GPU and are promoted to their own layers. The rules that are handled through compositing include:

> opacity*, position, rotation, scale

*: There are some special concerns with opacity that may cause a paint. See the reference article for more information.

References:

- High Performance Animations

## General CSS References

- Best Practices for a Faster Web App with HTML5
- HTML5 Best Practices for Designers
- HTML5 Essentials with Best Practices
- Tools for image optimization Here's a great read that includes an awesome list of tools for optimizing images as part of a CI process.