Code Guide

Standards for developing consistent, flexible, and sustainable HTML and CSS.

Created by @mdo · v4.0.0 · GitHub repo

Table of contents

Golden rule

Enforce these, or your own, agreed upon guidelines at all times. Small or large, call out what's incorrect. For additions or contributions to this Code Guide, please open an issue on GitHub.

Every line of code should appear to be written by a single person, no matter the number of contributors.

HTML

Syntax

- Don't capitalize tags, including the doctype.
- Use soft tabs with two spaces—they're the only way to guarantee code renders the same in any environment.
- Nested elements should be indented once (two spaces).
- Always use double quotes, never single quotes, on attributes.
- Don't include a trailing slash in self-closing elements—the HTML5 spec says they're optional.
- Don't omit optional closing tags (e.g. `</li>` or `</body>`).

```
<!doctype html>
<html>
  <head>
    <title>Page title</title>
  </head>
  <body>
    <img src="images/company-logo.png"
alt="Company">
    <h1 class="hello-world">Hello,
world!</h1>
  </body>
</html>
```

HTML5 doctype

Enforce standards mode and more consistent rendering in every browser possible with this simple doctype at the beginning of every HTML page. In keeping with the suggested syntax, keep it lowercase.

```
<!doctype html>
<html>
  <head>
    <!-- ... -->
  </head>
  <body>
    <!-- ... -->
  </body>
</html>
```

Language attribute

From the HTML5 spec:

> *Authors are encouraged to specify a lang attribute on the root html element, giving the document's language. This aids speech synthesis tools to determine what pronunciations to use, translation tools to determine what rules to use, and so forth.*

Read more about the `lang` attribute in the spec. Head to the IANA for a list of language codes.

```
<html lang="en">
  <!-- ... -->
</html>
```

IE compatibility mode

There's no need to include the Internet Explorer document compatibility `<meta>` tag these days, unless you need support for IE10 and older. The tag was dropped in IE11 and isn't used in Microsoft Edge (legacy or otherwise).

For more information, read this awesome Stack Overflow article.

```
<!-- IE10 and below only -->
<meta http-equiv="x-ua-compatible"
content="ie=edge">
```

Character encoding

Ensure proper content rendering by declaring an explicit character encoding. This also allows you to use regular characters instead of their HTML entities, like – instead of `&emdash;`, provided their encoding matches that of the document. For some reserved XML characters—like ampersand, non-breaking spaces, less/greater than, and quotes—you may still need to use the HTML character entities.

UTF-8 is the recommended encoding.

```
<head>
  <meta charset="utf-8">
</head>
<body>
  <p>Use an em dash like so—no HTML
entity required.</p>
</body>
```

CSS and JavaScript includes

Per HTML5 spec, typically there is no need to specify a `type` when including CSS and JavaScript files as `text/css` and `text/javascript` are their respective defaults.

HTML5 spec links

- Using link
- Using style
- Using script

```
<!-- External CSS -->
<link rel="stylesheet" href="code-guide.css">

<!-- In-document CSS -->
<style>
  /* ... */
</style>

<!-- JavaScript -->
<script src="code-guide.js"></script>
```

Practicality over purity

Strive to maintain HTML standards and semantics, but not at the expense of practicality. Use the least amount of markup with the fewest intricacies whenever possible.

```
<!-- Good -->
<button>...</button>

<!-- Not good -->
<div class="btn" onClick="...">...</div>
```

Attribute order

HTML attributes should come in this particular order for easier reading of code.

- `class`
- `id`, `name`
- `data-*`
- `src`, `for`, `type`, `href`, `value`
- `title`, `alt`
- `role`, `aria-*`
- `tabindex`
- `style`

Attributes that are most commonly used for identifying elements should come first—`class`, `id`, `name`, and `data` attributes. Miscellaneous attributes unique to specific elements come second, followed by accessibility and style related attributes.

```
<a class="..." id="..." data-toggle="modal" href="#">
  Example link
</a>

<input class="form-control" type="text">

<img src="..." alt="...">
```

Boolean attributes

A boolean attribute is one that needs no declared value. XHTML required you to declare a value, but HTML5 has no such requirement.

For further reading, consult the WhatWG section on boolean attributes

> *The presence of a boolean attribute on an element represents the true value, and the absence of the attribute represents the false value.*

If you *must* include the attribute's value, and **you don't need to**, follow this WhatWG guideline:

> *If the attribute is present, its value must either be the empty string or [...] the attribute's canonical name, with no leading or trailing whitespace.*

In short, **don't add a value**.

```
<input type="text" disabled>

<input type="checkbox" value="1"
checked>

<select>
  <option value="1" selected>1</option>
</select>
```

Reduce markup

Whenever possible, avoid superfluous parent elements when writing HTML. Many times this requires iteration and refactoring, but produces less HTML.

```
<!-- Not so great -->
<span class="avatar">
  <img src="...">
</span>

<!-- Better -->
<img class="avatar" src="...">
```

Editor preferences

Set your editor to the following settings to avoid common code inconsistencies and dirty diffs:

- Use soft-tabs set to two spaces.
- Trim trailing white space on save.
- Set encoding to UTF-8.
- Add new line at end of files.

Consider documenting and applying these preferences to your project's `.editorconfig` file. For an example, see the one in Bootstrap. Learn more about EditorConfig.

CSS

Syntax

- — Use soft tabs with two spaces—they're the only way to guarantee code renders the same in any environment.
- — When grouping selectors, keep individual selectors to a single line.
- — Include one space before the opening brace of declaration blocks for legibility.
- — Place closing braces of declaration blocks on a new line.
- — Include one space after `:` for each declaration.
- — Each declaration should appear on its own line for more accurate error reporting.
- — End all declarations with a semi-colon. The last declaration's is optional, but your code is more error prone without it.
- — Comma-separated property values should include a space after each comma (e.g., `box-shadow`).
- — Use space-separated values for color properties (e.g., `color: rgb(0 0 0 / .5)`). See the Colors section for more information.
- — Don't prefix property values or color parameters with a leading zero (e.g., `.5` instead of `0.5` and `-.5px` instead of `-0.5px`).
- — Lowercase all hex values, e.g., `#fff`. Lowercase letters are much easier to discern when scanning a document as they tend to have more unique shapes.
- — Use shorthand hex values where available, e.g., `#fff` instead of `#ffffff`.
- — Quote attribute values in selectors, e.g., `input[type="text"]`. They're only optional in some cases, and it's a good practice for consistency.
- — Avoid specifying units for zero values, e.g., `margin: 0;` instead of `margin: 0px;`.

Questions on the terms used here? See the syntax section of the Cascading Style Sheets article on Wikipedia.

```
// Bad CSS
.selector, .selector-secondary,
.selector[type=text] {
  padding:15px;
  margin:0px 0px 15px;
  background-color:rgba(0, 0, 0, 0.5);
  box-shadow:0px 1px 2px #CCC,inset 0
1px 0 #FFFFFF
}

// Good CSS
.selector,
.selector-secondary,
.selector[type="text"] {
  padding: 15px;
  margin-bottom: 15px;
  background-color: rgb(0 0 0 / .5);
  box-shadow: 0 1px 2px #ccc, inset 0
1px 0 #fff;
}
```

Declaration order

Property declarations should be grouped together in the following order:

- Positioning
- Box model
- Typographic
- Visual
- Misc

Positioning comes first because it can remove an element from the normal document flow and override box model related styles. The box model—whether it's flex, float, grid, or table—follows as it dictates a component's dimensions, placement, and alignment. Everything else takes place *inside* the component or without impacting the previous two sections, and thus they come last.

While `border` is part of the box model, most systems globally reset the `box-sizing` to `border-box` so that `border-width` doesn't affect overall dimensions. This, combined with keeping `border` near `border-radius`, is why it's under the Visual section instead.

Preprocessor mixins and functions should appear wherever most appropriate. For example, a `border-top-radius()` mixin would go in place of `border-radius` properties, while a `responsive-font-size()` function would go in place of `font-size` properties.

For a complete list of properties and their order, please see the property order for Stylelint used by Bootstrap.

```
.declaration-order {
  // Positioning
  position: absolute;
  top: 0;
  right: 0;
  bottom: 0;
  left: 0;
  z-index: 100;

  // Box model
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  width: 100px;
  height: 100px;

  // Typography
  font: normal 14px "Helvetica Neue",
sans-serif;
  line-height: 1.5;
  color: #333;
  text-align: center;
  text-decoration: underline;

  // Visual
  background-color: #f5f5f5;
  border: 1px solid #e5e5e5;
  border-radius: 3px;

  // Misc
  opacity: 1;
}
```

Logical properties

Logical properties are alternatives to directional and dimensonal properties based on abstract terms like *block* and *inline*. By default, block refers to the vertical direction (top and bottom) while inline refers to the horizontal direction (right and left). You can begin to use these values in your CSS in all modern, evergreen browsers.

Why use logical properties? Not every language flows left-ro-right like English, so the writing mode needs to be flexible. With logical properties, you can easily support languages that can be written horizontally or vertically (like Chinese, Japanese, and Korean). Plus, they're usually shorter and simpler to write.

Additional reading:

- CSS Logical Properties and Values – MDN
- CSS Logical Properties and Values — CSS Tricks
- CSS Writing Modes – MDN

```
// Without logical properties
.element {
  margin-right: auto;
  margin-left: auto;
  border-top: 1px solid #eee;
  border-bottom: 1px solid #eee;
}

// With logical properties
.element {
  margin-inline: auto;
  border-block: 1px solid #eee;
}
```

Colors

With the support of CSS Color Levels 4 in all major browsers, `rgba()` and `hsla()` are now aliases for `rgb()` and `hsl()`, meaning you can modify alpha values in `rgb()` and `hsl()`. Along with this comes support for new space-separated syntax for color values. For compability with future CSS color functions, use this new syntax.

Regardless of your color values and syntax, always ensure your color choices meet WCAG minimum contrast ratios (4.5:1 for 16px and smaller, 3:1 for larger).

Additional reading:

- Smashing Magazine - A Guide To Modern CSS Colors
- `rgb()` - MDN

```
.element {
  color: rgb(255 255 255 / .65);
  background-color: rgb(0 0 0 / .95);
}
```

Avoid `@imports`

Compared to `<link>`s, `@import` is slower, adds extra page requests, and can cause other unforeseen problems. Avoid them and instead opt for an alternate approach:

— Use multiple `<link>`elements
— Compile your CSS with a preprocessor like Sass or Less into a single file
— Concatenate your CSS files with features provided in Rails, Jekyll, and other environments

For more information, read this article by Steve Souders.

```
<!-- Use link elements -->
<link rel="stylesheet" href="core.css">

<!-- Avoid @imports -->
<style>
  @import url("more.css");
</style>
```

Media query placement

Place media queries as close to their relevant rule sets whenever possible. Don't bundle them all in a separate stylesheet or at the end of the document. Doing so only makes it easier for folks to miss them in the future. Here's a typical setup.

```
.element { ... }
.element-avatar { ... }
.element-selected { ... }

@media (min-width: 480px) {
  .element { ... }
  .element-avatar { ... }
  .element-selected { ... }
}
```

Single declarations

In instances where a rule set includes **only one declaration**, consider removing line breaks for readability and faster editing. Any rule set with multiple declarations should be split to separate lines.

The key factor here is error detection—e.g., a CSS validator stating you have a syntax error on Line 183. With a single declaration, there's no missing it. With multiple declarations, separate lines is a must for your sanity.

```
// Single declarations on one line
.span1 { width: 60px; }
.span2 { width: 140px; }
.span3 { width: 220px; }

// Multiple declarations, one per line
.sprite {
  display: inline-block;
  width: 16px;
  height: 15px;
  background-image: url("../img/
sprite.png");
}
.icon           { background-position: 0
0; }
.icon-home      { background-position: 0
-20px; }
.icon-account   { background-position: 0
-40px; }
```

Shorthand notation

Limit shorthand declaration usage to instances where you must explicitly set all available values. Frequently overused shorthand properties include:

- `padding`
- `margin`
- `font`
- `background`
- `border`
- `border-radius`

Usually we don't need to set all the values a shorthand property represents. For example, HTML headings only set top and bottom margin, so when necessary, only override those two values. A `0` value implies an override of either a browser default or previously specified value.

Excessive use of shorthand properties leads to sloppier code with unnecessary overrides and unintended side effects.

The Mozilla Developer Network has a great article on shorthand properties for those unfamiliar with notation and behavior.

```
// Bad example
.element {
  margin: 0 0 10px;
  background: red;
  background: url("image.jpg");
  border-radius: 3px 3px 0 0;
}

// Good example
.element {
  margin-bottom: 10px;
  background-color: red;
  background-image: url("image.jpg");
  border-top-left-radius: 3px;
  border-top-right-radius: 3px;
}
```

Nesting in preprocessors

Avoid unnecessary nesting in preprocessors whenever possible—keep it simple and avoid reverse nesting. Consider nesting only if you must scope styles to a parent and if there are multiple elements to be nested.

Additional reading:

- Nesting in Sass and Less

```
// Without nesting
.table > thead > tr > th { … }
.table > thead > tr > td { … }

// With nesting
.table > thead > tr {
  > th { … }
  > td { … }
}
```

Operators in preprocessors

For improved readability, wrap all math operations in parentheses with a single space between values, variables, and operators.

```
// Bad example
.element {
  margin: 10px 0 @variable*2 10px;
}

// Good example
.element {
  margin: 10px 0 (@variable * 2) 10px;
}
```

Comments

Code is written and maintained by people. Ensure your code is descriptive, well commented, and approachable by others. Great code comments convey context or purpose. Do not simply reiterate a component or class name. Use the `//` syntax when writing CSS with preprocessors. When shipping CSS to production, remove all comments.

Be sure to write in complete sentences for larger comments and succinct phrases for general notes.

```
// Bad example
// Modal header
.modal-header {
  ...
}

// Good example
// Wrapping element for .modal-title and
.modal-close
.modal-header {
  ...
}
```

Class names

- Keep classes lowercase and use dashes (not underscores or camelCase). Dashes serve as natural breaks in related class (e.g., `.btn` and `.btn-danger`).
- Avoid excessive and arbitrary shorthand notation. `.btn` is useful for *button*, but `.s` doesn't mean anything.
- Keep classes as short and succinct as possible.
- Use meaningful names; use structural or purposeful names over presentational.
- Prefix classes based on the closest parent or base class.
- Use `.js-*` classes to denote behavior (as opposed to style), but keep these classes out of your CSS.

It's also useful to apply many of these same rules when creating custom properties and preprocessor variable names.

```
// Bad example
.t { ... }
.red { ... }
.header { ... }

// Good example
.tweet { ... }
.important { ... }
.tweet-header { ... }
```

Selectors

— Use classes over generic element tags for more explicit and reliable styling that isn't dependent on your markup.

— Avoid using several attribute selectors (e.g., `[class^="..."]`) on commonly occuring components. Browser performance is known to be impacted by these.

— Keep selectors short and strive to limit the number of elements in each selector to three.

— Scope classes to the closest parent `only` when necessary (e.g., when not using prefixed classes).

Additional reading:

— Scope CSS classes with prefixes

— Stop the cascade

```
// Bad example
span { ... }
.page-container #stream .stream-item
.tweet .tweet-header .username { ... }
.avatar { ... }

// Good example
.avatar { ... }
.tweet-header .username { ... }
.tweet .avatar { ... }
```

Child and descendant selectors

When necessary, it may be helpful to use the child combinator (>) to limit the cascade of some styles in elements like `<table>`s that are often recursively nested. Use it to limit styles to the immediate children elements of a parent element to avoid unnecessary overrides later on.

```
.custom-table > tbody > tr > td,
.custom-table > tbody > tr > th {
  /* ... */
}
```

Organization

- Organize sections of code by component.
- Develop a consistent commenting hierarchy.
- Use consistent white space to your advantage when separating sections of code for scanning larger documents.
- When using multiple CSS files, break them down by component instead of page. Pages can be rearranged and components moved.

```
//
// Component section heading
//

.element { ... }
```

```
//
// Component section heading
//
// Sometimes you need to include
optional context for the entire
component. Do that up here if it's
important enough.
//

.element { ... }

// Contextual sub-component or modifer
.element-heading { ... }
```

Follow @mdo