HTML CSS JavaScript

Udacity Frontend Nanodegree Style Guide

Introduction

This style guide acts as the official guide to follow in your projects. Udacity evaluators will use this guide to grade your projects. There are many opinions on the "ideal" style in the world of Front-End Web Development. Therefore, in order to reduce the confusion on what style students should follow during the course of their projects, we urge all students to refer to this style guide for their projects.

General Formatting Rules

Capitalization

Use only lowercase.

All code has to be lowercase. This applies to HTML element names, attributes, attribute values (unless text/CDATA).

Not Recommended:

```
<A HREF="/">Home</A>
```

Recommended:

```
<a href="/">Home</a>
```

Trailing Whitespace

Remove trailing white spaces.

Trailing white spaces are unnecessary and can complicate diffs.

Not Recommended:

```
<p>What?</p>__
```

Recommended:

```
<p>What?</p>
```

If using Sublime Text, this can be done automatically each time you save a file by adding the following to your User Settings JSON file (you should be able to find this within Sublime Text's menu):

```
"trim_trailing_white_space_on_save": true
```

Indentation

Indentation should be consistent throughout the entire file. Whether you choose to use tabs or spaces, or 2-spaces vs. 4-spaces - just be consistent!

General Meta Rules

Encoding

Use UTF-8 (no BOM).

Make sure your editor uses UTF-8 as character encoding, without a byte order mark. Specify the encoding in HTML templates and documents with `<meta charset="utf-8">`.

Comments

Explain code as needed, where possible.

Use comments to explain code: What does it cover, what purpose does it serve, and why is the respective solution used or preferred?

Action Items

Mark todos and action items with `TODO:`.

Highlight todos by using the keyword `TODO` only, not other formats like `@@`. Append action items after a colon like this: `TODO: action item`.

Recommended:

```
<!-- TODO: add other fruits -->
<ul>
    <li>Apples</li>
    <li>Oranges</li>
</ul>
```

HTML Style Rules

Document Type

Use HTML5.

HTML5 (HTML syntax) is preferred for all HTML documents: `<!DOCTYPE html>`.

Do not close self-closing elements, ie. write `<br>`, not `<br />`.

HTML Validity

Use valid HTML.

Using valid HTML is a measurable baseline quality that ensures proper HTML usage and contributes to learning about technical requirements and constraints.

Not Recommended:

```
<title>Page Title</title>
<article>This is an article.
```

Recommended:

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Page Title</title>
    </head>
    <body>
        <article>This is an article.</article>
    </body>
</html>
```

Semantics

Use HTML according to its purpose.

Use elements for what they have been created for. For example, use heading elements for headings, `p` elements for paragraphs, `a` elements for anchor, etc. Using HTML according to its purpose is important for accessibility, reuse and code efficiency reasons.

Not Recommended:

```
<div onclick="goToRecommendations();">All
recommendations</div>
```

Recommended:

```
<a href="recommendations/">All recommendations</a>
```

Multimedia Fallback

Provide alternative contents for multimedia.

For multimedia, such as images, video, or animated objects via canvas, make sure to offer alternative access. For images that means use of meaningful alternative text and for video and audio transcripts and captions, if available.

Providing alternative contents is important for accessibility reasons. A blind user has few cues to tell what an image is about without the `alt` attributes, and other users may have no way of understanding what video or audio contents are about either.

For images whose alt attributes would introduce redundancy and for images whose purpose is purely decorative which you cannot immediately use CSS for, use no alternative text, as in `alt=""`.

Not Recommended:

```
<img src="udacity.png">
```

Recommended:

```
<img src="udacity.png" alt="Udacity logo">
```

Separation of Concerns

Separate structure from presentation from behavior.

Strictly keep structure (markup), presentation (styling), and behavior (scripting) apart, and try to keep the interaction between the three to an absolute minimum.

That is, make sure documents and templates contain only HTML and HTML that is solely serving structural purposes. Move everything presentational into style sheets, and everything behavioral into scripts. In addition, keep the contact area as small as possible by linking as few style sheets and scripts as possible from documents and templates.

Separating structure from presentation from behavior is important for maintenance reasons. It is almost always more expensive to change HTML documents and templates than it is to update style sheets and scripts.

Entity References

Do not use entity references.

There is no need to use entity references like —, ”, or ☺, assuming the same encoding (UTF-8) is used for files and editors as well as among teams.

The only exceptions apply to characters with special meaning in HTML (like < and &) as well as control or “invisible” characters (like no-break spaces).

Not Recommended:

```
The currency symbol for the Euro is &ldquo;&eur;&rdquo;.
```

Recommended:

```
The currency symbol for the Euro is “€”.
```

type Attributes

Omit type attributes for style sheets and scripts.

Do not use type attributes for style sheets and scripts. Specifying type attributes in

these contexts is not necessary as HTML implies `text/css` and `text/javascript` as defaults. This can be safely done even for older browsers

Not Recommended:

```
<link rel="stylesheet" href="css/style.css" type="text/css">
```

Recommended:

```
<link rel="stylesheet" href="css/style.css">
```

Not Recommended:

```
<script src="js/app.js" type="text/javascript"></script>
```

Recommended:

```
<script src="js/app.js"></script>
```

HTML Formatting Rules

General Formatting

Use a new line for every block, list or table element and indent every such child element.

Independent of the styling of an element (as CSS allows elements to assume a different role per display property), put every block, list or table element on a new line.

Also, indent them if they are child elements of a block, list or table element (if you run into issues around whitespace between list items it's acceptable to put all `li`

elements in one line).

Recommended:

```
<blockquote>
    <p><em>Space</em>, the final frontier.</p>
</blockquote>

<ul>
    <li>Moe</li>
    <li>Curry</li>
    <li>Larry</li>
</ul>

<table>
    <thead>
        <tr>
            <th scope="col">Income</th>
            <th scope="col">Taxes</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <td>$5.00</td>
            <td>$4.50</td>
        </tr>
    </tbody>
</table>
```

HTML Quotation Marks

When quoting attribute values, use double quotation marks.

Not Recommended:

```
<a href='login/' class='btn btn-secondary'>Login</a>
```

Recommended:

```
<a href="login/" class="btn btn-secondary">Login</a>
```

HTML CSS JavaScript

Udacity Frontend Nanodegree Style Guide

Introduction

This style guide acts as the official guide to follow in your projects. Udacity evaluators will use this guide to grade your projects. There are many opinions on the "ideal" style in the world of Front-End Web Development. Therefore, in order to reduce the confusion on what style students should follow during the course of their projects, we urge all students to refer to this style guide for their projects.

General Formatting Rules

Capitalization

Use only lowercase.

All code has to be lowercase. This applies to CSS selectors, properties and property values (with the exception of strings).

Not Recommended:

```
color: #E5E5E5;
```

Recommended:

```
color: #e5e5e5;
```

Trailing Whitespace

Remove trailing white spaces.

Trailing white spaces are unnecessary and can complicate diffs.

Not Recommended:

```
border: 0;__
```

Recommended:

```
border: 0;
```

If using Sublime Text, this can be done automatically each time you save a file by adding the following to your User Settings JSON file (you should be able to find this within Sublime Text's menu):

```
"trim_trailing_white_space_on_save": true
```

Indentation

Indentation should be consistent throughout the entire file. Whether you choose to use tabs or spaces, or 2-spaces vs. 4-spaces - just be consistent!

General Meta Rules

Encoding

Use UTF-8 (no BOM).

Make sure your editor uses UTF-8 as character encoding, without a byte order mark. Do not specify the encoding of style sheets as these assume UTF-8.

Comments

Explain code as needed, where possible.

Use comments to explain code: What does it cover, what purpose does it serve, and why is the respective solution used or preferred?

Action Items

Mark todos and action items with `TODO:`.

Highlight todos by using the keyword `TODO` only, not other formats like `@@`. Append action items after a colon like this: `TODO: action item`.

Recommended:

```
/* TODO: add button elements */
```

CSS Style Rules

CSS Validity

Use valid CSS.

Using valid CSS is a measurable baseline quality that ensures proper CSS usage and allows you to spot CSS code that may not have any effect and can be removed.

ID and Class Naming

Use meaningful or generic ID and class names.

Instead of presentational of cryptic names, always use ID and class names that reflect the purpose of the element in question or that are otherwise generic.

Names that are specific and reflect the purpose of the element should be preferred as these are most understandable and the least likely to change.

Generic names are simply a fallback for elements that have no particular meaning different from their siblings. They are typically needed as helpers.

Not Recommended:

```
.p-998 { … }
.btn-green { … }
```

Recommended:

```
.gallery { … }
.btn-default { … }
```

Type Selectors

Avoid qualifying ID and class names with type selectors.

Unless necessary (for example, with helper classes), do not use element names in conjunction with IDs or classes. Avoiding unnecessary ancestor selectors is useful for performance reasons.

It is also considered bad practice to use IDs in your CSS files. There are no situations where IDs provide a benefit over classes. If you need to use a unique name for an element, use a class. (The only benefit IDs provide is speed, and is only beneficial on pages with thousands of similar elements.)

Not Recommended:

```
ul#example { … }
div.error { … }
```

Recommended:

```
.example { … }
.error { … }
```

Shorthand Properties

Use shorthand properties where possible.

CSS offers a variety of shorthand properties (like `padding` rather than explicitly setting `padding-top`, `padding-bottom`, etc.) that should be used whenever possible, even in cases where only one value is explicitly set.

Using shorthand properties is useful for code efficiency and understandability. The `font` shorthand property is recommended when setting all font related properties but is not required when making minor modifications. When using the font shorthand property, keep in mind that if font size and family are not included browsers will ignore entire font statement.

Not Recommended:

```
border-top-style: none;
font-family: palatino, georgia, serif;
font-size: 100%;
line-height: 1.6;
padding-bottom: 2em;
padding-left: 1em;
padding-right: 1em;
padding-top: 0;
```

Recommended:

```
border-top: 0;
font: 100%/1.6 palatino, georgia, serif;
padding: 0 1em 2em;
```

0 and Units

Omit unit specification after `0` values.

Not Recommended:

```
margin: 0em;
padding: 0px;
```

Recommended:

```
margin: 0;
padding: 0;
```

Leading 0s

Include leading `0`s in decimal values for readability.

Not Recommended:

```
font-size: .8em;
```

Recommended:

```
font-size: 0.8em;
```

Hexadecimal Notation

Use 3-character hexadecimal notation where possible.

Not Recommended:

```
color: #eebbcc;
```

Recommended:

```
color: #ebc;
```

ID and Class Name Delimiters

Separate words in ID and class names by a hyphen.

Do not concatenate words and abbreviations in selectors by any characters (including none at all) other than hyphens in order to improve understanding and scannability.

Not Recommended:

```
.demoimage { … }
.error_status { … }
```

Recommended:

```
.demo-image { … }
.error-status { … }
```

Hacks

Avoid user agent detection as well as CSS "hacks"—try a different approach first.

It's tempting to address styling difference over user agent detection or special CSS filters, workaround and hacks. Both approaches should be considered an absolute last resort in order to achieve and maintain an efficient and manageable code base. Consider if the intended style is absolutely critical to the functionality of your application or can the "offending" user agent "live without it".

CSS Formatting Rules

Block Content Indentation

Indent all block content, that is rules within rules as well as declarations to reflect hierarchy and improve understanding.

Recommended:

```
@media screen, projection {
    html {
        background: #fff;
        color: #444;
    }
}
```

Declaration Stops

Use a semicolon after every declaration for consistency and extensibility reasons.

Not Recommended:

```
.test {
    display: block;
    height: 100px
}
```

Recommended:

```
.test {
    display: block;
    height: 100px;
}
```

Property Name Stops

Always use a space after a property name's colon, but no space between property and colon, for consistency reasons.

Not Recommended:

```
font-weight:bold;
padding : 0;
margin :0;
```

Recommended:

```
font-weight: bold;
padding: 0;
margin: 0;
```

Declaration Block Separation

Always use a single space between the last selector and the opening brace that begins the declaration block.

Not Recommended:

```
.video-block{
    margin: 0;
}

.audio-block
{
    margin: 0;
}
```

Recommended:

```
.video-block {
    margin: 0;
}

.audio-block {
    margin: 0;
}
```

Selector and Declaration Separation

Always start a new line for each selector and declaration.

Not Recommended:

```
h1, h2, h3 {
    font-weight: normal; line-height: 1.2;
}
```

Recommended:

```
h1,
h2,
h3 {
    font-weight: normal;
    line-height: 1.2;
}
```

Rule Separations

Always put a blank line (two line breaks) between rules.

Recommended:

```
html {
    background: #fff;
}

body {
    margin: auto;
    width: 50%;
}
```

CSS Quotation Marks

Use double quotation marks for attribute selectors or property values. Do not use quotation marks in URI values (`url()`).

Not Recommended:

```
@import url("css/links.css");

html {
    font-family: 'Open Sans', arial, sans-serif;
}
```

Recommended:

```
@import url(css/links.css);

html {
    font-family: "Open Sans", arial, sans-serif;
}
```

CSS Meta Rules

Section Comments

If possible, group style sheet sections together by using comments. Separate sections with new lines.

Recommended:

```
/* Header */
.header {
    …
}

.header-nav {
    …
}


/* Content */
.gallery {
    …
}

.gallery-img {
    …
}


/* Footer */
.footer {
    …
}
```

```
.footer-nav {
    …
}
```

HTML CSS JavaScript

Udacity Frontend Nanodegree Style Guide

Introduction

This style guide acts as the official guide to follow in your projects. Udacity evaluators will use this guide to grade your projects. There are many opinions on the "ideal" style in the world of Front-End Web Development. Therefore, in order to reduce the confusion on what style students should follow during the course of their projects, we urge all students to refer to this style guide for their projects.

General Formatting Rules

Trailing Whitespace

Remove trailing white spaces.

Trailing white spaces are unnecessary and can complicate diffs.

Not Recommended:

```
var name = "John Smith";__
```

Recommended:

```
var name = "John Smith";
```

If using Sublime Text, this can be done automatically each time you save a file by adding the following to your User Settings JSON file (you should be able to find this within Sublime Text's menu):

```
```

```
"trim_trailing_white_space_on_save": true
```

Indentation

Indentation should be consistent throughout the entire file. Whether you choose to use tabs or spaces, or 2-spaces vs. 4-spaces - just be consistent!

General Meta Rules

Encoding

Use UTF-8 (no BOM).

Make sure your editor uses UTF-8 as character encoding, without a byte order mark.

Comments

Explain code as needed, where possible.

Use comments to explain code: What does it cover, what purpose does it serve, and why is the respective solution used or preferred?

Action Items

Mark todos and action items with `TODO:`.

Highlight todos by using the keyword `TODO` only, not other formats like `@@`. Append action items after a colon like this: `TODO: action item`.

Recommended:

```
// TODO: add other fruits
```

JavaScript Language Rules

var

Always declare variables with `var`.

When you fail to specify `var`, the variable gets placed in the global context, potentially clobbering existing values. Also, if there's no declaration, it's hard to tell in what scope a variable lives.

Constants

If a value is intended to be constant and immutable, it should be given a name in all capital letters, like `CONSTANT_VALUE`. Never use the `const` keyword as it's not supported by all browsers at this time.

Semicolons

Always use semicolons.

Relying on implicit insertion can cause subtle, hard to debug problems. Semicolons should be included at the end of function expressions, but not at the end of function declarations.

Not Recommended:

```
var foo = function() {
    return true // Missing semicolon
} // Missing semicolon

function foo() {
    return true;
}; // Extra semicolon
```

Recommended:

```
var foo = function() {
    return true;
```

```
};

function foo() {
    return true;
}
```

Wrapper Objects for Primitive Types

There's no reason to use wrapper objects for primitive types, plus they're dangerous. However, type casting is okay.

Not Recommended:

```
var x = new Boolean(0);
if (x) {
    alert('hi');    // Shows 'hi' because typeof x is truthy object
}
```

Recommended:

```
var x = Boolean(false);
if (x) {
    alert('hi');    // Show 'hi' because typeof x is a falsey boolean
}
```

Closures

Yes, but be careful.

The ability to create closures is perhaps the most useful and often overlooked feature in JavaScript. One thing to keep in mind, however, is that a closure keeps a pointer to

its enclosing scope. As a result, attaching a closure to a DOM element can create a circular reference and thus, a memory leak.

Not Recommended:

```
function foo(element, a, b) {
    element.onclick = function() { /* uses a and b */ }
    }
```

Recommended:

```
function foo(element, a, b) {
    element.onclick = bar(a, b);
}

function bar(a, b) {
    return function() { /* uses a nd b */ }
}
```

for-in loop

Only for iterating over keys in an object/map/hash.

`for-in` loops are often incorrectly used to loop over the elements in an array. This is however very error prone because it does not loop from `0` to `length - 1` but over all the present keys in the object and its prototype chain.

Not Recommended:

```
for (var key in arr) {
console.log(arr[key]);
}
```

Recommended:

```
var len = array.length;
for (var i = 0; i < len; i++) {
    console.log(array[i]);
}

// or...

array.forEach(function(val) {
    console.log(val);
});
```

Multiline String Literals

Do not use.

The whitespace at the beginning of each line can't be safely stripped at compile time; whitespace after the slash will result in tricky errors; and while most script engines support this, it is not part of the specification.

Not Recommended:

```
var myString = 'A rather long string of English text, an error
 message \
    actually that just keeps going and going -- an error \
    message that is really really long.';
```

Recommended:

```
var myString = 'A rather long string of English text, an error
 message' +
```

```
    'actually that just keeps going and going -- an error' +
    'message that is really really long.';
```

Array and Object Literals

Use Array and Object literals instead of Array and Object constructors.

Not Recommended:

```
var myArray = new Array(x1, x2, x3);

var myObject = new Object();
myObject.a = 0;
```

Recommended:

```
var myArray = [x1, x2, x3];

var myObject = {
    a: 0
};
```

JavaScript Style Rules

Naming

In general, `functionNamesLikeThis`, `variableNamesLikeThis`, `ClassNamesLikeThis`, `methodNamesLikeThis`, `CONSTANT_VALUES_LIKE_THIS` and `filenameslikethis.js`.

Code Formatting

Because of implicit semicolon insertion, always start your curly braces on the same line

as whatever they're opening.

Recommended:

```
if (something) {
    // Do something
} else {
    // Do something else
}
```

Single-line array and object initializers are allowed when they fit on one line. There should be no spaces after the opening bracket or before the closing bracket:

Recommended:

```
var array = [1, 2, 3];
var object = {a: 1, b: 2, c: 3};
```

Multiline array and object initializers are indented one-level, with the braces on their own line, just like blocks:

Recommended:

```
var array = [
    'Joe <joe@email.com>',
    'Sal <sal@email.com>',
    'Murr <murr@email.com>',
    'Q <q@email.com>'
];

var object = {
    id: 'foo',
```

```
    class: 'foo-important',
    name: 'notification'
};
```

Parentheses

Only where required.

Use sparingly and in general only where required by the syntax and semantics.

Strings

For consistency single-quotes (`'`) are preferred over double-quotes (`"`). This is helpful when creating strings that include HTML:

Recommended:

```
var element = '<button class="btn">Click Me</button>';
```

Tips and Tricks

True and False Boolean Expressions

The following are all false in boolean expressions:

- `null`
- `undefined`
- `''` the empty string
- `0` the number

But be careful, because these are all true:

- `'0'` the string
- `[]` the empty array
- `{}` the empty object

Conditional Ternary Operator

The conditional ternary operator is recommended, although not required, for writing concise code. Instead of this:

Not Recommended:

```
if (val) {
    return foo();
} else {
    return bar();
}
```

You can write this:

Recommended:

```
return val ? foo() : bar();
```

&& and ||

These binary boolean operators are short-circuited and evaluate to the last evaluated term. `||` has been called the default operator because instead of writing this:

Not Recommended:

```
function foo(name) {
    var theName;
    if (name) {
        theName = name;
    } else {
        theName = 'John';
    }
```

```
}
```

You can write this:

Recommended:

```
function foo(name) {
    var theName = name || 'John';
}
```

`&&` is also used for shortening code. For instance, instead of this:

Not Recommended:

```
if (node) {
    if (node.kids) {
        console.log(node.kids);
    }
}
```

You can do this:

Recommended:

```
if (node && node.kids) {
    console.log(node.kids);
}
```