An Informal Naming Convention

Table of Contents

To ensure uniformity in the project, all group members must enforce the prescribed naming methodology as stipulated in this document. All pull requests will be immediately rejected or responded with “Request changes” before approval for merging is granted.

If you feel strongly for the naming rules created, or if you wish to modify it to better suit our purposes, you are welcome to discuss it with the team and if your concern is reasonable enough, a revision to this document shall be made.

Directories

Always keep into mind the purpose behind creating a new directory. Foremost consideration would be to logically group together files that are generally dealing with the same area of concern but with specific or key differences. This purpose should then be reflected into the name of the directory.

As much as possible, limit a folder's name into a single word, and only create directories up to three levels deep starting from the root folder, for file path length considerations.

Example: You wish to create custom styles for your home page that are not used anywhere else.

Root
- css
  - base
  - components
  - utilities
  - home (your new folder)

Source Files

For context, source files in this document shall refer to files ending with **.html, .css, .js, .php**. Source files should be intrinsically related to the parent directory in which they are located in. What this means is that the parent's folder name should reflect the general function or scope by which the files inside it is concerned with.

Source files can have one or more words composing its name excluding its scope (more on this), but as usual the shorter, the better. Typically, this is how we should name our source files for clarity's sake:

If source file is a common / global file, whose function is easily understandable by everyone:

Root
- css
  - base
    - styles.css (we know that this is concerned about global CSS configs)
    - fonts.css (we know that it contains CSS for the fonts of the entire project)

If source file is for a specific functionality, typical name format follows:

(parent folder / general membership) – (specific scope) – (specific functionality).filetype

Root
- css
  - home (our previous example)
    - home-header.css (we can easily decipher that it is for the home page's header)
    - home-footer.css (same thing here)

- home-footer-spacing.css (three words since it is very specific)
- home-footer-colors.css (again, specific)

Example: You wish to create a JavaScript file to define behavior for our custom Card components.

```
Root
  └ js
      └ card
          └ card.js (general class definition of a Card, initial configs, etc)
          └ card-input.js (defining custom functions for Card input)
          └ card-input-validation.js (supposing we implement validation for the input)
          └ card-transform.js (defining custom Card behavior with different user interactions)
```

Other File Types

Files that do not belong to the previously defined source file type of files typically follow the same format of naming but without certain differences. For instance, images have different variations with regards to color scheme, sizing, and look.

For other file types, the format should be as follows:

(parent folder / general membership) – (specific scope) – (variation) – (sizing) – (count).filetype

Example: Saving a local copy of the Home page's Hero element image variations.

```
Root
  └ assets
      └ images
          └ home
          │   └ home-hero-256x256.jpg
          │   └ home-hero-512x512.jpg
          │   └ home-hero-1024x1024.jpg
          │   └ home-hero-dark-256x256.jpg
          │   └ home-hero-dark-512x512.jpg
          │   └ home-hero-dark-1024x1024.jpg
          └ favicon-128x128.png
          └ favicon-256x256.png
          └ logo-512x512-1.jpg
          └ logo-512x512-2.jpg
          └ logo-dark-512x512-1.jpg
          └ logo-dark-512x512-2.jpg
```

Source Code

A clear distinction between naming convention standards for different languages shall be made here. In a typical frontend set-up, the **camelCase standard** shall be enforced, whereas for **.php** files, the **snake_case standard** is implemented.

Local and Global Variables

In typical fashion of naming variables, a short and concise variable name would be much appreciated. Single letter variables are *unacceptable*, unless they are to be used as a counter or index variable. It must be meaningful and immediately understandable to the first-time viewer of the code. Finally, the name must make sense in the context it lives on – you wouldn't name a variable *username* when the type of data you're expecting to store in that variable is numerical or totally unrelated to user information.

Global variables are to be *capitalized* and shall have _ for separating words.

Examples of acceptable variable names include:

```
var MAX_PRODUCTS_DISPLAYED = false;
var PRODUCT_INDEX_START = 0;
var PRODUCT_INDEX_INCREMENT = 5;

/* good example */
function appendProduct(product) {

  let productContainer = document.querySelector('#container');
  let product = document.createElement('div');

  product.innerText = product.name;
  productContainer.appendChild(product);
}

/* bad example */
function appendProduct(p) {

   let container = document.querySelector('#container');  /* container for what? */
   let div = document.createElement('div'); /* acceptable, but ambiguous */

   div.innerText = p.name;  /* what does p even mean? paragraph? */
   container.appendChild(div);
}
```

Functions and Class Methods

Naming functions well can help confusions down the road. If one alone can grasp a function's purpose by merely reading its name, we can say that the function has been named well. To avoid any ambiguities with function names, our style of naming functions shall be as follows:

(verb) – (adjective) – (object being modified/created/deleted/etc)

The adjective or object may be missing, but the verb must always be present in the function name.

Example:

```
function createNewStoreProduct() { }  (all three modifiers exist)
function displayUserProfile() { } (verb and object exist)
function displayDuplicates() { } (verb and adjective exist)

Class Partner extends User {

  createNewListing() { }

  modifyExistingListing() { }
}
```