Frontend Code Standards

Overview

- The objective of this document is to define the coding standard.
- It is important to follow all the rules defined in this document.
- Every line of code should appear to be written by a single person, no matter the number of contributors.
- Why code standards?
    - Consistency – like single person typed it.
    - Readability – easy to read the source code.
    - Maintainability – easy to modify the source code.
- Technologies used:
    - Jade (http://jade-lang.com)
    - Less (http://lesscss.org)
    - jQuery (http://jquery.com)
    - Handlebars (http://handlebarsjs.com)
    - Grunt (http://gruntjs.com)
    - Node.js (http://nodejs.org)

Structure

- Separation of content, presentation and behavior.
- Indentation: two spaces should be used as the unit of indentation.

Naming conventions

- HTML:
    - Identifiers (names, ids and classes) contain only the characters [a-z0-9] and the hyphen (-).
    - Separate words in ID and CLASS names by a hyphen (-).
    - Avoid unnecessary long names.
    - Choose semantic names based on functionality, not on appearance or position.
- CSS:
    - Avoid using ID selectors in CSS.
    - Use CLASS selectors:
        - Suffixes: -block, -list, -item, -form, -btn, -group for specific case
        - Prefixes: block-, list-, item-, form-, btn-, group- for general case
        - Prefixes for elements: select-, input-, textarea-, width-, color-, editor-
        - Block wrapper: outer, inner, content, group, wrap
        - General block naming: slideshow, slider, carousel, gallery, banner, accordion, calendar, datepicker
        - Common classes: wrapper, container, main, primary, secondary, sidebar, header, footer, overlay, nav, slogan, loading, thumb, preview, highlight, featured, related, panel, module, box, layer, tab, rating, caption, description, breadcrumb, paging, social, toolbar, toolbox, tooltip, active, inactive, current, focus, warning, error, success
        - Sprite classes: wi-general, wi-text, wi-icon, wi-button, wi-box, wilayer, wi-form, wi-form-elements, wi-corner, wi-frame
    - Avoid naming: id="id", name="name", class="class", name="submit", name="reset"
    - Abbreviation images:
        - Background: **bgd-**
        - Photo: **photo-**
        - Button: **btn-**
        - Logo: **logo-**
        - Icon: **icon-**
- JavaScript:
    - Identifiers (variables, methods) contain only the characters [a-z0-9].
    - Capitalization:
        - functionNamesLikeThis
        - methodNamesLikeThis
        - variableNamesLikeThis
        - EnumNamesLikeThis
        - ClassNamesLikeThis
        - CONSTANTS_LIKE_THIS
    - Be descriptive
- File naming:
    - The file name should be stored in and delivered as file-name.ext
    - Avoid naming a file with a period "."
- Every line of code should appear to be written by a single person, no matter the number of contributors.

HTML

- Use **p** tags for paragraph delimiters instead of multiple **br** tags.
- Use **div** tags to wrap labels and controls in a form.
- Use **fieldset** tags to group related elements in a form.
- Use **label** fields to label each form field, the for attribute should associate itself with the input field, so users can click the labels.
- Use **h1** tag for page title, **h2** tags for block title, **h3-h6** tags for smaller heading in content.
- The **form** element should has the same values of **name** and **id** property.
- A **form** tag should has its **action**, **method** attributes.
- Form elements (input, textarea, select) must have **name** attribute.
- The **id** attribute must be unique within the document.
- Do not use the **size** attribute on your input fields. Use CSS width instead.
- Tables should not be used for page layout, they should be used for tabular data only.
- Make use of **thead**, **tbody** and **th** tags (and **scope** attribute) when appropriate.
- Do not use all caps or all lowercase titles in markup, use CSS **text-transform: uppercase/lowercase** instead.
- The layer should be placed before closing of **body** tag. Prefer to get the content of layer from Ajax or generate from JavaScript.
- Avoid to use **http** and **https** protocols in the same page, use // instead.
- Use HTML encoded characters (&copy instead of ©).
- Use HTML5 custom data attributes (data-).
- Use **microformats**, **microdata** when appropriate.
- Always use double quotes, never single quotes, on attributes.
- Nested elements should be indented once (two spaces).
- Don't include a trailing slash in self-closing elements (HTML5 doctype).
- Use attribute order: class, id, name, data-, src, for, type, href, title, alt, aria-, role.
- Make sure all tags are nested properly.
- Do not put a block element inside an inline element.
- Do not nest a **p** tag in a heading **h1-h6** tag, and vice versa.
- A link should have its **title** attribute.
- An image should have its **alt** attribute.
- Some tags come in pairs (ul – li, ol – li, dl – dt & dd).
- Add comments on some closing tags to indicate what element you're closing.

CSS

- CSS Sprites:
    - A sprite combines multiple images into a one large image and using CSS background-position to only display parts of it.
    - This is a technique for making webpage faster because it reduces the number of HTTP requests in the page.
    - Do sprite for all backgrounds, icons, bullets, buttons, special text fonts, custom form elements, frames, boxes, layers, etc.
    - Do not sprite for logos, particular images and photos.
    - Do not make sprites too large to avoid memory usage problem.
- CSS Specificity:
    - Specificity determines which CSS rule is applied by the browsers.
    - If two selectors apply to the same element, the one with higher specificity wins.
    - Using !important overrides all specificity no matter how high it is. Avoid using it if possible.
    - Understand cascading and selector specificity so you can write very terse and effective code.
- CSS Values:
    - Colors:
        - All color values are written in the hexadecimal format and using lowercase
    - Units:
        - Use pixel (px) unit – corresponds to actual pixels on the screen – for margin, padding
        - Use em (em) unit – corresponds to the specified point size of the font – for text
        - Use percentage (%) unit for fluid width/height content
    - Fonts:
        - Always specify a fallback generic font
        - Font names with spaces must surrounded by double-quotes
- CSS Shorthand:
    - CSS shorthand is preferred because of its terseness
    - Follow the TRBL acronym
    - Common shorthand properties: margin, padding, font, background, border, border-radius, transition, transform, list-style
- CSS Box Model: Margin, Border, Padding, Content
- Use a reset CSS file to avoid browser inconsistencies
- Use a minimal number of style sheets
- Properties should be listed in group similar:
    - Display & Flow (display, visibility, float, clear)
    - Positioning & Floats (position, top, right, bottom, left, z-index)
    - Dimensions (width, -width, height, -height, overflow)
    - Margins & Paddings (margin, padding)
    - Borders & Outline (border, outline)
    - Background (background)
    - Typography (font-, line-height, text-, -spacing, white-space, verticalalign, color, list-style)
    - Opacity & Cursors (opacity, cursor)
- Avoid using !important if possible

- Use the link tag to include, never use @import
- Quote attribute values in selectors, e.g. input[type="text"]
- Avoid specifying units for zero values
- Avoid using style inline in the HTML file
- Avoid using single line CSS because it can cause issues with version control
- Single declarations on one line
- Multiple declarations, one per line
- Prefer CLASS selector than ID selector
- Write selectors that are optimized for speed
- For mobile version, should cut as separate images, do not use sprite images

JavaScript

- Indentation:
  - Avoid lines longer than 80 characters
  - When an expression will not fit on a single line, break it according to these general principles: break after a comma, break before an operator, align the new line with the beginning of the expression at the same level on the previous line.

```
var result = 0,
  longName = '';

var result = longExp1 + longExp2
             - longExp3;

var result = (longExp) ? longResult1
                       : longResult2;

if((longCondition1 && longCondition2)
  || longCondition3){
  // Code here
}
```

- Comments:
  - Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly yourself) who will need to understand what you have done
  - Make comments meaningful. Focus on what is not immediately visible
  - Comments should not be enclosed in large boxes drawn with asterisks or other characters
  - Comments should never include special characters such as form-feed and backspace
  - Multiline comments should be

```
/*
comment here
*/
```

  - Inline comments should be

```
// comment here
```

- Declarations:
  - One declaration per line is recommended since it encourages commenting

```
var result = 0, // comment here
longName1 = '',
longName2 = '', longName3 = '';
```

  - Do not put different types on the same line
  - Try to initialize local variables where they're declared
  - No space between a method name and the parenthesis '()'

```
function methodName(){
  // to do
}
```

  - Open brace '{' appears at the end of the same line as the declaration statement

```
function methodName(){
  // do to
}
```

  - Closing brace '}' starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the '}' should appear immediately after the '{'

```
function methodName(){
  // to do
}
function empty(){}
```

- Methods are separated by a blank line
- JavaScript does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages.
- Put declarations only at the beginning of blocks

- Statements:
  - Simple Statements:
    - Each line should contain at most one statement
  - Compound Statements: Compound statements are statements that contain lists of statements enclosed in '{ }' (curly braces):
    - The '{' should be at the end of the line that begins the compound statement
    - The '}' should begin a line and be indented to align with the beginning of the line containing the matching '{'
  - Conditional statements:
    - **if** statement

```
if(condition1){
  // to do if condition1 is true
}
else if(condition2){
  // to do if condition2 is true
}
else{
  // to do if neither condition1 nor condition2 is true
}
```

    - **switch** statement

```
switch(expression){
  case expression1:
    // to do 1
    break;
  case expression2:
    // to do 2
    break;
  default:
    // to do if expression is different from expression1 and expression2
}
```

  - Loop statements
    - **for** statement

```
for(initialization; condition; update){
  // to do
}
for(variable in object){
  // to do
}
```

    - **while** statement

```
while(condition){
  // to do
}
```

    - **do** statement

```
do{
  // to do
}
while(condition);
```

    - **try...catch** statement

```
try{
  // to do
}
catch(e){
  // to do
}
finally{
  // to do
}
```

- White Space
  - Blank spaces should be used in the following circumstances:
    - A blank space should not be used between a function value and its '('
    - All binary operators except '.' (period), '(' (left parenthesis) and '[' (left bracket) should be separated from their operands by a space

```
var result = longExp1 + longExp2;
var result = (longExp1 + longExp2);
var result = jsonData['result'];
```

- No space should separate a unary operator and its operand except when the operator is a word such as typeof

```
var x = 0;
x++
--x;
var y = -x;
typeof x;
```

- Each ';' (semicolon) in the control part of a for statement should be followed with a space

```
for(var i = 0, j = 10; i < j; i++){
  // to do
}
```

- Whitespace should follow every ',' (comma)

```
var x = 10, y = 15;
function methodName(param1, param2, param3){
  // to do
}
```

- Global variables should be minimized
- Variable declarations must start with **var** keyword
- Variables and functions should be declared before used
- Constants or configuration variables should be at the top of the file
- JS expressions must end with a semi-colon
- Don't rely on the user-agent string, do proper feature detection
- Don't use **document.write** function
- Avoid using inline script in the HTML file
- Avoid using **eval** function
- All Boolean variables should start with "is", "has"
- Create functions which can be generalized, take parameters, and return values
- Do not send too many function parameters
- Minimizing repaints and reflows
- Do not compare **x === true**, use **(x)** instead
- Use **[value1, value2]** to create an array
- Use **{member: value}** to create an object
- Comment your code! It helps reduce time spent troubleshooting JavaScript functions
- End of file with a newline
- Avoid:
  - Too much happening in a loop
  - Too much happening in a function
  - Too much recursion
  - Too much DOM interaction

Checklist

- Top Rules
  - You must report project status and timing daily
  - You must follow Git rules
  - You must follow Code standards
  - You must not copy and paste code, only use source code manager has approved
  - You must perform functional test of your code, do boundary and branch test
  - You must follow design, storyboard, requirement and schedule provided by manager
  - If requirement is not 100% clear, you must ask, never assume
  - Raise issue to manager as soon as possible
  - Do not copy or send code offsite, all code is owned by Sutrix client
  - Never test on client's production environment
  - Golden rule: **Complete your work on time with high quality**
- General
  - The page is validated by W3C
  - The page is checked on big screen resolution

    - The page is checked on supported browsers
    - The links and buttons should have rollover effects
    - Put CSS at top
    - Put Javascript at bottom
    - Using CSS Sprite for images
    - Check Y!Slow / PageSpeed tool at least grade B
    - Create libraries for special text fonts, images, banners and backgrounds
    - No Javascript error message on supported browsers
    - The forms must have validation functions
    - Input restriction follow the requirements in the storyboard
    - Avoid script inside HTML file, use external JS file
    - Flash detection is required
- Source Code
    - Source code must not contain any words related to Sutrixmedia
    - No Sutrix text in source code
    - No Vietnamese text in source code
    - No Developer Name or Email in source code
- HTML Source
    - The structure supports dynamic content
    - The HTML code is well structured and semantic
    - Do not scale images in HTML
    - Each image has a relevant ALT attribute
    - Each link and input button has a relevant TITLE attribute
    - Favicon is set on each page
    - Each page has a unique title
    - Avoid using TABLE tag, should be change to DIV tag
    - Contact and meeting information use microformats
    - For .Net project: use one form, prefer CLASS selector than ID selector
- CSS Source
    - Make sure the page has CSS print
    - Each section in the CSS file must have comment
    - CSS code is well structured and avoid CSS hack
    - Avoid using @import, !important
- JavaScript Source
    - Global objects should exist in namespaces
    - Variables and functions should be declared before used
    - Remove unused variables and functions
    - Avoid using eval / classic alert function
    - Use GET method for Ajax request
    - Ajax call should have loading icon
    - Include necessary JS file only

Code Review

- HTML Source
    - Check for layout, alignment, spacing, pa-ing
    - Code does not contain inline JavaScript event listeners
    - Code does not contain inline style attributes
    - Code does not contain deprecated elements & attributes
    - Code does not use all caps or all lowercase title
    - Page begins with a valid DTD (HTML5 doctype)
    - Code is indented using 2 spaces
    - Code is semantic
    - Tags and attributes are in lowercase
    - Tags are closed and nested properly
    - Tables are only used to display tabular data
    - Element IDs are unique
    - Code validates against the W3C validator
    - Code validates against WCAG level 2
    - Form elements are paired with labels elements containing the for attribute
    - Form label / input pairs are wrapped with a block level element
    - Forms are logically arranged within fieldsets
    - Page has a proper outline (H1-H6 order)
    - Alt attributes exist on all img elements
    - Events and styles applied to :hover are also applied to :focus
    - Tabindex order is logical and intuitive
    - Appropriate use of HTML inputs (e.g. email, phone) to trigger corresponding on-screen keyboards
- CSS Source
    - Consistent naming conventions are used
    - CSS validates against the W3C validator
    - CSS sprites are used to combine images
    - CSS shorthand is used for properties that support it

- JavaScript Source
    - No syntax/runtime errors and warnings in the code
    - No deprecated functions in the code
    - A method should not be larger than 40 lines of code. The basic idea is always a function should be viewable in single screen without scrolling
    - Check that each function is doing only a single thing. That is never function named createCustomer should delete the existing customer and create it again
    - Always try to initialize the variable before using that in a function
    - Always try to use constants in the left hand side of the comparison. That is instead of doing if (variable == 2) always use if (2 == variable)
    - If function returns something, there is a return method in all cases
    - Make sure user input is checked for value
    - No magic numbers. There should be no magic numbers like 6, 10, etc… Any numbers like this should be define as a constant. And Constants should be well commented about the purpose
    - Always try to have unit test for the new piece of code. In ideal condition we should have 100% unit test coverage
    - No functions or variables in the global namespace
    - Write efficient code, especially in loops
    - Don't append to the DOM in a loop. If possible do not read from it. It's slow
    - Keep style in css, use classes
    - Use the best jQuery selector if possible
    - Don't select an object in jQuery more than once, use chaining or put it in a variable
    - Minimize http requests
    - Validate user input before making http requests
    - Don't use jQuery $.each
    - Handle specific errors
    - Provide the user with enough info to remedy the situation when errors occur if possible
    - Don't put an else right after a return. Delete the else, it's unnecessary and increases indentation level
    - Use two spaces for indentation
    - When fixing a problem, check to see if the problem occurs elsewhere in the same file, and fix it everywhere if possible
    - End the file with a newline
    - Declare local variables as near to their use as possible
    - Do not compare x == true or x == false. Use (x) or (!x) instead. x == true, in fact, is different from if (x)! Compare objects to null, numbers to 0 or strings to "" if there is chance for confusion
    - Make sure that your code doesn't generate any strict JavaScript warnings, such as duplicate variable declaration, mixing return; with return value; and undeclared variables or members
    - Use [value1, value2] to create a JavaScript array in preference to using new Array(value1, value2)
    - Use {key1: value1, key2: value2} to create a JavaScript object
    - If having defined a constructor you need to assign default properties it is preferred to assign an object literal to the prototype property
    - Use regular expressions, but use them wisely
    - Code has been run through JSLint (http://jslint.com) or JSHint (http://jshint.com)
- Code Review Tools
    - DOM Monster bookmarklet (http://mir.aculo.us/dom-monster)
    - HTML: Link Checker (http://validator.w3.org/checklink)
    - HTML: Internationalization Checker (https://validator.w3.org/i18n-checker)
    - HTML: Markup Validator (http://validator.w3.org)
    - CSS: CSS Validator (http://jigsaw.w3.org/css-validator)
    - CSS: CSSLint (http://csslint.net)
    - JavaScript: JSLint (http://www.jslint.com)
    - JavaScript: JSHint (http://www.jshint.com)
    - Performance: YSlow (http://developer.yahoo.com/yslow)
    - Performance: PageSpeed (https://developers.google.com/speed/pagespeed)
    - Accessibility: CynthiaSays (http://www.cynthiasays.com)
    - Accessibility: AChecker (http://achecker.ca)

Unit Test

- HTML
    - Test on cross browsers
    - Test on different resolutions
    - Test on real devices
    - Test on CSS disabled
    - Use HTMLHint
- CSS
    - Check pixel perfect
    - Use CSSLint
- JavaScript
    - Use JSHint
    - Use Qunit
    - Use Jasmine

Jade

- Jade is a terse language for writing HTML templates.
    - Produces HTML
    - Supports dynamic code
    - Supports reusability (DRY)

Less

- Less is a CSS pre-processor, meaning that it extends the CSS language, adding features that allow variables, mixins, functions and many other techniques that allow you to make CSS that is more maintainable, themable and extendable.

jQuery

- site.js

```
/**
 * @name Site
 * @description Global variables and functions
 * @version 1.0
 */

var Site = (function($, window, undefined) {
  'use strict';

  var privateVar = null;
  var privateMethod = function() {
    // to do
  };

  return {
    publicVar: privateVar,
    publicMethod1: privateMethod
  };

})(jQuery, window);

jQuery(function() {
  Site.publicMethod1();
});
```

- l10n.js

```
/**
 *  @name L10n
 *  @description Localization
 *  @version 1.0
 */
var L10n = {
  group: {
    key: 'value'
  }
};
```

- plugin.js

```
/**
 *  @name plugin
 *  @description description
 *  @version 1.0
 *  @options
 *    option
 *  @events
 *    event
 *  @methods
 *    init
 *    publicMethod
 *    destroy
 */
;(function($, window, undefined) {
  'use strict';

  var pluginName = 'plugin';
  var privateVar = null;
  var privateMethod = function(el, options) {
    // to do
  };

  function Plugin(element, options) {
    this.element = $(element);
    this.options = $.extend({}, $.fn[pluginName].defaults, this.element.data(), options);
    this.init();
  }

  Plugin.prototype = {
    init: function() {
      var that = this;
      this.vars = {
        key: 'value'
```

```
      };
      // initialize
      // add events
    },
    publicMethod: function(params) {
      // to do
      $.isFunction(this.options.onCallback) && this.options.onCallback();
      this.element.trigger('customEvent');
    },
    destroy: function() {
      // remove events
      // deinitialize
      $.removeData(this.element[0], pluginName);
    }
  };

  $.fn[pluginName] = function(options, params) {
    return this.each(function() {
      var instance = $.data(this, pluginName);
      if (!instance) {
        $.data(this, pluginName, new Plugin(this, options));
      } else if (instance[options]) {
        instance[options](params);
      }
    });
  };

  $.fn[pluginName].defaults = {
    key: 'value',
    onCallback: null
  };

  $(function() {
    $('[data-' + pluginName + ']').on('customEvent', function(){
      // to do
    });
    $('[data-' + pluginName + ']')[pluginName]({
      key: 'custom'
    });
  });

}(jQuery, window));
```

Handlebars

- Handlebars provides the power necessary to let you build semantic templates effectively with no frustration.
- Handlebars is largely compatible with Mustache templates. In most cases it is possible to swap out Mustache with Handlebars and continue using your current templates.

Grunt

- The JavaScript Task Runner
- Why use a task runner? In one word: automation. The less work you have to do when performing repetitive tasks like minification, compilation, unit testing, linting, etc, the easier your job becomes. After you've configured it through a Gruntfile, a task runner can do most of that mundane work for you and your team with basically zero effort.

Node.js

- Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.