JavaScript Coding Guidelines

Contributors

Souvik Basu

[souvikbasu@gmail.com](mailto:souvikbasu@gmail.com)

@souvikbasu

References

<http://google.github.io/styleguide/javascriptguide.xml>

<https://github.com/rwaldron/idiomatic.js>

<http://contribute.jquery.org/style-guide/js/>

<http://javascript.crockford.com/code.html>

<http://jsbeautifier.org/>

Most of the guidelines mentioned here adhere to Google Coding standards for JavaScript. Wherever needed we have modified the rules to suit our needs (e.g. instead of 80 char line we use 120 chars as that works well for all our displays; and many more).

Tools

Web Essentials for Visual Studio 2013 (<http://vswebessentials.com/>)

Google Closure Compiler tool from (<https://visualstudiogallery.msdn.microsoft.com/de3d7c40-356e-446b-b14e-e8ccd57c57d2>)

Coding pitfalls

Variable Declaration

Always use ‘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 (e.g., it could be in the Document or Window just as easily as in the local scope). So always declare with var.

Put multiple comma-separated declarations in a single var statement, but only if they don’t have assignments. Assignments must be in their own var statement. Like this:

var paybackPeriod, analysisPeriod;

var message = 'Something';

var sections = [];

var paybackPeriod,

analysisPeriod = 0,

message = 'Something',

sections = [];

Declare all variables just before they are used even though JavaScript has function scope. This is for better readability.

var paybackPeriod;

//.. (statements)

paybackPeriod = 10;

//.. (statements)

var paybackPeriod = 10;

Declare iterator variables inside for loop and not at the beginning of function.

var i = 0;

...

for(; i < 4; i++) {

}

for(var i = 0; i < 4; i++) {

}

Semicolon

Always explicitly end a statement with a semicolon

JavaScript has Automatic Semicolon Insertion (ASI). But this leads to a lot of errors.

More info: <http://google.github.io/styleguide/javascriptguide.xml?showone=Semicolons#Semicolons>

var foo = function() {

return true;

}; // semicolon here.

function foo() {

return true;

} // no semicolon here.

Function Declaration vs Function Expression

if (x) {

function foo() {}

}

While most script engines support Function Declarations within blocks it is not part of ECMAScript. Worse implementations are inconsistent with each other and with future EcmaScript proposals. ECMAScript only allows for Function Declarations in the root statement list of a script or function. Instead use a variable initialized with a Function Expression to define a function within a block:

if (x) {

var foo = function() {};

}

Also, function declaration is invoked before any code is executed. This might add some performance hit depending upon the code structure. Use function expression wherever possible, except in places were the code is called multiple times as the function object will be created multiple times in such cases.

For better debugability you can also use named function expressions wherever necessary.

var foo = function foo() {};

This will show the function name in browser tools stack trace.

Exceptions

Throw custom exceptions wherever possible

throw {

name: "Divide By Zero Error",

level: "Critical",

message: "Illegal mathematical operation to divide by zero",

toString: function(){return this.name + ": " + this.message;}

};

If there are exception types that will be thrown in multiple places, you can create an exception class for reusability. Derive from Error class.

function DivideByZeroException(message) {

this.name = "Divide By Zero Error";

this.level = "Critical";

this.message = message;

this.toString = function(){return this.name + ": " + this.message;}

}

DivideByZeroException.prototype = new Error();

function ArgumentNullException(message) {

this.name = "Argument Null Error";

this.level = "Critical";

this.message = message;

this.toString = function(){return this.name + ": " + this.message;}

}

ArgumentNullException.prototype = new Error();

try {

if (divider === 0) {

throw new DivideByZeroException ();

} else if (arg == null) {

throw new ArgumentNullException ();

} else {

throw {

name: "Some other specific error",

level: "Error",

message: "Some error type has occurred",

toString: function(){return this.name + ": " + this.message;

}

}

}

catch (e) {

if (e instanceof DivideByZeroException) {

alert("Divide by zero error");

} else if (e instanceof ArgumentNullException) {

alert("Argument null erorr");

}

}

Standard Features

Always prefer standard features

string[3]

string.charAt(3)

Adding method to an object

Add methods to object prototype

Foo.prototype.bar = function () {

/\* ... \*/

};

The preferred style for other properties is to initialize the field in the constructor:

/\*\* @constructor \*/

function Foo() {

this.bar = value;

}

Adding/deleting properties to an object at a later point is a costlier operation. Avoid changing the shape of an object at runtime.

More info: <https://developers.google.com/v8/design#prop_access>

Deleting object property

Foo.prototype.dispose = function () {

delete this.property\_;

};

Foo.prototype.dispose = function () {

this.property\_ = null;

};

Same reason as above. In modern JavaScript engines, changing the number of properties on an object is much slower than reassigning the values.

Closures

If closure does not use all variables from outer scope then create the closure in a separate function call. 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. For example, in the following code:

function foo(element, a, b) {

element.onclick = function() { /\* uses a and b \*/ };

}

The function closure keeps a reference to element, a, and b even if it never uses element. Since element also keeps a reference to the closure, we have a cycle that won't be cleaned up by garbage collection. In these situations, the code can be structured as follows:

function foo(element, a, b) {

element.onclick = bar(a, b);

}

function bar(a, b) {

return function() { /\* uses a and b \*/ };

}

eval()

Do not use eval() as that can be used for script injection

var userInfo = eval(feed);

var email = userInfo['email'];

If the feed was modified to include malicious JavaScript code, then if we use eval then that code will be executed.

var userInfo = JSON.parse(feed);

var email = userInfo['email'];

with() {}

Do not use with(){} as it can cloud semantic of the code

switch() {}

Avoid switch(){} as it is easy to miss a break; statement for a case block. That will cause the flow to go into the case statement below.

But can be useful when there are a large number of cases - especially when multiple cases can be handled by the same block, or fall-through logic (the default case) can be leveraged. If you are using switch statement, make sure he code is reviewed.

for-in() {}

Do not use for-in loop. Instead use for loop with an iterator variable.

Multiline string literals

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

actually that just keeps going and going -- an error \

message to make the Energizer bunny blush (right through \

those Schwarzenegger shades)! Where was I? Oh yes, \

you\'ve got an error and all the extraneous whitespace is \

just gravy. Have a nice day.';

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 ECMAScript.

Use string concatenation instead:

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

'actually that just keeps going and going -- an error ' +

'message to make the Energizer bunny blush (right through ' +

'those Schwarzenegger shades)! Where was I? Oh yes, ' +

'you\'ve got an error and all the extraneous whitespace is ' +

'just gravy. Have a nice day.';

Observe that plus operator is at the end of previous line and not at beginning of next line. This is for readability.

Similarly for logical operator also, put them at the end of line in case of multiline expression. Also indent the additional lines to differentiate from the body

if (firstCondition() &&

secondCondition() &&

thirdCondition()) {

doStuff();

}

Type Coercion

Prefer === and !== over == and !=

== and != looks for truthy values and ignores object type. e.g. 1 == “1”

Type Checking

To determine type of an object, use following

String:

typeof variable === "string"

Number:

typeof variable === "number"

Boolean:

typeof variable === "boolean"

Object:

typeof variable === "object"

Array:

Array.isArray( arrayLikeObject )

undefined:

Global Variables:

typeof variable === "undefined"

Local Variables:

variable === undefined

Properties:

object.prop === undefined

Checking for item in array

if ( array.indexOf( "a" ) >= 0 ) {

// ...

}

Checking that array is empty

if (!array.length === 0)

Evaluate falsy-ness

if (!array.length)

Checking that array is not empty

if (array.length > 0)

Evaluate truthiness

if (array.length)

Checking that string is empty

if (string === "")

Evaluate falsy-ness

if (!string)

Checking that string is not empty

if (string !== "")

Evaluate truthiness

if (string)

Checking that a reference is true/false

if (foo === true)

if (foo === false)

Evaluate truthiness

if (foo)

if(!foo)

Be careful, this will also match: 0, "", null, undefined, NaN. If you must test for a boolean false, then use

if ( foo === false ) ...

Checking for null

When only evaluating a ref that might be null or undefined, but NOT false, "" or 0,

if ( foo === null || foo === undefined )

Take advantage of == type coercion, like this:

'if ( foo == null )

Using == will match a `null` to BOTH `null` and `undefined` but not `false`, "" or 0

null == undefined

Immediately-Invoked Function Expression (IIFE)

Use IIFE for every modules to avoid cluttering the global namespace. This is a good practice to create private variables in JavaScript.

(function( global ) {

var Module = (function() {

var data = "secret";

return {

// This is some boolean property

bool: true,

// Some string value

string: "a string",

// An array property

array: [ 1, 2, 3, 4 ],

// An object property

object: {

lang: "en-Us"

},

getData: function() {

// get the current value of `data`

return data;

},

setData: function( value ) {

// set the value of `data` and return it

return ( data = value );

}

};

})();

// Other things might happen here

// expose our module to the global object

global.Module = Module;

})( this );

More info: <http://benalman.com/news/2010/11/immediately-invoked-function-expression/>

Early return

Return from function early on for better readability

function returnLate( foo ) {

var ret;

if ( foo ) {

//...

ret = "foo";

} else {

ret = "quux";

}

return ret;

}

function returnEarly( foo ) {

if ( foo ) {

// ...

return "foo";

}

return "quux";

}

Strict mode

Enforce strict mode in js code. There are 2 ways to do it – either at the top of the file or inside an IIFE. Always use strict mode inside a IIFE so that it does not enforce strict mode on other js file code after concatenation into a single file

'use strict';

ei.MAX\_SECTIONS\_PER\_TEMPLATE = 50;

ei.MAX\_STATEMENTS\_PER\_EXEC\_SUMM\_SECTION = 4;

ei.DEFAULT\_LOCALE = 'en';

(function () {

'use strict';

ei.MAX\_SECTIONS\_PER\_TEMPLATE = 50;

ei.MAX\_STATEMENTS\_PER\_EXEC\_SUMM\_SECTION = 4;

ei.DEFAULT\_LOCALE = 'en';

}());

Use JSHint and JSCS

Enforce Javascript styling using JSHint and JSCS. Create config files in project as shown in Appendix section

To ignore a block of code from a particular styling, escape as follows:

/\* jshint -W101 \*/ // Line is too long

routes: {

'': 'templateAdmin',

'template-admin': 'templateAdmin',

'template-detail?mode=:mode': 'templateDetailWithMode',

'template-detail': 'templateDetail',

'template-configuration?mode=:mode': 'templateConfiguration',

'generate-report?scenarioId=:scenarioId&scenarioName=:scenarioName&buildingId=:buildingId&reportConfigurationId=:reportConfigurationId&analysisPeriod=:analysisPeriod': 'generateSavedReport',

'generate-report?scenarioId=:scenarioId&scenarioName=:scenarioName&buildingId=:buildingId&analysisPeriod=:analysisPeriod': 'generateReport'

},

/\* jshint +W101 \*/

JavaScript types

Enforce JavaScript types using JSDoc. Specify access modifier as public, private or protected

/\*\*

\* Some value.

\* @type {!Object}

\* @private

\*/

this.myValue\_ = value;

Append method or property with an underscore if it marked private to indicate that it is not publicly accessible and final

Compiling

Compile code with Google Closure Compiler using the visual studio external tool (mentioned in the tools section)

Otherwise, go to <http://closure-compiler.appspot.com/home> and compile the all.js file.

This will ensure that object type and access modifiers are verified to be intact within the code.

Style

Naming convention

functionNamesLikeThis

variableNamesLikeThis

ClassNamesLikeThis

EnumNamesLikeThis

methodNamesLikeThis

CONSTANT\_VALUES\_LIKE\_THIS

foo.namespaceNamesLikeThis.bar

filenameslikethis.js.

Use readable function and parameter names

function q(s) {

return document.querySelectorAll(s);

}

function query(selector) {

return document.querySelectorAll(selector);

}

Names should be descriptive but not excessively so.

var thisIsAVariableThatStoresTheAnalysisPeriod

var analysisPeriod

Looping index is an exception. They can have names i, j, k, etc. in for loop.

for(var i = 0; i < 4; i++) {

Prefix jQuery elements with $ for readability. This tells the developer immediately that it’s a jQuery element.

var $header = $('#header');

Tabs and Whitespace

Use 4 space instead of tabs for indentation

function something() {

return 0;

}

function something() {

return 0;

}

Unary special-character operators (e.g., !, ++) must not have space next to their operand.

Any , and ; must not have preceding space.

Any ; used as a statement terminator must be at the end of the line.

Any : after a property name in an object definition must not have preceding space.

The ? and : in a ternary conditional must have space on both sides.

No filler spaces in empty constructs (e.g., {}, [], fn())

Code formatting

Use explicit curly braces for all block statements including single line if statements. Also start braces on the same line as the statement, not on a new line. This is to take care against automatic semicolon insertion that might cause unexpected behavior of code.

if (something)

x = 0;

else

y = 0;

if (something) {

// ...

} else {

// ...

}

Use space after if and while statements before parenthesis. No space inside parenthesis.

if(something) {

x = 0;

}

if (something) {

x = 0;

}

No space after anonymous function before parenthesis

function (something) {

...

}

function(something) {

...

}

Single-line array and object initializers are allowed when they fit on a line:

var arr = [1, 2, 3]; // No space after [ or before ].

var obj = {a: 1, b: 2, c: 3}; // No space after { or before }.

Multiline array initializers and object initializers are indented 4 spaces, with the braces on their own line, just like blocks.

// Object initializer.

var inset = {

top: 10,

right: 20,

bottom: 15,

left: 12

};

// Array initializer.

this.rows\_ = [

'"Slartibartfast" <fjordmaster@magrathea.com>',

'"Zaphod Beeblebrox" <theprez@universe.gov>',

'"Ford Prefect" <ford@theguide.com>',

'"Arthur Dent" <has.no.tea@gmail.com>',

'"Marvin the Paranoid Android" <marv@googlemail.com>',

'the.mice@magrathea.com'

];

// Used in a method call.

goog.dom.createDom(goog.dom.TagName.DIV, {

id: 'foo',

className: 'some-css-class',

style: 'display:none'

}, 'Hello, world!');

Long identifiers or values present problems for aligned initialization lists, so always prefer non-aligned initialization. For example:

CORRECT\_Object.prototype = {

a: 0,

b: 1,

lengthyName: 2

};

Not like this:

WRONG\_Object.prototype = {

a : 0,

b : 1,

lengthyName: 2

};

Never use parentheses for unary operators such as delete, typeof and void or after keywords such as return, throw as well as others (case, in or new).

Strings

Use single quote for string

var message = "do not use double quote";

var message = 'use single quote';

Line character count limit

120

Function arguments

When possible, all function arguments should be listed on the same line. If doing so would exceed the 120-column limit, the arguments must be line-wrapped in a readable way.

// Parenthesis-aligned, one argument per line. Emphasizes each

// individual argument.

function bar(veryDescriptiveArgumentNumberOne,

veryDescriptiveArgumentTwo,

tableModelEventHandlerProxy,

artichokeDescriptorAdapterIterator) {

// ...

}

Blank lines

Use newlines to group logically related pieces of code.

doSomethingTo(x);

doSomethingElseTo(x);

andThen(x);

nowDoSomethingWith(y);

andNowWith(z);

Do not keep more than 2 blank lines anywhere.

Put exactly 2 empty lines between function declarations, expressions, properties

var func1 = function() {

// ...

}

var func2 = function(){}

End a file with a blank line

Ternary operator

var x = a ? b : c;

var z = a ?

moreComplicatedB :

moreComplicatedC;

Dot operator (method chaining)

Put dot at the end of the line and not at beginning of next line. Indent 4 space w.r.t the top line

var x = foo.bar().

doSomething().

doSomethingElse();

Documentation

Use JSDoc documentation (<http://usejsdoc.org/>)

/\*\*

\* Some class, initialized with a non-null value.

\* @param {!Object} value Some value.

\* @constructor

\*/

function MyClass(value) {

/\*\*

\* Some value.

\* @type {!Object}

\* @private

\*/

this.myValue\_ = value;

}

/\*\*

\* A JSDoc comment should begin with a slash and 2 asterisks.

\* Inline tags should be enclosed in braces like {@code this}.

\* @desc Block tags should always start on their own line.

\*/

/\*\*

\* Illustrates line wrapping for long param/return descriptions.

\* @param {string} foo This is a param with a description too long to fit in

\* one line.

\* @return {number} This returns something that has a description too long to

\* fit in one line.

\*/

project.MyClass.prototype.method = function(foo) {

return 5;

};

To list down elements use html li tag

/\*\*

\* Computes weight based on three factors:

\* items sent

\* items received

\* last timestamp

\*/

It'll come out like this:

Computes weight based on three factors: items sent items received last timestamp

Instead, do this:

/\*\*

\* Computes weight based on three factors:

\* <ul>

\* <li>items sent

\* <li>items received

\* <li>last timestamp

\* </ul>

\*/

Use the following JSDoc tags whenever applicable

@name name - the name of the ngdoc document

@param {type} name description - describes a parameter of a function

@returns {type} description - describes what a function returns

@requires - normally indicates that a JavaScript module is required; in an Angular service it is used to describe what other services this service relies on

@property - describes a property of an object

@description - used to provide a description of a component in markdown

@link - specifies a link to a URL or a type in the API reference.

@example - specifies an example that will be formatted as a code block

@deprecated - specifies that the following code is deprecated and should not be used

@this - specifies what this refers to in the context of a documented function

The type must be wrapped in {} curly braces; e.g. {Object|Array} Parameters can be made optional by putting the [name] in square brackets; e.g. @param {boolean} [ownPropsOnly=false] Descriptions can contain markdown formatting

Add copyright info at the top of the main JavaScript file

/\*\*

 \* @preserve Copyright © MeanHub, Inc. All rights reserved.

 \*/

Comments

Use single line comments above a logical block of code wherever needed. If you need to put multiline comments then use multiple single line comments. Do not use comment beside a code. Comments are always preceded by a blank line and should start with a capital letter.

var balance = principal + interst; //don’t write comment here

// Add principal and interest to see balance

var balance = principal + interest;

// Some detailed comments about what the below

// code is doing

var balance = principal + interest;

Shorthands

while (x != null) {

you can write this shorter code (as long as you don't expect x to be 0, or the empty string, or false):

while (x) {

And if you want to check a string to see if it is null or empty, you could do this:

if (y != null && y != '') {

But this is shorter and nicer:

if (y) {

For return:

if (val) {

return foo();

} else {

return bar();

}

you can write this:

return val ? foo() : bar();

Binary boolean operators are short-circuited, and evaluate to the last evaluated term.

"||" has been called the 'default' operator, because instead of writing this:

/\*\* @param {\*=} opt\_win \*/

function foo(opt\_win) {

var win;

if (opt\_win) {

win = opt\_win;

} else {

win = window;

}

// ...

}

you can write this:

/\*\* @param {\*=} opt\_win \*/

function foo(opt\_win) {

var win = opt\_win || window;

// ...

}

"&&" is also useful for shortening code. For instance, instead of this:

if (node) {

if (node.kids) {

if (node.kids[index]) {

foo(node.kids[index]);

}

}

}

you could do this:

if (node && node.kids && node.kids[index]) {

foo(node.kids[index]);

}

or this:

var kid = node && node.kids && node.kids[index];

if (kid) {

foo(kid);

}

However, this is going a little too far:

node && node.kids && node.kids[index] && foo(node.kids[index]);

Iteration

Node lists are often implemented as node iterators with a filter. This means that getting a property like length is O(n), and iterating over the list by re-checking the length will be O(n^2).

var paragraphs = document.getElementsByTagName('p');

for (var i = 0; i < paragraphs.length; i++) {

doSomething(paragraphs[i]);

}

It is better to do this instead:

var paragraphs = document.getElementsByTagName('p');

for (var i = 0, paragraph; paragraph = paragraphs[i]; i++) {

doSomething(paragraph);

}

Appendix

Config file for JSHint

Create .jshintrc at project level folder as follows

{

"maxerr" : 500, // {int} Maximum error before stopping

// Enforcing

"bitwise" : true, // true: Prohibit bitwise operators (&, |, ^, etc.)

"camelcase" : false, // true: Identifiers must be in camelCase

"curly" : true, // true: Require {} for every new block or scope

"eqeqeq" : true, // true: Require triple equals (===) for comparison

"forin" : true, // true: Require filtering for..in loops with obj.hasOwnProperty()

"immed" : true, // true: Require immediate invocations to be wrapped in parens e.g. `(function () { } ());`

"indent" : 4, // {int} Number of spaces to use for indentation.

// It no longer provides warnings about indentation levels.

// You can still use it to set your tab-width but it will be used only for character locations in other warnings.

"latedef" : true, // true: Require variables/functions to be defined before being used

"newcap" : true, // true: Require capitalization of all constructor functions e.g. `new F()`

"noarg" : true, // true: Prohibit use of `arguments.caller` and `arguments.callee`

"noempty" : true, // true: Prohibit use of empty blocks

"nonew" : false, // true: Prohibit use of constructors for side-effects (without assignment)

"plusplus" : false, // true: Prohibit use of `++` & `--`

"quotmark" : "single", // Quotation mark consistency:

// false : do nothing (default)

// true : ensure whatever is used is consistent

// "single" : require single quotes

// "double" : require double quotes

"undef" : true, // true: Require all non-global variables to be declared (prevents global leaks)

"unused" : "strict", // true: Require all defined variables be used

"strict" : false, // true: Requires all functions run in ES5 Strict Mode

"maxparams" : 5, // {int} Max number of formal params allowed per function

"maxdepth" : false, // {int} Max depth of nested blocks (within functions)

"maxstatements" : 50, // {int} Max number statements per function

"maxcomplexity" : false, // {int} Max cyclomatic complexity per function

"maxlen" : 120, // {int} Max number of characters per line

// Relaxing

"asi" : false, // true: Tolerate Automatic Semicolon Insertion (no semicolons)

"boss" : true, // true: Tolerate assignments where comparisons would be expected

"debug" : true, // true: Allow debugger statements e.g. browser breakpoints.

"eqnull" : true, // true: Tolerate use of `== null`

"es5" : false, // true: Allow ES5 syntax (ex: getters and setters)

"esnext" : false, // true: Allow ES.next (ES6) syntax (ex: `const`)

"moz" : false, // true: Allow Mozilla specific syntax (extends and overrides esnext features)

// (ex: `for each`, multiple try/catch, function expression…)

"evil" : false, // true: Tolerate use of `eval` and `new Function()`

"expr" : true, // true: Tolerate `ExpressionStatement` as Programs

"funcscope" : false, // true: Tolerate defining variables inside control statements"

"globalstrict" : false, // true: Allow global "use strict" (also enables 'strict')

"iterator" : false, // true: Tolerate using the `\_\_iterator\_\_` property

"lastsemic" : false, // true: Tolerate omitting a semicolon for the last statement of a 1-line block

"laxbreak" : false, // true: Tolerate possibly unsafe line breakings

"laxcomma" : false, // true: Tolerate comma-first style coding

"loopfunc" : false, // true: Tolerate functions being defined in loops

"multistr" : false, // true: Tolerate multi-line strings

"proto" : false, // true: Tolerate using the `\_\_proto\_\_` property

"scripturl" : false, // true: Tolerate script-targeted URLs

"shadow" : false, // true: Allows re-define variables later in code e.g. `var x=1; x=2;`

"sub" : false, // true: Tolerate using `[]` notation when it can still be expressed in dot notation

"supernew" : false, // true: Tolerate `new function () { ... };` and `new Object;`

"validthis" : false, // true: Tolerate using this in a non-constructor function

// Environments

"browser" : true, // Web Browser (window, document, etc)

"couch" : false, // CouchDB

"devel" : true, // Development/debugging (alert, confirm, etc)

"dojo" : false, // Dojo Toolkit

"jquery" : true, // jQuery

"mootools" : false, // MooTools

"node" : false, // Node.js

"nonstandard" : false, // Widely adopted globals (escape, unescape, etc)

"prototypejs" : false, // Prototype and Scriptaculous

"rhino" : false, // Rhino

"worker" : false, // Web Workers

"wsh" : false, // Windows Scripting Host

"yui" : false, // Yahoo User Interface

// Custom Globals

"globals" : { "angular": false } // additional predefined global variables

}

Config file for JSCS

Create .jscsrc at project level folder as follows

{

"disallowCommaBeforeLineBreak": false,

"disallowDanglingUnderscores": false,

"disallowEmptyBlocks": true,

"disallowImplicitTypeConversion": [ "string" ],

"disallowKeywordsOnNewLine": [ "else" ],

"disallowKeywords": [ "with" ],

"disallowMixedSpacesAndTabs": true,

"disallowMultipleLineBreaks": false,

"disallowMultipleLineStrings": true,

"disallowMultipleVarDecl": true,

"disallowPaddingNewlinesInBlocks": true,

"disallowQuotedKeysInObjects": true,

"disallowSpaceAfterBinaryOperators": true,

"disallowSpaceAfterKeywords": [ "for", "while", "do", "switch" ],

"disallowSpaceAfterLineComment": false,

"disallowSpaceAfterObjectKeys": true,

"disallowSpaceAfterPrefixUnaryOperators": [ "!" ],

"disallowSpaceBeforeBinaryOperators": false,

"disallowSpaceBeforeBlockStatements": null,

"disallowSpaceBeforePostfixUnaryOperators": [ "++", "--" ],

"disallowSpacesInAnonymousFunctionExpression": {

"beforeOpeningRoundBrace": true

},

"disallowSpacesInConditionalExpression": false,

"disallowSpacesInFunctionDeclaration": {

"beforeOpeningRoundBrace": true

},

"disallowSpacesInFunctionExpression": {

"beforeOpeningRoundBrace": true

},

"disallowSpacesInNamedFunctionExpression": {

"beforeOpeningRoundBrace": true

},

"disallowSpacesInsideArrayBrackets": true,

"disallowSpacesInsideObjectBrackets": true,

"disallowSpacesInsideParentheses": true,

"disallowTrailingComma": true,

"disallowTrailingWhitespace": true,

"disallowYodaConditions": true,

"maximumLineLength": 120,

"requireBlocksOnNewline": true,

"requireCamelCaseOrUpperCaseIdentifiers": true,

"requireCapitalizedConstructors": true,

"requireCommaBeforeLineBreak": true,

"requireCurlyBraces": [ "if", "else", "for", "while", "do", "try", "catch" ],

"requireDotNotation": true,

"requireKeywordsOnNewLine": null,

"requireLineFeedAtFileEnd": true,

"requireMultipleVarDecl": true,

"requireOperatorBeforeLineBreak": true,

"requirePaddingNewlinesInBlocks": false,

"requireParenthesesAroundIIFE": true,

"requireSpaceAfterBinaryOperators": true,

"requireSpaceAfterKeywords": [ "if", "else", "return", "try", "catch" ],

"requireSpaceAfterLineComment": true,

"requireSpaceAfterObjectKeys": false,

"requireSpaceAfterPrefixUnaryOperators": null,

"requireSpaceBeforeBinaryOperators": true,

"requireSpaceBeforeBlockStatements": true,

"requireSpaceBeforePostfixUnaryOperators": null,

"requireSpacesInAnonymousFunctionExpression": null,

"requireSpacesInConditionalExpression": true,

"requireSpacesInFunctionDeclaration": {

"beforeOpeningCurlyBrace": true

},

"requireSpacesInFunctionExpression": {

"beforeOpeningCurlyBrace": true

},

"requireSpacesInNamedFunctionExpression": {

"beforeOpeningCurlyBrace": true

},

"requireTrailingComma": false,

"safeContextKeyword": true,

"validateIndentation": 4,

"validateJSDoc": {

"checkParamNames": true,

"requireParamTypes": true

},

"validateLineBreaks": "CRLF",

"validateQuoteMarks": "'"

}

Config file for JSBeautify

Create .jsbeautifyrc at project level folder as follows

{

"braceStyle": "collapse",

"breakChainedMethods": false,

"e4x": false,

"evalCode": false,

"indentChar": " ",

"indentLevel": 0,

"indentSize": 4,

"indentWithTabs": false,

"jslintHappy": false,

"keepArrayIndentation": false,

"keepFunctionIndentation": false,

"maxPreserveNewlines": 3,

"preserveNewlines": true,

"spaceBeforeConditional": true,

"spaceInParen": false,

"unescapeStrings": false,

"wrapLineLength": 120

}

Config node package

Create package.json at project level folder as follows. Add/delete modules as needed by the project

{

"name": "NeonLite.Compatible",

"version": "1.0.0",

"description": "HTML app for generating MeanHub reports",

"main": "all.js",

"dependencies": {

"gulp-ruby-sass": "^1.0.1",

"gulp": "^3.8.11",

"js-beautify": "^1.5.6",

"yargs": "^3.7.2"

},

"devDependencies": {

"gulp": "^3.9.0",

"gulp-jsbeautifier": "0.0.8"

},

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

"author": "Souvik Basu <souvikbasu@gmail.com>",

"license": "Copyright © MeanHub, Inc. All rights reserved."

}

Run npm install to install the packages mentioned in the package.json file.

Config gulp file

Create gulpfile.js at project level folder as follows. Modify file paths as per project structure

var gulp = require('gulp');

var prettify = require('gulp-jsbeautifier');

var jsFiles = [

'static/app.js',

'static/config.js',

'static/init.js',

'static/utility.js'

];

var htmlFiles = [];

var cssFiles = [];

gulp.task('format-js', function () {

gulp.src(jsFiles, { base: './' })

.pipe(prettify({ config: '.jsbeautifyrc', mode: 'VERIFY\_AND\_WRITE' }))

.pipe(gulp.dest('./'));

});

gulp.task('prettify-html', function () {

gulp.src(htmlFiles, { base: './' })

.pipe(prettify({ indentSize: 2 }))

.pipe(gulp.dest('./'));

});

gulp.task('prettify-css', function () {

gulp.src(cssFiles, { base: './' })

.pipe(prettify({ indentSize: 2 }))

.pipe(gulp.dest('./'));

});

Run gulp format-js to beautify JavaScript files.

To run gulp tasks from Visual Studio, install VS extension at <https://visualstudiogallery.msdn.microsoft.com/8e1b4368-4afb-467a-bc13-9650572db708>