JavaScript
Best Practices Guide

Version: 2013.11.19

This section includes best practices regarding development of JavaScript source code. Details on each checklist item are discussed later in the document.

If your project uses AMD module loaders, please also see the ***Javacript - AMD*** document.

Checklist

General Recommendations

- ❑ Avoid global variables
- ❑ Define all local `var`'s at the top of each function
- ❑ Use explicit comparators (ie "===") over coerced comparators (ie "==")
- ❑ Use shortcuts for default value assignment
- ❑ Always put braces on the right of block definitions
- ❑ Always use semi-colons at the end of a line
- ❑ Never leave dangling commas in lists and object definitions
- ❑ Never use "eval()"
- ❑ Never leave `debugger` statements in code
- ❑ Always follow standard variable and function naming conventions
- ❑ Never use hyphens in variable names
- ❑ Never test variables against text strings for triggered events

- ❑ Never hard code strings within program code or templates
- ❑ Always use double quotes for NLS'd strings
- ❑ Never hard code usernames or passwords in client processed code
- ❑ Never persist passwords locally

Logging and Comments

- ❑ Use standard logging with useful details
- ❑ Use the proper logging levels
- ❑ Use multiple arguments to log, not string concatenation
- ❑ Comment all complex logic
- ❑ Use JSDoc syntax for all public module, function, and variables

Performance

- ❑ Be suspicious of logic inside loops
- ❑ Keep the DOM as small as possible
- ❑ Minimize DOM access / manipulation
- ❑ Ensure all 3rd party libraries are optimized

Supplemental Discussions

- ❑ App Startup

- ❑ Linting

Discussions

> *Just because JavaScript permits sloppy programming, it is not a license to exploit such poor practice.*

JavaScript is a powerful language, but care should be taken to write good clean code. Since JS is loosely typed, which enables variables to be easily coerced into different types, it is far too easy to inadvertently convert a string to a number, or worse a NaN (Not a number!).

```
var s = "Hello World";
// Well this looks like it would remove "World" from the string
var x = s - " World";
console.log("Hello: ", x);   // Displays: "Hello: NaN" !?!
```

This error caused by trying to remove a substring using the above syntax -- which is valid in some languages -- looks proper. But, JavaScript thinks you want to perform a math operation, and the results are not a valid number giving us a `NaN` primitive. The point is to be careful, and know your language.

General Recommendations

The following are general recommendation on JavaScript coding practices.

Avoid global variables

Always use the `var` statement to define local variables. Otherwise the variable is placed into the global scope, which is almost always bad.

This does not apply to browser provided globals such as: `window`, `navigator`, `document`, `console`, etc.

Using AMD based loaders, eliminates globally namespaced packages/modules.

Define all local var's at the top of each function

This is not required but leads to cleaner code and shows intent by the programmer. Local variables are at “function” scope, meaning they are accessible anywhere within the enclosing function, not just within enclosing `{ ... }` blocks like most languages. Random “`var`” definitions throughout a function are sloppy and can lead to errors, such as unintended variable re-assignment.

This also applies to `for( var i in ...) { ... }` blocks as well. The “i” variable is not scoped to the for block, but its outer containing function. This is just such a common practice in most other languages, that people expect this to be the case in JavaScript too. Its not, and can result in unintended side-effects.

Use explicit comparators (ie ===) over coerced comparators (ie ==)

Using coerced comparators actually says convert the value types to match and then compare them. All the following evaluate to true:

- `"hello" == 1`
- `"" == 0`
- `1 == true`

Using the triple equals (`===`), or not equals (`!==`) forces a pure comparison of explicit values with no coercion.

Use shortcuts for default value assignment

Make your code faster and more readable by using logical OR (“`||`”) during assignment operations to allow for default values.

Example: Do a simple validation on input args. This ensures we can work on an expected input object

```
// "args" defaults to empty object if not defined
args = args || {};
// We know args is an object so this is safe to do
var greeting = args.greeting || "Good day";
```

Another option on the second example is to “mixin” a “defaults” map into the arguments object. Mixins are provided by most libraries, or can be done simply by iterating the values map, and applying the values to the target map.

```
var defaultValues = { a:1, b:true, c:"hello" };
args = args || {};
lang.mixin( args, defaultValues );
```

Put braces on the right of block definitions.

This is not a personal style choice. Its simply safer to do this in JavaScript. Taken from a Talk by Doug Crawford

- Good:
```
return {
  ok : true
};
```

- Bad:
```
return     // returns undefined, not the expected object!
{          // This block is valid, but no-ops!
  ok : false
};
```

Never use “eval()”

As the saying goes -- “eval is evil”. With that said, you should rarely if ever use it directly. Its far to easy for bad things to happen.

If you truly know what you are doing, and must use the eval() call, the source must be from trusted source to avoid security vulnerabilities.

Never leave dangling commas in lists and object definitions

This used to cause IE to throw errors. All browsers now allow it, but its sloppy programming. This issue is commonly seen in define/require statements and after the last method/function of a declared class.

Bad Examples:

- `var x = [0,1,2,3,];`
- `var y = {a:1, b:”B”, c:x, };`

Tip when generating visual lists, use an Array and join the results. Example:

```
console.log( "List of names: ", nameList.join(", ") );
```

Proper JSON objects will fail validation and parsing due to dangling commas as well, so just don’t do it anywhere.

Always use semi-colons at the end of a line

Yes, JavaScript will do a best guess on end of lines, so they are not technically required. But, it does not always infer the programmer’s intention. So don't guess how it will be interpreted by the compiler.

Never leave debugger statements in code

The "`debugger`" statement can be used to force a browser breakpoint during manual testing. This can aid developers in quickly locating bugs, but it will cause optimization build failures if left in the code. Its better to locate the desired code location and manually set breakpoint using browser tools.

Always follow standard variable and function naming conventions

- Normal variables are: `camelCase`
- Private variables use leading underscore: `_myPassword`
- Constants are all uppercase: `var PI = 3.14159;`

Never use hyphens in variable names.

These can get confused with subtraction attempts.

Example:

```
var hello-world = "Hi there";
```

Results in an attempt to subtract world from hello, throwing errors.

Never test variables against text strings for triggered events

Condition checking should be made against variables or to the reference node that triggered an event, rather than against multilingual values. This will reduce the chance of errors when UI values change, but the logic checking for them is mistakenly not updated. It should also improve the maintainability of the code itself by actually clearly showing what the actual conditions are for a given action.

Example: Consider checking against an Enumeration rather than one of two text translations. e.g. This is brittle coding:

```
if (acct.type.match(/^LINE OF CREDIT.*/) ||
    acct.type.match(/^LIGNE DE CREDIT.*/)) {
```

This would be a much better and extensible way of testing this condition

```
if (acct.type === this.acct.TYPES.credit) {
```

Another common anti-pattern is checking the target.label of an onClick event to determine the source button. It is much better to use an "id", "class", or other identifier that won't break when the app gets internationalized.

Never hard code strings within program code or templates

Always assume your app will be translated eventually. At the very least, create a local map of key values for strings. Example:

```
this.text = {
  "title"  : "Greetings",
  "firstName" : "First name",
  …
};
```

Always use double quotes for NLS'd strings.

Many languages make heavy use of single quotes as part of the language. Translators typically do not know "JSON" style stucture, and will introduce invalid strings when apostrophes are introduced to single quoted string as shown below:

English base: `{ "finish" : 'The end of the story' }`
French: `{ "finish" : 'La fin de l'histoire' } //Error!`

Never hard code usernames or passwords in client processed code

This is sloppy and lazy. Even test and demo users and passwords left in the code can become security holes.

Never persist passwords locally

Never persist passwords in localStorage / sessionStoreage / Cookies. Even encrypted -- or much worse, encoded -- storing passwords creates security holes that can be exploited by an attacker. Let the user determine if/when passwords are persisted through normal browser mechanisms.

Server-side resources may also require passwords. Again, there is never a valid reason to persist these on the client. Do not do it!

Logging and Comments

Logging and comments should be used liberally throughout your code. There is no performance penalty for doing either, as they are generally removed during the build phase. Some may argue that logging is a substitute for good debugging skills and code stepping, but general consensus says it can be a valuable development aid in logic comprehension.

Use standard logging with useful details

Developers should be liberal about logging using `console.log()` and its sister functions. When logging, make the contents meaningful. You should include the source module and function, a message containing descriptive details, along with any stateful variables.

Never use comments like "`@@@@@@@ I am here!!!! @@@@@@@`". It is sophomoric, provides no valuable information, and is replacing proper debugging techniques. On a similar vein, do not use alert() statements as a substitute for logging. They tend to get left in the code and can sometimes embarrassingly make it into production.

Use the proper logging level

The console and WL.logger both support various levels for logging. Examples are:

- debug
- log
- warn
- error

By using the proper level for a given situation you can manage what gets logged better. During a build, the default for Dojo is to remove all logging below "error". This means that any console.error() calls will remain in the code. But, a common anti-pattern is to use console.log() calls everywhere, even within a failure's catch() block. The problem, is after a build, any potential error conditions that would be logged in development, are now removed from the production code and can be lost.

Use multiple arguments to log, not string concatenation

A common anti-pattern is to use string concatenation within console logging as shown here:

```
console.error("Invalid arguments provided: " + args);
```

This fails, as the actual output generated to has run a toString() on the args objects, and will likely show up as

```
Invalid arguments provided: [Object]
```

Well, that didn't help us much did it? Instead use, multiple arguments and the console will show the actual object (or a JSON version of the object under hybrid apps.)

```
console.error("Invalid arguments provided:", args);
```

Comment all complex logic

JavaScript code is generally pretty readable. Sometimes code gets complex to solve complex problems. It is not always practical to refactor code just to make it more understandable to the novice developer. Conversely, if a seasoned programmer cannot grok your “elegant” (read “overly complex”) code quickly, then you might be obfuscating for the sake of pride. Regardless, comment your complex code to aid comprehension. It just makes life easier for everyone.

Regular expressions are another area where regardless of its complexity, can be difficult to figure out quickly. Put a simple comment in, and everyone will know what the RegExp is intending to accomplish.

Use JSDoc syntax for all public module, function, and variables

JSDoc is way to annotate JavaScript code. You should always properly comment your public functions and variables so that documentation can be generated. Private functions and variables should also be properly documented as well, but by their nature are less important in this regard. Any reusable code that might eventually be contributed to a library (public or private) must be properly documented. No exceptions.

Note: Prior to version 2, the Dojo Toolkit used its own special syntax for source documentation. As of version 2, Dojo is adopting JSDoc syntax like the rest of the world.

Performance

Be suspicious of activity inside loops

Iterating through data or elements using a loop is normal. But, consider everything that happens within that loop acts as a multiplier in terms of potential negative performance. Look for obvious performance evils occurring within loops such as DOM access / manipulations, storage access, AJAX calls, math calculations,etc. See if you can refactor any of these heavy burdens outside of the loop.

Keep the DOM as small as possible

Mobile app performance drops as the DOM size and complexity grows. Therefore, you should be proactive in maintaining a small DOM. Keeping the DOM as small is possible greatly improve rendering and reflow operations. This can be done by destroying views after they are transitioned out, removing unneeded nodes, etc.

Two basic options are to:

- Destroy the view and all its children, saving state if needed. This can actually get quite complex to manage if the child widget and/or state is complex. But, if its

unlikely the user will revisit the view again, or there is little stateful concerns, then prune the fat, and clean up memory and DOM resources.
- Use a DomCache. This involves offloading the view's node tree to a "domCache" (DomFragment) variable after they have been transitioned out. Then if the view is needed again, check the cache and reinsert it back into the master DOM. This is good for views likely to be revisited, or that require a lot of state and setup.

Minimize DOM access / manipulation

The DOM is very slow compared to everything else in your app. Try to minimize access to the DOM as much as possible. If you will be accessing a certain node frequently, then get it once (eg document.getElementByID), and store the results in a local variable. Also, its best to build up a large DOM fragment and insert it into the DOM once, rather than iterating and building nodes directly into the DOM.

Ensure any external libraries are pre-optimized

Be aware that you not only need to optimize your own custom JavaScript, but you must also ensure that you are using optimized version of any 3rd party libraries. Typically, minimized libraries have a ".min" in their names. Look for that or the libraries web site to ensure you are using an optimized version of the library instead of the developer's source version.

Performance References:

- Google's Make the Web Faster
- Mobile App performance
- Performance Tooling - An excellent talk by Paul Irish of Google.

App Startup

First impressions are critical for any Web App. This is even more important for attention distracted mobile users. Your app must load fast enough to keep the user from doing something else, or simply abandoning the app altogether. Application startup time is critical to users, and should always be a primary concern during development. If your app starts too slowly, user may never actually wait to give it a fair chance. There are many approaches to improve app startup as shown below.

Load only enough to put something interesting on the glass

Carefully examine and control what gets loaded during startup. Using careful examination, you can extract critical HTML, CSS, and JavaScript modules for the startup

phase. There is plenty of time later to load everything else. It is a very common anti-pattern to load everything up front. While this makes for simple app management, its a crush on the browser, and user satisfaction will suffer.

- Create a simple bootstrap module that can get the user to the initial home or login screen. Do not bundle everything into a master layer here. You want the bare minimum.
- The initial DOM body should be as minimal as possible. Do not include the entire site's collection of potential views by default. You can inject them later.
- Secondary splash or loading screens should be avoided if possible. These are annoying and actually increase the perceived loading time.

Load anything else needed immediately in the background.

Once the initial app has loaded and the user is looking at a main landing / login page, you can then load other important modules and styles in the background. Users will typically look at the initial view for at least a couple seconds before taking any action. This user based delay can be exploited to your benefit. Now is the time to load any secondary modules, extra CSS, updated messages, real time data, etc.

Load everything else either on demand, or pre-fetch on likely user flow

Use analytics to determine typical story flows. Once the app is started, on demand loading of views and resources is typically acceptable performance wise. You really don't want to load everything if the user is only going to visit 20% of the entire app's full potential.

If a user is likely to go from view A to view C, go ahead and preload view C in the background. The user experience will be improved as there is no delay on the new view transition.

Remove views from memory when no longer needed.

This is not really a startup issue, but is related enough to loading to be considered here. This is especially valuable for CPU and memory constrained mobile devices. See the discussion above on keeping the DOM small.

Linting

Lint is a classic method for static code analysis. Since JavaScript is unfortunately forgiving with sloppy syntax, that can lead to hard to depict errors, Linting tools should be part of every web developer's toolbox. Many IDEs and editors do a good job of detecting syntax errors, but there is still a place for standalone command line invocable Lint checkers as part of continuous

integration.

Listed below are several Lint tools specifically for JavaScript

JSLint - The grandfather of JavaScript lint tools by Doug Crockford. Sometimes frowned upon by frontend developers as too opinionated, and newer tools are now available that perform similar behavior. It will also do a good job of detecting invalid HTML code. Read the JSLint documentation. Then re-read it until you understand and accept (almost) all the rules defined. You will be a better JavaScript programmer afterwards.

JSHint - A popular fork of the original JSLint. Available as an extension for many editors, as well as an eclipse plugin.

Recommended JSHint options

```
{
      "asi"          : false,
      "bitwise"      : false,
      "boss"         : true,
      "browser"      : true,
      "camelcase"    : true,
      "couch"        : false,
      "curly"        : true,
      "debug"        : false,
      "devel"        : true,
      "dojo"         : true, // false if not using Dojo
      "eqeqeq"       : true,
      "eqnull"       : true,
      "es5"          : true,
      "evil"         : true,
      "expr"         : true,
      "forin"        : false,
      "globalstrict" : false,
      "immed"        : true,
      "indent"       : 4,
      "iterator"     : true,
      "jquery"       : true, // false is not using Jquery
      "lastsemic"    : false,
      "latedef"      : true,
      "laxbreak"     : true,
      "loopfunc"     : true,
      "maxdepth"     : 3,
      "maxparams"    : 3,
      "newcap"       : true,
```

```
    "noarg"        : true,
    "noempty"      : true,
    "nonew"        : true,
    "nonstandard"  : true,
    "nomen"        : false,
    "onecase"      : false,
    "onevar"       : false,
    "passfail"     : false,
    "plusplus"     : false,
    "proto"        : true,
    "quotmark"     : "single",
    "regexp"       : false,
    "regexdash"    : true,
    "scripturl"    : true,
    "shadow"       : false,
    "smarttabs"    : true,
    "strict"       : false,
    "sub"          : true,
    "supernew"     : true,
    "trailing"     : true,
    "validthis"    : true,
    "undef"        : false,
    "unused"       : true,
    "white"        : false
    "wsh"          : false
}
```

One or more of the following toolkit specific entries should be added and set to true as appropriate: `dojo, jquery, mootools, node, prototypejs, rhino`

ESHint - A new command line JS linter that enables a plugable list of tests. It shows promise as its more flexible than JSHint, based on NodeJS and AMD and the popular Esprima AST parser.

JSONLint Although JSON formatted documents are not formally compliant to the JavaScript syntax, they are commonly used in front-end projects and this lint tool does a great job of catching common problems in JSON files.