Coding Style guidelines

(AngularJS & ES6)

Table of Contents

[Overview 3](#_Toc448739578)

[Angular coding style/best practices 3](#_Toc448739579)

[es6 coding style/best practices 5](#_Toc448739580)

[ESlint options 14](#_Toc448739581)

# Overview

This document is intended to provide consistent and helpful coding conventions for the Drysdale Web development.

Babel - is used to transpile the code written in ES6 to ES5.

ESLint – is the linting tool of choice to help enforce the consistent coding standards across the teams. The options (in the .eslintrc file) are appended to this document in the end.

# Angular coding style/best practices

1. **Single Responsibility**
   1. Define only 1 component per file. The component can be any of the following – Module, Service, Controller, Directive, Component, etc.
2. **Controllers**
   1. **Keep Controllers Focused** - Define a controller for a view, and try not to reuse the controller for other views. Instead, move reusable logic to factories and keep the controller simple and focused on only its view.
   2. **Controller Logic** – Keep controller logic simple. Avoid any controller logic for making data operations and interacting with data to a factory. Make data services responsible for XHR calls, local storage, stashing in memory, or any other data operations
   3. **Use dependency injection** properly to inject the required services into the controllers to make it reusable
3. **Directives**
   1. **One Directive per file**
   2. **Clean up** - Directives should clean up after themselves. Use element.on('$destroy', ...)  to run a clean-up function when the directive is removed.
   3. **Manipulate DOM in a Directive** - When manipulating the DOM directly, use a directive. If alternative ways can be used such as using CSS to set styles or the [animation services](https://docs.angularjs.org/api/ngAnimate), Angular templating, [ngShow](https://docs.angularjs.org/api/ng/directive/ngShow) or [ngHide](https://docs.angularjs.org/api/ng/directive/ngHide), then use those instead. For example, if the directive simply hides and shows, use ngHide/ngShow
   4. **Provide a unique Directive prefix** - Provide a short, unique and descriptive directive prefix such as salesCustomerInfo which would be declared in HTML as sales-customer-info
4. **$rootScope**
   1. Avoid using or adding any kind of data into the $rootScope
   2. Do not broadcast/emit from the $rootScope to propagate events to the child controllers. If you have a use case for it, let’s discuss as a team before implementation. Instead, use $scope.emit, etc.
   3. event.stopPropogation();
5. **Angular $ wrapper services**
   1. **$document and $window –** Use the Angular’s wrapped services instead of document and window, as it is more easily testable and helps avoid having to mock these
   2. **$timeout and $interval -** These services are wrapped by Angular and more easily testable and handle Angular's digest cycle thus keeping data binding in sync.
6. **Angular’s Service vs Factory**
   1. Essentially, factories are functions that *return the object*, while services are *constructor functions of the object* which are instantiated with the new keyword. Both are singletons in the app
   2. The `module.service()` method was specifically designed when services are defined as classes (or instantiable types)
   3. With that knowledge, **Always use Service** whenever youneed to share any reusable logic. Since the service is simply a constructor function we can write the services using the ES6 classes and inject them into the angular module

class MyService {

sayHello() {

console.log('hello');

}

}

appModule.service('MyService', MyService);

1. **ng-if vs ng-show**
   1. **Always use ng-if**, as it removes the elements from the DOM and all the associated bindings and handlers. Thereby better performance as the DOM is not cluttered with unwanted elements
   2. Ng-if is executed at a higher priority than ng-show/ng-hide and it also creates a child scope
   3. Use ng-show/ng-hide, if for some reason you want to preserve the handler even when the element is hidden.
2. **Global vs Module level constants**
   1. Create constants and refer them in the app instead of hard-coding values directly in the code
   2. **Create one app level constant file for global constants,** andimport it in the app component and inject it into the appModule.constant(‘CONSTANT’, {})
   3. Do the same thing for the module level constants as well, if needed
3. **Error Handling**
   1. Always make sure to check for null object before accessing it in the code especially if the data s coming from an API
   2. The application should not throw any undefined errors under any scenario. So, handle them with proper try/catch if you’re expecting a failure in some situations
   3. We’ll have a global exception handler at the app level to catch all Javascript errors and log it to the console and to the backend in case of any failures. As a developer, make sure to check for errors in your browser’s console at all times

# es6 coding style/best practices

1. **Naming Convention**
   1. **Use camelCase** for functions and variables. -- ***likeThis***
   2. **Use PascalCase** for classes / constructor functions. --- ***LikeThis***
   3. Use all lowercase for file names in order to avoid confusion on case-sensitive platforms. Filenames should end in .js and should contain no punctuation except for - delimiters.
   4. **Private properties and methods** must be formatted ***\_likeThis, \_likeThis()***
   5. **For constant** values use **const** declaration and ALL\_CAPITAL\_ CASE naming convention --- ***LIKE\_THIS.*** e.g.,const NORMAL\_FILE\_TYPE = 0;
2. **Variables**
   1. var must not be used.
   2. **const must be used by default**. Use let only when variable will be (re)assigned a different value later in the code
   3. const (or let) must be used exactly once per variable declaration. (eslint rule: one-var)

// avoid

var a = 1;  
const a = 1, b = 2, c = 3;

// recommended

const a = 1;

const b = 2;

let c = 3;

1. **One Class per file**
   1. Define only 1 class per file and the file name should match the class name. If the class is named "Car" then the file should be named "car.js"
2. **Ternary Operators**

Ternary operator (?:) should be avoided. It must not be used if:

* 1. The whole expression cannot fit on one line
  2. The expression involve other ternary operators
  3. The members of the expression are too complex to keep it readable

// avoid

const foo = fooBar(bar + qux) \* qux ? fooBar(barFoo({ bar })[qux]).foo : fooBar(barFoo({ qux })[foo]).baz;

// avoid

const foo = qux ? (bar ? qux : baz) : bar;

// acceptable

const foo = bar === baz ? bar + 1 : baz - 1;

// recommended

const foo = bar ? baz : qux;

1. **If/else vs Switch construct vs Object literals**
   1. Avoid using switch construct , use the **if-else if-else** instead. There a multiple issues with the switch statements like forced break statements, fall-through, etc.
   2. If you end up writing many if statements (>4), consider using [Object literals](https://toddmotto.com/deprecating-the-switch-statement-for-object-literals/) (refer to the [link](https://toddmotto.com/deprecating-the-switch-statement-for-object-literals/))

// avoid

var type = 'coke';

var drink;

switch(type) {

case 'coke':

drink = 'Coke';

break;

case 'pepsi':

drink = 'Pepsi';

break;

default:

drink = 'Unknown drink!';

}

console.log(drink); // 'Coke'

// recommended – for simple String

function getDrink (type) {

const drinks = {

'coke': 'Coke',

'pepsi': 'Pepsi',

'lemonade': 'Lemonade',

'default': 'Default item'

};

return 'The drink I chose was ' + (drinks[type] || drinks['default']);

}

var drink = getDrink('coke');

console.log(drink); //The drink I chose was Coke

// recommended – for complex functions

function getDrink (type) {

const drinks = {

'coke': function () {

return 'Coke';

},

'pepsi': function () {

return 'Pepsi';

},

'lemonade': function () {

return 'Lemonade';

},

'default': function () {

return 'Default item';

}

};

return (drinks[type] || drinks['default'])();

}

var drink = getDrink('coke');

console.log(drink); // Coke

1. **Undefined**
   1. Do not use undefined in the conditional statements, instead use "void 0", It's a little faster and simpler

// avoid

if (typeof obj === 'undefined') or if(obj === undefined)

// recommended

if (obj === void 0)

1. **Do not cuddle else or catch**

|  |  |
| --- | --- |
| // recommended if (foo) {  bar(); }  else if (baz){  qix(); }  else {  qux();  } | // recommended  try {  bar();  }  catch (err) {  baz();  } |

1. **Loops**
   1. Prefer array methods to avoid loops, if you need loop for whatever reason prefer "for of" and if it's not a good fit then while, plain for loops is a last resort.

// avoid

const count = xs.length;

let sum = 0;

for (let i = 0; i < count; i++) {

sum = sum + xs[i];  
}

// good

let sum = 0;

for (let x of xs) {

sum = sum + x

}

// recommended

xs.reduce((sum, x) => sum + x)

1. **Functions**
   1. Regular (**named and hoisted**) function declarations should be used instead of anonymous functions by default.
   2. **new** **must not** be used with functions which are not ES6 classes.

// avoid

function A() {}

const a = new A();

// recommended

class A {}

const a = new A();

* 1. When using anonymous functions, **arrow function should be used** instead of anonymous function

const a = [1, 2, 3];

// avoid

i) for (let i = 0; i < a.length; i++){ ... }  
ii) a.reduce(function(x, y) { return x + y; }, 0);

Just use the arrow function instead -

// recommended

a.reduce((x, y) => x + y, 0);

* 1. **Parens must be used for arrow functions parameters**, even when there is only one parameter

// avoid

const triple = x => 3\*

// recommended

const triple = **(**x**)** => 3\*x;

* 1. **Use Arrow functions to preserve "this"**

When using anonymous functions, use arrow functions. Arrow functions should be used instead of Function.prototype.bind when applicable. self / \_this / that must not be used

|  |
| --- |
| // avoid  function FooCtrl (FooService) {  this.foo = 'Hello';  FooService.doSomething(function (response) {  this.foo = response;  }.bind(this)); //passing the context  } |
| // avoid  function FooCtrl (FooService) {  var that = this; //storing the reference to this  that.foo = 'Hello';  FooService  .doSomething(function (response) {  that.foo = response;  });  } |
| // recommended  function FooCtrl (FooService) {  this.foo = 'Hello';  FooService.doSomething((response) => {  this.foo = response;  });  } |

* 1. **Reuse functions** where possible, creating closures on every call has worse performance

// avoid

const foo = (nums) => {

return nums.filter(function(x) {

return x % 2;

}).reduce(function(a, b) {

return a + b;

});

}

// recommended

const isOdd = (x) => x % 2

const sum = (x, y) => x + y

const foo = (nums) => nums.filter(isOdd).reduce(sum);

* 1. **Function Parameters -** Use default values for parameters when needed and put default parameters at last

// avoid

function f(param) {

if (param === void 0){ // unwanted param or arguments check

param = 12;

}

console.log(param);

}

// recommended   
function runApp(param = 12) {

console.log(param);

}

OR

function runApp(x, y, param = 12) {

console.log(param);

}

1. **Promises**
   1. Do not use the callback form for the async calls like "asyncCall.then(successFunction, errorFunction)". Instead, always **use the .then() & .catch()**form

asyncCall()

.then(function(result) {

// handle success

})

.then(function(result) {

// handle success

})

.catch(function(err) {

// handle error

});

The catch function catches all errors including the ones that might happen inside the .then() handlers – Source: [Promises-Catch](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/catch)

* 1. Do not use nested promises, **use Promise.all()** instead
* **Waiting until all promises are fulfilled** -

Promise.all([asyncCall1, asyncCall2])   
  .then(function(results) {  
    // do something with result1 and result2  
    // available as results[0] and results[1] respectively  
  })  
  .catch(function(err) { /\* ... \*/

    //handle error from both calls

  });

* **Data dependency -** If the second async call depends on the data from the first call, Resolve the first async call and pass it to the second call

asyncCall1()

.then(function(result1) {

return Promise.all([result1, asyncCall2(result1)]);

})

.then(function(results) {

// do something with results array

})

.catch(function(err){

//handle error

});

* **Do not assume when the async call will be fullfilled** in the .then() handler, instead always return a promise from the first handler if you want to make second handler depend on the first one

asyncCall1().then(function(result) {

return asyncCall2(result); //returns the promise

})

.then(function() {

asyncCall3(); // asyncCall2 has resolved

});

* **Use Promise.resolve() & Promise.reject()** to handle synchronous code that might throw error and use it in the Promise chain or use Promise.cast() when needed

function makeSyncAsync() {

return Promise.resolve().then(function(){

// execute synchronous code that might throw

return value;

});

}

# ESlint options

Our .eslintrc configurations are based on the StandardJS style guidelines along with some overrides as per our drysdale needs (refer below)

**Source:**

ESLint - <http://eslint.org/docs/rules/>

StandardJS - <http://standardjs.com/>

|  |  |
| --- | --- |
| Rules/Options | Notes |
| {  "ecmaFeatures": {  "modules": true  },  "env": {  "browser": false,  "es6": true,  "node": true  },  "globals": {  "document": false,  "navigator": false,  "window": false,  "describe": false,  "beforeEach": false,  "expect": false,  "inject": false,  "it": false,  "xit": false  },  "rules": {  "accessor-pairs": 2,  "arrow-spacing": [2, { "before": true, "after": true }],  "block-spacing": [2, "always"],  "brace-style": [2, "1tbs", { "allowSingleLine": true }],  "comma-dangle": [2, "never"],  "comma-spacing": [2, { "before": false, "after": true }],  "comma-style": [2, "last"],  "constructor-super": 2,  "curly": [2, "multi-line"],  "dot-location": [2, "property"],  "eol-last": 2,  "eqeqeq": [2, "allow-null"],  "generator-star-spacing": [2, { "before": true, "after": true }],  "handle-callback-err": [2, "^(err|error)$" ],  "indent": [2, 2, { "SwitchCase": 1 }],  "key-spacing": [2, { "beforeColon": false, "afterColon": true }],  "new-cap": [2, { "newIsCap": true, "capIsNew": false }],  "new-parens": 2,  "no-array-constructor": 2,  "no-caller": 2,  "no-class-assign": 2,  "no-cond-assign": 2,  "no-const-assign": 2,  "no-control-regex": 2,  "no-debugger": 2,  "no-delete-var": 2,  "no-dupe-args": 2,  "no-dupe-class-members": 2,  "no-dupe-keys": 2,  "no-duplicate-case": 2,  "no-empty-character-class": 2,  "no-empty-label": 2,  "no-eval": 2,  "no-ex-assign": 2,  "no-extend-native": 2,  "no-extra-bind": 2,  "no-extra-boolean-cast": 2,  "no-extra-parens": [2, "functions"],  "no-fallthrough": 2,  "no-floating-decimal": 2,  "no-func-assign": 2,  "no-implied-eval": 2,  "no-inner-declarations": [2, "functions"],  "no-invalid-regexp": 2,  "no-irregular-whitespace": 2,  "no-iterator": 2,  "no-label-var": 2,  "no-labels": 2,  "no-lone-blocks": 2,  "no-mixed-spaces-and-tabs": 2,  "no-multi-spaces": 2,  "no-multi-str": 2,  "no-multiple-empty-lines": [2, { "max": 1 }],  "no-native-reassign": 2,  "no-negated-in-lhs": 2,  "no-new": 2,  "no-new-func": 2,  "no-new-object": 2,  "no-new-require": 2,  "no-new-wrappers": 2,  "no-obj-calls": 2,  "no-octal": 2,  "no-octal-escape": 2,  "no-proto": 2,  "no-redeclare": 2,  "no-regex-spaces": 2,  "no-return-assign": 2,  "no-self-compare": 2,  "no-sequences": 2,  "no-shadow-restricted-names": 2,  "no-spaced-func": 2,  "no-sparse-arrays": 2,  "no-this-before-super": 2,  "no-throw-literal": 2,  "no-trailing-spaces": 2,  "no-undef": 2,  "no-undef-init": 2,  "no-unexpected-multiline": 2,  "no-unneeded-ternary": [2, { "defaultAssignment": false }],  "no-unreachable": 2,  "no-unused-vars": [2, { "vars": "all", "args": "none" }],  "no-useless-call": 2,  "no-with": 2,  "one-var": [2, { "initialized": "never" }],  "operator-linebreak": [2, "after", { "overrides": { "?": "before", ":": "before" } }],  "padded-blocks": [2, "never"],  "quotes": [2, "single", "avoid-escape"],  "radix": 2,  "space-after-keywords": [2, "always"],  "space-before-blocks": [2, "always"],  "space-before-function-paren": [0, {"anonymous":"always", "named":"never"}],  "space-before-keywords": [2, "always"],  "space-in-parens": [2, "never"],  "space-infix-ops": 2,  "space-return-throw-case": 2,  "space-unary-ops": [2, { "words": true, "nonwords": false }],  "spaced-comment": [2, "always", { "markers": ["global", "globals", "eslint", "eslint-disable", "\*package", "!", ","] }],  "use-isnan": 2,  "valid-typeof": 2,  "yoda": [2, "never"],  //em-ngen-master-overrides - errors & warnings  "semi": [2, "always"],  "no-extra-semi": 2,  "semi-spacing": [2, { "before": false, "after": true }],  "max-len": [2, 120, 4, {"ignoreComments": true, "ignoreUrls": true}],  "camelcase": [2, {"properties": "always" }],  "guard-for-in": 2,  "wrap-iife": [2, "outside"],  "no-use-before-define": 2,  "no-undefined": 2,  "max-nested-callbacks": [2, 3],  "max-depth": [2, 5],  "max-params": [2, 10],  "complexity": [2, 5],  "no-useless-concat": 2,  "strict": [2, "global"],  "no-script-url": 2,  //warnings  "no-empty": 1  }  } |  |

Drysdale specific overrides

"semi": [2, "always"],

"no-extra-semi": 2,

"semi-spacing": [2, { "before": false, "after": true }],

"max-len": [2, 120, 4, {"ignoreComments": true, "ignoreUrls": true}],

"camelcase": [2, {"properties": "always" }],

"guard-for-in": 2,

"wrap-iife": [2, "outside"],

"no-use-before-define": 2,

"no-undefined": 2,

"max-nested-callbacks": [2, 3],

"max-depth": [2, 5],

"max-params": [2, 10],

"complexity": [2, 5],

"no-useless-concat": 2,

"strict": [2, "global"],

"no-script-url": 2,

//warnings

"no-empty": 1