UI Code Review Checklist

**Disclaimer:** Below rules are only applied for OmniVista UI web development project due to some unique characteristics.

# Javascript / AngularJS

## Always check your javascript code for JSHint error and fix it immediately!

Because WebStorm has this feature, so there is no excuse for not using it!  
Following this rule will help eliminate a large set of bugs.

## Follow the variable naming convention

* If the variable is a constant, all characters are **capitalized**. Each word is separated by ‘**\_**’ (underscore)
  + E.g. var CONSTANT\_NAME = ‘http://google.com’;
* Any other kinds of variable have to follow **camelCase**
  + E.g. var iAmALongVaraibleName = 100;
* **AngularJS** controller must have the first character capitalized and capitalize the 1st character of each word.
  + E.g. module.controller(‘**MainController**’, [function(){ … }])

## Don’t use magic number or index that would make code difficult to understand (code style from 1990).

Example:

**BAD:**

**$scope.icons[0].disabled = false; // nobody can look at this code and understand what is icons[0]**

**GOOD:**

**$scope.icons.add.disabled = false;**

**BAD:**

**function doSomething() {**

**…**

**return 12; // what the hell is ‘12’!?**

**}**

**GOOD:**

**function doSomething() {**

**…**

**return APP\_CONST.ENABLED;**

**}**

## Any AngularJS module must be defined together with Dependecies Injection list

This will help avoiding error when code is minified.

This includes all kind of AngularJS modules: controller, service, directives,…

**BAD:**

module.directive(‘myCoolDirective’, function (**$compile**, **$timeout**) { … });

**GOOD:**

module.directive(‘myCoolDirective’, [**‘$compile’, ‘$timeout’**, function (**$compile, $timeout**) { … }]);

**Or**

function myCoolDirective(**$compile, $timeout**) {

// …

}

myCoolDirective.$inject = [**‘$compile’, ‘$timeout’**];

module.directive(‘myCoolDirective’, myCoolDirective);

## Don’t $watch ‘true’ unless there is no other way!

You should really think twice before $watch true!

**BAD:**

**$scope.$watch(‘arrayData’, listenerFn, true);**

**GOOD:**

**$scope.$watch(‘arrayData.length’, listenerFn);**

**$scope.$watch(‘arrayData’, listenerFn);**

## Use typeof(obj) === ‘undefined’ to check for ‘undefined’

## If a String is used more than 1 time, promote it to a CONSTANT variable and reuse it.

**BAD:**

**var a = ‘Oh my God’;**

**if (a === ‘Oh my God’) {**

**//…**

**}**

**GOOD:**

**var GOD = ‘Oh my God’;**

**var a = GOD;**

**if (a === GOD) {**

**//…**

**}**

## Do not create directive with replace: true

This feature will be removed in the next AngularJS version. So it’s better to ignore this feature.

There is also chance that you will get **memory leak** when using replace: true.

From now on, it’s recommended to use directive as **Attribute** instead of **Element**

## If you do $compile a template manually, be sure to compile it with a new scope.

Because you can’t be sure that when you remove the template, it won’t create memory leak.

**BAD:**

**$compile(template)($scope);**

**GOOD:**

**var childScope = $scope.$new();**

**$compile(template)(childScope);**

## When removing the manually compiled template, you have to destroy the scope associated with the template first. Otherwise, memory leak could happen.

**BAD:**

**var childScope = $scope.$new();**

**var compiledElement = $compile(template)(childScope);**

**…**

**compiledElement.remove();**

**compiledElement = null;**

**$childScope.$destroy();**

**$childScope = null;**

**GOOD:**

**var childScope = $scope.$new();**

**var compiledElement = $compile(template)(childScope);**

**…**

**$childScope.$destroy();**

**$childScope = null;**

**compiledElement.remove();**

**compiledElement = null;**

**or**

**// setup a ‘destructor’**

**$childScope.$on(‘$destroy’, function () {**

**compiledElement.remove();**

**compiledElement = null;**

**});**

**…**

**$childScope.$destroy();**

**$childScope = null;**

# CSS

## Always use REM instead of PX for unit, except html and @media queries

<html> font-size is pre-defined as **62.5%,** so you can divide the unit by 10 to get the corresponding REM.

Don’t use REM at @media since you will have to divide it by 16 instead. That makes code harder to understand as we are not Math genius!

**BAD:**

**.item-list {**

**width: 100px;**

**}**

**@media (max-width: 20rem) {**

**// …**

**}**

**GOOD:**

**.item-list {**

**width: 10rem;**

**}**

**@media (max-width: 320px) {**

**// …**

**}**

## Have namespace for your CSS classes.

* The namespace can be as short as a single character, or as long as 5 characters
* Where possible, the namespace should be a meaningful shorthand
* In class names, the namespace must be followed by a single dash
* Views should be treated as individual components

**BAD:**

**.item-list {**

**// ...**

**}**

**.dropdown-item-list {**

**// ...**

**}**

**.xyz-item-list {**

**// ...**

**}**

**GOOD:**

**.ddl-container {**

**// ...**

**}**

**.ddl-item-list {**

**// ...**

**}**

**.ddl-item {**

**// ...**

**}**

## Naming classes rules.

* Must be all-lowercase
* Words must be separated by single dashes
* As short as possible, but as long as necessary
* Don't abbreviate words carelessly
* Name things consistently
* Meaningful description of the elements that should use it
* Keep your non-prefix word count below 4

**BAD:**

**.ddlItem {**

**// ...**

**}**

**.ddl-item-container-text {**

**// ...**

**}**

**.ddl-foo-bar-baz {**

**// ...**

**}**

**GOOD:**

**.ddl-item {**

**// ...**

**}**

**.ddl-selected {**

**// ...**

**}**

**.ddl-item-selected {**

**// ...**

**}**

## Never use ID in style sheet file! Never!

While the id attribute might be fine in HTML and JavaScript, it should be avoided entirely inside stylesheets. Few reasons:

* ID selectors are not reusable
* Priority nightmares

**BAD:**

**#ur-name {**

**// ...**

**}**

**GOOD:**

**.ur-name {**

**// ...**

**}**

## Tag Names.

Tag names in selectors follow a few rules:

* Application level styles that are only overridden in a few places are okay to use tag name selectors
* Not semantic. Avoid where possible, use class names instead
* Fine to use when there's a ton of elements under the same namespace that need a small tweak
* Don't overqualify (a.foo)

**BAD:**

.ddl-container **button** {

background-color: #f00;

}

**Or**

**button**.ddl-container {

background-color: #f00;

}

**GOOD:**

**button** {

padding: 5px;

margin-right: 3px;

}

.ddl-button {

background-color: #f00;

}

## Selectors and Nesting.

Styles shouldn't need to be nested more than three (four at worst) levels deep. This includes pseudo-selectors.

If you find yourself going further, think about re-organizing your rules (either the specificity needed, or the layout of the nesting).

**BAD:**

**.dg-container .sg-container .sg-title {**

**font-size: 1.1em;**

**}**

**.dg-container .sg-title span:before {**

**// ...**

**}**

**GOOD:**

**.sg-title-icon:before {**

**// ...**

**}**

**.dg-container .sg-title {**

**font-size: 1.1em; // larger segment title inside dialogs**

**}**

## @Media Queries.

Approach your styles in a Mobile First manner. Generally you add more things as you get more real state. Mobile First logically follows

**BAD:**

**.co-field {**

**width: 400px;**

**color: #f00;**

**}**

**@media only screen and (min-width: 768px) {**

**.co-field {**

**width: 120px;**

**color: initial;**

**}**

**}**

**GOOD:**

**.co-field {**

**width: 120px;**

**}**

**@media only screen and (min-width: 768px) {**

**.co-field {**

**width: 400px;**

**color: #f00;**

**}**

**}**

# HTML

## Double check your template to see if it has a wrapper element already

When you create template either by using \*.html or ‘<template strings>’ you must have a wrapper element. Else the AngularJS template framework will not completely remove your compiled template and causing memory leak in the application.

Example of good and bad template:

|  |  |
| --- | --- |
| **Have a wrapper element** | Scattered elements |
| <**div**>   <**div class="form-group ag2-fix-ov-select"**>      <**label class="sortby-label col-sm-1"**>{{ 'unifiedAccess.byod.sortBy' |i18next}}:</**label**>      <**div id="sortby" class="sortby-select col-sm-4"**>        <**ov-select ng-model="$parent.selectSwitchSortBy" ng-options="item.name for item in $parent.columnList"**  **ng-change="updateSortField($parent.selectSwitchSortBy)"**></**ov-select**>      </**div**>   </**div**>   <**br**>   <**br**>   <**ov-slick-double-list sdl-config="$parent.DblListCfg" id="switchPortSelect" input-list="$parent.list2" selected-list="$parent.list1"**></**ov-slick-double-list**>  </**div**> | <**div class="form-group ag2-fix-ov-select"**>    <**label class="sortby-label col-sm-1"**>{{ 'unifiedAccess.byod.sortBy' |i18next}}:</**label**>    <**div id="sortby" class="sortby-select col-sm-4"**>      <**ov-select ng-model="$parent.selectSwitchSortBy" ng-options="item.name for item in $parent.columnList"**  **ng-change="updateSortField($parent.selectSwitchSortBy)"**></**ov-select**>    </**div**>  </**div**>  <**br**>  <**br**>  <**ov-slick-double-list sdl-config="$parent.DblListCfg" id="switchPortSelect" input-list="$parent.list2" selected-list="$parent.list1"**></**ov-slick-double-list**> |

## Always add ‘id’ to support Automation testing whenever possible. ID must have namespace to avoid collision.

**BAD:**

**<button>Click meh!<button>**

**Or**

**<button id=”button1”>Click meh!<button>**

**GOOD:**

**<button id=”app-btn-clickMe”>Click meh!<button>**

## Always try your best to produce semantic HTML!

* Use <form> whenever you want to collect input from users.
* Inside <form> element, there should always be <fieldset> to group related inputs together and inside <fieldset> there is <legend> to show the name of the input group.
  + Each input field must have a <label>
  + If the input it is a radio button or checkbox type, make sure that user can select it by clicking on the <label>.
* Try to maximize the use of new html5 elements such as <nav>, <menu>, <aside>, <section>, …