**KwIDF Expansion Project**

**Software Development Best Practices**

**Revision Control Log**

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Date** | **Description** | **By** |
| Draft 0 | 09/16/2016 | First Draft | MT |
| 0 | 09/23/2016 | Original | MT, SSI |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

Contents

[1 General 4](#_Toc462392891)

[1.1 Objectives 4](#_Toc462392892)

[1.2 Scope 4](#_Toc462392893)

[2 Best Practices 5](#_Toc462392894)

[2.1 General Guidelines 5](#_Toc462392895)

[2.2 General Coding Standards 5](#_Toc462392896)

[2.3 General File Naming convention 6](#_Toc462392897)

[2.4 General File Comments conventions 6](#_Toc462392898)

[2.5 Technology-specific practices 8](#_Toc462392899)

[2.5.1 Javascript 8](#_Toc462392900)

[2.5.2 AngularJS2 15](#_Toc462392901)

[2.5.3 HTML5 18](#_Toc462392902)

[2.5.4 CSS 19](#_Toc462392903)

[2.6 File Directory Structure 20](#_Toc462392904)

[2.7 Logs 31](#_Toc462392905)

# General

## Objectives

This document intends to present a set of formal and informal rules for software development and code programming, to be observed by all members of the project development team, and verified by the quality assurance team and the management before the development is deployed to production.

## Scope

This document present a set of rules to be strictly observed by all persons involved in the design, development, revision and quality assurance of all pieces of code and related files related with this project.

These rules are not meant to be exhaustive; it just give general guidelines. Any team member may suggest other practices, not explained or included in this document, which may help with the maintainability, dependability, efficiency and usability of all code to be written for this project.

The set practices listed herein includes, but are not limited to:

* General Guidelines
* General Coding Standards
* Technology-Specific practices
  + HTML5, Angular2JS, Bootstrap, JavaScript
  + ASP.NET / C#
  + Charting
  + Other technologies
* Development file / directory structures
* Code changes control
* Installation scripts
* Testing scripts / procedures

# Best Practices

## General Guidelines

1. Always adopt the “Keep It Simple” (KIS) approach. All development components must be built keeping simple, readable and easy understandable. Follow the Occam’s razor principle: “the simplest explanation or strategy tends to be the best one”.
2. All files, databases, tables, strings, configuration files and any other text of data container must use UTF-8 (standard ASCII). No unicode or special coding to be used, unless otherwise instructed.
3. Any software tool to be used must come from a trusted source and approved by the technical management. Use of tools or special applications downloaded from the internet or not obtained from legitimate sources is forbidden.
4. The use of any library, piece of code, class that is not coming from an untrusted source is prohibited. If any piece of code from any non-formal source is to be used, it must previously approved by the technical management.

## General Coding Standards

1. Adopt cross-platform, cross-browser web development practices. Any web development must be assure to run in any available (and reasonably recent) web browser.
2. Always separate code from configuration. Ensure configuration is stored externally. This is mandatory. NO code must have any string, number or other configuration information embedded into the code language. Any configuration, title, text, string, scalar of vector number must be placed in a separate file. This file must be integrated into the code files at run time or assembly time as fits best.
3. Build modular component and systems with integration and configurability in mind. Most of the features must be configurable. Configuration must rely either on configuration files or database tables’ contents.
4. Maintain a level of abstraction between your components and your dependencies. Adopt the Model-View-Controller (MVC) paradigm.
5. Keep supportability, maintainability and upgrade in mind when designing you applications. All files must include headers explaining the scope and intent of the file, along with the modification history and other pertaining information. Comments should be used throughout the file; classes, methods, functions and other code component must be fully commented.

## General File Naming convention

1. All file names must be written in lowercase.
2. Use consistent names for all files following that describes the file component feature, then (optionally) its type. For example:

avengers.component.js, page.configuration.json

1. Avoid the use of numbers in file names, unless to provide file version information.
2. Avoid use of special characters. In the case of “.” (point), “-“ (dash) and “\_” (underscore), you may use it for word separation within the file name.
3. Use the standard extension text according to the file type. Examples:

myfile.doc (word document)

myfile.js (JavaScript)

myfile.xml (XML file)

myfile.ts (TypeScript)

myfile.json (JSON file)

myfile.html (HTLM5 file)

myfile.log (log file)

## General File Comments conventions

1. ALL source files (unless is not possible to include it) must come with a file text header, indicating information about the file. This text must be placed in a way that the file considers it as comment. The file header must be standard JavaDoc. Detailed JavaDoc documentation may be found at

<http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html>

Following table shows the basic tags that must be used in any file.

|  |  |  |
| --- | --- | --- |
| **Tag** | **Use** | **Qty per file** |
| @author | Describes the author(s) | n≥1 |
| @version | Provides software version entry | n≥1 |
| @since | Describes when this functionality first existed | n=1 |
| @see | Describes a link to other element of documentation | 0≤n≤1 |
| @param | Describes a method or function parameter | n>0 |
| @return | Describes return value | 0≤n≤1 |
| @throws | Describes an exception that may the thrown from this method / function | n≥0 |

The standard file text header should look like this:

/\*\*

\* @filename xxxxx.yyy

\*

\* Here comes the full description of the file: What contains, what

\* is its purpose.

\*

\* @author John Doe

\* @version 0.0 date: 09/12/2016

\* @version 0.1 date: 09/15/2016

\* @since 09/01/2016

\* @see app.component.ts, app.service.ts

\*

\* Copyright HALLIBURTON – All rights reserved

\*/

1. A text header must be placed and on top of each function, class, enum, method and interface. It must be updated by the programmers every time the file changes after the creation date.

Example:

/\*\*

\* Validates a chess move.

\*

\* Use doMove(int theFromFile, int theFromRank, int theToFile, int theToRank)

\*

\* @param theFromFile file from which a piece is being moved

\* @param theFromRank rank from which a piece is being moved

\* @param theToFile file to which a piece is being moved

\* @param theToRank rank to which a piece is being moved

\* @return true if the move is valid, otherwise false

\*

\* @throws OutOfRangeException if any of the params is out of range

\*/

boolean isValidMove(int theFromFile, int theFromRank, int theToFile, int theToRank) {

// ...body

}

1. Be generous with comments. It is useful to leave information that will be read at a later time by people (possibly your future self) who will need to understand what you have done and why. The comments should be well-written and clear, just like the code they are annotating. An occasional nugget of humor might be appreciated. Frustrations and resentments will not.
2. It is important that comments be kept up-to-date. Erroneous comments can make programs even harder to read and understand.
3. Make comments meaningful. Focus on what is not immediately visible. Don't waste the reader's time with stuff like:

i = 0; // Set i to zero.

1. Use line comments.

## Technology-specific practices

### Javascript

Most of the User Interface (UI) coding will be done in JavaScript or Angular2JS, which code is TrueScript. The coding standard for related files are as follows.

**JavaScript Files**

JavaScript programs should be stored in and delivered as .js files.

JavaScript code should not be embedded in HTML files unless the code is specific to a single session. Code in HTML adds significantly to page weight with no opportunity for mitigation by caching and compression.

**Whitespace**

Where possible, these rules are consistent with centuries of good practice with literary style. Deviations from literary style should only be tolerated if there is strong evidence of a significant benefit.

Blank lines improve readability by setting off sections of code that are logically related.

Blank spaces should always be used in the following circumstances:

A keyword followed by ( *left parenthesis* should be separated by a space. Spaces are used to make things that are not invocations look less like invocations, so for example, there should be space after if or while.

while (true) {

A blank space should not be used between a function value and its invoking ( *left parenthesis*. This helps to distinguish between keywords and function invocations.

The word function is always followed with one space.

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

All binary operators should be separated from their operands by a space on each side except . *period* and ( *left parenthesis* and [ *left bracket*.

Every *, comma* should be followed by a space or a line break.

Each *; semicolon* at the end of a statement should be followed with a line break.

Each *; semicolon* in the control part of a for statement should be followed with a space.

Every statement should begin aligned with the current indentation. The outermost level is at the left margin. The indentation increases by 4 spaces when the last token on the previous line is { *left brace*, [ *left bracket*, ( *left paren*. The matching closing token will be the first token on a line, restoring the previous indentation.

The ternary operator can be visually confusing, so ? *question mark* always begins a line and increase the indentation by 4 spaces, and : *colon* always begins a line, aligned with the ? *question mark*. The condition should be wrapped in parens.

var integer = function (

value,

default\_value

) {

value = resolve(value);

return (typeof value === "number")

? Math.floor(value)

: (typeof value === "string")

? value.charCodeAt(0)

: default\_value;

};

If . *period* is the first character on a line, the indentation is increased by 4 spaces.

Avoid excessively long lines. When a statement will not fit nicely on a single line, it may be necessary to break it. It is best to break after a { *left brace*, [ *left bracket*, ( *left parenthesis*, , *comma*, or before a . *period*, ? *question mark*, or : *colon*. If such a break is not feasible, then break after an operator and continue on the next line with 8 spaces added to the current indentation. Those 8 spaces do not change the current indentation.

Clauses (case, catch, default, else, finally) are not statements and so should not be indented like statements.

Tabs and spaces should not be mixed. We should pick just one in order to avoid the problems that come from having both. Personal preference is an extremely unreliable criteria. Neither the tab nor the space offers a powerful advantage over the other. Fifty years ago, tab had the advantage of consuming less memory, but Moore's Law has eliminated that advantage. Space has one clear advantage over tab: there is still no reliable standard for how many spaces a tab represents, but it is universally accepted that a space occupies a space. So always use spaces. You can edit with tabs if you must, but make sure it is spaces again before you commit. Maybe someday we will finally get a universal standard for tabs, but until that day comes, the better choice is spaces.

**Variable Declarations**

Use strict. All variables should be declared before used. JavaScript does not require this, but doing so makes the program easier to read and makes it easier to detect undeclared variables that may become implied. Implied global variables should never be used. Use of global variables should be minimized.

It is preferred that each variable declarative statement and comment. They should be listed in alphabetical order if possible.

var currentEntry; // currently selected table entry

var level; // indentation level

var size; // size of table

A JavaScript var does not have block scope, so defining variables in blocks can confuse programmers who are experienced with other C family languages.

**Function Declarations**

All functions should be declared before they are used. Inner functions should follow the *var* statement. This helps make it clear what variables are included in its scope.

There should be no space between the name of a function and the ( *left parenthesis* of its parameter list. There should be one space between the ) *right parenthesis* and the { *left curly brace* that begins the statement body. The body itself is indented four spaces. The } *right curly brace* is aligned with the line containing the beginning of the declaration of the function.

function outer(c, d) {

var e = c \* d;

function inner(a, b) {

return (e \* a) + b;

}

return inner(0, 1);

}

This convention works well with JavaScript because in JavaScript, functions and object literals can be placed anywhere that an expression is allowed. It provides the best readability with inline functions and complex structures.

function getElementsByClassName(className) {

var results = [];

walkTheDOM(document.body, function (node) {

var array; // array of class names

var ncn = node.className; // the node's classname

// If the node has a class name, then split it into a list of simple names.

// If any of them match the requested name, then append the node to the list of results.

if (ncn && ncn.split(" ").indexOf(className) >= 0) {

results.push(node);

}

});

return results;

}

If a function literal is anonymous, there should be one space between the word function and the ( left parenthesis. If the space is omitted, then it can appear that the function's name is function, which is an incorrect reading.

div.onclick = function (e) {

return false;

};

that = {

method: function () {

return this.datum;

},

datum: 0

};

Use of global functions should be minimized.

When a function is to be invoked immediately, the entire invocation expression should be wrapped in *parenthesis* so that it is clear that the value being produced is the result of the function and not the function itself.

var collection = (function () {

var keys = [];

var values = [];

return {

get: function (key) {

var at = keys.indexOf(key);

if (at >= 0) {

return values[at];

}

},

set: function (key, value) {

var at = keys.indexOf(key);

if (at < 0) {

at = keys.length;

}

keys[at] = key;

values[at] = value;

},

remove: function (key) {

var at = keys.indexOf(key);

if (at >= 0) {

keys.splice(at, 1);

values.splice(at, 1);

}

}

};

}());

**Names**

Names should be formed from the 26 upper and lower case letters (A .. Z, a .. z), the 10 digits (0 .. 9), and \_ *underscore*. Avoid use of international characters because they may not read well or be understood everywhere. Do not use $ dollar sign or \ *backslash* in names.

Most variables and functions should start with a lower case letter.

Constructor functions that must be used with the new prefix should start with a capital letter. JavaScript issues neither a compile-time warning nor a run-time warning if a required new is omitted. Bad things can happen if new is not used, so the capitalization convention is the only defense we have.

Constants should be in all caps.

**Statements**

Simple Statements

Each line should contain at most one statement. Put a ; *semicolon* at the end of every simple statement. Note that an assignment statement that is assigning a function literal or object literal is still an assignment statement and must end with a semicolon.

JavaScript allows any expression to be used as a statement. This can mask some errors, particularly in the presence of semicolon insertion. The only expressions that should be used as statements are assignments, invocations, and delete.

**Compound Statements**

Compound statements are statements that contain lists of statements enclosed in { } *curly braces*.

The enclosed statements should be indented four more spaces.

The { *left curly brace* should be at the end of the line that begins the compound statement.

The } *right curly brace* should begin a line and be indented to align with the beginning of the line containing the matching { left curly brace.

Braces should be used around all statements, even single statements, when they are part of a control structure, such as an *if* or *for* statement. This makes it easier to add statements without accidentally introducing bugs.

**Labels**

Statement labels should be avoided. Only these statements should be labeled: while, do, for, switch.

**return Statement**

The return value expression must start on the same line as the return keyword in order to avoid semicolon insertion.

**if Statement**

The *if* class of statements should have the following form:

if (condition) {

statements

}

if (condition) {

statements

} else {

statements

}

if (condition) {

statements

} else if (condition) {

statements

} else {

statements

}

**for Statement**

A *for* class of statements should have the following form:

for (initialization; condition; update) {

statements

}

**while Statement**

A while statement should have the following form:

while (condition) {

statements

}

**do Statement**

A do statement should have the following form:

do {

statements

} while (condition);

Unlike the other compound statements, the do statement always ends with a ; semicolon.

**switch Statement**

A switch statement should have the following form:

switch (expression) {

case expression:

statements

default:

statements

}

Each case is aligned with the switch. This avoids over-indentation. A case label is not a statement, and should not be indented like one.

Each group of statements (except the default) should end with break, return, or throw. Do not fall through.

**try Statement**

The try class of statements should have the following form:

try {

statements

} catch (variable) {

statements

}

try {

statements

} catch (variable) {

statements

} finally {

statements

}

**continue Statement**

Avoid use of the continue statement. It tends to obscure the control flow of the function.

**with Statement**

The with statement should not be used.

**{} and []**

Use {} instead of new Object(). Use [] instead of new Array().

Use arrays when the member names would be sequential integers. Use objects when the member names are arbitrary strings or names.

***, comma* Operator**

Avoid the use of the comma operator. (This does not apply to the comma separator, which is used in object literals, array literals, var statements, and parameter lists.)

**Assignment Expressions**

Avoid doing assignments in the condition part of if and while statements.

Is

if (a = b) {

a correct statement? Or was

if (a == b) {

intended? Avoid constructs that cannot easily be determined to be correct.

**Confusing Pluses and Minuses**

Be careful to not follow a + with + or ++. This pattern can be confusing. Insert parentheses between them to make your intention clear.

total = subtotal + +myInput.value;

is better written as

total = subtotal + (+myInput.value);

so that the + + is not misread as ++. Avoid ++.

**eval is Evil**

The eval function is the most misused feature of JavaScript. Avoid it.

eval has aliases. Do not use the Function constructor. Do not pass strings to setTimeout or setInterval.

### AngularJS2

**General**

Angular2 coding must strictly adhere to Angular2 coding standards as per Angular Style Guide.

This may be found at <https://angular.io/styleguide>.

**Feature File Names**

Use consistent names for all components following a pattern that describes the component’s feature then (optionally) its type. My recommended pattern is feature.type.js.

* Provides a consistent way to quickly identify components.
* Provides pattern matching for any automated tasks.

/\*\*

\* common options

\*/

// Components

avengers.component.ts

// Services/Factories

logger.service.ts

/\*\*

\* recommended

\*/

// constants

constants.js

constants.ts

constants.json

// routes

avengers.routes.ts

// configuration

avengers.config.ts

avengers.config.js

avengers.config.json

// directives

avenger-profile.directive.js

avenger-profile.directive.spec.js

**Test File Names**

Name test specifications similar to the component they test with a suffix of spec.

* Provides a consistent way to quickly identify components.
* Provides pattern matching for karma or other test runners.

/\*\*

\* recommended

\*/

avengers.controller.spec.ts

logger.service.spec.ts

**Component Names**

Use consistent names for all component named after their feature. Use Pascal Casing (Camel Casing with first letter in Uppercase) for components, as they are constructors.

* Provides a consistent way to quickly identify and reference controllers.
* Pascal Casing is conventional for identifying object that can be instantiated using a constructor.

/\*\*

\* recommended

\*/

// avengers.conponent.ts

//

Import { Component } from 'angular2/core';

@Component ({

selector: 'av-sel',

templateUrl: 'app/avengers/avenger.component.html'

})

Export class AvengersComponent {

avengerName: string = 'Iron Man';

}

**Component Name Suffix**

Append the controller name with the suffix component. The component suffix is more commonly used and is more explicitly descriptive.

**Components**

When there are multiple components, the main component file is named app.component.ts while other dependent component are named after what they represent. For example, an admin component is named admin.component.js. The respective registered component names would be app and admin.

* Provides consistency for multiple component apps, and for expanding to large applications.
* Provides easy way to use task automation to load all component definitions first, then all other angular files (for bundling).

**Configuration**

Separate configuration for a component into its own file named after the component. A configuration file for the main app component is named app.config.js. A configuration for a component named admin.component.js is named admin.config.js. The same applies with .ts files or .json files.

* Separates configuration from component definition, components, and active code.
* Provides an identifiable place to set configuration for a component.

**Variable and Function Naming Conventions**

|  |  |  |
| --- | --- | --- |
| Type | Rule | Example |
| Variable, object | Camel Casing | pageTitle |
| Private variable, private object | Starting with “\_” (Underscore) then Camel Casing | \_internalCounter |
| Class | Pascal Casing (first letter Uppercase) | AppComponent |
| Function | Camel Casing | getProducts() |
| Interface | Starting with “I”, then  Pascal Casing | IProduct[] |
| Constants | All Uppercase | HTTPS\_PORT |
| Component Class | Pascal Casing, append “Component” | ProductListComponent |
| Service Class | Pascal Casing, append “Service” | ProductService |
| Pipe Class | Pascal Casing, append “Pipe” | ProductFilterPipe |
| Directive | All lowercase.  Two words separated by “-“ (dash).  <component>-<function> | app-label |
| Pipe | Camel Casing | productFilter |

**Use Interfaces**

Enforce use of explicit declarations. Use Interfaces where required. DO NOT USE any[].

### HTML5

1. In creating new templates make sure you added html5 doctype:

<!doctype html>

1. Make sure all tags are being closed rightly with not mixing in nesting tags.

<!-- correct -->

<h1> Heading</h1>

<p>some text.....</p>

<!-- wrong -->

<H1>Heading</h1>

<p>some text.....

1. Only lower case html is allowed, do not use upper case tags.

<!-- correct -->

<h1> heading here </h1>

<!-- wrong -->

<H1> heading here </H1>

1. Avoid inline styles; structure should always be separate from presentation, presentation only exists in our CSS files and should be referred to from head tag.

<h1 style=”color: blue;”> heading here </h1>

1. CSS files should only be inserted within the <head> tag, which should be within our master page layout, if you want to add any more classes, this should be added to the one of our customized CSS, in our case it is dsis-theme.css

<head>

<title>...</title>

<link rel="stylesheet" href="css/style.css">

</head>

1. Use requirejs for loading JS files based on their AMD (asynchronous module definition).

Show angular directive for how to use requirejs

1. Use sematic HTML5 elements to make sense of page structure, avoid using the unnecessary “divitus” (full of divs and divs for everything) approach.

Instead, make use of HTML5 elements

e.g.

<body>

<header>...</header>

<article>...

<section>...</section>

</article>

<footer>...</footer>

</body>

1. Form elements

In using form elements, make sure to use HTML5 attributes discretely based on how well each one is supported within our list of supported browsers, a good reference for all listed elements and attributes to use based on browser support is http://caniuse.com/

1. Since this application is highly modularized and the desired output is each component to work independently regardless of its context, avoid using #ids unless on necessity when knowing that this element will never be duplicated within the same page, as known ids should always be unique within the pages. Always use classes for presenting, as CSS classes can always be repeated within the same page causing no HTML errors.
2. Use HTML beautify, to make the HTML look cleaner before integrating it with other code, this will easy maintaining and debugging afterwards.
3. Finally and the most important of all, before integrating your code with backend services. Make sure it works separately with dummy / mock data, validated and doesn’t have any errors. Use those files as bases for your UI Unit Testing.
4. Use browser plugins for validating your HTML5 code like HTML validator & Validity in Chrome or HTML5 Validator Add-on in Firefox, or you can use online tools to validate your code before integrating it like <https://html5.validator.nu/>.

### CSS

Group similar CSS blocks into one area. The way the grouping is chosen is entirely up to the programmer. However, those are some standards:

@resets – taking away default browser margins, padding, fonts, colors, others.

@fonts – paragraphs, headings, blockquotes, links, code.

@navigation – the main core website navigation links.

@layout – wrapper, container, sidebars.

@header & @footer – these may vary based on design. Possible styles include links and unordered lists, footer columns, headings, sub-navs.

When grouping stylesheets, the tagging system can help a lot. Examples:

/\*\* @group footer \*/

#footer { styles... }

/\*\* @group footer, small fonts, columns, external links \*\*/

You could alternatively add a bit of extra detail in each comment block. I choose to keep things simple and straightforward so the stylesheets are easy to skim. Commenting is all about documentation so as long as you understand the writing it’s good to go!

A complete set of standards on CSS coding may be found at: <https://google.github.io/styleguide/htmlcssguide.xml#CSS_Style_Rules>

## File Directory Structure

The use of Visual Studio for development is proposed.

This is the proposed directory structure to be used:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **KWIDF - UI APP Folder Structure** | | |  |  |  |  |  |  | **Comments** |
| Repository |  |  |  |  |  |  |  |  | Repository name |
|  | □ | **KWIDF** |  |  |  |  |  |  | root folder |
|  |  |  | □ | **app** |  |  |  |  |  |
|  |  |  |  |  | □ | main.ts | app.component.ts|html |  |  | Contains bootstrap ts file, app components ts file. Html is the folder it contains the html templates for the specific components. |
|  |  |  |  |  | □ | **css** |  |  | Css folder. It contains css files specific to main app module. |
|  |  |  |  |  | □ | **config** |  |  | Config folder. It contains the configuration elements in config.json file specific to main app module. |
|  |  |  |  |  | □ | **images** |  |  | Images folder. It contains the images specific to main app module. |
|  |  |  | □ | **bin** |  |  |  |  | .Net assemblies for Angular2 |
|  |  |  | □ | **css** |  |  |  |  | Css folder. It contains css files common for the application. |
|  |  |  | □ | **fonts** |  |  |  |  | Fonts folder. It contains fonts files common for the application. |
|  |  |  | □ | **images** |  |  |  |  | Images folder. It contains the images common for the application. |
|  |  |  | □ | **module-a** |  |  |  |  | This is the template for any module. This is root folder of the module. |
|  |  |  |  |  | □ | **css** |  |  | Css folder. It contains css files specific to this module. |
|  |  |  |  |  | □ | **config** |  |  | Config folder. It contains the configuration elements in config.json file specific to this module. |
|  |  |  |  |  | □ | **images** |  |  | Images folder. It contains the images specific to this module. |
|  |  |  |  |  | □ | **component-a** |  |  | We need to create a folder for each component. The component folder contains the components specific files. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  |  |  | □ | component-a.ts|html|css | pipe | component-a.service.ts | component-a.model.ts | html and css are the folders which contains the group of component relates html and css files. |
|  |  |  |  |  | □ | **component-b** |  |  | The component folder contains the components specific files. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  |  |  | □ | component-b.ts|html|css | pipe | component-b.service.ts | component-b.model.ts | The component folder contains the components specific files. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  | □ | **shared** |  |  | This is the shared folder. It contains the common componets, css, html, service and model files across all components in a module. |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts | The shared folder contains the common components files in a specific module. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  | □ | module-a.component.ts|html|css | pipe |module-a.service.ts |  |  | module component file, html, css, pipe, service files. |
|  |  |  |  |  | □ | main.ts |  |  | bootstrap file for the module. |
|  |  |  |  |  | □ | index.html |  |  | module html file. |
|  |  |  | □ | **config** |  |  |  |  | Config folder. It contains the configuration elements in config.json file common to the application. |
|  |  |  | □ | **shared** |  |  |  |  | The shared folder contains the common components files for the application. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  | □ | **scripts** |  |  |  |  | This folder contains the thirdparty scripts such as bootstrap, jquery, chart related. |
|  |  |  | □ | **node\_modules** |  |  |  |  | This is application generated folder. Contains the modules related to angular2, array… |
|  |  |  | □ | **typings** |  |  |  |  | This is application generated folder. Contains the modules related to type scripts. |
|  |  |  | □ | **module-sps** |  |  |  |  | This is the folder structure for sps module. Please refer module-a folder structure. |
|  |  |  |  |  | □ | **css** |  |  |  |
|  |  |  |  |  | □ | **config** |  |  |  |
|  |  |  |  |  | □ | **images** |  |  |  |
|  |  |  |  |  | □ | **cmp-sps-a** |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-sps-a.ts|html|css | pipe | cmp-sps-a.service.ts | cmp-sps-a.model.ts |  |
|  |  |  |  |  | □ | **cmp-sps-b** |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-sps-b.ts|html|css | pipe | cmp-sps-b.service.ts | cmp-sps-b.model.ts |  |
|  |  |  |  |  | □ | **shared** |  |  |  |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts |  |
|  |  |  |  |  | □ | module-sps.component.ts|html|css | pipe | service |  |  |  |
|  |  |  |  |  | □ | index.html |  |  |  |
|  |  |  |  |  | □ | main.ts |  |  |  |
|  |  |  | □ | **module-esp** |  |  |  |  | This is the folder structure for esp module. Please refer module-a folder structure. |
|  |  |  |  |  | □ | css |  |  |  |
|  |  |  |  |  | □ | config |  |  |  |
|  |  |  |  |  | □ | images |  |  |  |
|  |  |  |  |  | □ | cmp-esp-a |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-esp-a.ts|html|css | pipe | cmp-esp-a.service.ts | cmp-esp-a.model.ts |  |
|  |  |  |  |  | □ | cmp-esp-b |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-esp-b.ts|html|css | pipe | cmp-esp-b.service.ts | cmp-esp-b.model.ts |  |
|  |  |  |  |  | □ | shared |  |  |  |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts |  |
|  |  |  |  |  | □ | module-esp.component.ts|html|css | pipe | service |  |  |  |
|  |  |  |  |  | □ | index.html |  |  |  |
|  |  |  |  |  | □ | main.ts |  |  |  |
|  |  |  | □ | **module-wpe** |  |  |  |  | This is the folder structure for wpe module. Please refer module-a folder structure. |
|  |  |  |  |  | □ | **css** |  |  |  |
|  |  |  |  |  | □ | **config** |  |  |  |
|  |  |  |  |  | □ | **images** |  |  |  |
|  |  |  |  |  | □ | **cmp-wpe-a** |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-wpe-a.ts|html|css | pipe | cmp-wpe-a.service.ts | cmp-wpe-a.model.ts |  |
|  |  |  |  |  | □ | **cmp-wpe-b** |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-wpe-b.ts|html|css | pipe | cmp-wpe-b.service.ts | cmp-wpe-b.model.ts |  |
|  |  |  |  |  | □ | **shared** |  |  |  |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts |  |
|  |  |  |  |  | □ | module-wpe.component.ts|html|css | pipe | service |  |  |  |
|  |  |  |  |  | □ | index.html |  |  |  |
|  |  |  |  |  | □ | main.ts |  |  |  |
|  |  |  | □ | **module-os** |  |  |  |  | This is the folder structure for os module. Please refer module-a folder structure. |
|  |  |  |  |  | **□** | **css** |  |  |  |
|  |  |  |  |  | □ | **config** |  |  |  |
|  |  |  |  |  | □ | **images** |  |  |  |
|  |  |  |  |  | □ | **cmp-os-a** |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-os-a.ts|html|css | pipe | cmp-os-a.service.ts | cmp-os-a.model.ts |  |
|  |  |  |  |  | □ | **cmp-os-b** |  |  |  |
|  |  |  |  |  |  |  | □ | cmp-os-b.ts|html|css | pipe | cmp-os-b.service.ts | cmp-os-b.model.ts |  |
|  |  |  |  |  | □ | **shared** |  |  |  |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts |  |
|  |  |  |  |  | □ | module-os.component.ts|html|css | pipe | service |  |  |  |
|  |  |  |  |  | □ | index.html |  |  |  |
|  |  |  |  |  | □ | main.ts |  |  |  |
|  |  |  | □ | **charts** |  |  |  |  | This is the folder structure for charts components. |
|  |  |  |  |  | □ | **css** |  |  | Css folder. It contains css files specific to charts components. |
|  |  |  |  |  | □ | **config** |  |  | Config folder. It contains the configuration elements in config.json file specific tocharts components. |
|  |  |  |  |  | □ | **images** |  |  | Images folder. It contains the images specific to charts components. |
|  |  |  |  |  | □ | **cmp-os-a** |  |  | The component folder contains the components specific files to various chart types. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  |  |  | □ | cmp-os-a.ts|html|css | pipe | cmp-os-a.service.ts | cmp-os-a.model.ts |  |
|  |  |  |  |  | □ | **cmp-os-b** |  |  | The component folder contains the components specific files to various chart types. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  |  |  | □ | cmp-os-b.ts|html|css | pipe | cmp-os-b.service.ts | cmp-os-b.model.ts |  |
|  |  |  |  |  | □ | **shared** |  |  | The shared folder contains the common components files for the various chart types. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts |  |
|  |  |  | □ | **common** |  |  |  |  | This is the folder structure for common components like log, authentication… |
|  |  |  |  |  | □ | **css** |  |  | Css folder. It contains css files specific to common components. |
|  |  |  |  |  | □ | **config** |  |  | Config folder. It contains the configuration elements in config.json file specific to common components. |
|  |  |  |  |  | □ | **images** |  |  | Images folder. It contains the images specific to common components. |
|  |  |  |  |  | □ | **cmp-log** |  |  | Logging service component. |
|  |  |  |  |  |  |  | □ | cmp-log.ts|html|css | pipe | cmp-log.service.ts | cmp-log.model.ts |  |
|  |  |  |  |  | □ | **cmp-service** |  |  | Http service component. |
|  |  |  |  |  |  |  | □ | cmp-service.ts|html|css | pipe | cmp-service.service.ts | cmp-service.model.ts |  |
|  |  |  |  |  | □ | **cmp-auth** |  |  | Authentication service component. |
|  |  |  |  |  |  |  | □ | cmp-auth.ts|html|css | pipe | cmp-auth.service.ts | cmp-auth.model.ts |  |
|  |  |  |  |  | □ | **shared** |  |  | The shared folder contains the common components files for the common services. Such as component.ts, pipe.ts, service.ts, model.ts, html, css. |
|  |  |  |  |  |  |  | □ | component-shared.ts|html|css | pipe | component-shared.service.ts | component-shared.model.ts |  |
|  |  |  | □ | index.html |  |  |  |  | Landing page |
|  |  |  | □ | favicon.ico |  |  |  |  | Favourite icon |
|  |  |  | □ | package.json |  |  |  |  | contains information about the pacakages/modules used in the application |
|  |  |  | □ | packages.config |  |  |  |  | contains information about the assemblies used in the application. Specific to Visual studio. |
|  |  |  | □ | systemjs.config.js | |  |  |  | provides information to a module loader about where to find application modules, and registers all the necessary packages. |
|  |  |  | □ | tsconfig.json |  |  |  |  | defines how the TypeScript compiler generates JavaScript from the project's files |
|  |  |  | □ | typings.json |  |  |  |  | provides additional definition files for libraries that the TypeScript compiler doesn't natively recognize |
|  |  |  | □ | web.config |  |  |  |  | Application configuration file. Provides additional definition for the website. |
|  |  |  | □ | README.md |  |  |  |  | Generate the html summary of the project and used in github. |
|  |  |  | □ | CHANGELOG.md |  |  |  |  | Change log file |
|  |  |  | □ | .csproj |  |  |  |  | Visual studio project file |
|  | □ | packages |  |  |  |  |  |  | Folder contains the .Net assemblies for the application |
|  | □ | .sln |  |  |  |  |  |  | Solution file |

## Logs

1. All software components, applications, or script must generate appropriate log records to keep track their operations.
2. Logs must be centralized. At server level, they must be placed in a common directory or set of files. The log centralization will depend of the type of server technology to be used.
3. From the client side, all logs must be sent to the browser log console.
4. Logs must include the following information:

* Date and time that the log occurs.
* Log severity: Severity level of the log, according to this list:
  + INFO (No error, just informational)
  + WARN (Warning – App may keep running – error is tolerated)
  + ERROR (Fatal Error – app may stop functioning)
  + DEBUG (Debugging – Messages to be logged when in debugging mode only)
* Application Identification – name of the application that generated the log.
* File – that contains the code that generated the log.
* Method, routine, function – routine that generated the log.
* Message – free text containing the specific message information.

1. All log message must be 1 line only, except DEBUG in which they may be more than one line.
2. A typical log message:

09/18/2016 10:59:01 [ERROR] (spsapp, main.component.ts, getProduct) product ID is out of range

1. At application level, the generation of logs must be controlled by selecting the appropriate level. That is, each application must have a configurable constant, named LOGLEVEL, which determined which message shown be logged at runtime. The rule for logging is the following:
   * LOGLEVEL=DEBUG -> All logs are recorded.
   * LOGLEVEL=INFO -> All logs are recorded, except DEBUG.
   * LOGLEVEL=WARN -> Only WARN and ERROR logs are recorded.
   * LOGLEVEL = ERROR -> Only ERROR logs are recorded.