Front-end to-be system - draft

# Architecture

Application should be designed so as to be extendable. A general barebones application structure will provide JSP templates, JavaScript application namespace and CSS styling. Features should be added to the application in the form of modules. New features should be aligned with the barebones application, reusing and extending the main application framework.

# JSP Templating

Application will define basic extendable templates. Pages will extend these templates avoiding any repetition of any DOM elements, JavaScript code and CSS styles.

Generally, two basic templates should suffice:

1. Main template: This template is for the main page layout. All complete main pages (like home) should extend this main template.
2. Page Fragment template: This template is for dynamic page fragments that are requested via AJAX requests.

Based on requirements, additional templates can be added.

General guidelines:

1. Avoid JavaScript code in templates.
2. Main template should include application CSS files at the top of the page. JavaScript libraries and application JavaScript files must be included at the bottom of the page.
3. Load page components by making AJAX calls based on necessity. Do not preload or include lesser accessed page components.
4. Application level messages and other data required on the client side should be assigned to properly namespaced JavaScript variables in a separate JSP file.
5. Avoid unnecessary markup. Examples for unnecessary markup: Hidden input fields for maintaining application state. Remove unnecessary comments and commented markup.
6. Indent code by two spaces. Use double quotes for assigning attributes.

# HTML Design

Use HTML5boilerplate as base for designing template. Markup should be semantically correct. Scripting or styling information should not be attached to markup. CSS class names can be used generously for styling and scripting purposes but IDs should be used sparingly.

Guidelines:

1. Minimize use of IDs in the HTML markup. IDs should be assigned only to global container or placeholder elements for reference. All widgets, page fragments should be identified with CSS class names.
2. Force Google chrome frame rendering.
3. Main page should only have library, necessary plugin and application CSS & JavaScript code included.
4. Page fragments that are loaded dynamically should load plugin and other application module code dynamically after checking for the module’s availability.
5. Apart from function calls and initializations, page fragments and main page should not have function or module definitions.
6. Avoid adding placeholders or containers for JavaScript plugins in HTML. Rather, check for existence of such containers and then create them if necessary.
7. No inline styling or JavaScript should be present.
8. Use “data-“attributes on elements for assigning data.

<li data-asset-id="34343">Asset</li>

1. HTML containers for particular features or stories should use appropriate class names for styling and scripting purposes.

**HTML**

<div class=”module1”> <p class=”asset”>Asset A</p> </div>

<div class=”module2”> <p class=”asset”>Asset B</p> </div>

**CSS**

.module1 .asset { color: #000022; }

.module2 .asset { color: #002200; }

# CSS Design

CSS for the application should be built on HTML5Boilerplate’s CSS. Application’s CSS will define the overall website style. Modules, features, stories should have their CSS added in separate files. Using LESSC, compile CSS from different files into one main stylesheet file and multiple other files based on need.

Guidelines:

1. Do not repeat / recreate the styles specified in the “normalize.css”.
2. Avoid inline styling.
3. Use LESS CSS.
4. Use LESS variables to define the commonly used CSS values like colors, widths, etc.

Variables allow specifying widely used values in a single place, and then reusing them throughout the style sheet, making global changes as easy as changing one line of code.

@color: #4D926F;

#header {

color: @color;

}

h2 {

color: @color;

}

1. Use nested rules rather than constructing long selector names to specify inheritance. In LESS, selectors can be nested inside other selectors. This makes inheritance clear and style sheets shorter.

#header {

h1 {

font-size: 26px;

font-weight: bold;

}

p {

font-size: 12px;

a {

color: @linkColor;

&:hover {

border-width: 1px;

}

}

}

}

1. Avoid margin, padding to position any element. Use float instead.
2. Avoid defining the same styles instead add the classes that is pre-defined.

#header{

~~float: left;~~ /\* use the class “.floatLeft” instead of defining here.\*/

padding: 0;

}

.floatLeft{

float: left;

}

<div id=”header” class=”floatLeft”>header</div>

1. Use semantic CSS class names. If there are different states defined for a particular element, it must correspond to the state.

@red: #FF0000;

@blue: #0000FF;

.critical {

color: @red;

}

.active {

color: @blue;

}

The element should be assigned the state “critical” than simply the color “red”.

# JavaScript Design

JavaScript code for the application will define an application namespace containing objects for utility functions, application related data, functions and modules.

Each feature or story should be added as modules or widgets to the application namespace. These modules should be decoupled from the HTML markup.

Use JSHint to detect errors and coding convention violations.

Guidelines:

1. Application will define a global namespace which will hold all the application utility functions, data and modules.
2. Separate namespace for UI utilities, models, cached data, models and modules.

**app.js**

var APP = window.APP || { };

APP.UI = APP.UI || {utilities: {}, tabs: {}};

APP.messages = APP.messages || {}; // for general application messages

APP.modules = APP.modules || {};

APP.widgets = APP.widgets || {};

**utilities.js**

APP.utilities = APP.utilities || {};

**userprofile.js –(modules and widgets should add to the APP.widgets, APP.modules namespace)**

(function(APP){

var userprofile = APP.widgets.userprofile = {};

userprofile.load = function(){

// load userprofile

}

// define more user profile functions

}(APP));

1. APP.widgets is for containing small widgets , like popups, modals, utility bars, dropdowns. APP.modules is for containing features/stories, page or tab components.
2. Add common utility functions to APP.utilities.
3. No loose functions or variables should be defined. They should be moved to appropriate namespace.
4. Decouple JavaScript from markup. Avoid hardcoding DOM IDs in JavaScript functions. Minimize accessing elements using class names. Pass DOM references to functions/modules instead and use these as context in them. Assign and reuse references to DOM elements.

function renderAsset(el, data){

$( '.title', el).text(data.title); // use el as context

$( '.desc', el).html(data.description);

}

var el = $( '.asset-container'); // save references to elements

renderAsset(el, data); // pass DOM reference

attachEvent(el, eventMap); // reuse element references

1. Use handlebar.js for client side templates. Application level templates can stay in application’s templates.js file. Templates specific to modules should ideally be in module’s JavaScript code.

APP.module.assetList = function(){

var assetList = { templates: {}};

assetList.templates.list = “<li>${name}</li>”; // add templates here

var assetList.load = function (container, data){

assetList.templates.highlight = data.templates.highlight;

/\* assign any templates dynamically only when required \*/

// load

}

// more functions…

return obj;

};

1. Maintain references to objects, application states and DOM elements at application or module level.
2. Cache necessary application data in APP.cache or APP.modules.moduleName.cache
3. Keep DOM modifications to a minimum. Assign CSS classes to elements rather than applying individual CSS styles dynamically.
4. Application should have one initialization function which initializes and sets up the UI components such as tabs. This initialization function would be the single point where the application functionality code starts execution.