UI Technical Requirements

VERSION: V1.0

Checklist

Please use the below checklist to assess the conformance of your delivery with MSCI's requirements. MSCI will also use this checklist to evaluate the delivered product before we sign-off on it. These are not absolute requirements and there is some room for flexibility to change them based on actual requirements.

- [ ] Project structure and folder use is OK
- [ ] Source code is properly commented
- [ ] Naming conventions and name-spacing is correctly used
- [ ] HTML source code Markup is tested as valid HTML5 and is encoded in UTF-8
- [ ] HTML DOM is properly structured
- [ ] HTML uses a responsive mobile friendly design
- [ ] CSS version and pre-processor versions are OK
- [ ] CSS frameworks are supported
- [ ] JavaScript version is supported
- [ ] JavaScript 3rd party libraries are supported
- [ ] JavaScript code conventions are respected
- [ ] JavaScript error handling and logging is properly setup
- [ ] JavaScript bootstrapping is compatible with MSCI's requirements

- [ ] Analytics vendor is supported by MSCI
- [ ] Data-sets are stored in a dedicated JSON and the widget can be redrawn via an APi call to show the new values
- [ ] Configuration is stored in a dedicated JSON and the widget can be redrawn via an APi call to reflect the config change
- [ ] Analytics requirements are well documented
- [ ] Browser compatibility is respected
- [ ] The Integration test has been carried out according to the test specs
- [ ] IFRAMEs are not used
- [ ] Performance metrics (size, speed) are within acceptable values

Project Structure

NOTE: For projects where the delivered package is in 100% native Liferay project format please skip to the chapter about "Deliveries and Handover" on this link: https://bodhi.msciapps.com/x/q2JyD.
The below structure is for a non-Liferay format UI development which will be converted and migrated onto Liferay by the MSCI IT/Web team.

You have to create and hand-over a well structured package to MSCI when the project is completed. It has to contain:

- Human readable **Source code files** which can be used in MSCI's development environment to understand, review, modify and rebuild the project.
- A **Distribution package** which is an example of a pre-built ready-ti-deploy binary, It does not need to be human readable and it can be bundled indemnified and ready for deployment.
- **3rd party dependencies** in any form (donwloaded libraries, NPM modules, configuration files, etc.)
- Auto-generated **JSDoc files** explaining the API nevel usage of the methods, functions and classes in the project
- **Manual documentation** including installation guide, release and change notes and project manifest (a catalog of all the source code files and dependencies with their filenames, location and short description)
- **Project meta files** and configuration (package.json for NPM, WebPack, Grunt config files, etc.), basically anything that is required to rebuild the sources into the distributable version from scratch

Folder structure

The folder structure should follow the standard NPM/Webpack structure's recommendations:

```
MSCI-SAMPLE-UI-PROJECT
 dist
  css
  fonts
  html
  images
  js
  app.js
  index.html
 doc
  jsdoc
  release-notes.txt
  build-instructions.txt
  manifest.txt
 node_modules
 src
  css
   main.css
   navigation.css
  fonts
   roboto.ttl
  images
   background.gif
  lib
   jQuery-1.3.4-bundle.js
   msci-common-lib.js
  html
   navigation.html
   menu.html
  js
   analytics.js
   footer.js
  app.js
  index.html
 gulpfile.js
 package.json
 webpack.config.js
```

When creating the soucecode package please follow the below rules:

- "**src/**" is the source folder in a human readable pre-build format with release version of the library with license headers and usage description
    - "**src/lib**" contains 3rd party libraries in a human readable format with licensing info, source repository and locations where it was obtained from
    - "**src/{html,css,images}**" contain static elements HTML pages, fragments, CSS stylesheets and imagery
- "**dist/**" is the output folder of the compilation/transpilation and build process(es) it is the "binary" which cna be used to install in a web application or on the website, its contents should be minified, bundled and optimized for the runtime environment
- "**doc/**" contains automatically generated JSDOC files, release and installation notes and a disctionary/manifets file to verview the project components

Preferred Bundling Stack

MSCI prefers to use NPM as the core for the Javascript build and dependency management. NPM can be augmented with extra compile, build or package time utilities as part of the NPM ecosystem.

The preferred build stack consists of the below technologies:

- **NPM** fpr compile and build time dependency management and build
- **Webpack** for build task management (bundling, minifiaction, pre-processing, tranpilation) and the governance model for the run-time dependency (module) system (based on ECMA2016 modules)

HTML

Version

The preferred content type is HTML5 using the below DOCTYPE:

```
<!DOCTYPE html>
```

The page templates should be validated through http://validator.w3.org/.

Encoding

The character encoding of the UI source code files (HTML, CSS, Javascript) should be in UTF-8 and the language should be properly declared wherever possible.

Identifiers and Naming conventions

DOM elements have to be marked with identifiers in the form of stylesheet class names or IDs (if they are unique) when they are functional parts of the DOM. Try to avoid using **ID attributes** in the HTML markup to avoid having to resolve later conflicts between independent widgets embedded into the same page. Please use **CSS class names** instead.

- Placeholders for dynamically changing data
- Have event handlers attached to them
- Can be mutated by Javascript
- Represent a logical structure, like sections, headers, footers, navigation, hover content, etc.

Naming Conventions

Naming conventions to respect

- Class names and identifiers must be compact yet still carry meaning. The meaning should hint at the function and/or structural location of the identified HTML element.
- They must also be part of a namespace to avoid name collisions.
- Avoid using general and frequently used CSS class names , like 'wrapper', 'content', 'box' etc., since they can easily be used in other parts of the destination page.
- Avoid using frame-work specific CSS class names for custom purposes, especially the ones, that are being used in Bootstrap, like 'col', 'row', 'hidden'. If you still use them, then use it in a way, recommended by the Bootsrap documentation or please reset/override them inside the widget wrapper only.

"NAMESPACEPART-NAMEPART1{-...-NAMEPARTn}{-NUMBER}"

Examples:

- Bad examples:
    - class="123ZH4"
    - class="margins"
- Good examples:
    - class="msci-navigationFloat-2"
    - class="msci-perfChart-margins-1"
    - class="perfchart-padding"

Comments

The HTML source code should be appropriately commented and pretty-formatted with indentation:

- functionally important blocks or elements must include a short label and description
- blocks must be marked with a START and END comment to help visual identification of boundaries

Example comment style:

```
<!-- START: Footer -->
<div id=”footer”>
 <div id="footer_inner"></div>
</div>
<!-- // END: Footer -->
```

Structure

The HTML design should reflect the visual page layout as best as it can. WYSIWYG stands true here too, meaning that visually looking at the page and navigating across parts and components should be consistent with what is in the HTML structure:

- Avoid self-modifying code which generates the final HTML using Javascript. IN other words the rendered HTML should closely follow tghe downloaded HTML. Exemptions are certainly possible for example when 3rd party libraries render special widgets (charts, animatin, etc.) but anything else must come from the server.
- Horizontal and vertical positioning and neighborhood of components should follow the order of definition in the downloaded HTML. For example the footer should be defined at the bottom of the source code and not floated across using special CSS or Javascript.
- Do not use dynamic client-side inclusion of HTML sub-templates for things like the footer, header, etc.

Make use of the the HTML header tags to convey special META information for SEO and other purposes.

The use of HTML5 Semantic elements is encouraged (http://www.w3schools.com/html/html5_semantic_elements.asp) do denote semantically different sections.

The dynamic elements of the page (i.e. those areas which contain content that can change) should also be tested with content of varying length, to ensure the basic template will not break if either content expands or the font size is increased, e.g. a dynamic element that contains a list of headlines in a “Latest News” section of a homepage.

CSS

To style the HTML markup CSS is preferred over the use of HTML tags. Furthermore

- Inline style definitions are not allowed
- HTML property or tag based styling or skinning is not allowed, for example
    - use images with CSS classes instead of using the <img/> tag for icons & backgrounds.
    - padding and margin HTML attributes are not allowed
- “CSS Sprites” should be used for managing icon states.
- CSS should be used for layout of all elements in the page

Version

CSS version 3 must be used.

Structure and file separation

Static CSS code should be contained in as few files as possible. The recommendation is to have a one central main file “css/main.css” A separate print version of the CSS needs to be provided, called “css/print.css”. This can be used to hide certain navigation elements or banner ads from the printed version, and also address situations where light text is used on dark backgrounds.

Pre-Processors

For CSS pre-processors SASS must be used. The delivered source package must contain everything that is required to do the compilation and build of the final static CSS elements:

- The SASS source files
- An example build project
    - with configuration and project files that can be executed in NPM, Grunt or Gulp and result in the static compiled CSS files
    - documentation on how to perform the setup and build
- The static already compiled CSS files must also be included in a dedicated project folder or ZIP package

CSS Frameworks

For structured UI widget elements, like form fields, grid layouts, sections, tables, etc. a CSS layout/design framework should be used to stay coherent and to allow easier skinning and theming. Liferay 7.1 is using Clay, a Bootstrap 4 derivative for this so the suggested practice is to use Clay as the default CSS (widget) framework.

Javascript

Javascript Version

Run-time required version is **ES5**,
The source can be written in at most in ECMA2015 (ES6) Javascript.
However in the final distributable package it must be compiled/transpiled into **ES5**. This requirement stems from the relatively high penetration of Internet Explorer. (See Browser statistics below)

3rd Party Libraries

The out of the box supported and accepted Javascript libraries and frameworks are listed below.

- Liferay 7.1 provided standard Javascript libraries
- jQuery v3.4.0 (see https://help.liferay.com/hc/en-us/articles/360036751511-What-is-the-default-jquery-version-used-in-Liferay-7-1-)
- Charting and Data visualization
    - Highcharts v5.0.14 (2017-07-28)
    - Altair Vega chart libraries:
        - `"msci-vega@4-v1.0.0" package:`
            - Vega v4.3.0 (https://cdn.jsdelivr.net/npm//vega@4.3.0)
            - Vega-lite v2.6 (https://cdn.jsdelivr.net/npm//vega-lite@2.6.0)
            - Vega-embed v3.24.0 (https://cdn.jsdelivr.net/npm//vega-embed@3.24.0)
        - `"msci-vega@5-v1.0.0" package:`
            - Vega 5.10.0 (https://cdn.jsdelivr.net/npm//vega@5.10.0)
            - Vega-lite 4.0.2 (https://cdn.jsdelivr.net/npm//vega-lite@4.0.2)
            - Vega-embed v6.5.2 (https://cdn.jsdelivr.net/npm//vega-embed@6.5.2)
        - `"msci-vega@5-v1.1.0" package:`
            - Vega 5.10.1 (https://cdn.jsdelivr.net/npm//vega@5.10.1)
            - Vega-lite 4.8.1 (https://cdn.jsdelivr.net/npm//vega-lite@4.8.1)
            - Vega-embed v6.5.2 (https://cdn.jsdelivr.net/npm//vega-embed@6.5.2)
    - Three.js
- Animation
    - GreenSock GSAP
- Analytics
    - Hotjar
    - Demandbase
    - GA
    - Pardot
- Other
    - Font Awesome 3.2.1 (which is part of the clay.css)

There can be other libraries considered, subject to pre-approval from MSCI. The considerations will be done along the below lines:

- justification, i.e. what is the benefit of on boarding a new library (performed by implementation team)
- licensing (performed by MSCI)
- cost analysis (performed by MSCI)
- compatibility check (performed by MSCI) with the existing frameworks used on the site
- 3rd party vendor and security-architecture review (performed by MSCI)

Code Conventions

MSCI does not maintain a custom code convention and/or notation. Instead, Dougles Crockford's Javascript coding conventions and styles from this document should be adhered to:
https://www.crockford.com/code.html

It contins short and meaningful recommendations in the following areas:

- Comments
- Whitespaces
- Declarations
- Names and Labels
- Statements

Error handling and Debugging

- Provide source-maps along with bundles to ease debugging
- Log in NONE, INFO, WARN and ERROR mode to the console and provide the means to change verbosity (by default it should be none)
- For any web-service call use try-catch style error handling and anticipate and handle any erroneous or unexpected situations
- Errors must never be silently ignored, empty catch blocks are not allowed.
- Errors must be displayed on screen if they effect usability or functionality of the web pages, if not then a WARNING must be logged onto the console
- Debug handles and functionality must not have an adverse impact neither on performance nor on security if so, than the build flow must be configurable to rebuild the DIST version with DEBUGGING turned OFF

Modularization and Scoping

With modularization and scoping aim to make the source code meet the following rules:

- **Isolation**: to minimize unforeseen and unhanded interaction between your code and the rest of the code on the website pages (Liferay core UI functionality, other MSCI page level customizations).
- **Reusability and Composition**: Write once use many principle.

To achieve this you have to make the code modular. Since the preferred language Javascript compatibility level syntax is ES5 the run-time technique to achieve this is by using namespaced IIFEs (Immediately Invoked Funcion Expressions). For the compile time source code however you can use ES6 standard modules and then have the bundler (WebPack) transpile them into IIFEs.

Refernce article: https://tylermcginnis.com/javascript-modules-iifes-commonjs-esmodules/

Bootstrapping

Bootstrapping will be handled differently if the delivered project is just a visual mock-up or is already in a Liferay format (converted to a portlet).

Liferay Portlet format

For using Javascript in portlet template code there are two standard Liferay style options:

- A; embedd the JS code in the portlet and let Liferay minify and bundle it, then bootstrap the code from the portlet template HTML file
- B; prepare a distributable version and the MSCI Web team will add that to our GTM (Google Tag Manager) rules.

Standalone JS app or Mockups

Prepare a distributable version using the project's bundler mechanism and let the entry-point HTML page initialize and bootstrap the application. The inclusion and bootstrap mechanisms should be documented.

Website Analytics

If there are any analytics requirements in scope then they have to be clearly defined as part of the project documentation:

- analytics framework to be used
- UI events to be used as triggers for analytics reporting
- message to be sent upon a triggered event
- placeholder for the event

Any new project must be prepared to support the following analytics integration requirements:

- Has to be compatible with MSCI's analytics technologies (see below). Compatibility means both not breaking existing analytics code and the ability to feed events into our analytics stack.
- The DOM structure must have named placeholders with IDs where analytics triggers can be easily attached to
- The Javascriot and DOM events have to be made available for capturing by event handlers (e.g do not sink and terminate an onClick event)
- Working in tandem with MSCI's Marketing department provide a list of all the events, elements and actions that will be used for tracking

MSCI's analytics stack

The below list summarizes the available analytics platforms at MSCI. Exact requirmeents and use-cases that would/should be built on top of them have to be discussed with MSCI's Digital Marketing and IT/Web teams for feasibility.

- **Google Analytics:** To be used to capture metrics on usage and website visitor interaction.
- **Pardot**: To capture usage and interaction from marketing email campaign initiated website activity
- **Hotja**r: To capture UX and UI performance and metrics
- **DemandBase**: For B2B Account Based Marketing

Browser Support

Our client-base is mostly B2B type coming from corporate networks with a natural tendency towards using older browser stacks. Although in the last couple of years IE browsers have drastically lost in popularity, and most of our users are now on Chrome, IE is still a factor we have to count with and make the delivered code compatible for IE11.

Browser Support Requirements

The web pages must be tested to work on the below operating systems:.

- Windows 10
- Mac OS
- IPhone
- Android

Browsers:

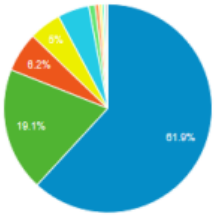
- Internet Explorer 11
- Microsoft Edge (latest, latest-1)
- Firefox (latest, latest-1)
- Chrome (latest, latest-1)

Note: Apart from IE, the latest major version and the one before that with the highest minor version numbers must be supported. For example in December /2019 Chrome latest was **80.0.3987** and the version before that would have been: **79.0.3945**

Statistics

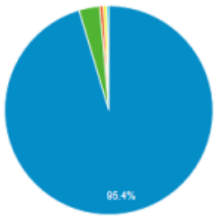
Browsers visiting MSCI.com August 2019

1.	Chrome	52,832	61.88%
2.	Internet Explorer	16,334	19.13%
3.	Edge	5,261	6.16%
4.	Firefox	4,242	4.97%
5.	Safari	4,175	4.89%
6.	Lore Document Collector 73448a19d5663f28c9f6d61e0c40d583	826	0.97%
7.	Samsung Internet	523	0.61%
8.	Opera	389	0.46%
9.	Safari (in-app)	322	0.38%
10.	Android Webview	146	0.17%



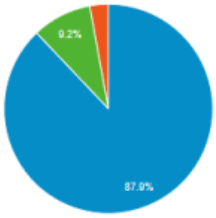
IE browser versions visiting msci.com August 2019:

1.	11.0	15,532	95.36%
2.	8.0	527	3.24%
3.	7.0	92	0.56%
4.	9.0	80	0.49%
5.	10.0	57	0.35%



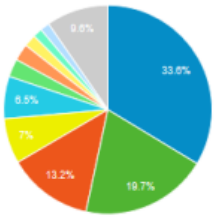
Different devices visiting msci.com August 2019:

		91,789 % of Total: 100.00% (91,789)	91,789 % of Total: 100.00% (91,789)
1.	desktop	80,696	87.91%
2.	mobile	8,454	9.21%
3.	tablet	2,643	2.88%



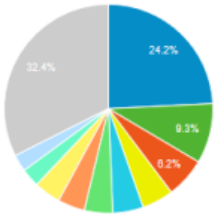
Mobile device branding visiting msci.com August 2019:

1.	Apple	3,734	33.64%
2.	Samsung	2,191	19.74%
3.	Microsoft	1,467	13.22%
4.	Huawei	779	7.02%
5.	Xiaomi	727	6.55%
6.	(not set)	310	2.79%
7.	OnePlus	290	2.61%
8.	Motorola	209	1.88%
9.	Google	162	1.46%
10.	Sony	161	1.45%



Screen resolution visiting msci.com August 2019:

1.	1920x1080	22,664	24.21%
2.	1366x768	8,700	9.29%
3.	1536x864	5,837	6.23%
4.	1280x1024	4,566	4.88%
5.	1680x1050	4,427	4.73%
6.	1280x720	4,099	4.38%
7.	1920x1200	3,927	4.19%
8.	1440x900	3,870	4.13%
9.	1600x900	2,715	2.90%
10.	1280x800	2,462	2.63%



Data Sources

- Data must be separated from Code and Markup. Please provide the data-set which is displayed in a separate JSON object which is then parsed by JavaScript and used to fill-in the charts and other data representation elements on the page
- If the data is larger then a few hundred records, the display widgets must use AJAX calls to fetch the data from a remote JSON/WS APi
- If visuals depend on the data, then the change of of data must allow a refresh of the visuals.
    - for example if the the underlying data-set changes for a bar chart then the bars must be resized. This must be done via exposing and an APi call, something like doRefresh() which can be invoked manually
    - doRefresh() will redraw the chart with the modified data-set

Configuration

- If there is a requirement to allow easy change of different aspects of the UI elements for example for non-technical personnel, than the variable parameters must be externalized from the Code and Markup and placed into separate configuration
- The configuration parameters must be stored in a dedicated JSON object which can be easily modified with actual config values during the server side template generation
    - The code must expose an APi call, something like applyConfig() to allow re-initializing the Widget with the new configuration values
    - applyConfig() will reinitialize the widget so that the widget's new behavior or look and feel will reflect the modified configuration

Integration Tests

Since the widgets that will be developed can be part of an existing page on an existing website (www.msci.com) it has to integrate without any errors into its surrounding page. We explicitly **disallow using IFRAMEs** for integration due to the fact that IFRAMEs are hard to maintain from a SEO, UX and UI theme-ing perspective. Because of this please make sure that the delivered asset has been put through a series of integration tests.

During development and before hand-over you have to test the integration of the new widget by setting up a page which mimics the www.msci.com environ ment where the widget will be published at go-live. This page will have all the static elements (CSS, images, HTML structure) cloned from an existing web-page. You will need to ask MSCI Marketing to provide you with a sample live page on www.msci.com which you can clone into your local development environment and test the embedding and integration of the asset you are developing. Once testing is complete you will need to include a zipped version of the integration page with all of the elements of your asset. The test page will need to show that the following elements are included and the page and its static integrity is still OK.

Acceptance criteria:

- Layout ans structure stays the same
- New asset CSS does not override or modify the surrounding elements and the embedding page's CSS rules
- Page re-flow is not changed, i.e. when resizing the page to different viewport sizes the elements will not start to overlap, overflow, get hidden or become inaccessible
- The theme (skin) based design including fonts, colors, spacing, positions, sizes is retained

Meta Tags

Please use the following meta tags by default in the head section of the test page:

```
<meta http-equiv="X-UA-Compatible" content="IE=edge">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<meta content="initial-scale=1.0, width=device-width" name="viewport">
```

CSS Scope and Integration Test

Please embed MSCI!s default stylesheets in the head section of the test page. Please ask the MSCI Marketing or the IT/Web team to verify in which environment the graphics are going to be displayed (MSCI Theme / MSCI Wealth Management Theme / MSCI Research Theme). Based on the environment use the HEAD section to add the appropriate environment specific rules. With this in place all the built in styles will be provided by default (font families, font sizes, colors, grid system, etc.). It will also can prevent style conflicts or unwanted side-effects.

```
<head>...
<!—NEW MSCI THEME -->
<link rel="stylesheet" href=" https://www.msci.com/o/msci/css/main.css">
...
</head>
```

Add 'aui' as a CSS class to the HTML element, which is required to make the base Liferay theme's css rules work.

```
<html class="aui">
</html>
```

If the widget will be presented in the Wealth Management Theme, please add the following markup around the code:

```
 <div class="wrapper content_wrapper">
    <section class="msci-wealth-management-wrapper">
        <!-- Widget Markup comes here -->
    </div>
 </div>
```

HINTS

Make sure that your CSS will take effect only to the html code of the asset. Create a wrapper div around the HTML of the asset and add a unique CSS class name to it (do not use id!)

Accepted usage for example:

...

Do not use this:

...

In CSS prefix all CSS selectors to target the elements inside your wrapper only to prevent unwanded effects outside of the your code on the destination page.
Accepted usage for example:

```
.rs-graphics-wrapper .box {
 }
```

Do not use this:

```
.box {
 }
```

JavaScript Scope and Integration Test

Include your asset's JavaScript libraries on the test page and bootstrap your widget. Check that their intended scope and effect stays within the desired boundaries and there is no scope creep whereby JavaScript would effect unwanted components or sections of the page.

- **Error check:** There are no errors displayed on the console. If there are analyze them and asses their impact. Inclusion of your widget/asset on the page should not break on collision with other libraries that make up part of the embedding page.
- **Visual check:** Visual changes introduced by your code are limited to the scope of your widget. Nothing else should get resized, get hidden, moved around, etc.

Hints

Make sure that **your JavaScript code manipulates the elements inside the wrapper only**! Filter jQuery selectors to your wrapper to avoid manipulating the HTML code accidentally around your widget on the page.

Accepted usage for example:

```
var $rsGraphicsWrapper = $('.rs-graphics-wrapper');
$('.box_main', $rsGraphicsWrapper).click(function() {
// ...
});
```

Do not use this:

```
$('.box_main').click(function() {
// ...
});
```

Layout Test

Please inquire about the embedding's **page layout (and any grid system)** (1 column, 2 columns 50-50%, 2 columns 30-70%, ...) and the new asset's intended placeholder location. The test page should request the existing/desired layout with the new asset embedded into the desired placeholder location. Implement additional media query rules based on the layout if required.

HINTS

Do not use fix heights or sizes for the embedding container, because these assets will likely be deposited among a host of textual content pieces and fixed sizes will limit the natural flow of the components. White spaces can appear above or below the asset or the .
Please use **desktop first approach** in the media queries, since our theme is based on this, and it is easier to apply any fine-tuning or fixes to the widget.

Preferred usage of media queries for example:

```
@media(max-width:767px) {
 ...

}
```

Not recommended usage:

```
@media(min-width:768px) {
 ...

}
```

Web Content Accessibility

The Design should follow the Web Content Accessibility Guidelines (WCAG 2.0) where possible. We should adhere to Level A compliance (at a minimum) as defined in the official guidelines at http://www.w3.org/TR/WCAG20

Animation and Interactions

Avoid the use of Adobe Flash technologies. Do not use Adobe Flex. Any animation and interaction must be created using HTML5 frameworks like Three.js and GreenSock GSAP.

Forms development

Form design rules:

- When designing forms use the standard Lifreay Clay or Bootstrap (V4) form designations and components
- Provide the necessary client-side data input validation code (i.e. required fields, length constraints, date/e-mail format requirements). The form design should include elements for any error messages to appear.
- The preferred validation framework shall be based on Liferay Clay/Bootstrap and/or the jQuery validation framework. http://docs.jquery.com /Plugins/Validation;
- Form structure
    - use sections to delimit logical groups
    - fields should have an associated <label> tag
- Forms pages may require CAPTCHA validation (or an alternative). If this is the case then we prefer the use of Google's reCaptcha.

Pop-ups and IFRAMEs

The use of browser windows pop-ups is not allowed; instead the use of inline modal windows (through the use of CSS/JavaScript) is preferred.

IFRAMEs are not allowed. Widget integration should be done via adding a DIV tag which contains the widget!s DOM subtree.

Images and Artwork

In general, we expect web-optimized JPGs/GIFs for one-off artwork. For larger projects – or artwork that is intended to be re-purposed in the future – we expect the original, layered PSDs (compatible with Photoshop CS4). Appropriate use of vector versus raster (bitmap) based images is expected. If Adobe Illustrator is used in the design process, then the original Illustrator files should also be provided.

Images should be sensibly and descriptively named (i.e. not "logo.gif", "ad.jpg"). Use hyphens in filenames instead of underscores.
No resizing in HTML. Small, medium and large images should be supplied instead.
No hard coded text to be placed in images. Alt text to be provided for all images.

Screenshots of the delivered static pages must be be provided for the following screen resolutions:

- 768px for Small phones (iOS, Android 6.0+)
- 1024px for Medium Tablets (iOS, Android 6.0+)
- 1280px for Desktop/Laptop + Mobile devices (Windows, Mac)
- 1920px for HD Desktop/Laptop (Windows, Mac)

Performance

We expect pages to render **within 2 seconds** using Chrome latest on a standard office desktop.
Page rendering times are affected by a number of factors one of which is size. To assist with page rendering the following limitations are proposed:

- Total Page limit size limit – including all Javascript/CSS/flv – 300k
- Single Image size limit – 95k for desktop or tablet, 20k for mobile devices
- Animations (ads) – 500k
- Elements (unique HTTP requests) – less than 80
- Number of Javascript files – maximum of 4
- Number of CSS files – maximum of 2

In-page refresh

In case when user input is sent back to the servers, avoid using full page refresh after form submission. Use ajax calls instead and re-render only the sections that need updating.
Ajax calls can be achieved via Portlet Resource calls, or Service Builder JSON Webservice.