**Frontend engineering - Best practices**

**Frontend Engineering code standards & best practices**

**Overview**

This document contains guidelines for web applications built by Ticketmaster. It is to be readily available to anyone who wishes to check the iterative progress of our best practices. If you have any feedback, please email

<mailto:Matthew.french@livenation.co.uk>

<mailto:Mohamed.jama@livenation.co.uk>

This document's primary motivation is two- fold: 1) code consistency and 2) best practices. By maintaining consistency in coding styles and conventions, we can ease the burden of legacy code maintenance, and mitigate risk of breakage in the future. By adhering to best practices, we ensure optimized page loading, performance and maintainable code.

**General Practices**

**Readability vs Compression**

Readability is preferable over file-size savings when it comes to maintaining existing files. Whitespace is encouraged, where appropriate. There is no need for any developer to purposefully compress HTML or CSS, nor obfuscate JavaScript.

**Markup**

**HTML5**

HTML5 is the latest version of HTML. It attempts to solve issues found in previous iterations of HTML and addresses the needs of web applications, an area previously inadequately covered by HTML.

We will use HTML5 features when appropriate. By appropriate we mean that they are cross browser compatible, or use a solid polyfill to support HTML5 tags / features.

HTML5 streamlines the amount of code that required. Meaningless attributes have been dropped, and the DOCTYPE declaration has been simplified significantly. Additionally, there is no need to use CDATA to escape inline JavaScript, formerly a requirement to meet XML strictness in XHTML.

Pages should be tested against the W3C validator, to ensure that the markup is well formed. 100% valid code is not a goal, but validation certainly helps to write more maintainable code as well as debugging code.

**Template**

The template from the HTML5 Boilerplate and Modernizr should provide the perfect basis.

**Doctype**

The HTML5 Doctype, which triggers standards mode in browsers, is the standard doctype and is required.

All markups should be delivered as UTF-8. It should be designated in both the HTTP header and the head of the document.

Setting the character set using <meta> tags:

*<meta charset="utf-8">*

**General Markup Guidelines**

The following are general guidelines for structuring your HTML markup. Always use markup, which represents the semantics of the content in the document being created.

1. Use actual P elements for paragraph delimiters as opposed to multiple BR tags. BR tags are reserved for a forced, single line break inside a logical block of content.
2. Items in list form should always be contained in a UL, OL, or DL, never a set of DIVs or Ps.
3. Use label fields to label each form field, the **FOR** attribute should associate itself with the input field, so users can click the labels. cursor:pointer; on the label is wise, as well. note 1, note 2
4. Do not use the size attribute on your input fields. The size attribute is relative to the font-size of the text inside the input. Instead use css width.
5. When using tables ( for tabular data display only ) make use of THEAD, TBODY, and TH tags (and Scope attribute) when possible.

**CSS**

**General principles**

Add CSS through external files, minimizing the # of files, if possible. It should always be in the HEAD of the document. Use the LINK tag to include, never @import.

Including a stylesheet:

*<link rel="stylesheet" type="text/css" href="myStylesheet.css" />*

Never use inline styling:

*<p style="font-size: 12px; color: #FFFFFF">Lorem ipsum</p>*

* Never include styles inline in the document, either in a style tag or on the elements. CSS styles that are inlined in HTML documents get downloaded every time the HTML document is requested. In addition, the style of elements is adjusted mid-layout, which causes the browser to potentially repaint the page more than necessary.
* Use the CSS reset included with html boiler plate.
* IDs are generally unnecessary. They are restrictive. In some cases you may need IDs to target specific elements, but there is little to no performance benefit to doing so. It is usually more efficient and flexible to use jQuery selectors based on the Sizzle selector engine.
* Understand cascading and selector specificity so you can write very terse and effective code. [More info](http://coding.smashingmagazine.com/2007/07/27/css-specificity-things-you-should-know/)

Write selectors that are optimized for speed. Where possible, avoid expensive CSS selectors, especially in the key selector (the selector furthest to the right). For example, avoid the \* wildcard selector. Don't qualify ID selectors or class selectors with element types (e.g. div#id or div.class). This is especially important with web applications where speed is paramount and there can be thousands or even tens of thousands of DOM elements. The developer / nightly version of Google Chrome has a CSS selector profiler [more info](http://forum.boagworld.com/discussion/comment/45270#Comment_45270)

**CSS Formatting**

At minimum, format CSS with selectors on one line and each property on its own line. The declarations are indented.

As an enhancement to that style, related or child styles and additional 2 or 4 spaces. That allows for hierarchical scanning and organization and makes (for some people) an easier-to-read stylesheet.

For selectors that have 1 or 2 declarations, place the declarations on 1 line.

.some-class li a { text-decoration: none; }

.some-class ol li a {

color: #003366;

font-family: arial;

list-style-type: lower-roman;

}

**Classes vs. IDs**

You should only give elements an ID attributes if they are unique. They should be applied to that element only and nothing else. Classes can be applied to multiple elements that share the same style properties. Things that should look and work in the same way can have the same class name. IDs are restrictive in that you are guaranteeing there is only one element with that name. That is a fairly large assumption about a page that should be avoided.

**Naming Conventions for Selectors**

It is always preferable to name something, be it an ID or a class, by the nature of what it is rather than by what it looks like. For instance, a class name of bigBlueText for a special note on a page is quite meaningless if it has been changed to have a small red text color. Using a more intelligent convention such as noteText is better because when the visual style changes it still makes sense.

**Internet explorer bugs**

Inevitably, when all other browsers appear to be working correctly, any and all versions of Internet Explorer will introduce a few nonsensical bugs, delaying time to deployment. While we encourage troubleshooting and building code that will work in all browsers without special modifications, sometimes it is necessary to use conditional if IE comments for CSS hooks we can use in our stylesheets.

Fixing IE

*<!--[if lt IE 7]> <html class="no-js ie ie6" lang="en"> <![endif]-->*

*<!--[if IE 7]> <html class="no-js ie ie7" lang="en"> <![endif]-->*

*<!--[if IE 8]> <html class="no-js ie8" lang="en"> <![endif]-->*

*<!--[if IE 9]> <html class="no-js ie9" lang="en"> <![endif]-->*

*<!--[if gt IE 9]><!--><html class="no-js" lang="en"><!--<![endif]-->*

This is the preferred method as opposed to using css hacks or swapping out css files in the conditional IE comments.

If you are using the htmlboilerplate and/or modernizr, the method above is available.

**Images**

* Use CSS image sprites.
* Check to see if the icon / image you require already exists even as a single image or within a sprite image.
* Use data uri to embed your image within cached css stylesheets. If using data uri, it is recommended to use in conjunction with modernizr , so you can provide a fallback for browsers that do not support data uri. You can use. [More info](http://www.websemantics.co.uk/online_tools/image_to_data_uri_convertor/%20to%20encode%20your%20images)
* Use Smushit to reduce image file sizes, both prior to encoding data-uri images and for traditional external file-based images. [More info](http://www.smushit.com/ysmush.it/)

*.datauri .tooltip:after {*

*background-image:*

*url('')*

*}*

*.no-datauri .tooltip:after { background: url(/static/ui\_toolkit/v1.0/images/uitk-sprite.png) 100% -133px no-repeat;}*

**JavaScript**

**JavaScript Libraries**

All new pages / projects should attempt to use Vanilla Javascript or use jQuery.

jQuery is the utility library of record. Cross-site domain-specific functionality (like date functions) should be built as jQuery plug-ins. Page or Pod specific functionality should be written in simple JS includes.

ALL JavaScript is REQUIRED to be loaded after DOM-Ready. This has a profound effect on the behavior of the scripts. While execution order can still be managed in script queues, you cannot expect JavaScript to have been executed during the course of HTML parsing and rendering. The HTML parses and renders first, THEN scripts are executed.

**General Coding Principles**

* All JavaScript code should be housed in external files.
* Name variables and functions logically: For example: popUpWindowForAd rather than myWindow.
* Strive to create **functions** which can be generalized, take parameters, and return values. This allows for substantial code reuse and, when combined with includes or external scripts, can reduce the overhead when scripts need to change.
* Comment your code! It helps reduce time spent troubleshooting JavaScript functions.
* AGAIN COMMENT YOUR CODE.
* Minimize global variables - the less globals you create, the better. Generally one, for your application namespace, is a good number.
* Is some JS functionality on your page requires options that can only be set at runtime, create a JavaScript object to hold the options in the HEAD and load it late. Do not try to run functions inline to create the options or use the options. They should simply be available in scope when your script eventually executes post DOM-Ready.
* For onclick events, make use of the data-onclick="someFunc()" that can be detected in core.js. This is to avoid inline JS or use event listeners for your click events
* Use JSON (JavaScript Object Notation) for hierarchical data structures.
* [Be idiomatic about your JavaScript](https://github.com/rwldrn/idiomatic.js/)

**Event Delegation**

When assigning unobtrusive event listeners, it is typically acceptable to assign the event listener directly to the element(s) which will trigger some resulting action. However, occasionally there may be multiple elements which match the criteria for which you are checking, and attaching eventlisteners to each one might negatively impact performance. In such cases you should use event delegation instead.

jQuery's delegate() is preferred over live() for performance reasons. Once jQuery 1.7 is in use, on() would be the preferred event handler.

**Debugging**

Even with the best of validators, inevitably browser quirks will cause issues. There are several invaluable tools, which will help to refine code integrity and loading speed. It is important that you have all of these tools available to you, despite the browser you primarily use for development.

The following is a list of helpful debuggers and speed analyzers...

* Firefox: Firebug, Page Speed, YSlow, Web developer toolbar. FF11 has it's own inspect tool.
* Google Chrome: Developer Tools, Web developer toolbar
* Safari: Web Inspector
* Opera: Dragonfly
* Internet Explorer 6-7: Developer Toolbar
* Internet Explorer 8-10: Developer Tools

**Accessibility**

It is advisable to follow the [W3C checklist of checkpoints for accessibility](http://www.w3.org/TR/WCAG10/full-checklist.html)

**Performance**

**Optimize delivery of CSS and JavaScript**

Take note of the [Yahoo Performance Guidelines1.](http://developer.yahoo.com/performance/)

* Reduce image file size using smush it
* Avoid inline <script> blocks
* Where possible CSS should be located in the <head> of the document
* Make use of the minification tools available
* Do not simply concatenate all the files together. This guarantees serial loading of the resources, when what we really want is 3-5 parallel loading files to maximize the browser's HTTP connections.
* Make a concerted effort to reduce http requests overall - watch browser connections and try to set up requests so that there are never more than 3-5 requests needed to a single subdomain at a time. This prevents 'waiting' for a connection to open up.

**Measure performance**

Teams should measure the performance of their pages using tools such as:

* YSlow
* Page speed
* Chrome canary profiler

**Code Reviews**

Why should I have a code review?

Code reviews are a strategic investment of time to mitigate risk on a project.

**Code reviews increase the overall level of knowledge across projects**

Since code reviews involve members from within and outside a project team, techniques and best practices are easily shared throughout the entire FE team.

**Code reviews eliminate bugs before they are reproduced from a few templates into multiple pages**

Ideally, code reviews are conducted early in the development process, before full production of pages begins. When templates are reviewed by the team and run through multiple validation tools and browsers, bugs can and will appear. This is the ideal time for bugs to be fixed.

**Code reviews give a set of fresh eyes an opportunity to spot issues with code**

Reviewers from outside a project team can spot issues with code more easily than authors of markup, who have been working with code for a longer amount of time.

**Who should participate in a code review?**

Ultimately, the Frontend Lead on a project is required to ensure that proper code review procedures are followed. Persons required in the review process depends on the code for review. I.e. If the review is for Frontend code, ask the Frontend engineering team to review the code.