Frontend Development Guideline

General Guidelines

Frontend Definition

To start this guidelines, firstly lets define what's a front-end and a back-end. In software engineering, the terms "**front-end**" and "**back-end**" are distinctions which refer to the separation of concerns between a presentation layer and a data access layer respectively. So when we talk about front-end development, what we're really talking about is developing the part of the web that we can see and interact with. The front-end usually consist of two parts, web design and front-end web development (HTML, CSS and JavaScript). In this guidelines, we will mostly talk regarding the latter which is writing codes using HTML, CSS and JavaScript, how to produce a high quality page with good performance and maintainable codes.

Three Pillars of Frontend Development

There are already plenty of guides and best practices that we can found on the web regarding writing HTML, CSS and JavaScript, but for the sake of simplicity let's take a look at the Three Pillars of Frontend Development, which are:

1. Separation of content, presentation and behavior
2. Markup should be well formed, semantically correct and generally valid
3. JavaScript should progressively enhance the experience.

What these pillars does is lays a groundwork that allows us to focus our priorities. Its practical, for instance, writing generally valid code but not stringently enforcing W3C validation. The point is to have code that works and that others can understand. With that, other things start to fall into place, like performance and maintainability.

1. Separation of content (HTML), presentation (CSS) and behavior (JavaScript)

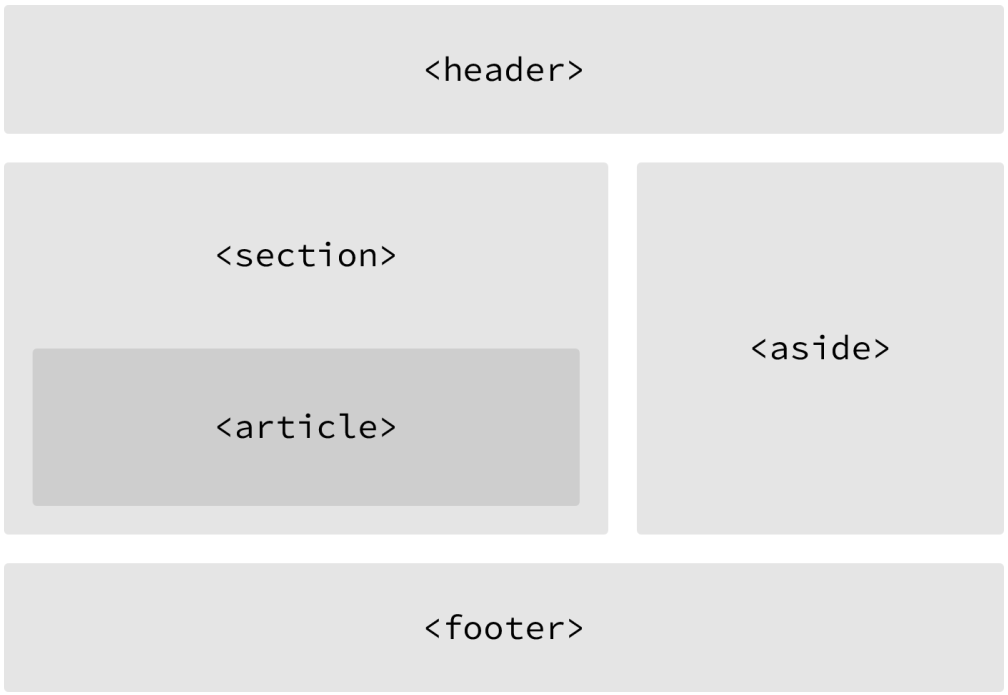
Separation of concern is a programming principle that encourages grouping functionality in ways that reduce overlap. The core concept is that by improving the separation of concerns, we improve a program's clarity and durability, separation of content, presentation and behavior also vanquished the `<font>` tags, spacer gifs and inline JavaScript that polluted our website, the old web structure presentation and behavior lived in one later but this principle divided them. Here is some example of how we can achieved separation of content, presentation and behavior:

1. HTML, CSS and JavaScript files divided into their own files
2. The content (HTML markup) is dumb, that means that its style and behavior agnostic. Usually, this means writing it first, and only minimally changing it to accommodate JavaScript and CSS when necessary
3. Classes are content oriented and not presentational (ex. 'content-info' vs. 'right-col')
4. Elements are styled using clever selectors (pseudo-classes, attribute selectors, etc.) beside classes and ids
5. JavaScript should be unobtrusive, it selects, operates on, and injects markup to aid behaviors

2. Markup should be well formed, semantically correct and generally valid

Semantics within HTML is the practice of giving content on the page meaning and structure by using the proper element. Semantic code describes the *value* of content on a page, regardless of the style or appearance of that content. There are several benefits to using semantic elements, including enabling computers, screen readers, search engines, and other devices to adequately read and understand the content on a web page. Additionally, semantic HTML is easier to manage and work with, as it shows clearly what each piece of content is about.

For the longest time the structure of a web page was built using divisions (`<div>`, `<span>`). The problem was that divisions provide no semantic value, and it was fairly difficult to determine the intention of these divisions. Fortunately HTML5 introduced new structurally-based elements, including the `<header>`,`<nav>`, `<article>`, `<section>`, `<aside>`, and `<footer>` elements. All of these new elements are intended to give meaning to the organization of our pages and improve our structural semantics. They are all block-level elements and do not have any implied position or style. Additionally, all of these elements may be used multiple times per page, as long as each usage reflects the proper semantic meaning. For more information regarding the usage of these structurally-based elements, please take a look in here and here.



One possible example of HTML5 structural elements giving meaning to the organization of our pages (http://learn.shayhowe.com/html-css/getting-to-know-html/)

Some of the general markup guidelines are as follows:

1. Use actual `<p>` elements for paragraph delimiters as opposed to multiple `<br>` tags.
2. Make use of `<dl>` (definition lists) and `<blockquote>`, when appropriate.
3. Items in list form should always be housed in a `<ul>`, `<ol>`, or `<dl>`, never a set of `<div>` or `<p>`.
4. Use `<label>` fields to label each form field, the `for` attribute should associate itself with the input field, so users can click the labels. Setting cursor to pointer on the label is wise, as well.
5. 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`.
6. Place an HTML comment on some closing div tags to indicate what element you're closing. It will help when there is lots of nesting and indentation.
7. Tables shouldn't be used for page layout.
8. Use microformats and/or microdata where appropriate, specifically hCard and adr.
9. Make use of `<thead>` , `<tbody>`, and `<th>` tags (and scope attribute) when appropriate.
10. Use the least amount of markup when possible as this will make the code more maintainable as well as lightweight.
11. Avoid using old legacy HTML tags such as `<b>`, `<i>`, `<center>`, `<font>`.

3. JavaScript should progressively enhance the experience.

Progressive enhancement is a layered approach or a guidance for building websites using HTML for content, CSS for presentation, and JavaScript for interactivity. If for some reason JavaScript breaks, the site should show some notification or users will still be able to see some content. If the CSS doesn't load correctly, the HTML content should still be there with meaningful hyperlinks. This approach typically yields a result that is robust, fault tolerant, and accessible. People that have the latest devices will get the most progressively enhanced experience, and people on slow connections or less capable devices will still be able to access the information. Websites don't need to look the same in every browser, they just need to deliver core content and functionality. So at its core, progressive enhancement is not about specifics like whether our website works with JavaScript, CSS3 Animations, etc. but it is about thinking about a web from the content and gradually enhancing the experience with JavaScript.

This approach is not a rule to build websites, but more as a reminder or a guide because there are some occasions where it can't be applied as a whole such as when we build a web applications or website using JavaScript MVC Frameworks like Angular, Backbone and Ember where it is dependent on the availability of JavaScript. But when possible, we can applied this approach for a robust, fault tolerant and accessible web.

Software bugs are costly to fix and revisiting codes can often takes quite some time; so it is critical to reduce the time it takes to understand code, either written by our self some time ago or written by another developer in the team. It's critical for the business and the developer's happiness as well, because in the end, developers would all rather develop something new and exciting instead of spending hours and days maintaining old legacy code. Therefore, it is crucial to be able to produce a maintainable code.

Here are some best practices that we can use when writing JavaScript:

1. Maintainable code, which mean the codes are:
    a. Readable
    b. Consistent (please refer here for principles to write a consistent JavaScript)
    c. Documented
    d. Looks as if it was written by the same person
2. Minimizing globals, learn more about global namespace pollution in here.
3. If we are using jQuery, write a smarter jQuery.
4. Read more about others JavaScript best practice here.

References:

- http://www.bbc.co.uk/guidelines/futuremedia/technical/semantic_markup.shtml
- http://code.tutsplus.com/tutorials/the-essentials-of-writing-high-quality-javascript--net-15145
- http://isobar-idev.github.io/code-standards/#_pillars_of_front_end_development
- http://viget.com/extend/client-side-separation-of-concerns-are-we-doing-it-wrong
- http://learn.shayhowe.com/html-css/getting-to-know-html/
- http://icant.co.uk/articles/pragmatic-progressive-enhancement/
- https://github.com/rwaldron/idiomatic.js