Design Spike: Research CSS Plan

See Jira Story <https://jira.intuit.com/browse/QBSE-8575>

# Pain Points with existing QuickBase CSS

* Large css files, typical of large legacy apps
  + Everyone adds to the bottom, adding to tech debt
  + File size deters us from looking for existing rule
* Not much reuse leads to unnecessary inconsistencies in users experience
  + Very specific selections
  + Specificity of descendants in rules effect performance
  + Editor inheritance (aka copy/paste)
    - then later tweaks don't happen in all places automatically
    - border-radius needs browser prefixes every time if missed leads to inconsistencies on different browsers
* Fear of change effecting too much
  + Hard to tell effect of changes
  + Vicious cycle adding more specific rules

## Analysis of current QuickBase Css files for dups

grep -r –e [^a-z]*<term>*[^a-z] . | wc -l

|  |  |
| --- | --- |
| Key | Count |
| font-size | 552 |
| h1-h6 | 150 |
| margin | 2023 |
| padding | 2178 |
| color | 2086 |
| #ff | 298 |
| tr | 623 |
| td | 799 |

A lot of duplication and specification we can do better in the new stack ☺ with some forethought and putting in some abstractions.

Here I was not looking for exact values, but a general order of magnitude.

# CSS Goals for re-architecture

* Reuse for more consistency in what we present
* Living Style Guide for up-to-date reference and for fostering reuse
* Theme-able for flexibility and branding features
* Ease of change for rapid experimentation and development

# CSS Methodologies

Some the latest thinking around creating maintainable CSS

## OOCSS

Object Oriented Cascading Stylesheets

Defined by Nicole Sullivan, see <https://github.com/stubbornella/oocss/wiki>

OOCSS is a way of writing CSS that avoids duplicating code by separating styles into layered abstractions.

In OOCSS a CSS “object” is a repeating visual pattern, which can be abstracted into an independent snippet. That object can then be reused throughout a site. i.e. find all the button and then a promo-button.

### Separation of structure from skin

Separate the visual features style statements (colors, gradients, shadow, borders) from the invisible features / structure (width, height, overflow) into separate classes. i.e.

In the stylesheet use this separation.

.widget {

width: 500px;

min-height: 200px;

overflow: auto;

}

.skin {

border: solid 1px #ccc;

background: linear-gradient(#ccc, #222);

box-shadow: rgba(0, 0, 0, .5) 2px 2px 5px;

}

then in html have:

<div class="widget skin">

For everything that gives a widget its widget-ness does in the widget class statements. Any optional variations on the object

### Separation of containers and content

Separate the visual features style statements (colors, gradients, shadow, borders) from the invisible features / structure (width, height, overflow) into separate classes. i.e.:

In the stylesheet if we have this decedent selector for level 3 heading in the sidebar

#sidebar h3 {

font-family : Arial, san-serif;

font-size : .8em;

color : #777;

}

Then for another area we want the exact same header like in footer with the exception of font size we would have to add the footer container to the selector list and another rule to override or worse just copying everything from sidebar h3 for footer h3

#sidebar h3, #footer h3 {

font-family : Arial, san-serif;

font-size : .8em;

color : #777;

}

#footer h3 {

font-size : 1.5em;

}

Instead have a class the a describes the meaning of the object say for example called article-header and article-header-featured to describe what it is. And remove the id selectors so these type of header can be used anywhere.

.article-header {

font-family : Arial, san-serif;

font-size : .8em;

color : #777;

}

.article-header-featured {

font-size : 1.5em;

}

then in html

<sidebar> <h3 class="article-header">…

<footer> <h3 class="article-header article-header-featured">…

<section> <h3 class="article-header">…

This ends up with duplication and styles that are declared with an id'd descendent selector are *not reusable*. An object should look the same no matter where you put it. Better to have instead of h3 a class like 'category' and style it for consistency. When you style with just id for a rule selector you are making the style a singleton, it can't be reused on the page. Instead use a class to keep it flexible.

OOCSS encourages us to give more forethought to what is common among different elements, and then separate those common features into modules, or objects, that can be reused. You can still use IDs in html but avoid IDs in css selectors for best reuse.

### Benefits:

The benefits are performance improvements and maintainability. End up with a lot less CSS using this methodology.

Using multiple classes in the HTML you can simulate inheritance so that you can stop copy and pasting in the CSS file

Heavy reuse of CSS code, so less CSS code needed.

* Smaller files less time downloading,
* Every reuse of style makes it faster to develop features and a more consistent interface.
* Fewer styles means the less time the rendering engine spends calculating “computed values”

No longer adding to the bottom of the stylesheet building up tech debt, instead with some modularity we will be able to locate and reuse and extend what we have.

OOCSS is about writing tiny DRY awesome CSS.

Challenges:

Lots of classes that aren't really objects but utilities are necessary to implement OOCSS. Changes to these non-semantic classes mean changes to your html.

HTML bloat with potentially lots of class attributes needed on elements.

## SMACSS

Scalable and Maintainable Cascading Stylesheets

SMACSS is a also a concept of designing consistent CSS to maintain flexibility throughout projects and within a team. Similar to OOSCC Learn more [here](https://smacss.com/).

## Sass

Syntactically Awesome StyleSheets

A set of extensions to the css language that lets you express more with css. It's about writing tiny DRY Sass. Provides missing features of a language, variables and functions.

Sass doesn't have an opinion but it's an excellent tool for documenting the patterns you design.

Main Features of Sass

* Variables
* Nesting
* Mixins
* Inheritance

### Variables

Use variable for all the colors, common margins etc. i.e.

$defaultWidth : 200px

width : $defaultWidth

You can also do math on variables, makes it easier to make things come out consistently. Have a level of indirection between the color codename variable and a variable for its use case. i.e. have $red:#f00 and $errorColor: $red then use $errorColor in rules

### Nesting

You can indent the tr rule under the table curly braces , and the td and th rules nested within the tr etc . Sass will compile the nested items as descendent selectors "table tr", "table tr td". All your selectors end up scoped to where they are nested.

Though don't get to crazy follow the inception rule 2 to 3 levels max, any more will be slow as it generates long selectors.

### Mixins

A way of including chunks of code in another part of the stylesheet. Good when you need reuse in many place but not necessary want a class for it in the html.

/\*\* Sass \*\*/

@mixin clearfix {

zoom:1;

&:before, &:after {

content:" ";

display:table;

}

&:after {

clear:both;

}

}

.box {

@include clearfix;

}

/\*\* Generated CSS \*\*/

.box {

zoom:1;

}

.box &:before,

.box &:after {

content:" ";

display:table;

}

.box &:after{

clear: both;

}

Copies the mixin contents into the location of the include statement so you don’t need to have this fake clearfix class in your html, making it hard for novices to understand.

Careful if you have lots of pseudo classes on mixins or extends, check the resulting css to make sure its generating what you'd do.

Optimize mixin usage, only use when you would not want a class in the html., like border-radius, clearfix. And since is being replicated use when there are only a small number of property, value pairs in it. DRY Sass != DRY CSS

### Selector Inheritance

Sass uses @extend to simulate inheritance (so we can stop repeating ourselves) he @extend directive in Sass lets you inherit styles from another selector without duplicating everything like a @mixin.  A trade of @extend add some bloat in selectors vs @mixin some bloat in property value pairs.

/\*\* Sass \*\*/

.btn {

border: 1px solid #ccc;

border-radius: 6px;

}

.primaryBtn {

@extend .btn;

border-color: $green;

}

.secondaryBtn {

@extend .btn;

border-color: $orange;

}

/\*\* Generated CSS \*\*/

.btn, .primaryBtn, .secondaryBtn {

border: 1px solid #ccc;

border-radius: 6px;

}

.primaryBtn {

border-color: $green;

}

.secondaryBtn {

border-color: $orange;

}

### Placeholders

Use when you want to abstract something but don’t need the class itself in the stylesheet.

%message {}

.error {@extends %message; …}

.warning {@extends %message; …}

.info {@extends %message; …}

These stub class declarations can be use as a base set of common prop/val pairs and other classes can @extend it but it doesn't get output in the generated css

## Why Sass and not Less?

* Sass has all that Less has and more - see Compass rich set
* Sass support real functions Less just has guards no loops, for etc. Richer control
* Colors must be hex in Less not human readable in vars, params etc.
* Math is a confusing in Less it has no parentheses
* Sass has extends and Less doesn't it just has mixins which make copies of statements

## Compass

Provides a boatload of mixins and a toolbox for Sass. Provides provide a huge collection of Sass mixins to handle CSS3 features so you don’t have to deal with vendor prefixes or CSS hacks. [Best practice](http://compass-style.org/help/tutorials/best_practices/) to have rhythm units in the new rem (root element) measurement which there is a mixin for with fallback to pixels.

## OOCSS / SMACSS and Sass

It makes the most sense to the best practices from these methodologies along with Sass. So we get the best of both worlds: modular CSS without bloated, hard-to-maintain HTML.

OOCSS/ SMACSS is what you're trying to say in your architecture of styles and Sass is a way of saying it.

Sass uses @include to simulate mixins. OOCSS doesn’t have a good way to support mixins, tends to duplicate code. So if you have common css statements like clearfix or 7pixel border radius, they can be keep in 1 place and reused without adding a .clearfix class as needed with plain OOCSS.

We will also derive even more maintainability with the use of variables for structure and skin separations. The variables include file will have all the hard coded values for font-size, color pallets etc. so that changing the variable will often all that’s needed to update the design.

Similarly use of modules and placeholder css classes will allow styling changes to happing in the css without need to modify the html unless there is new content

See button example in doc example directory.

## Style Guides

From <https://github.com/davidhund/styleguide-generators>

*Pattern Libraries (or*Style Guides*) are a helpful tool when developing webapps. Read more about Creating Style Guides at this*[*A List Apart article*](http://alistapart.com/article/creating-style-guides)*.*

*Maintaining a*static*Pattern Library (in HTML/CSS) is hard work and errorprone. There are, however, various tools that help us*generate*a dynamic Pattern Library or*[*'Living Style Guide'*](https://speakerdeck.com/hagenburger/style-guide-driven-development)*.*

I took a look at <http://livingstyleguide.org/> see examples, and generated from \_.scss from our repo. It's promising so this or something similar is worth while for automation of a living guide moving forward.

# Guidelines

## Files

* Live in the assets/css directory .
* Use .scss for sass file extension
  + Use the standard CSS syntax i.e. SCSS not the Sass alternative without curly braces and semicolons
* Prefix filename with \_ to have that scss partial imported at build time

## File Internals

* Use comments at major sections
* Document appropriate use case of the style
* Make use of partials to make the files/merges reasonable, no super long files nor super many partials

## CSS/SCSS rules

There will obviously be times when some of these rules will be broken, but overall, these are good habits to develop and will give us stylesheets that are smaller and easier to maintain.

* Avoid the putting descendent selectors too high in the DOMtree, rarely use location-dependent styles. (i.e. avoid use .sidebar h3) A good rule of thumb is that anything within the body of a container is clearly a separate object.
* Appropriate use of the cascade (.module .inner) is when the sub-node is really part of the larger parent object, if inner can't can’t exist on its own.
* Avoid IDs as styling hooks, it limits reuse
* Avoid attaching classes to elements in your stylesheet (i.e. don’t do div.header or h1.title)
* Use for example class="media" for a image/video component rather than directly styling the <img> element
* Except in some rare cases, avoid using !important
* DRY - don't repeat yourself , Write dry sass
* Have a configVariables file with all the hard coded constants and add another variable to have a layer of indirection between the value and the usage. So styling changes only need to change the style sheet not the html content.