CSS for Modern Apps

Part 3 - Modular CSS

Johan Steenkamp / @johanstn

Part 1 we discussed CSS and using less specific selectors, grid systems, and using SCSS with more specific selectors to isolate modules.

Part 2 we tackled the Holy Grail layout

In Part 3, we'll look at the really hard CSS problems and how we can solve them by applying what we have learnt. We'll do this using an innovative package management tool to realize a simple component approach.

What makes CSS hard?

- Vertical centering
- Equal height columns
- Browser inconsistencies
- Tricks, hacks, weird property combinations

Not really hard problems - Googling and remember browser tricks and hacks

In parts 1 and 2 I showed how you can solve these problems.

A quick recap...

CSS

- position (static, relative, fixed, absolute)
- float (left, right)
- clear (left, right, both)
- display (block, inline, inline-block, none)
- margin (auto), padding
- box-model (box-sizing)
- .clearfix hack ({clear:both; overflow: auto;})

Although browsers are better requirements are more complex with mobile devices, screen sizes and orientations to think about.

So you still need a solid foundation by really understanding of basics of CSS layout.

First read these articles and tutorial:

1. CSS Positioning 101: http://alistapart.com/article/css-positioning-101
   Take note of how relative positioning works.

2. CSS Floats 101: http://alistapart.com/article/css-floats-101
   This article explains why parent elements containing floated elements do not expand to completely surround them.

3. Review the excellent Learn Layout step-by-step tutorial: http://learnlayout.com/
   This tutorial gives you everything you need to tackle layouts and also has a section on Flexbox.

Learning by experimenting with positioning and floating using the project files in the `/css-position-float` project folder from Part 2.

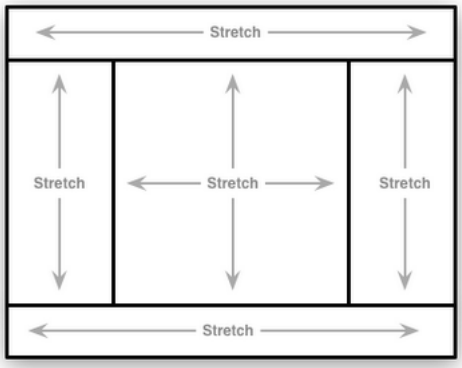
Should you use floats or inline-block? http://www.vanseodesign.com/css/inline-blocks/

Can you fix this in-line block layout? http://codepen.io/jsteenkamp/pen/vOGWem

Holy Grail Layout

5 panel layout (header + 3 columns + footer)

- float
- calc(…)
- CSS tables
- flexbox
- *grid system (skeleton)*



Demonstrated a number of solutions:

1. Floating
2. Float and calc(…)
3. CSS tables
4. Flexbox

Another option is using a grid system like Skeleton (Bootstrap, Foundation …) but the footer may still need to work.

Use Chrome and enabling the DevTools: https://developer.chrome.com/devtools

Install Pesticide to toggle element outlining: http://pesticide.io/

SCSS (Sass)

- @import
- $variable
- @mixin / @include
- @extend
- %placeholder
- Interpolation #{%placeholder}
- Source maps

Using SCSS we can be DRY (Don't Repeat Yourself) and modularize our CSS

Read the SASS Basics section http://sass-lang.com/guide

Note: SCSS is the newer version of Sass. SCSS files are valid CSS, this is not true for Sass.

Getting started with SCSS is easy after installing some tools/plugins.

Popular Web IDEs support SCSS via plugins so you can write SCSS and output CSS.

variable: `$my-color: red;`
mixin: `@mixin border-radius($radius) -> @include border-radius (10px)`
extend: `.column { … } -> @extend .column`
placeholder: `%column{ … } -> @extend %column`
interpolation: `using SCSS in strings (in other CSS rules)`

Source maps: You can set your SCSS pre-processor to output a source map. This is used by your web browser dedugging tools (Chrome: DevTools) to map the generated CSS code to the corresponding SCSS code for debugging.

Without a source map you would have to figure out where in the SCSS file the CSS you are examining was coded. This is complicated if you are using imported SCSS

files and other features like mixins, extends and computed styles (iterators, loops ...).

You do not need to write you own mixins. There are solutions like this one for Flexbox:

https://github.com/mastastealth/sass-flex-mixin

You can also use a task runner like Gulp to process your SCSS and automatically add browser specific extensions using plug-ins like autoprefixer:

https://github.com/postcss/autoprefixer

Really Hard CSS Problems

- Scoping styles
- Specificity conflicts
- Non-deterministic matches
- Dependency management
- Removing unused code

Selectors are the biggest determining factor in how scalable your CSS is

Familiarize youself with CSS selectors: http://code.tutsplus.com/tutorials/the-30-css-selectors-you-must-memorize--net-16048

Non-deterministic matches: Watch out for combinators such as " ", ">", "+", and "~"), especially the last two (adjacent and sibling)!

Example of non-deterministic matching: http://codepen.io/jsteenkamp/pen/NqNvbZ

Also happens in apps where CSS is different due to order in which views are loaded. Very hard to reproduce.

Selectors determine how scalable your CSS is.

Most developers use OOCSS and naming conventions like BEM to manage (not solve) these problems

OOCSS

OOCSS goal is reusable code, maintainability, performance

- Class selectors
- No element type selectors
- No ID selectors
- No descendant combinators
- Presentational classes in markup

.BLOCK{__ELEMENT[--MODIFIER]}

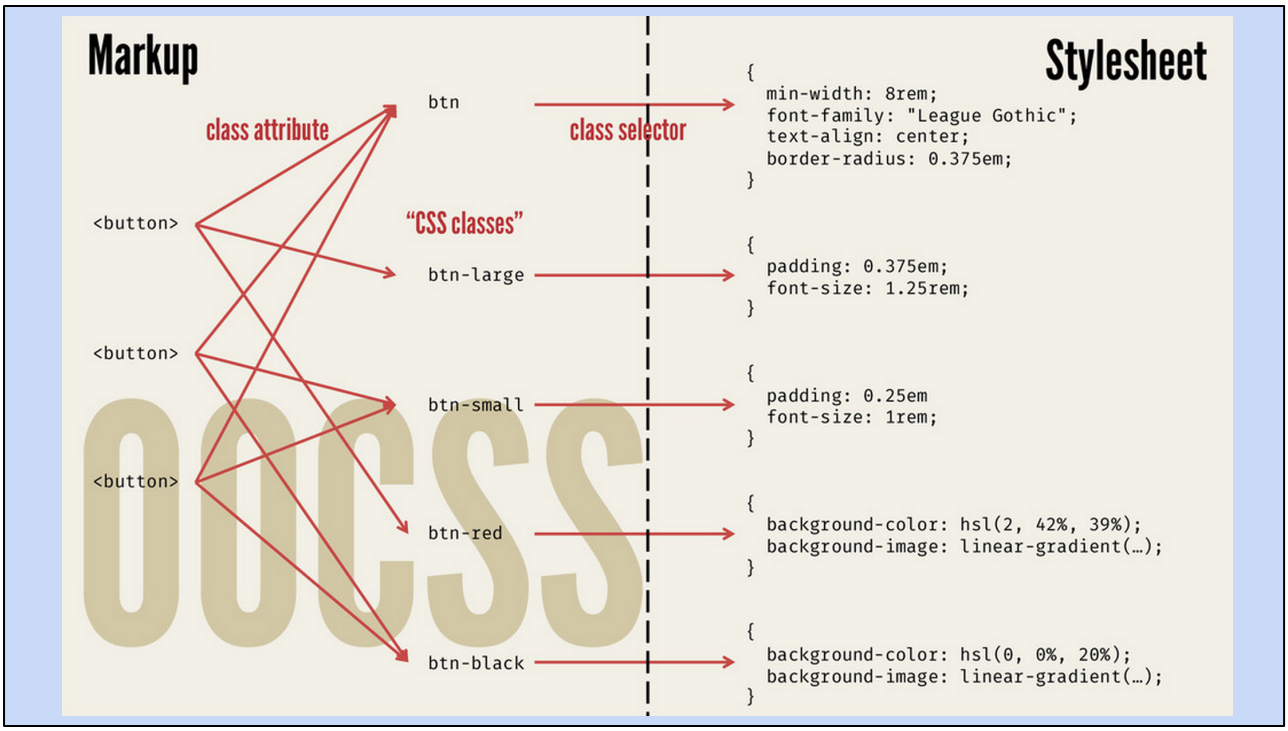
OOCSS - “object” is a repeating visual pattern that can be abstracted into an independent snippet.

This was about how the developers deal with component development in CSS now.

CSS as a technology does not provide us a solution.

BEM is a naming convention used to “fix” CSS.

The Bootstrap CSS framework is effectively >600 “globals”!



OOCSS - goal is reusable code, maintainability, performance

- Class selectors
- No element type selectors
- No ID selectors
- No descendant combinators
- Presentational classes in markup

Example with 3 buttons:

- Send Order = big, red
- Change Address = small, black
- Change Basket = small, black

Example

```
<button class=”btn btn-large btn-red”>Send</button>

<i class="mdi mdi-format-list-bulleted icn--
medium content-area__header-icon"></i>
```

/assets/icons/css/materialdesignicons.scss

/assets/styles/common/icons.scss

/assets/styles/common/content-area.scss

Example from current project.

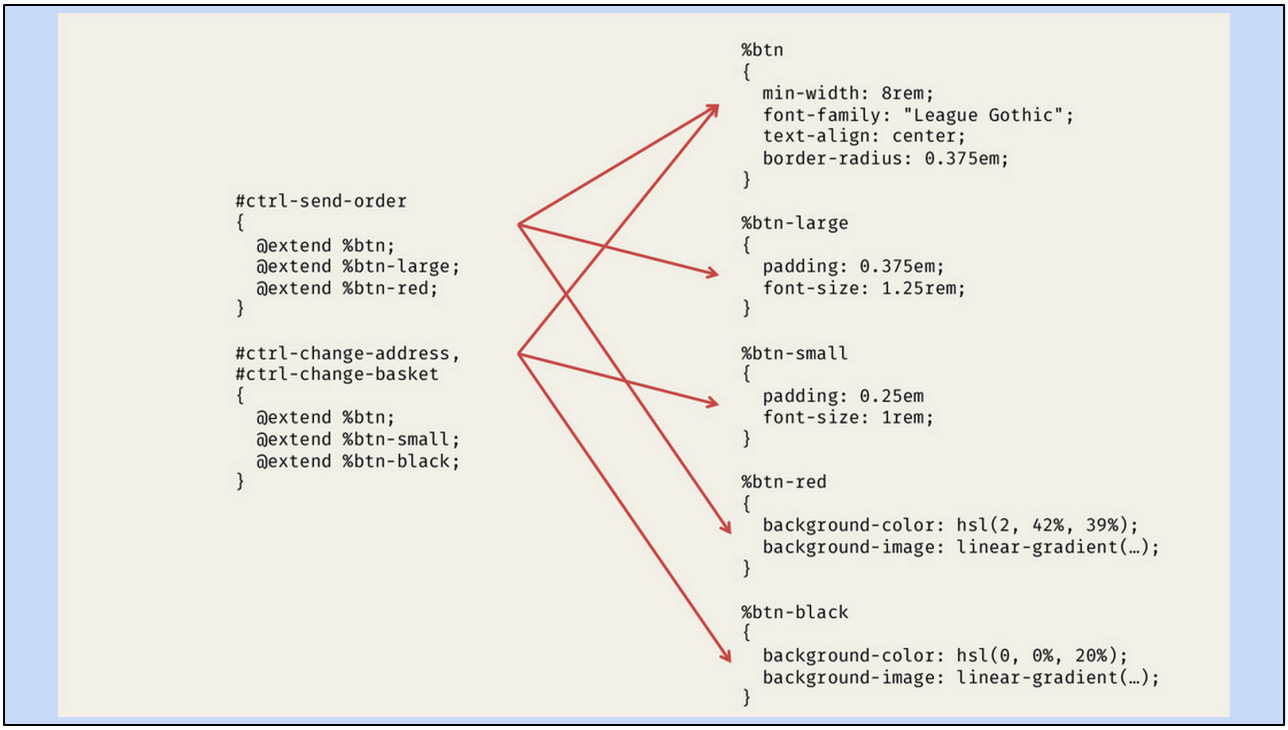
MindBEMding - bigger presentational HTML and .distributed__ugly--funky CSS!

BEM gets complicated due to combined classes. Gets worse as developers add more classes to “fix” things they cannot track down.

Hard to debug, lots of searching and switching between folders/files, can use source maps but can get complicated with `@imports`.

You have to manually add `@imports` to your project SCSS file import hierarchy.

The order of your `@imports` is important to apply correct style cascades.

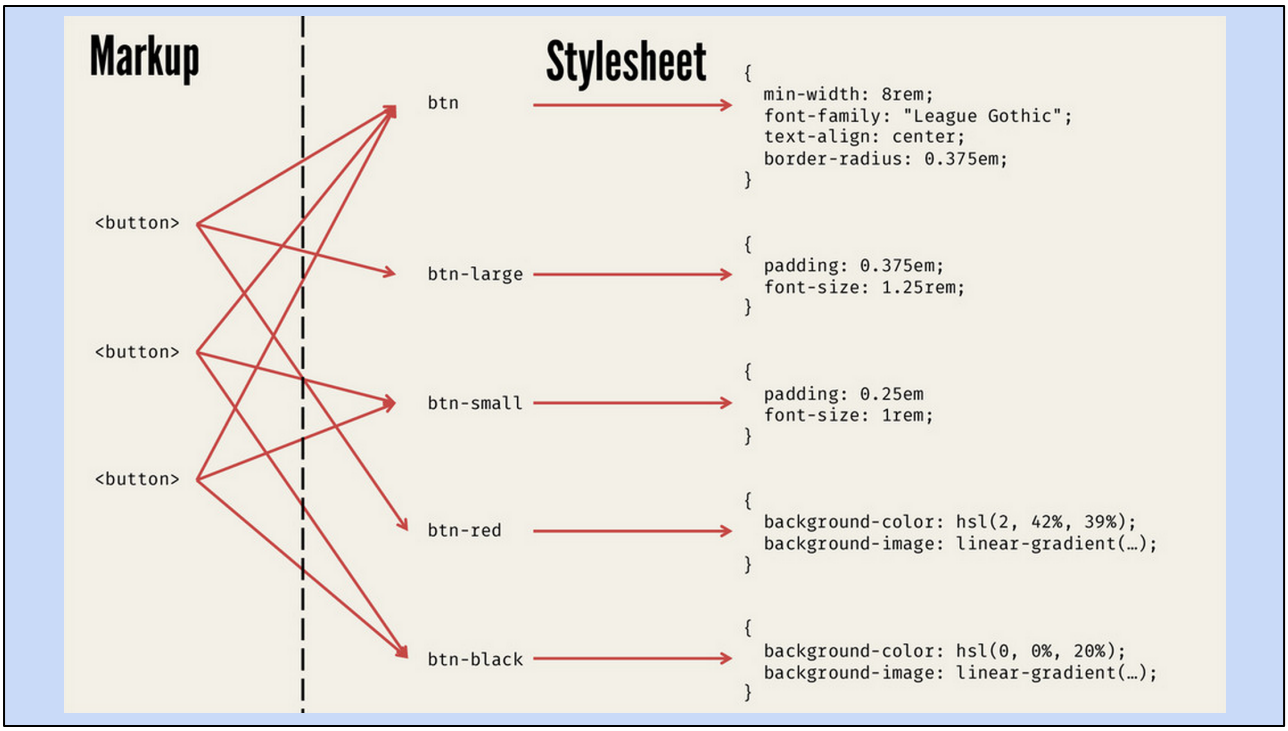



```
<button type=”submit” id=”ctrl-send-order”>Send
Order</button>
<button type=”submit” id=”ctrl-change-address”>Change
Address</button>
<button type=”submit” id=”ctrl-change-basket”>Change
Basket</button>
```

Use specific targeted selector and SCSS to manage building required styles efficiently.

So we have reusable and maintainable styles (code).

Recap question - Why should we use `%placeholder` with `@extend`?



SCSS allows us to remove presentational markup and create reusable and maintainable styles.

Example

```
<button id="ctrl-send-order">Send</button>


<md-icon
   ↪ md-svg-icon="editor-format-list-bulleted">
</md-icon>
```

/styles/md/icons.scss

The real question isn't "how do I write good CSS?" its "how should we change CSS to make it harder to mess up?"

Leveraging SCSS

- Good HTML - No presentational markup
- Good CSS - DRY by using SCSS $variable, @mixin, @extend, %placeholder

How do I modularize and isolate this (harder to mess up) - for example if I want to change the icon color in a specific component?

Modular

```
<my-component>
  <md-icon
      ↪ md-svg-icon="editor-format-list-bulleted">
  </md-icon>
</my-component>

----- SCSS -----
my-component {
   md-icon { color: red; }
}
```

Modularity - we still need to be aware of specificity

- Wrapper component (element, ID, class - ideally use what is already available but watch specificity)
- SCSS to enforce selector hierarchy = specificity
- Single component specific styles.scss file
- Easy to debug

What does good CSS look like?

- Selectors that match exactly the elements you want without accidentally matching the elements you don't
- Selectors are not overly verbose or repetitive
- Selectors are resilient to change
- Selectors are adaptable to any and all future design requirements

Cool but if I have lots of components how do include all of these separate CSS files?

<demo>jspm</demo>

JavaScript Package Manager - jspm: http://jspm.io/

Package manager built on top of the ES6 module loader.

Import JS, HTML, and CSS, ES6 transpilation (Traceur / Babel), super-cool SFX bundle command.

CSS plugin: https://github.com/systemjs/plugin-css

More about client-side package management see: http://blog.npmjs.org/post/101775448305/npm-and-front-end-packaging

Demo using In Browser Design project main application toolbar in `/components/layout/layout.html`

Change the color of the hamburger menu icon to red:

```
&[md-svg-icon="navigation-menu"] {
 color: red;
}
```

Use DevTools to explore styles. Still hard to find element - can we improve this?

How about using custom wrapper element? Downside?

Remember specificity: https://developer.mozilla.org/en-US/docs/Web/CSS/Specificity

Inline style (in HTML style attributes) comes first, followed by ID selectors, then class selectors and eventually element-name selectors.

We have a higher specificity selector on body and then `@import "md/md"` (this was to ensure we can override the Material Design CSS).

Try to define rules only once for a certain selector, and group all rules belonging to that selector.

Selector Specificity

```
*               /* a=0 b=0 c=0 -> specificity = 0   */
li              /* a=0 b=0 c=1 -> specificity = 1   */
ul li           /* a=0 b=0 c=2 -> specificity = 2   */
ul ol+li        /* a=0 b=0 c=3 -> specificity = 3   */
h1 + *[rel=up]  /* a=0 b=1 c=1 -> specificity = 11  */
ul ol li.red    /* a=0 b=1 c=3 -> specificity = 13  */
li.red.level    /* a=0 b=2 c=1 -> specificity = 21  */
#x34y           /* a=1 b=0 c=0 -> specificity = 100 */
#s12:not(FOO)   /* a=1 b=0 c=1 -> specificity = 101 */
```

Ascending order of specificity - list of selectors is by increasing specificity:

- Universal selectors
- Type selectors
- Class selectors
- Attributes selectors
- Pseudo-classes
- ID selectors
- Inline style

A selector's specificity is calculated as follows:

- Count the number of ID selectors in the selector (= a)
- Count the number of class selectors, attributes selectors, and pseudo-classes in the selector (= b)
- Count the number of type selectors and pseudo-elements in the selector (= c)
- Ignore the universal selector

CSS3 specification: http://www.w3.org/TR/css3-selectors/#specificity

What have we achieved?

- Scope styles to a particular component
- Hidden implementation details
- Removed presentational markup
- Simplified CSS
- Modular code

What have we achieved?

- Scope styles to a particular component
- Hidden implementation details

Gunnar Bittersmann - CSS preprocessors for the best of both worlds (From the Front 2014)

https://speakerdeck.com/gunnarbittersmann/css-preprocessors-for-the-best-of-both-worlds-from-the-front-2014

While purists propagate to not mix content structure (HTML), presentation (CSS) and behavior (JavaScript) layers for the reason of maintainability.

Concepts like OOCSS aim at the same goal, but from a different angle: reusable CSS code, at the expense of bloated, presentational markup, violating the separation of concerns.

Can’t we have both, clean markup and reusable CSS?

The benefits of both approaches without the drawbacks?

Yes, we can. Take the OO concept out of the HTML and put it where it belongs in the style sheet. Not directly into the CSS though, but into an intermediate layer provided by a CSS preprocessor like SCSS/Sass.

Next...

Part 4 - Web Components

Johan Steenkamp / @johanstn

Next... Part 4 Web Components