CSS starter pack: Thoughts and ideas

One of the things I would have killed for when I came back to doing (serious) web design was a set of guidelines, stylesheets and examples of how to use responsive design and typography on the web. This is my attempt at creating such a guide. It's far from perfect but it's a starting point.

I don't hide the fact that I'm opinionated. There are reasons why I choose to do things a given way and I respect that you may disagree with me, that's your call.

SASS/SCSS versus CSS

The styles in this guide will be created using SASS 3.4 and the Ruby SASS interpreter. There is a C implementation of SASS (libsass) that is supposed to be on par with Ruby SASS. In my experience libsass does not work as well as the Ruby version so I decided it's ok to add the Ruby dependency to the project.

The CSS output of the SASS process will also be available and you can work from the CSS. Just remember that if you work from the CSS you are responsible for updating your stylesheets when changes happen or when you want to do things differently.

Using Normalize.scss

We need to make sure that our content looks the same in as many browsers as possible. Normalize.css and its SCSS version make a good job of smoothing the remaining differences in styles and their application across browsers. It has been imported in our `main.scss` stylesheet.

If you don't want Normalize as part of yout stylesheet, remove the statement `@import 'partials/normalize/import-now';` from `main.scss`.

Open Type classes

The project includes Utility OpenType from Kenneth Normandy. The classes in

Utility OpenType provide fine grained control over OpenType font features.

Not all OTF fonts offer all features. But it is still a good tools to have if you choose your fonts carefully .

using the Golden Ratio and modular scales

The first thing we'll do is define a modular scale to use in the content. I've chose to use modular scale by Scott Kellum and Tim Brown as a way to make it easier to create a scale that works across sizes and accross devices.

There is also a SASS plugin (will work with Compass as well as vanilla SASS). To install in vanilla SASS follow these instructions:

- Download the latest version modular scale plugin from Github
- Unzip the content to your project (I used the `src/sass` directory)
- In your main SCSS file `@import 'modular-scale/modular-scale`

For example, to define the font sizes of the 6 heading elements (h1 through h6), we could do something like this:

```
h1 {
  font-size: ms(4);
  line-height: 1.2;
}
h2 {
  font-size: ms(3);
  line-height: 1.1;
}
h3 {
  font-size: ms(2);
  line-height: 1.1;
}
h4 {
  font-size: ms(1);
  line-height: 1.1;
  text-transform: uppercase;
```

```
  word-spacing: .2em;
}
h5 {
  font-size: ms(1);
  line-height: 1.1;
  text-transform: uppercase;
}
h6 {
  font-size: ms(1);
  line-height: 1.1;
}
```

and the resulting CSS looks like this:

```
h1 {
  font-size: 6.8541em;
  line-height: 1.2;
}

h2 {
  font-size: 4.23607em;
  line-height: 1.1;
}

h3 {
  font-size: 2.61803em;
  line-height: 1.1;
}

h4 {
  font-size: 1.61803em;
  line-height: 1.1;
  text-transform: uppercase;
  word-spacing: .2em;
}

h5 {
```

```
  font-size: 1.61803em;
  line-height: 1.1;
  text-transform: uppercase;
}

h6 {
  font-size: 1.61803em;
  line-height: 1.1;
}
```

The one thing I don't like about the sizes we get using the Golden Ratio is that they are too big. The Modular Scale makes available different scales. If we use the change the scale to 15:16, also known as a minor second scale. The values of our headers will also change. To make these changes, add the following variables to your stylesheet, or to your variables file:

```
$ms-base: 1em;
$ms-ratio: 1.067;
```

In this scales the headings become harder to differentiate:

```
h1 {
  font-size: 1.29616em;
  line-height: 1.2;
}

h2 {
  font-size: 1.21477em;
  line-height: 1.1;
}

h3 {
  font-size: 1.13849em;
  line-height: 1.1;
}
```

```
h4 {
  font-size: 1.067em;
  line-height: 1.1;
  text-transform: uppercase;
  word-spacing: .2em;
}

h5 {
  font-size: 1.067em;
  line-height: 1.1;
  text-transform: uppercase;
  word-spacing: .2em;
}

h6 {
  font-size: 1.067em;
  line-height: 1.1;
}
```

We'll try one more scale, the major third scale (1.25). We'll change the variables to those below:

```
$ms-base: 1em;
$ms-ratio: 1.25;
```

And the resulting CSS

```
h1 {
  font-size: 3.8147em;
  line-height: 1.2;
}

h2 {
  font-size: 3.05176em;
  line-height: 1.1;
}
```

```
h3 {
  font-size: 2.44141em;
  line-height: 1.1;
}

h4 {
  font-size: 1.95313em;
  line-height: 1.1;
}

h5 {
  font-size: 1.5625em;
  line-height: 1.1;
}

h6 {
  font-size: 1.25em;
  line-height: 1.1;
}
```

There are other scales available. You can play with them to find the scale that suits your fonts, your design and your tastes the best

We'll pair the modular scale with media queries to change the sizes for smaller form factors.

Modular scale includes functions for a number of classic design and musical scale ratios. You can add your own ratios as well.

Function	Ratio	Decimal value
$phi	1:1.618	1.618
$golden	1:1.618	1.618
$double-octave	1:4	4
$major-twelfth	1:3	3
$major-eleventh	3:8	2.667

$major-tenth	2:5	2.5
$octave	1:2	2
$major-seventh	8:15	1.875
$minor-seventh	9:16	1.778
$major-sixth	3:5	1.667
$minor-sixth	5:8	1.6
$fifth	2:3	1.5
$augmented-fourth	1:√2	1.414
$fourth	3:4	1.333
$major-third	4:5	1.25
$minor-third	5:6	1.2
$major-second	8:9	1.125
$minor-second	15:16	1.067

Defining a list of variable to use

For a long time SASS was the only game in town when it came to using variables or variable-like constructs for your stylesheets. CSS caught up with a more dynamic concept of variables.

I love CSS variables but, for building a framework SASS variables will work best. If necessary we can convert some of the SASS variables discussed below into CSS variables to use after transformation.

Some variables to think about:

- Colors: We can create variables for colors in the Material Design Palette, for example, and then use SASS functions to create lighter and darker versions of each color without having to do the calculations manually.
- Default font sizes: Since we'll be changing some (if not all) font size attributes we may as well store them in variables. However we need to remember that we're using modular scales to generate the sizes

Fonts loading and use

Rather than let font loading remain atth e mercy of the network we'll use Font Face Observer to dynamically load the fonts and swap them when ready. This may cause a little user cognitive dissonance but it's better than the alternatives of browsers taking forever or not loading the font.

setting up fonts

We'll use the following mixin to create our `@font-face` declarations

```
@mixin font-declaration(  $font-family,
                          $font-file-name,
                          $weight:normal,
                          $style:normal) {
  @font-face {
    font-family: '#{$font-family}';
    src: '#{$font-family}'urlme}.eot');
    src: url('#{$font-file-name}.eot?#iefix') format('embedded-opentype')
    url('#{$font-file'#{$font-file-name}.eot?#iefix'url  url('#{$font-file
 '#{$font-file-name}.woff'urltf') format('truetype'),
      // working with m') format('url('#{$font-file-name}.ttf') format('tr
      // working with myFonts requires the svg file 'svg');'svg'// to be
      // url('#{$font-file-name}.svg#wf') format('svg');
    url('#{$font-file-name}.svg##{$font-family}') format('svg');
    font-weight: $weight;
    font-style: $style;
  }
}
```

And use them as follows:

```
@include font-declaration('notomon'notomono_regular'/fonts/notomono-regula
@include font-declaration('notosans_regular',
  '../fonts/notosans-regular-webfont', normal,  normal);
'notosans_regular'@include'notosans_bold',
```

```
  '../fonts/notosans-bold-webfont',  700,   normal);
@include font-decl'notosans_bold'@includec',
  '../fonts/notosans-italic-webfont',   normal, italic);
@include font-declaration('notosa'../fonts/notosans-italic-webfont'@inclu
  '../fonts/notosans-bolditalic-webfont', 700,  italic);
```

to create the`@font-face` declarations.

Note that we create 4 `@font-face` declarations for our notosans:

- One for regular
- One for bold
- One for italics
- One for bolditalics

We do this to prevent faux bold and faux italics that we'll discuss in more details later.

Loading fonts with Font Face Observer

Once we have our `@font-face` declarations done we need to actually load the fonts. We will use Font Face Observer to automate the process and to ensure that we don't get flashes of unstyled text.

The first thing to do is to modify our SCSS file to add a plain body tag with our backup fonts. We also add a `.fonts-loaded` class to the body declaration that uses the web fonts for the body tag.

We also need to add a `fonts-failed` class to take into account that the fonts may fail to load for a variety of reasons

```
/* Basic font stack*/
body {
  font-family: Verdana, sans-serif;
}
```

```
/* Font stack when fonts are loaded */
.fonts-loaded body {
  font-family: "notosans_regular", Verdana, sans-serif;
}

"notosans_regular"/* Same font stack as basic but for when font loading fa
.fonts-failed body {
  font-family: Verdana, sans-serif;
}
```

In Javascript we'll create add a script tag linking to `fontfaceobserver.js`

```
<script src="fontfaceobserver.js"></script>
```

To load a single font we create variables to hold a `FontFaceObserve` with the name of the fonts we defined in the `@font-face declaration` and a reference to `document.documentElement`.

We add a class to the document to indicate tht fonts are loading.

We load the font and that creates a promise. If the promise succeeds then we remove the `font-loading` class and replace it with `fonts-loaded`.

If the font fails to load the promise will reject and jump to the catch statement. This will replace the `font-loading` class with `font-failed`. We can use this failed class to create a backup font stack to use when web fonts are not available.

```
var font = new FontFaceObserver('notosans-regular');
var html = document.documentElement;

html.classList.add('fonts-loading');

font.load().then(() => {
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-loaded');
}).catch(() => {
  html.classList.remove('fonts-loading');
```

```
  html.classList.add('fonts-failed');
});
```

We can work loading multiple fonts simultaneously by defining multiple `FontFaceObserver` declaration and then using `promise.all` to make sure that all the fonts are loaded before proceeding to change the classes as appropriate depending on success or failure.

```
var mono = new FontFaceObserver('notomono_regular');
var sans = new FontFaceObserver('notosans_regular');
var italic = new FontFaceObserver('notosans_italic');
var bold = new FontFaceObserver('notosans_bold');
var bolditalic = new FontFaceObserver('notosans_bolditalic');

var html = document.documentElement;

html.classList.add('fonts-loading');

Promise.all([
  mono.load(), sans.load(), bold.load(), italic.load(), bolditalic.load()
]).then(() => {
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-loaded');
  console.log('All fonts have loaded.');
}).catch(() =>{
  html.classList.remove('fonts-loading');
  html.classList.add('fonts-failed');
  console.log('One or more fonts failed to load')
});
```

One important note. In this example we've chosen an all or nothing approach. Either all fonts load successfully or we'll use none of them. I want to make sure that if we use web fonts we remain consistent in the usage. It would look ugly if we were to mix and match the web font stack with the backup.

Preventing faux fonts

In all our previous examples we've loaded 4 fonts for our Noto Sans font. We do this to prevent faux italic and faux fonts. When the browser finds italic or bold fonts it'll look to see if if a corresponding font (for example notosans_bold) is available either as a system font or as a loaded web font.

If it is then the browser will use it and everything is fine. If it's not available the browser will synthesize an equivalent to the font we need.

Needless to say synthetic fonts are not the best for any serious typographic work. They may look different in each browser and they generally look different than the main font.

To prevent any of these problems we associate the font with the correct font (bold, italic or bold italic) to make sure that the content is displayed with the correct font.

```
strong, b {
  font-family: "noto_san"noto_sansbold"em, i font-family: "noto_italic";
}

stro"noto_italic"strong em,
strong, i,
b em,
b i {
  font-family: "noto_sansbold_italic";
}
```

Taking advantage of SASS to play with colors

I'm bad at remembering headecimal, rgba or hsl colors. These fuctions will help convert the hexadecimal values

```
$base-color: #AD141E;
```

Darken & Lighten These two adjust the Lightness of the color's HSL values. Sass will parse our hex color variable to hsl, then make the adjustments. You call them on the color with a percentage value between 0 and 100. I usually stay in the range of 3-20%.

```
darken( $base-color, 10% )
lighten( $base-color, 10% )
```

Saturate, & Desaturate

These will will adjust the Saturation of the colors HSL values, much like Darken and Lighten adjusted the Lightness. Again, you need to give a percentage value to saturate and desaturate.

```
saturate( $base-color, 20% )
desaturate( $base-color, 20% )
```

Adjust-hue

This adjusts the hue value of HSL the same way all of the others do. Again, it takes a percentage value for the change.

```
adjust-hue( $base-color, 20% )
```

Adding Alpha Transparency

Using our hex color we can do a few things to get it to be a little transparent. I stick to rgba as it comes most naturally to me which takes a color and a value from 0 to 1 for the alpha.

```
rgba( $base-color, .7 )
```

With these techniques we can build color palettes for our base colors. for example:

```
$red-base: #AD141E;
$red-l10: lighten( $red-base, 10%);
$red-l20: lighten( $red-base, 20%);
$red-l30: lighten( $red-base, 30%);
$red-l40: lighten( $red-base, 40%);
$red-l50: lighten( $red-base, 50%);
$red-d10: darken( $red-base, 10%);
$red-d20: darken( $red-base, 20%);
$red-d30: darken( $red-base, 30%);
$red-d40: darken( $red-base, 40%);
$red-d50: darken( $red-base, 50%);
```

Using `$red-base` we've used SASS to create 5 progressively lighter and 5 progressively darker shades of the color without having to learn what the hexadecimal values are for each of the 10 colors we created automatically.

We can repeat the process for each of the colors that we want to work with. We can take the colors from The New Defaults or a palete from Material Paletteand build a 5 up and 5 down scale for each color.

Color manipulation functions can also be used directly in selectors. The example below show some ways we can use the color functions inside selectors.

```
.example {
  border: 1px solid darken($base-color, 20%);
  text-shadow: 0 -1px 0 darken($base-color, 10%);
  box-shadow: 10px 5px 5px lighten($base-color, 20%));
}
```

The stylesheets provide the following list of colors along with their names:

```
//Greens
$turqoise: #1abc9c;
$green-sea: #16a085;
$emerald: #2ecc71;
$nephritis: #27ae60;
$green: #4caf50;
$light-green: #8bc34a;
$lime: #cddc39;
//Blues
$river: #3498db;
$belize: #2980b9;
$asphalt: #34495e;
$midnight-blue: #2c3e50;
$blue: #2196f3;
$light-blue: #03a9f4;
$cyan: #00bcd4;
$teal: #009688;
//Reds
$alizarin: #e74c3c;
$pomegranate: #c0392b;
$red: #f44336;
//Oranges
$carrot: #e67e22;
```

```
$pumpkin: #d35400;
$dull-orange: #f39c12;
$orange: #ff9800;
$blood-orange: #ff5722;
$amber: #ffc107;
//Yellows
$sunflower: #f1c40f;
$yellow: #ffeb3b;
//Purples + Pinks
$rebecca-purple: #663399;
$amethyst: #9b59b6;
$plum: #8e44ad;
$purple: #9c27b0;
$deep-purple: #673ab7;
$pink: #e91e63;
$indigo: #3f51b5;
//Browns
$brown: #795548;
//Greys
$grey: #9e9e9e;
$gun-metal: #607d8b;
$asbestos: #7f8c8d;
$concrete: #95a5a6;
$silver: #bdc3c7;
//Whites
$clouds: #ecf0f1;
$paper: #efefef;
```

Media queries to control typography and layout

Media queries are awesome. They allow developers (us) to change elements of the design based on arbitrary criteria…. We finally can customize our designs to fit on the target devices rather than the other way around.

If we were to create a query for every device in the market we would be stuck in the process for a long, long time. Instead we'll build queries for device sizes using the following mixin. It's broken in 2 sections:

- A nested map containing this information:
    - The names of the breakpoint
    - The minimum width
    - The maximum width
    - The DPI resolution
- A function that checks if the specified breakpoint exists in the nested table and, if it does, builds a query with the information for that breakpoint

```
@mixin media-queries( $qtype ) {
  $value: screen;
  $breakpoints: (
    // smartphones, portrait iPhone, portrait 480x320 phones (Android)
    ( phone-portrait,     20em,   30em,   1),
    // smartphones, portrait iPhone, portrait 480x320 phones (Android)
    ( phone-portrait-r,   20em,   30em,   2),
    // smartphones, Android phones, landscape iPhone
    ( phone-landscape,    30em,   37.5em, 1 ),
    // smartphones, Android phones, landscape iPhone
    ( phone-landscape-r,  30em,   37.5em, 2 ),
    // portrait tablets, portrait iPad, e-readers (Nook/Kindle),
    // landscape 800x480 phones (Android)
    ( table-portrait,     37.5em, 50em,   1 ),
    // portrait tablets, portrait iPad, e-readers (Nook/Kindle),
    // landscape 800x480 phones (Android)
    ( table-portrait-r,   37.5em, 50em,   2 ),
    // tablet, landscape iPad, lo-res laptops ands desktops
```

```
    ( table-landscape,    50em,   64em,   1),
    // tablet, landscape iPad, lo-res laptops ands desktops
    ( table-landscape-r,  50em,   64em,   2),
    // big landscape tablets, laptops, and desktops
    ( large,              50em,   80em,   1),
    // big landscape tablets, laptops, and desktops
    ( large-r,            50em,   80em,   2),
    // hi-res laptops and desktops
    ( xlarge,             80em,   100em,  1),
    // hi-res laptops and desktops
    ( xlarge-r,           80em,   100em,  2)
  );

  @each $item in $breakpoints {
    @if $qtype == nth($item, 1) {

    @media #{$value} and (min-width: nth($item, 2))
      and (max-width: nth($item, 3))
      and (-webkit-min-device-pixel-ratio: nth($item, 4)) {
        @content;
      }
    }
  }
}
```

We can then use the name for a specific family of devices, for example large (desktops) like this (using modular scale to calculate the size of the h1 element):

```
@include media-queries( large ) {
  h1 {
    font-size: ms(4);
  }
}
```

Which produces the following result:

```
@media  screen and (min-width: 50em) and (max-width: 80em)
        and (-webkit-min-device-pixel-ratio: 1) {
  h1 {
    font-size: 2.44141em;
  }
}
```

The `@content` directive allow us to add data inside the mixin.

Because we've named our queries with devcie type and orientation the media queries will be easier to read and understand. Someone coming to our code should have no problem reasoning what devices the queries apply to.

TODO: Test if we can chain queries using our existing media-queries Mixin and what would it take to change it if it doesn't

Layouts: The Next Generation

Rather than work with floats and positioned content we'll concentrate on 3 new(ish) layout technologies: Multi column, flexbox and grid

General considerations

The table below, taken from Responsive Typography by Jason Pamental (O'Reilly) gives an idea of the different devices sizes and the elements sizes as related to them. We've also put the equivalent print sizes for comparison.

		Print	Desktop (xl)	Desktop	Tablet	Phone
Body	Font Size	12pt	16px (1em)	16px (1em)	16px (1em)	16px (1em)
	Line Height	1.25	1.375	1.375	1.375	1.375
	Line Length	60 - 75	60 - 75	60 - 75	60 - 75	60 - 75
H1	Font Size	36pt (3em)	48px (3em)	48px (3em)	40px (2.5em)	32px (2em)
	Line Height	1.25	1.05	1.05	1.125	1.25
H2	Font Size	24pt (2em)	36px (2.25em)	36px (2.25em)	32px (2em)	26px (1.625em)
	Line Height	1.25	1.25	1.25	1.25	1.15384615
H3	Font Size	18pt (1.5em)	28px (1.75em)	28px (1.75em)	24px (1.5em(	22px (1.375em)
	Line Height	1.25	1.25	1.25	1.25	1.13636364
H4	Font Size	14pt (1.667em)	18px (1.125em)	18px (1.125em)	18px (1.125em)	18px (1.125em)

	Line Height	1.25	1.25	1.25	1.222222	1.111111
Blockquote	Font Size	24pt (2em)	24px (1.5em)	24px (1.5em)	24px (1.5em)	20px (1.25em)
	Line Height	1.458333	1.458333	1.458333	1.458333	1.25

The length of the line and the size of the font and its line height will affect the look of the layout and the layout will affect the size of the font and its line height. This is important to keep in mind as we look at these new layout options.

Multi Columns

We've been able to do multi column layouts for a while now but it's only been recently, when I've been pushed to look at layouts beyond the holy grail that I've been able to look at columns and their limitations in modern browsers.

The SCSS mixin below allows us to create multiple columns (2 by default) with a 16px gap between columns, attempting to balance the content among the columns. This will create the number of columns specified, regardless of available space.

```
@mixin column-attribs ($cols: 2, $gap: 1em, $fill: balance, $span: none){
  // How many columns?
  column-count: $cols;
  // Space between columns
  column-gap: $gap;
  // How do we fill the content of our columns, default is to balance */
  column-fill: $fill;
  // Column span, defaul is not to span columns */
  column-span: $span;
}
```

Columns are a prime candidate to use autoprefixer. I have removed the prefixed attribute in my working code and will only prefix the attributes as needed in the build phase. It is important that you test this thoroughly in your target browsers.

We first import the mixin into our master style sheet. Then we call it using

syntax like this:

```
/* Column predefined classes */
.cols-2 {
  @include column-attribs(2, 2em);
}

.cols-3 {
  @include column-attribs(3);
}
```

The first class will create two column text with 2em between columns. This will take all available space which may look bad in larger displays.

The second class will create a three column box using all the available screen space. This may not look as intended on phones or other small form factors. Again, test in your target devices and consider using media queries to adjust the layout to fewer or no columns in those smaller devices.

There is an attribute that uses specific width as the basis for the columns. We can modify the `column-attribs` mixin to use columns of a specific width. I will take out the prefixed versions of the values to avoid confusion. They will be prefixed at build time.

`fixed-column-attribs` uses the `column-width` attribute to take a fixed width value (default is 15em) and create as many columns as can possibly fit on screen with a 1em gap between them.

```
@mixin fixed-column-attribs ($col-width: 15em,
                             $gap: 1em, $fill: balance, $span: none){
  // How many columns?
  column-width: $col-width;
  // Space between columns
  column-gap: $gap;
  // How do we fill the content of our columns, default is to balance */
  column-fill: $fill;
  // Column span, defaul is not to span columns */
  column-span: $span;
```

```
}
```

These are the basics, we're not covering all of the multi column specifications. In particular we're not covering how to do column separators between columns.

Flexbox

Flexbox is one of the new CSS modules finalized recently. It has a long and checkered story. There are 3 different implementations for browsers throughout its history. We wil only work with the standard as specified

We define 4 different mixins for flexbox declarations. The forward mixins work by creating a flex layout that goes "with the grain" of the layout (left to right in rows) and top to bottom in columns) where the back mixins go against the grain in the reverse direction.

The SCSS mixins look like this:

```
@mixin flex_row_forward() {
  display: flex;
  flex-flow: row wrap;
}

@mixin flex_row_back() {
  display: flex;
  flex-flow: row-reverse wrap;
}

@mixin flex_column_forward() {
  display: flex;
  flex-flow: column wrap;
}

@mixin flex_column_back() {
  display: flex;
  flex-flow: column-reverse wrap;
}
```

```
    @mixin flex_item() {
      display: flex;
      flex: 1 1 auto;
    }
```

We then use the mixins to generate classes. The idea is to provide basic flexbox functionality without loosing the flexibility of adding additional functionality. The classes look like this:

```
.flexbox-row-forward {
  @include flex_row_forward();
}

.flexbox-row-back {
  @include flex_row_back();
}

.flexbox-col-forward {
  @include flex_column_forward();
}

.flexbox-col-back {
  @include flex-column_back();
}

.flex-item {
  @include flex_item();
}
```

The class definitions produce the following CSS. We can then add additional rules to the class selectors depending on what additional needs our project has.

```
    .flexbox-row-forward {
      display: flex;
      flex-flow: row wrap;
```

```
}

.flexbox-row-back {
  display: flex;
  flex-flow: row-reverse wrap;
}

.flexbox-col-forward {
  display: flex;
  flex-flow: column wrap;
}

.flexbox-col-back {
  display: flex;
  flex-flow: column-reverse wrap;
}

.flex-item {
  display: flex;
  flex: 1 0 auto;
}
```

Grids

I'm really excited about grids. They provide a native alternative to systems like 960.gs, Foundation and Bootstrap when it comes to laying out the content of our sites.

I consider grids part of a bespoke design process that is harder to learn than what we've covered so far. Rather than provide full templates I will work through an example layout matching the image below, as close as possible.

We'll use the following mixins to generate the grid and place items on it.

```
@mixin generate-grid($columns: 12, $gap: 10px) {
```

```
  display: grid;
  grid-gap: #{$gap};
  grid-template-columns: repeat(#{$columns}, 1fr);
  grid-template-rows: 1fr;
}
```

The `generate-grid` mixin will create the grid itself. By default it'll create a 12 equal columns and add 10 pixel gaps between columns. The idea is that we can create flexible grids bigger or smaller than 12 columns depending on your design needs and how you want to position the content.

Each row will take 1 fraction of the height of the page.

The `generate-grid` mixin is not responsive by default. To make it responsive we need to modify it slighty to provide both minimum and maximum for each column.

```
@mixin generate-fluid-grid($min-size: 300px, $max-size: 1fr, $gap: 10px) {
  display: grid;
  grid-gap: #{$gap};
  grid-template-columns: repeat(auto-fill, #{$min-size}, #{$max-size});
  grid-template-rows: 1fr;
}
```

`generate-fluid-grid` will create as many columns that are at least `min-size` (300 pixels by default) and no larger than `max-size` (1fr by default). This is similar on how we work with flexbox.

The `place-item` mixin is used to place an element in a specific position.

```
@mixin place-item($col_start: 1, $col_end: 1, $row_start: 1, $row_end: 1) {
  grid-column: #{$col_start} / #{$col_end};
  grid-row: #{$row_start} / #{$row_end};
}
```

This will place content on the grid using column/rown placement. Essentially, the mixin will tell the browser the column start/end and row start/end position. We

can then add additional CSS attributes to the item we're placing.

For Grid I will insist on using Feature Queries to make sure that browsers support the feature before going crazy with it. It would look something like this:

```
@supports (display: grid) {
  /* Put all the rules to work with grids here */
}
```

and use it like this to generate a 12 column grid with a 10 pixel gap between columns using the mixin's default values:

```
@supports (display: grid) {

    .wrapper {
       @include generate-grid();
    }
}
```

we'll place the item indicated by class box1with the following data

- Column start (1)
- Column end (2)
- Row Start (1)
- Row End (3)

```
.box1 {
  @include place-item(1, 2, 1, 3);
}
```

This will produce the following CSS. Note that I've added a general boxes class to style all boxes at the same time with color (and size) and other generic attributes.

```
    .grid-wrapper {
      display: grid;
      grid-gap: 10px;
```

```
    grid-template-columns: repeat(15, 1fr);
    grid-template-rows: 1fr;
  }

  .boxes {
    background-color: #663399;
  }

  .box1 {
    grid-area: 1 / 2;
    grid-row: 1 / 3;
  }
```

Now back to our example layout. We'll create a media query to test for devices larger than 760px and create a 12 column grid.

I'm really excited to get (native) grids to browsers. It opens some seriously cool layout possibilities and my dream of doing magazine style layouts of the web become a real posibility.

We can create asymetrical column layouts. Our columns can have different widths and different heights and we can let the layout dictate the way we code and style our documents.

It is also possible to mix layouts using Flexbox, 3D work, transitions and animations and other web and CSS technologies with Grid. For example you can create a masthead using Flexbox and place that mastead in the top row of a Grid and place additional content below the masthead.

Or we could create a magazine like spread where the images overlap the text or viceversa and where we use columns to layout the content.

They may not match 100% the source material but will be much closer to what we've dared do until now.

Still missing: regions

This one is a gripe. We don't have a way to tie different portions of our layout. If you've worked on with inDesign this is equivalent to threading frames.

A few years ago there were proposals for CSS Regions. Because of a desire for technical purity from Håkon Wium Lie and performance issues from David Baron the proposal along with some early implementations in WebKit and Blink are not in active development (and the implementation Adobe contributed to WebKit was removed from Blink due to performance reasons) but it meant that a very promising feature is now missing from the web.

Imagine, if you will, a design where content is placed in different blocks and, in an ideal world, they would flow from top left and into other boxes until either all boxes are filled or the content has been placed.

There are new proposals for region-like functionality. Stay tuned

Final notes

Making and Breaking the Grid and *The Elements of Typographic Style* are two of my favorite books on design and, for the longest time, I thought about them only in their applications to books. Sitting at a full-day workshop with Jen Simmons about grids it has opened mind and reminded me of what can we do to expand the reach of the web. I will discuss some of them as a way to wrap this up.

What is exclusive to the web

We need to research and discover what is it that makes the web unique and different from other media. The web is not print (don't agree with this) has been a mantra from web designers and developers for as long as I remember. But I don't think that I've ever heard any of these people articulate why?

New layouts and technologies make the web look and work closer to print but it's different and interactive enough that we can turn these layouts into fully interactive experences.

How do we leverage the viewport

Regardless of how long a web page is, users only see a small section of the site at a given time. The size of the screen dictates how much of our content we can view at any given time.

How do we leverage this viewport for our content? I'm still trying to figure it out but wanted to throw it out there

Are we story boarding? Why not? Story boards work particularly well with some of the technologies we discuss here. Animators have always done in a storyboard first, storyboards work particularly way to layout position of content in a grid and how elements interact with each other.

Framing the content (film) / Blocking and working on interactions (theatre) Both film and theater frame the content, block the movements of actors and cameras and how the characters interact with each other.

Because we are we looking at our content through a viewport we need to be careful how different areas of the page show and interacts with the user.

Find your sources of inspiration Because we are no longer limited to the

3-column holy grail layout it's OK to look at other sources of inspiration. There is no reason why we can't look at print and see how closely we can match their designs with our technlogies.

Sure we may not be able to match print designs 100% on a feature by feature basis but we are much closer now than we've ever been. Let's experiment and play and discover all the possibilities that we have.

Everything is CRAP

One of the things I learned at An Event Apart is how much of the graphic design cannon we can leverage on our (web) designs. I was in Grad School for Instructional Technology when I first heard about CRAP and it has guided my design and my work outside design ever since.

The four components of CRAP:

Contrast If the elements (type, color, size, line thickness, shape, space, etc.) are not the same, then make them very different. Contrast is often the most important visual attraction on a page.

- Can you see the difference between your content, ads, headings, body copy and comments?

Repetition Repeat visual elements of the design throughout the piece. You can repeat color, shape, texture, spatial relationships, line thicknesses, sizes, etc.

- Do you have a consistent theme or brand throughout your site? Do you reuse the same colour, shapes, blockquotes, formatting for all of your articles?

Alignment Nothing should be placed on the page arbitrarily. Every element should have some visual connection with another element on the page. This creates a clean, sophisticated, fresh took,

- Does everything line up or have you got things centred, left aligned or out of place?

Proximity Items relating to each other should be grouped close together. When several items are in close proximity to each other, they become one visual unit rather than several separate units. This helps organize information and reduces clutter.

- Can you find everything you need on your page easily? What is it that your visitors are looking for?

(CRAP definition taken from http://www.dailyblogtips.com/crapthe-four-principles-of-sound-design/)

Links, Resources, References

- Grids
  - labs.jensimmons.net
  - rachelandrew.co.uk
- Modular Scale
  - Modular Scale
  - More Meaningful Typography -- Tim Brown, A List Apart
  - Web Font Loading Patterns -- Bram Stein
  - Molten Leading
  - Molten leading (or, fluid line-height) -- Tim Brown, Nice Web Type
- Viewport Sized Typography
  - Viewport Sized Typography -- CSS Trics
  - Viewport sized typography with minimum and maximum sizes -- Eduardo Boucas
  - FitText
  - CSS Viewport Units: vw, vh, vmin and vmax -- Chris Mills, Dev.Opera
  - Fluid Type
  - Precise control over responsive typography -- Mike Riethmuller
  - Font Face Observer -- Github
- Color
  - The color thesaurus -- Ingrid Sundberg
  - The New Defaults -- Dudley Storey
  - Material Design: Color -- Google
  - Material Palette
  - Materialize: Color
- Technical SASS/SCSS
  - All you ever need to know about SASS interpolation
  - Using Nested Sass Maps for TypeSetting
  - Using Sass Maps
  - Media Queries for Standard Devices
  - SASS: @content directive is a wonderful thing