Evolving Complexity: Design

I've always believed that there is no reason not to make our designs look like print. What makes a good print design (and an acceptable web equivalent) has changed over time. What looked good in 1994:

May not look as good as a modern site or application.

Just like our authoring tools and technologies, the design and layout techniques changed based on what we had available at the time.

This is what a table-based layout in the mid to late 1990's looked like. It used tables for layout and 1x1 spacer gif images.

```
<TABLE>
  <TR>
    <TD><IMG SRC="1x1.gif" WIDTH=300>
    <TD><FONT SIZE=42>Hello welcome to my <MARQUEE>Internet Web Home</MARQ
  </TR>
  <TR>
    <TD BGCOLOR=RED><IMG SRC="/cgi/webcounter.cgi">
  </TR>
</TABLE>
```

Around this time CSS came out and the big debate on whether we should design with tables or using CSS. I find it funny that, as late as 2009, there was still some discussion about it along with pros and cons for each option.

The next stage was designing with floats. We use margins and explicit sizing for elements in CSS. In this example we set the nav element is 200 pixels wide and floated to the left; the `section` element is placed 200 pixels from the left margin and, since it doesn't have a `float`attribute, it'll float after the element floated to the left.

```
nav {
```

```
  float: left;
  width: 200px;
}
section {
  margin-left: 200px;
}
```

When we apply the CSS to the HTML below we get a navigation section displayed on the left side and all the content, represented by the `section` elements, displayed to the right of the navigation.

```
<div class="clearfix">
  <nav>
    <ul>
      <li>
        <a href="float-layout.html">Home</a>
      </li>
      <li>
        <a href="float-layout.html">Taco Menu</a>
      </li>
      <li>
        <a href="float-layout.html">Draft List</a>
      </li>
      <li>
        <a href="float-layout.html">Hours</a>
      </li>
      <li>
        <a href="float-layout.html">Directions</a>
      </li>
      <li>
        <a href="float-layout.html">Contact</a>
      </li>
    </ul>
  </nav>

  <div>
    <p>
```

```
      This example works just like the last one. Notice we
      put a <code>clearfix</code> on the container. It's not
      needed in this example, but it would be if the
      <code>nav</code> was longer than the non-floated content.
    </p>
  </div>

  <div>
    <p>
      Lorem ipsum dolor sit amet, consectetur adipiscing elit.
      Phasellus imperdiet, nulla et dictum interdum, nisi lorem
      egestas  odio, vitae scelerisque enim ligula venenatis
      dolor. Maecenas nisl est, ultrices nec congue eget, auctor
       vitae massa.
    </p>
  </div>
</div>
```

Float layouts are not responsive. 200 pixels are two hundred pixels regardless of the device you're in. It also gets more complicated the more elements we have in our layout. That's where frameworks like Twitter Bootstrap and Zurb Foundation to appear between 2010 and 2011.

The earliest examples I've been able to find are for version 2 of the frameworks: Foundation for Sites 2.2.1 and Bootstrap 2.0.4

Both frameworks provided a 12-column grid that you could section in as many columns as you wanted. You had also to specify how many rows you wanted and each row could have a different number of columns.

In this section we're only concerning ourselves with the layout. Both Bootstrap and Foundation provide Javascript libraries for basic site layout elements like acordions and similar that required Javascript to function properly.

The first example uses the syntax provided by Foundation for Sites:

```
<div class="row display">
  <div class="four columns">
```

```
    .four.columns
  </div>
  <div class="four columns">
    .four.columns
  </div>
  <div class="four columns">
    .four.columns
  </div>
</div>
<div class="row display">
  <div class="six columns">
    .six.columns
  </div>
  <div class="six columns">
    .six.columns
  </div>
</div>
<div class="row display">
  <div class="twelve columns">
    .twelve.columns
  </div>
</div>
```

And the equivalent markup for Bootstrap:

```
<div class="row show-grid">
  <div class="span4">4</div>
  <div class="span4">4</div>
  <div class="span4">4</div>
</div>
<div class="row show-grid">
  <div class="span4">4</div>
  <div class="span8">8</div>
</div>
<div class="row show-grid">
  <div class="span6">6</div>
  <div class="span6">6</div>
```

```
</div>
<div class="row show-grid">
  <div class="span12">12</div>
</div>
```

We'll revisit the frameworks to see how they evolve in the Responsive Web landscape.

The term "Responsive Web Design" was coined by Ethan Marcotte in 2010 to represent the combination of Fluid Grids, Responsive Images, and Media Queries. These combined techniques reduce the need for specific sites or sub domains for mobile versus desktops.

This approach required changing the tools (if were not already using them) but it also required a change of thinking about design and about what was your site designed for. In his A List Apart Article Marcotte states that:

> Fluid grids, flexible images, and media queries are the three technical ingredients for responsive web design, but it also requires a different way of thinking. Rather than quarantining our content into disparate, device-specific experiences, we can use media queries to progressively enhance our work within different viewing contexts. That's not to say there isn't a business case for separate sites geared toward specific devices; for example, if the user goals for your mobile site are more limited in scope than its desktop equivalent, then serving different content to each might be the best approach.

Now that we've gotten an idea of what Responsive design is, let's look at the components.

A fluid grid uses relative units (`em` in this case) based on a fixed root element size. The comments for the rules show the calculation used to get the number.

```
#page {
  margin: 40px auto;
  padding: 0 1em;
  max-width: 61.75em; /* 988px / 16px = 61.75em */
```

```
}

h1 {
  margin-left: 14.575%; /* 144px / 988px = 0.14575 */
  width: 70.85%; /* 700px / 988px = 0.7085 */
}

.entry {
  float: left;
  width: 100%;
}

.entry h2,
.entry .content {
  float: right;
  width: 85.425%; /* 844px / 988px = 0.85425 */
}

.entry .info {
  float: left;
  margin-top: 0.72727em; /* 8px / 11px = 0.72727em */
  width: 12.551%; /* 124px / 988px = .12551 */
}
```

The result can be seen in this page.

Fluid images are different than the responsive images we've discused before. At their simplest they are images sized using percentages for the full width of the image.

```
<img src="motorcycle.jpg" style="max-width:100%;height:auto;"
alt="Motorcycle">
```

You can also size images relative to the overall page width.

For example, let's say you had an image that had a natural size of 500px × 300px in a 1200px wide document. Below 1200px, the document will be fluid. The calculation of how much width the image takes up as a percentage of the

document is easy:

```
(500 / 1200 ) × 100 = 41.66%
```

```
<img src="mount-mayon.jpg" style="float: right; width: 41.66%;"
alt="A photograph of Mount Mayon, Phillipines">
```

Dudley Storey's The New Code has an article on CSS Fluid Image Techniques for Responsive Site Design

The final element of our responsive web design is Media Queries. These are a CSS feature that lets you adjust or flat out change CSS selectors and rules based on criteria you choose.

In this example, taken from http://cssmediaqueries.com/ we can see two different media queries in action.

- The first query is triggeerd when the width of the screen is 1200px or wider. If true we swap the original image with a larger version.
- The second query only triggers when we print the page. We then remove the image altogether to save on toner or ink.

The idea is that we can tailor the content for the devices we are targeting without having to write entire sites dedicated to each class of devices we're targeting.

```
/* normal style */
#header-image {
  background-repeat: no-repeat;
  background-image: url('image.png');
}

/* show a larger image when you're on a big screen */
@media screen and (min-width: 1200px) {
  #header-image {
    background-image: url('large-image.png');
  }
}
```

```
/* remove header image when printing. */
@media print {
  #header-image {
    display: none;
  }
}
```

We are getting closer to the present. Thanks to the CSS working group and the willingness of browser vendors to work towards uniform specification support we can see the CSS we write become, slightly, easier.

CSS Variables allow you to create custom reusable elements directly in CSS. In this example the :root defines a `--main-color` variable.

We can reuse the variable anywhere in the style sheet. In this case we use it in the h1 element with the var() syntax.

```
:root {
  --main-color: #06c;
}

#foo h1 {
  color: var(--main-color);
}
```

We can use CSS variables to create color themes and a set of breakpoints. We define them in the `:root` element and then use them throughout the style sheet, without having to use SASS or Less or Stylus.

Flexbox is a single dimension layout tool that allows you to create layouts that are responsive by default.

The CSS defines 3 selectors:

- `.boxes` is the container for the flex elements. All its children automatically become flex items
- `.box` is the individual flex item. The most important rule is the width. This is what the wrapping will be based on
- The `img` element inside the boxes is set up as a fluid image taking 100% of

the parent's width

```
.boxes {
  padding: 0.5vw;
  flex-flow: row wrap;
  display: flex;
}

.box {
  margin: 0.5vw;
  border: 1px solid #444;
  padding: 0.5vw;
  flex: 1 0 auto;
  width: 300px;
}
.box img {
  width: 100%;
  height: auto;
}
```

The HTML uses the element we defined and it adds an extra class to each box so we can style it independently. We might want to add individual backgrounds based on position or make other changes to individual blocks of content.

```
<h1>Example Flexbox Gallery</h1>

<div class="boxes">
  <div class="box box1">
    <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/32795/love-the-
looking balloon" />
    <h3>The architecture rocks</h3>
    <p> </p>
  </div>
  <div class="box box2">
    <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/32795/sunset-10
    <h3>Amsterdam Sunset</h3>
    <p> ... </p>
  </div>
```

```
  <div class="box box3">
    <img src="https://s3-us-west-2.amazonaws.com/s.cdpn.io/32795/pdx-signp
Balloon" />
    <h3>Portland Signpost</h3>
    <p> ... </p>
  </div>
</div>
```

You make grids vertical or horizontal and you can nest them indefinitely. Using media queries you can change them from horizontal to vertical if necessary.

CSS Grid provides 2-dimensional layouts beyond what's possible with Flexbox alone. It replaces the old old frameworks' float layouts and provides a native alternative to the new responsive layouts that Foundation, Bootstrap and Other frameworks have made available in recent versions.

The CSS builds a 3 column 100 pixels per column layout with a gap (vertical and horizontal) of 10 pixels. We dson't explictly create rows, the placement algorythm will take care of that autommatically.

```
.wrapper {
  display: grid;
  grid-template-columns: 100px 100px 100px;
  grid-gap: 10px;
  background-color: #fff;
  color: #444;
}

.box {
  background-color: #444;
  border: 1px solid black;
  border-radius: 5px;
  color: #fff;
  padding: 20px;
  font-size: 150%;
}
```

As with the Flexbox we've added an additional class to the child items so we can

style them individually if we need to.

```
<div class="wrapper">
  <div class="box a">A</div>
  <div class="box b">B</div>
  <div class="box c">C</div>
  <div class="box d">D</div>
  <div class="box e">E</div>
  <div class="box f">F</div>
</div>
```

We can combine flexbox and grid in the same layout or we can use them for whatever they are best suited for.

RWD is still a thing but the more we play with the technologies available to us now we can get as close as we can to print layouts; sure there are some things that are missing like fragmentation to create regions for different pieces of text, but we're closer now than we've ever been.

Responsive design is great but it comes with a complexity price attached to it. Ethan released RWD to the world at the dawn of the mobile flood when desktop was still the predominant target for the web. Now we have to worry about pixel density and how will how images look in the newest 4x Retina display, we have many more combinations of tablets, phones, landscape versus portrait, and many more screen sizes across the spectrum.

Working towards complex layouts on the web

For the longest time we said that the web is not print and we shouldn't try to duplicate print layouts on the browser. Until not too long ago the tools were not there to even try, but that has changed with grid, flexbox, shapes and writing modes. We can do some very good (but incomplete) approximations of print layouts on the web.

The work of Jen Simmons, particularly her 2016 workshop demos and Layout Land video series have rekindled my love with experimenting with/in/on the web. She has created very complex layouts in her experimental layout lab.

If we work from a Progressive Enhancement paradigm we can work towards complex layouts on the web without having to use frameworks and libraries.

If the code defensively we can provide a baseline experience for all users and a better experience for the browsers that support the technologies we need to give the experience to them. For example, if we want to work with grid and ensure that the non-supporting browsers have a good experience we can work on something like this:

```
@supports (display: grid) {
  div {
    display: grid;
  }
}

@supports not (display: grid) {
  div {
    float: right;
  }
}
```

You can further customize content placement and layout with Media Queries and `@support`. You can be as detailed as your layout needs you to be.

This doesn't add complexity but adds resilience to our code by explicitly deciding what will happen if the feature we're working with is not supported.

Another part of the design process is accepting that content will not look the same everywhere and that you don't need all the libraries to make it the same everywhere.

And that is the key.