Complexity

There are many different things that have made me tired as a developer lately. I've seen that we've chosen to specialize in one or more frameworks either because the products you work with are moving towards a given library (looking at you, Polymer and Gutenberg) or because you've realized a given library does what you need better (happened to me with Gulp over Grunt).

But there are many areas where we can simplify. We don't need the kitchen sink for every project we create, and huge libraries or moving styling to Javascript when the platform gives you options to accomplish the same goal.

I'll look at this from three areas: simplifying the stack, simplifying the designa and simplifying the publishing process.

Evolving Complexity: The Stack

When I first started doing development, what we now call front-end, in 1994 it was as simple as it could be, just HTML tags and far fewer than we have now. Javascript in 1995 and CSS (1.0) in 1996 increased the complexity but it was still manageable.

I stopped building sites for a while around 2009 or 2010. I moved full time into curriculum development and instructional design. I tried to stay current but the truth is that my mind just wasn't in it.

When I came back it was a totally different, and far scarier, playing field. Even looking at the basic components of the web: HTML 4 had grown and morphed into HTML 5, CSS 2 was an order of magnitude (at least in the size of the spec and number of features) more complex, and Javascript was finally moving forward again in TC39.

As Frank Chimero writes in Everything Easy Is Hard Again:

> The complexity was off-putting at first. I was unsure if I even wanted tackle a website after seeing the current working methods. Eventually, I agreed to the projects. My gut told me that a lot of the new complexities in workflows, toolchains, and development methods are completely optional for many projects. That belief is the second thread of this talk: I'd like to

> make a modest defense of simple design and implementation as a better option for the web and the people who work there.

And that is just the basics. I had the same belief as Frank. That working with the technology would help but I see this cyclic "best tool ever" wave come over and over and feel like I'm swimming in a never ending pool with technology that will never stay in place long enough.

Trip Down Memory Lane

But let's take a step back and figure out how we got here:

In the beginning there was HTML (and HTTP 0.9 and 1.0) and content was mostly documents. The first public browsers were Viola, developed at UC Berkley, Arena (used as a testbed for new web technologies and now superseeded by Amaya), and NCSA Mosaic, the root of both Netscape (throug engineering staff from the project) and Internet Explorer (through licensing to Spyglass).

The first server side applications used CGI (Common Gateway Interface) scripts written in either Pelr or compiled in C; The spec is defined in RFC 3875.

This is what a ***hello world*** script written in Perl and using the `CGI::Simple` module looks:

```
#!/usr/bin/perl
use strict;
use warnings;

use CGI::Simple;
my $q = CGI::Simple->new;
print $q->header;

print "Hello World!";
```

For the longest time server side processing and round trips were the norm and HTML was the way we authored content, we didn't pay attention to applications beyond form submission and some other creative uses. As long as you knew Perl or C to write CGI or had access to some of the earlier marketplaces where people would share or sell their scripts you were good.

The early specifications for HTML were part of IETF through their RFC process. A search in the IETF archives produced the following list of RFC documents directly related to HTML 2.0:

When tables were introduced people thought, of course we can use tables to do layouts, can't we? Thus started the era of layout tables and spacer gifts. It also gave us some never dying gems like the Space Jam website, still mostly alive 21 years after the movie was first released.

At this point it's also worth noting that all the styles and style attributes were inline or attached to the elements. Even so the style attributes and elements were limited and we could "know all of HTML" easily

After we got tables we got CSS1 in 1996. As far as CSS recommendations go this is a skinny one, as you can see in the HTML version of the spec at the W3C site and, somewhat surprisingly, most of those properties still exist in the current CSS3 family of specifications.

In 1997, just a year later, we get CSS 2.0. It's a complete update to the 1.0 specification and provides a more complete coverage of styles for CSS. At this time, however, browser vendors were under no obligation to implement the specification rather than their own propetary styles and elements.

Here was another inflection point. Do we use CSS or do we use tables. The Web Standards Project was founded in 1998 to advocate to Netscape, Microsoft and developers to implement and use the existing W3C specifications in a way that will work across browsers.

Also in 1996 we see the release of Javascript 1.0. I think we've all heard the story of how Brendan Eich created the language in 10 days and based its syntax on Java (to please management at Sun and Netscape) with inspiration from Self and Scheme. Microsoft, who apparently didn't want to deal with licensing issues, reverse engineered Javascript and called it JScript. The two languages were similar but not identical.

Netscape server products provided Server-Side Javascript, an earlier precursor of what we now call Isomorphic Javascript applications.

Javascript became an international standard through ECMA (now Ecma International) as ECMA 262 with the publication of version 1.0 in 1997.

So we have the basic components of the web as we know it today, although

much simpler than their current versions.

To frame the next section, let's look at when initial versions of the browsers we know today were released. These are by no means the only the only browsers existing in the early/mid 1990s but those, I believe are the most important.

- Early Browsers
    - Netscape Navigator 1.0 was releasssed in 1994
    - Internet Explorer 1.0 was released in 1995
- Later Entrants
    - Opera First Public Release as shareware in 1996
    - Safari 1.0 Released June 23, 2003
    - Google Chrome first releassed in 2008

This is where the first big layer of complexity was added. For those of you who remember, the first browser war was between Netscape and Internet Explorer (Microsoft). They each introduced proprietary features to their browsers in the hope of attracting larger market share and, while this competition inspired inovation, the innovation was not shared across browsers or with a standards process and it left developers in the middle having to decide which features of which browser to implement and how many people using "the other" browser to exclude from your experience.

Still, it was possible to know everything there was to know for working on the web. The tags exclusive to th two major browsers were not many and many people chose not to use them anyway (anyone remember `blink` or `marquee`, `layers` or `document.all`?)

Oh, and if you really want to know, Microsoft won the browser war because they had access to the operating system and where able to bundle browser and operating system making it easier for people to use without downloading the competition's.

Does anyone remember when we used to do browser detection instead of feature detection? In its simplest form we tested if the browser name contained either `Netscape` or `Microsoft` and wrote code that would work on one browser but not the other.

```
browsername = navigator.appName;
if (browsername.indexOf('Netscape') != -1) {
```

```
  browsername = 'Netscape';
} else {
  if (browsername.indexOf('Microsoft') != -1) {
    browsername = 'Internet Explorer';
  } else {
    browsername = 'N/A';
  }
}
```

Eventhough browser detection was (and is) fraught with danger as most browsers kept the `mozilla` or `mozilla compatible` string as part of the user agent name; that's why I used the name of the vendor when testing if the browser is supported. Even that caution is not enough... Internet Explorer 3 for Macintosh supported a different set of features than the Windows version so we'd have to dig deeper into the browser detection tree and test for platform in addition to browser.

The example above only deals with the more simplistic case, only Netscape and IE where IE supported the same features in both Windows and Macintosh. I purposefully left out how to detect the early standards-supporting browsers where we would do basic feature detection, and wrongly assume that if a browser supported the feature we were testing for (`document.getElementByID` for example) it supported all the specs.

The next big wave of web complexity (either increase or decrease) came with frameworks. Several of the earliest frameworks were released in a short period between 2005 and 2006. Some of the better known Javascript frameworks and their release dates:

- jQuery initial anouncement: August, 2006
- Dojo Toolkit initial release: March 2005
- MooTools initial release: September, 2006
- Prototype initial release: February, 2005

The biggest reasons for using one of these frameworks were: to smooth the differences between features browsers supported in different ways, to avoid code duplication beyond what the differences in browser support already required and to provide a smoother developer experience.

This is where we start loosing control of the amount of CSS and Javascript that we have to learn. Each library does similar things but very differently.

```
// This assumes you already loaded the libraries

$(document).ready(function() {
  $('p').css('color', '#ff0000');
});

'p'//MooTools stuffndow.addEvent('domready', function() {
  $$('p').setStyle('color', '#ff0000');
});

// 'domready'// Prototypeph').setStyle({
  color: '#ff0000'
});

// Dojo 1.7+ '#ff0000'// Dojo 1.7+ (AMD)
require(['dojo'], function(dojo) {
  // Passing a node, a style property, and a value
  // changes the current display of the node and
  // returns the new computed value
  dojo.setStyle('myParagraph', 'color', '#ff0000');
});
```

This is one very simple example. If you wanted to move from one framework to another we had to evaluate all the code we wrote and how much we'd have to change it to make it work in the new platform. We also had to make sure that all platforms supported the same features so we wouldn't have to write them from scratch or add yet another library to the project.

Then we started what I call the ***let's be overwhelmed*** period where multiple frameworks, libraries and tools came out (and are still coming out) to do very specific things. Since Javascript frameworks are a special class of software frameworks, we'll keep the following definition in mind as we move through the discussion.

> In computer science, a software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. A software framework provides a standard way to build and deploy applications. A

> software framework is a universal, reusable software environment that provides particular functionality as part of a larger software platform to facilitate development of software applications, products and solutions.
>
> Wikipedia - Software Framework

I've seen over the years that it has become harder to decide what to learn and how much is enough or too much. The requirements have become much more complex too, thus adding a level of difficulties to the decisions you make:

- The web has become an application platform to the detriment of content sites
- It's not just desktop anymore, the form factors people use to access the web have increased and your app/content has to work well in all of them
- There are more users than ever... millions of them. Performance is now a thing that touches all aspects of front-end development

What I've seen is that most developers, particularly people who are just starting, feel the need to immediately get into the latest and greatest framework or library thinking it'll make the more marketable and it may, in the short term, without really learning fundamentals that will make it easier to transition to what will come next.

Those of us who've been around the block once or twice see the pattern and wonder how many people will migrate next time.

And it's got even more complicated; it's not just frameworks or HTML or CSS. It's the way you build your applications, the way you bundle assets (and if you should bundle assets), what code editor you use and how it helps your productivity, its performance and how you optimize your code to make the site as fast as possible, it's about how to implement PWAs for your application, it's older browser support, not breaking the web, it's about CORS, it's about CSS in Javascript and whether you should use it ot not.

The Gist below lists some frameworks and libraries that I'm aware of (not necessarily familiar with) and some collected from Wikipedia.

[gist https://gist.github.com/caraya/32ad55f05549b8b938f7d839a9ae89ba]

I think we're not asking the right questions when it comes to frameworks and related technologies. Rather than rehash all the questions I've asked about this I'll

refer you to Pam Selle's Choosing a JavaScript Framework and 13 Criteria for Evaluating Web Frameworks for some of the questions that you need to ask yourself when adopting a new stack or parts of a stack for your project.

And do yourself a favor and question every little piece of data you see in those *best frameworks* articles. You'll be happy you did.

So, given all the decisions that we have to make as developers I'm not surprised we see and talk about fatigue and about how many things that we need to know or learn and how much code it takes to complete a given task.

Do we need a framework/library for that?

The answer is that it depends. It depends on the project, requirements and team expertise. I've listed the requirement for a site or the front end of an application and what I use to address each of the concerns.

The issues I will address in this section are:

- a build system to automate repetitive tasks
- responsive layout
- web fonts and good typography
- responsive images
- lazy loaded images and video

Build System

I guess I'll date myself when I admit that I've used Make and Ant to build content from ebooks to early examples of web applications before I discovered Grunt, Gulp and associated tools.

It was some of the libraries and frameworks I started working with that dictate my move first go Grunt and then to Gulp. I've gotten to the point where I'm happy with my build system... just in time to choose between Rollup and Webpack.

I chose what I call ***Webpack lite***. I don't see the need to throw away my Gulp workflow just for bundling, even though Webpack may be able to duplicate my Gulp tasks; instead I integrated Webpack's bundling functionality into my Gulp workflow as another item to tackle during build.

Responsive Layout

Responsive layout is one of the few places where I use a framework. I started playing with SCSS a few years ago and I find myself writing it even when working with plain CSS. All CSS is valid SCSS but the reverse is not true. Responsive design has more to do with knowing what CSS to use when and how to use media queries to tweak the layout of your page.

Web fonts and Typography

I've written about font loading and optimization. For the most part this is a matter of using tools like Fontface Observer and its standard complementary tool, `font-display` to control the behavior of the downloadable fonts and their fallbacks, subsetting fonts so it'll only use the characters actually on your pages among other things.

The following script uses Fontface Observer to work with Noto Sans and Noto Mono. It loads them and notifies you if it's unable to. It then switches the class of the HTML element to use the web font or the backup you've assigned.

```
const mono = new FontFaceObserver('notomono-regular');
const sans = new FontFaceObserver('notosans-regular');
const italic = new FontFaceObserver('notosans-italics');
const bold = new FontFaceObserver('notosans-bold');
const bolditalic = new FontFaceObserver('notosans-bolditalic');

let html = document.documentElement;

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

Promise.all([mono.load(), sans.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');
  });
```

Responsive images

This is a combination of generating the images during the build process and being disciplined when writing your HTML or creating your templates that use the images in the sizes that you want and not have to worry about entering the `sizes` and `srcset` attributes for each image that you want to use.

There are many situations that we have to adapt our images for. Some of the use cases:

1. We want our images to be available in multiple resolutions and densities so that they scale well in fluid layouts and look good in Retina displays
2. Sometimes we might want to crop or change images to match the design of a site or layout, in essence, providing `art direction` for the project
3. Provide images in different formats for browsers that support them. We can provide WebP images for those browsers that support them and give PNG and JPEG to those browsers that don't support WebP

To see possible solutions to these use cases see the Responsive Images Community Group page of demos and Responsive Images Done Right: A Guide To And srcset

Templates and template engines would allow the automation of this process by creating templates that only require the name of the image and would paste it where appropriate in the image element and it's children. When I've used them I've always included templating as part of the build process.

Zell Liew wrote a good introductory article on modularizing content with templating engines and Gulp. It should be too hard to extend the idea to other build systems.

Lazy loading images and videos

Lazy loading, in this context, means that we don't load an asset until it appears on screen (or within a certain distance from the viewport) or the user clicks on it.

The Javascript lazy loading script uses intersection observers to detect when

the image is coming into view and load it when the conditions are met.

The video lazy loading I use creates full on replacement of the thumbnail replaces it with Youtube's iframe when the user clicks on the video. This type of lazy loading scripts are prime candidates to update using ES6 template literals to make it more concise and easier to read and reason through.

You may find other ways to create lazy loading scripts but the basic functionality is already here.

Simplify The 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 early 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-1
```

```
    <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.

Conclusion

Over the years the sheer number of tools that are available has become

staggering; at least it was for me. I had to consciously make a choice about what I really wanted to work with and in what areas. These were my choices:

- HTML
- CSS
- Javascript
    - Current Version
- Data Visualization
- WebGL

Much to my surprise, each one of these areas had a much bigger footprint than I thought it did.

The list has evolved over time to something like this:

- HTML
    - Hand-coded
    - Templates
        - Mustache
        - Handlebars
- CSS
    - Vanilla CSS (features are added all the time)
    - SCSS (Superset of CSS with programming-like constructs)
- Javascript
    - Current Version at the time (not including HTML5 APIs)
        - ES5
    - ES Next (not including HTML5 APIs)
        - ES6
        - ES7 + whatever is next
    - Typescript
- Data Visualization
    - D3 V3
    - D3 V4
- WebGL
    - Three.js
- Node.js
    - Expres.js
- Tooling
    - Transpilers
        - Babel
    - Script Runners

        - Grunt
        - Gulp
        - NPM
    - Module Bundlers
        - Rollup
        - Webpack

Where there is more than one option on the list they are listed from the ones I learned first and moved to different tools either because they performed a task better or because a framework or library I wanted to work with used those libraries or build tools (looking at you Polymer).

The thing is: none of the items on either list need a full framework to work with them; rather, we need to ask ourselves what do we need for the project and how not to go overboard. No more JS frameworks presents a view of why we should carefully evaluate the needs of the projects.

> For a long time there was a whole lot of inconsistency between browsers and we, as an industry, had to write frameworks to paper over them. The problem is that there was disagreement even on the fundamental issues among browsers, like how events propagate, or what tags to support, so every framework not only papered over the holes, but designed their own model of how the browser should work. Actually their own models, plural, because you got to invent a model for how events propagate, a model for how to interact with the DOM, etc. A lot of inventing went on. So frameworks were written, each one a snowflake, a thousand flowers bloomed and gave us the likes of jQuery and Dojo and MochiKit and Ext JS and AngularJS and Backbone and Ember and React. For the past ten years we've been churning out a steady parade of JS frameworks.
>
> — No more JS frameworks

We've spent the best part of 2 decades working over how to make the web work well, work consistently and work closer to the way we want them to.

In the beginning frameworks were necessary because browsers were incompatible and you had to make your code work the across browsers and, more recently, across form factors... but browsers have gotten better at implementing the same specs and in the same way. So the need to have normalization

frameworks like jQuery, MooTools, Dojo, and others has lessened from this point of view.

You Might Not Need jQuery and (Now More Than Ever) You Might Not Need jQuery but, depending on your the browsers you must support, frameworks like jQuery might still be needed as John-David Dalton and Paul Irish wrote in jQuery's browser bug workaround.

Another thing worth remembering is that when you use a framework you're not only buying into the functionality you need; it's an all or nothing proposition. Either you use the full set of capabilities the framework offers or you are wasting bandwidth sending over unnecessary bytes if you don't slim your framework down to what you're actually using.

One last thing about the complexity curve. Sooner or later it'll come a time when no one on your team will understand all of the code in your application. What happens when a key member of your team who understands the majority of your code leaves or when you decide to refactor the code or more to a different framework that has become the latest and greatest?

A separate issue, for me, is when to use a library versus when to use a framework versus when vanilla javascript will work, and how to tell the difference.

In When Does a Project Need React? Chris Coyier presents a list of items to consider when implemeting an application and whether to use React and its ecosystem or other less complex alternatives. I think we need to ask the same type of question about any framework... is React overkill? Angular? Polymer?

I'm not against frameworks and libraries, they have their place. What worries me is that we're reaching for frameworks when we don't neccessarily have to and by not doing so we save ourselves from fatigue.

References

- Simplify the stack
    - Everything Easy Is Hard Again
    - Complexity
    - CSS Tricks Newsletter #83
    - The increasing nature of frontend complexity
    - I finally made sense of front end build tools. You can, too.

        - What is the Future of Front End Web Development?
        - The What and Why of Javascript Frameworks
        - Owning the Role of the Front-End Developer
- Build Tools And Task Runners
    - JS Task Runners Comparison: Grunt vs Cake vs Gulp vs Broccoli
    - Comparison of Build Tools
- JS Fatigue
    - Why I'm Thankful for JS Fatigue. I know you're sick of those words, but this is different.
    - What is JavaScript Fatigue?
    - The Ultimate Guide to JavaScript Fatigue: Realities of our industry
    - JavaScript fatigue fatigue
- CSS In Javascript
    - CSS in JavaScript: The future of component-based styling
    - Stop using CSS in JavaScript for web development
- Simplify the design
    - What Screens Want
    - The Web's Grain
    - Designing in The Borderlands
    - A Simpler Page
    - Death to the waterfall
- The Web Standard Project
    - The Web Standards Project
    - WaSP History
    - WaSP Mission
    - Our Work Here is Done
- Responsive Web Design
    - Responsive Web Design
    - Responsive Web Design Examples
- Bootcamps
    - Complete guide to the top 24 coding bootcamps
    - Bootcamps won't make you a coder. Here's what will
- Conclusion
    - When Does a Project Need React?
    - Loading Third-Party JavaScript
    - Is React library or a framework?