Evolving 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 two areas where complexity has evolved over the years: the stack, and 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 for me to become good at, let alone master it.

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!";
```

And the same program written in C:

```
#include <stdio.h>
```

```
int main(void)
{
  printf("Content-type: text/html\n\n");
  printf("<html><title>Hello</title><body>\n");
  printf("Hello World!\n");
  printf("</body></html>");

  return 1;
}
```

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. 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 the web. At this time, however, browser vendors were under no obligation to implement the specification instead of their own propetary styles and elements.

Here was another inflection point. Do we use CSS or do we use tables? If we

use CSS, which browser do we support? 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 and that produced compatibility issues. Sounds familiar?

Netscape server products provided Server-Side Javascript, an earlier precursor of what we now call Isomorphic Javascript applications, even though it used different methods and syntax for server and client side.

Javascript became an international standard through ECMA (European Computer Manufacturing Association), 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 their experiences. The alternative was to duplicate code and create tailored experiences for each browser and having to do twice the work every time marketing wanted a change.

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`)

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.

```
var 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 browser features that worked in different ways on different browsers, to avoid code duplication 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 in very different ways.

```
// 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 to do very specific things. 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 as many of these new web users access content from low powered, low CPU, low storage capacity devices

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 based on the version of HTTP your server supports (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.

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.

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.

How many scripts do we need 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.

This works for me, right now, and it may not work for future projects. One thing you'll hear from me a lot is that you should test your tools in the context of your projects, your needs and your requirements.

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.

There are other ways to have richer extensions to CSS and they each have a different learning curve.

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');
  });
```

Note that this will display the text and update it only if the fonts load via a specified time, if not the fallback fonts will be used.

> This script also needs three classes for the body element:

- **fonts-loading** to indicate the fonts when the page first loads
- **fonts-loaded** to use when the fonts have loaded successfully
- **fonts-failed** is used when the fonts fail to load for whatever reaason

Variable Fonts for the win?

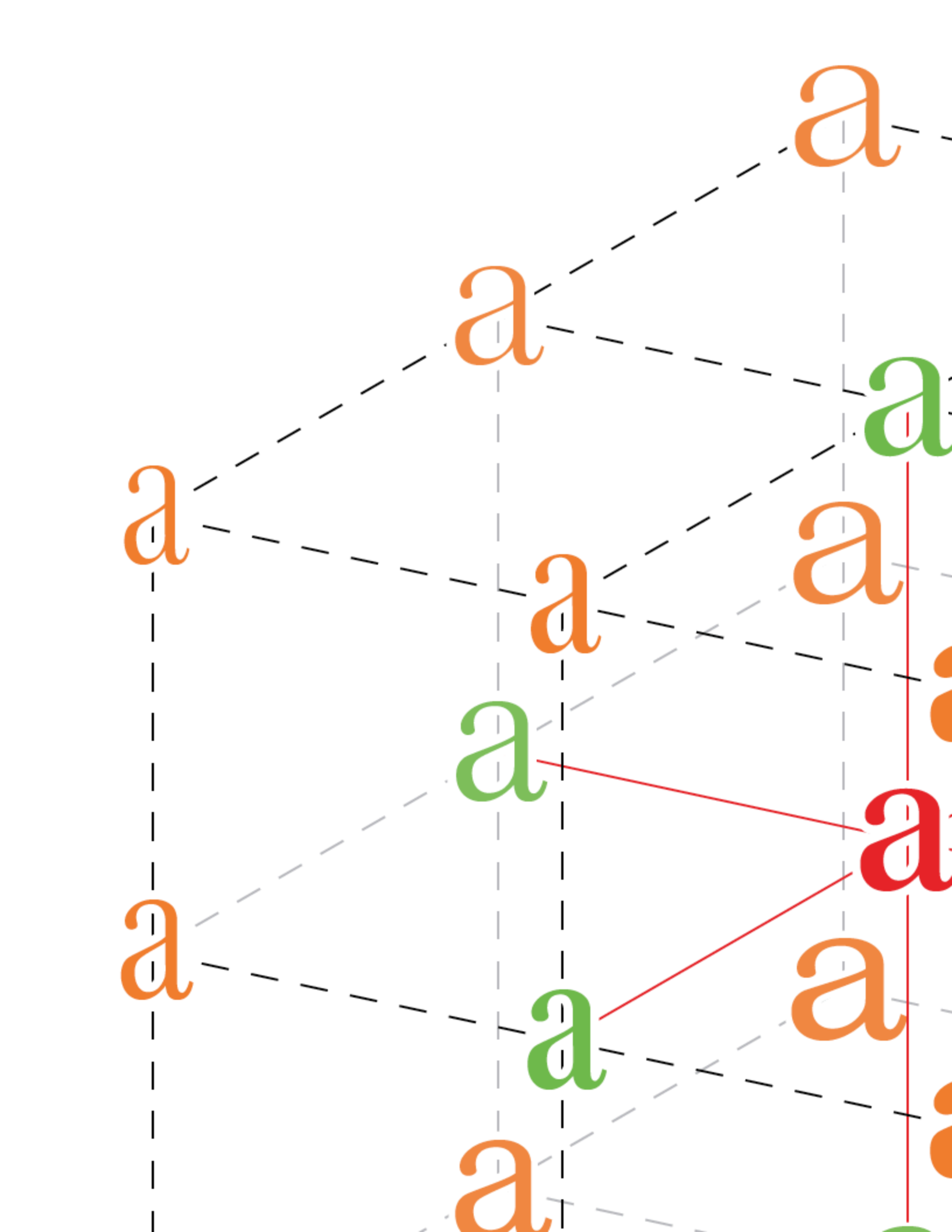

Figure 1: Normalised design space of a 3-axis variable font.[Typeface: Kepler, an Adobe Original designed by Robert Slimbach.]

The latest development in the world of fonts and typography is variable fonts. Without going into details we can say that:

> An OpenType variable font is one in which the equivalent of multiple individual fonts can be compactly packaged within a single font file. This is done by defining variations within the font, which constitute a single- or multi-axis design space within which many font instances can be interpolated. A variable font is a single font file that behaves like multiple fonts.
> John Hudson *https://medium.com/variable-fonts/https-medium-com-tiro-introducing-opentype-variable-fonts-12ba6cd2369*

The biggest issue with Variable Fonts is that they are new and they require a modern operating system and a recent browser for them to work. But in a modern operating system, Variable Fonts will work accross modern browsers (IE 11 is the only desktop browser where they won't work and Opera Mini and Opera Mobile are the only mobile browsers I'm aware of don't support them).

See variablefonts.io for a good introduction to using variable fonts.

Responsive images: srcset and sizes

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. Also see the discussion about the `<picture>` element, later in this post.

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.