Thiking about design

The hardest part of front-end design, for me, is how to put all the little pieces together, whether to use a build system and how to best address performance as a holistic concern.

This article will explore things we can do to put all the disparate elements together and some areas where we need to evaluate options before moving forward.

Note

The article assumes that you're familiar with HTML, CSS and Javascript and, if necessary, can write it by hand.

To illustrate the techniques, we'll use a static site for a hypothetical conference.

The site will have the following areas:

1. About the conference.
    1. What it is
    2. When will it happen
2. A list of presentations linked to the speakers
3. A list of speakers linked to their presentation abstracts
4. Information about the venue
5. The skeleton of a registration page

We will consider the following items as we discuss building the site.

1. Before we write code
    1. Where do we want to host our projects
    2. Understanding the W3C processs
    3. Understanding the TC39 process
    4. Caniuse.com is your friend, trust him
2. Responsive Web Design
3. CSS Grid and Flexbox
4. Media Queries: Still the way to go

5. Variable Fonts
    1. Typography with variable fonts
6. CSS Variables
7. Responsive images
8. Keeping ourselves honest with client hints
9. Performance suggestions
    1. Working with Javascript
    2. Performance budgets
    3. Performance measurement tools
        1. Lighthouse
        2. CrUX Report
10. Node and NPM
    1. Gulp as a Node-based build tool
11. Putting it all together

Not all the issues are directly related to design but they all affect design in one way or another.

Before we write code

Something that I always tend to forget when starting a new web project is to slow down and think about hosting and related items before I write code.

I know that for some this sounds like putting the horses before the cart but hear me out.

A lot of times the host we choose will dictate or influence how we publish our work or how we configure additional assets.

For example: Different servers and cloud providers have different ways to add mime types for new file formats so depending on where we choose to host a project we will have different ways to do the configuration.

Another example: If you're creating a site with Jamstack technologies, will you pay for a full fledged hosting provider or will you go with a free-tier plan from Netlify or Firebase Hosting. Will you use cloud functions from the same provider?

One final example: You need a database for your application and you've selected Postgresql. You now have to decide what version to use and make sure that the version you're developing with matches the version of Postgres matches what's on the server or you will have unexpected results.

These are the items you should figure out ahead of time. There's nothing more stressful than having things not working and only 10 minutes to figure out why before the site goes live.

There are other external factors we need to understand before we move forward. How does the W3C (the group that implements HTML, CSS and related standards) and TC39 (the group that implements Javascript and Javascript internationalization standards) work?

Understanding this will help us better target the technologies we use.

Understanding the W3C processs

The World Wide Web Consortium (W3C)

Understanding the TC39 processs

Caniuse.com is your friend, trust him

Responsive Web Design

Responsive Web Design is not a new thing in and of itself. When Ethan Marcotte wrote Responsive Web Design in A List Apart he didn't invent a brand new technology from scratch. He combined existing techniques and APIs like media queries and fluid images to accomplish good design for all form factors.

We've come a long way since then.

Media queries have become more flexible and powerful and are now supported in all major browsers.

Media Queries (the original specification, and Media Queries Level 4) present the existing work in media queries that has been ratified as recommendations (or candidate recommendations by the W3C) while Media Queries Level 5 represents the next set of iterations for Media Queries.

Fluid images still work as intended but for better performance we can use responsive images to provide the best experience for our users.

CSS Grids and Flexbox working together

One of the first things I thought about when CSS grids became a thing was how to incorporate it to my existing workflows without breaking what's already there.

The conclusion was to put the grid at the very bottom and place other design elements on top of it. Contrary to what some believe, Flexbox and CSS grids can work together and are not in competition.

Think of Flexbox as a one-dimensional layout, either row or column-based, not both.

CSS Grids, on the other hand, can place content in a two-dimensional grid, both column and row at the same time so it's more flexible. However, that doesn't mean we don't need Flexbox anymore.

Media Queries: Still the way to go

In the past people use to have separate sites for mobile and dekstop browsers. They also used to do browser detection instead of feature detection but that’s another story.

Having mutliple sites was hard on everyone. People had to remember the name of the mobile site, usually *m.site.com* and developers had to worry about keeping content synchronized

Variable Fonts

I've written a lot about variable fonts. To me, they are a wonderful addition to the web typography arsenal, even with their OS/Browser limitations.

It is not enough that a browser support variable fonts. The underlying operating system also has to support them or they won't work at all.

We also need to keep in mind that a variable font is usually larger than the single font. For example: Recursive is larger than Roboto Regular but it contains all the styles available to the font and will contain them even if you don't use them yet it is smaller than using the four files that I normally use when working on the web:

- Regular
- Italic
- Bold
- Bold Italic

Taking the regular Roboto font from Github, we will take the four font files that we'll need for a page. If we know we don't have italics or bold we can eliminate them but, in most cases, these are the files we'll need.

We compressed the font files with `woff2_compress`, part of the Woff2 reference implementation arriving at the following values.

Font	Size	format	Notes
Roboto Regular	136KB	WOFF2	
Roboto Italic	159KB	WOFF2	
Roboto Bold	137KB	WOFF2	
Roboto BoldItalic	161KB	WOFF2	

We then took the Google-commissioned Roboto Variable Font from Github. The first value on the table is the original True Type Font, for reference.

Like we did with the individual instances we compressed the font with `woff2_compress` for a 50% reduction on the weight of the font.

It is important to note that the compressed font has 36 named instances (combinations of weight, italics and width) and 22 Open Type Layout features.

Font	Size	Format	Notes
Roboto Variable Font	786KB	TTF	Baseline for Variable font comparison
Roboto Variable Font	340KB	WOFF2	Carries all condensed, bold italic and regular instances of the font
Roboto VF Subset	330.9KB	WOFF2	Subset of Latin characters

We can further reduce the size of the variable font by subsetting using tools like Glyphhanger to remove the characters from languages that we know we won't use.

```
glyphhanger --latin \
--subset=Roboto-VF.ttf  \
--formats=woff2
```

Rather than figure out everything on your own, you can use tools like Wakmaifondue (a play on the phrase *what can my font do?*) to see the type of font is (variable or not), the languages it supports, the instancces it has built in if it's a variable font, and, most important to me, the number of Open Type features that the font makes available.

Wakamaifondue also generates a CSS stylesheet that you can use as is or by incorporating it to your existing CSS styles.

Google fonts way of serving variable fonts

One of the biggest pain points of variable fonts was that they were not available in font CDNs like Google Fonts.

In October, 2019, Google released the new version of their font API that supported variable fonts. I wrote about it in Variable Fonts from Google Fonts

The way we import the fonts from Google Fonts changes when working with version 2 of the Google font API. To use Roboto Variable Font, the link we insert in HTML looks like this:

```
<!--
  The link should be in a single line.
  Broken down for readability
-->
<link
  href="https://fonts.googleapis.com/css2?
  family=Roboto+Slab:ital,wght@0,400;
  0,700;1,400;1,700&display=swap"
  rel="stylesheet">
```

and the CSS `@import` command changes in a similar way:

```
/*
  The link should be in a single line.
  Broken down for readability
*/
@import url('https://fonts.googleapis.com/css2?
  family=Roboto+Slab:ital,wght@0,400;
  0,700;1,400;1,700&display=swap');
```

The downside of using Variable Fonts from Google Fonts is that they don't have all the axes available to the font so if you've done work with the font before it may not do what you would expect, at least not without major work.

For example, Recursive from Google Fonts only provides the `wght` axis and none of the other axes the font provides when you use it locally.

They also lose all the named instances because in this example, each axis request individual values.

We can ask for value ranges by changing the values and replacing the single value with a range of values separated by two periods:

```
<!--
```

```
  This should be all in a single line
  Broken down for readability
-->
<link href="https://fonts.googleapis.com/css2?
family=Roboto+Slab:ital,wght@0,100..700;
1,100..700&display=swap" rel="stylesheet">
```

What this will do is request the full range of weights for both regular and italic versions of Roboto Slab and modify the CSS `@font-face` declaration to include the range.

With these modifications we get the full power of variable fonts. We can animate them, we can use values other than multiples of 100 and actually be creative with how we use them.

How to use variable fonts

Once we have loaded the font we need to register it for use using the [@font-face at-rule].

In most examples you will see something like this:

```
@font-face {
  font-family: "Open Sans"Open Sans"   url("/fonts/OpenSans-Regular-webfo
    format("woff2"),
    "woff2"url("/fonts/OpenSans-Regular-webfont.woff")
    format("woff");
}
```

This will load the specified font, associating it with the value of the `font-family` attribute and using defaults for everything else.

While it’s possible to load fonts from the user’s file system using `local` instead of `url`, you must be careful doing that as it may be used to fingerprint the user by identifying those users that have the font in question. This is a problem because the font may be a corporate resource only available to people who work at a given organization or have other restrictions to use.

To provide the browser with a hint as to what format a font resource is — so it can select a suitable one — it is possible to include a format type inside a `format()` function

The available types are:

- WOFF
- WOFF2
- True Type
- embedded-opentype
- svg

The browser will load the first font it can use so the order of the children in the `src` attribute matter. For example, if you have `woff2` and `ttf` versions of the font, you should load them like this:

```
@font-face {
  font-family: "Open Sans"Open Sans"   url("/fonts/OpenSans-Regular-webfon
    format("woff2"),
    "woff2"url("/fonts/OpenSans-Regular-webfont.ttf")
    format("true type");
}
```

For mordern browsers you will be working primarily with WOFF2 and, possibly, WOFF formats.

There are also a series of descriptors we can use to customize the font we're loading.

`font-stretch`
: The `font-stretch` value selects the width of the font,meaning if the font is condensed, regular or wide. Most modern browsers will also accepts two values to specify a range that is supported by a variable font, for example `font-stretch: 50% 200%;`

`font-weight`
: The `font-weight` value select the weight of the displayed font (how bold it is). Most modern browsers also accepts two values to specify a range that is supported by a variable font-face, for example `font-weight: 100 900;`

`font-variant`
: `font-variant` is the shorthand for `font-variant-*` attributes as defined in CSS Fonts Module Level 3 and CSS Fonts Module Level 4 specifications

`font-feature-settings`
: Allows control over advanced typographic features in OpenType fonts. The specification recommends that you use `font-variant-*` where possible and use `font-feature-settings` as a last resort if the equivalent font-variant attribute is not available

`font-variation-settings`
: Allows low-level control over OpenType or TrueType font variations, by specifying the four letter axis names of the features to vary, along with their variation values.

`unicode-range`
: The range of Unicode code points to use from the font.

CSS Variables

CSS Variables, also known as CSS Custom Properties, and its Houdini big siblings create live custom properties for our pages to use.

That’s the main difference between CSS variables and preprocessor variable like SASS or LESS.

Preprocessor variables are executed once at runtime; Every time we want to change the values of the properties we have to recompile the stylesheet

CSS Variables, on the other hand, are live. Every time we change the value, the new value will be reflected on the page elements that use it.

Houdini’s CSS Properties and Values API takes CSS Variables a step further by addressing the shortcomings of the CSS-only solution. The downside is that Houdini Custom Properties are not supported everywhere yet.

You can use Houdini Custom Properties, where supported, either via Javascript or CSS. The Javascript declaration looks like this:

```javascript
CSS.registerProperty({
  name: "--my-color",
  syntax: "<color>",
  initialValue: "black",
  inherits: false
});
```

The CSS declaration is very similar but it follows CSS syntax:

```css
@property --my-color {
  syntax: "<color>";
  initial-value: black;
  inherits: false;
}
```

Responsive Images

Images are the biggest portion, file size wise, of any webpage. Users download more images than any other component of the page; Javascript comes close but it’s not quite the same.

The problem is that, using a one size fits all approach, the images will look awful in retina displays and will download unneccesary pixels in smaller form factor devices, potentially consuming a large portion of the user’s metered bandwidth allocation.

While it’s true that they require more work upfront in preparing the images that we want to use, responsive images provide a better user experience, particularly for people on lower-end mobile devices.

There are two things we can do and one thing we **definitely** should not to to improve image size and performance.

Keeping ourselves honest with client hints

Performance suggestions

Performance budgets

Performance measurement tools

Lighthouse

CrUX Report

Node and NPM

Development has sadly gotten to the point where every little web project needs a build system of some sort or another. This section will discuss one such build system in the context of what we’ve discussed so far.

Gulp as our build tool

Putting it all together

As discussed in the introduction we’ll build a static site for a hypothetical conference with the following items

1. About the conference.
    1. What it is
    2. When will it happen
2. A list of presentations linked to the speakers
3. A list of speakers linked to their presentation abstracts
4. Information about the venue
5. The skeleton of a registration page