Improving Font Performance

Talking about web font performance is more than just talking about adding the fonts to a page using `@font-face`. That's just the beginning of our quest for performant fonts.

Because of their size fonts tend to be some of the largest components of any web pages. According to the HTTP Archive, the median of requested fonts is 98KB for Desktop and 83.4KB for mobile.

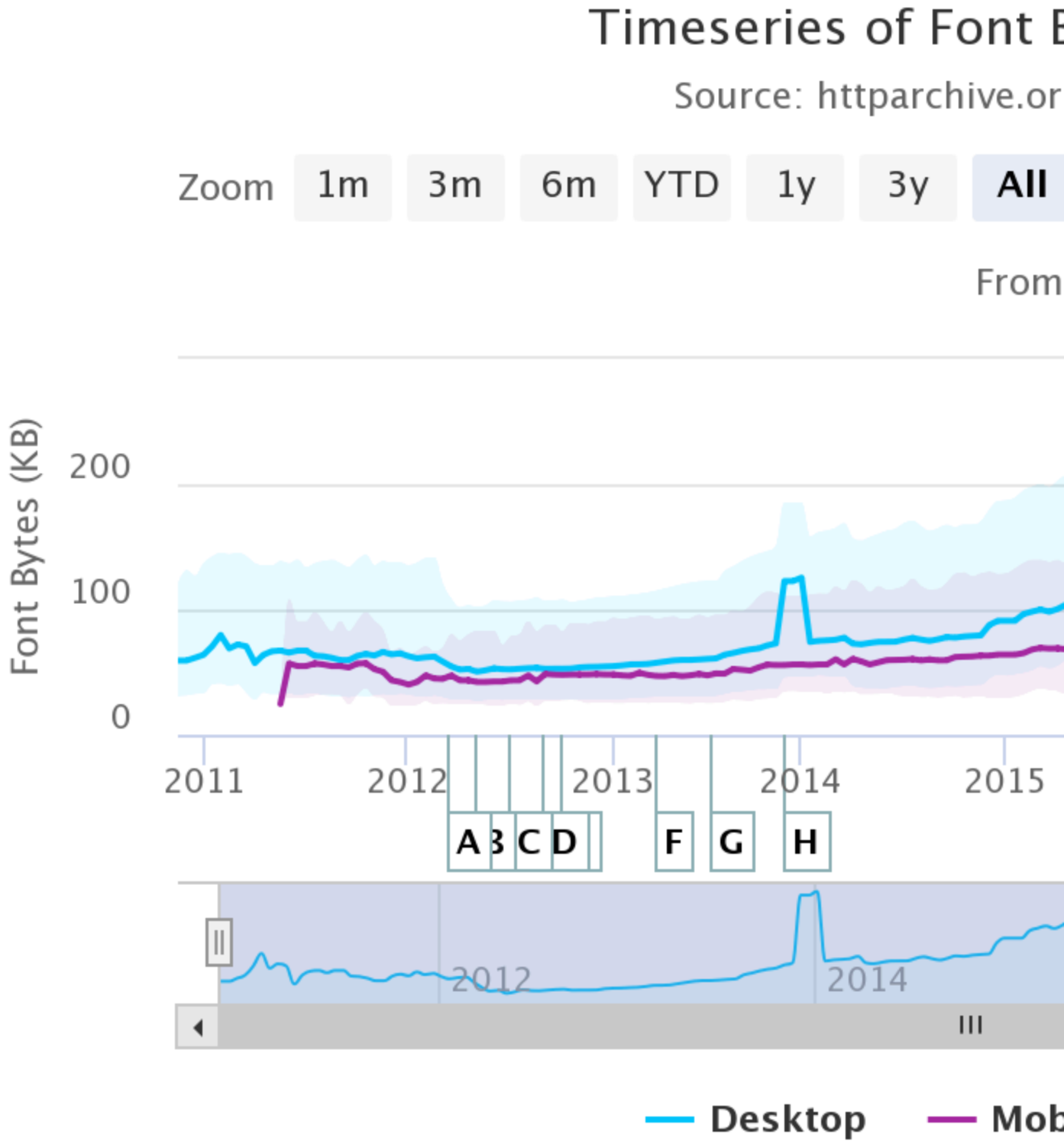

Figure 1: Comparison of median font sizes from 2010 to today

Making our fonts performant is particularly important when working with large variable fonts; Roboto VF is 976KB when compressed with WOFF2 and, if not cached, can significantly impact the performance of the page.

Background: @font-face and downloadble fonts

We tend to think of downloadable fonts as a new phenomenon but it isn't. In 1998 browsers started shipping support for the `@font-face` CSS declaration... now developers could use digital fonts in their pages, right?

Not quite. While browsers supported the syntax they supported different font formats: Netscape supported TrueDoc from Bitstream and Microsoft supported Embedded Open Type (EOT), which also provided a layer of encryption for their downloadable fonts.

The other problem was that you didn't have to own fonts in order to use them as downloadble assets. There was no way for foundries (the companies that created fonts) or developers to restrict access to the downloadble fonts.

Many foundries were concerned about piracy even in the encrypted EOT format so, for a long time, they witheld licenses for downloadble fonts.

Without good fonts, the whole idea of downloadble fonts fell off developers' radars and the whole idea lay dormant until 2009 when both Safari and Firefox (re)introduced `@font-face` on their browsers and the CSS Working Group introduced a standardized way to load fonts on the web.

While we had the specification for how to load fonts there was no standard font to use, we still had to account for all the different formats supported browsers at the time.

That's where the `bulletproof @font-face syntax` came in. It tried to support all browsers so we only declare the font once with multiple formats to acommodate different browsers. Over the years there have been multiple versions of the syntax, dating back to 2009:

- Bulletproof @font-face Syntax (2009)
- The New Bulletproof @font-face Syntax (2011)
- Further Hardening of the Bulletproof @font-face Syntax (2011), with "Extra Bulletproofiness" 😀
- How to Bulletproof @font-face Web Fonts (2011)

The next section explores how much it has evolved and whether we still need the

full syntax to work with Modern browsers.

The evolution of the bulleproof syntax

The original bulletproof @font-face syntax, first documented in Paul Irish's Bulletproof @font-face Syntax, looked something like this:

```
@font-face {
  font-family: Open Sans;
  src:  url("opensans.eot");
  src:  url("opensans.eot?#iefix") format("embedded-opentype"),
        "embedded-opentype"url("opensans.woff")      url("opensans.ttf")"
}
```

The original syntax accounted for the different formats browsers supported at the time and their idiosyncracies like the double declaration of EOT fonts.

In No @font-face syntax will ever be bulletproof, nor should it be Zach Leat describes the evolution of the bulletproof syntax and what the rationale for the removals are. The final `@font-face` syntax is:

```
@font-face {
  font-family: Open Sans;
  src:  url(opensans.woff2) format("woff2"),
        "woff2"url(opensans.woff) format("woff");
}
```

Browsers `src` attribute will use the first format that they support and, since most browsers that support WOFF2 will also support WOFF we want to place the smallest file first.

This assumes that most of your target users are in modern browsers, we're forcing older browsers to use system fonts:

Make sure that you test your font stacks in your target browsers, particularly if you're targeting emerging markets where users may be working with older

versions of operating systems to avoid device upgrades and, potentially expensive, software updates.

Compress fonts

Now that we've figured out the formats that we'll use we can start looking at the mechanics of preparing the fonts.

Most font providers will give you fonts in either TTF or OTF and we need to compress them for use in our pages.

WOFF and WOFF2 are font packaging formats and not font formats like TTF or OTF, they will shrink the size of the file and, possibly, add metadata to the resulting font, they will not change the font in any way.

WOFF with `sfnt2woff-zopfli` (Macintosh)

WOFF File Format 1.0 is a font packaging specification created in 2012 by the W3C's Web Fonts Working Group.

To create the WOFF version of the fonts I want to use, I've chosen `sfnt2woff-zopfli`. It will create smaller fonts than other WOFF packagers by using the Zopfli Compression Algorithm to compress the data at the cost of slower compression speeds.

To install sfnt2woff-zopfli in my Mac I used Homebrew and Bram Ramsteins `webfonttools` tap.

Run the following commands in your terminal:

```
brew tap bramstein/webfonttools
brew install sfnt2woff-zopfli
```

Once the process is complete, go to the directory containing your fonts and, for each font that you want to compress run the following command:

```
sfnt2woff-zopfli Roboto-min-VF.ttf
```

WOFF with `woff-tools` (Linux and Windows runing WSL)

Ubuntu systems and Windows 10 using the Window Subsystem for Linux and an Ubuntu Image work the same way. Make sure that you're in a terminal (for Windows users type `bash.exe` from Power Shell).

To install run the following command from your bash shell.

```
sudo apt-get woff-tools
```

WOFF2 with `woff2_compress` (all systems)

WOFF File Format 2.0 is an evolutionary development over WOFF 1.0 that provides better compression and smaller file sizes.

For WOFF2 I've chosen to use Google's WOFF2 Reference Implementation installed via Homebrew. Run the following command to install the WOFF2 tools and their dependencies.

```
brew install woff2
```

Once it's installed run the following command to compress your font:

```
woff2_compress Roboto-min-VF.ttf
```

To install WOFF2 in Linux (Ubuntu 18.04 LTS) and Windows (with WSL) run the following command to add the repository where the package is located. When prompted enter your Unix password.

```
sudo apt-add-repository universe
```

next, update the repository list

```
sudo apt-get update
```

Once the repository is configured and the list updated, you can install the package using `apt-get install`.

```
sudo apt-get install woff2
```

Subset fonts using Glyphhanger

Glyphhanger is a Node application that will help you reduce the size of your fonts by subsetting them.

There are times when we use just a few characters from a given font. For example if we use a font just for headings there are many characters that you will not use, a subset will get rid of these unnecessary characters making the fonts smaller and providing a faster download. Another example is when the font supports multiple languages that you know you won't need for your project (like Cyrilic when you know you will be working with English, Spanish, and French)

We will generate four different items from our font:

- The list of glyphs to use
- A latin characters subset
- A subset using only the characters in a given page
- A subset using the characters in all the local pages

Generating a list of the glyphs to use

The first thing to look at is generating the list of glyphs you want to subset to. This is different than creating the subset fonts, it will ony list the glyphs.

```
glyphhanger http://localhost:5000
```

You can also redirect the list to a file for later use

```
glyphhanger http://localhost:5000 > site-glyphs.txt
```

Subseting to the Latin Character set

The first experiment is to subset the font to use Latin characters only. Latin alphabets are used in the United States, Latin America and Western European countries

```
glyphhanger --latin \
--subset=Roboto-min-VF.ttf  \
--formats=woff-zopfli,woff2
```

This command will generate larger but more flexible subsets that will work with all pages in your site or application. But they are larger files, we can probably do better.

The next version will subset the font to use only the characters in the specified page, in this case the index page for the site.

Subsetting to a page

```
glyphhanger http://localhost:5000 \
--subset=Roboto-min-VF.ttf  \
--formats=woff-zopfli,woff2
```

This will create subset fonts in the specified formats (WOFF and WOFF2) along with the CSS @font-face declaration that you can drop in your stylesheet to use. The fonts and CSS will only have the characters used in that page.

Subseting with the spider

The previous subset is good for a single page application but breaks in multi-page sites as characters in the page we subset from may not be in the other pages in the site.

We can handle multiple pages by using Glyphhanger's spidering cpabilitiy by indicating that we want to use it (`--spider`) and the maximum number of URLs to

capture (`--spider-limit`).

```
glyphhanger --spider --spider-limits-30 \
--formats=woff2,woff-zopfli \
--subset=*.ttf \
http://localhost:5000
```

The result will be closer to the Latin subset but it will only contain the characters that exists in your site in the language it was written in. This becomes important when you use fonts like Roboto, Fira or other fonts designed to support multiple languages.

Looking at the sizes

Using Roboto Variable Font, downloaded from Github, as an example I got the following results when running the commands shown in prior sections.

Format	File Name	Size
TTF	Roboto-min-VF.ttf	2.2MB
WOFF	Roboto-min-VF.woff	1.3MB
WOFF Subset	Roboto-min-VF-subset.zopfli.woff	**104KB**
WOFF Latin Subset	Roboto-min-VF-latin-subset.zopfli.woff	**109KB**
WOFF Site Subset	Roboto-min-VF-site-subset.zopfli.woff	**1.3MB**
WOFF2	Roboto-min-VF.woff2	976KB
WOFF2 Subset	Roboto-min-VF-subset.woff2	**86KB**
WOFF2 Latin Subset	Roboto-min-VF-latin-subset.woff2	**977KB**
WOFF2 Site Subset	Roboto-min-VF-site-subset.woff2	**92KB**

The sizes are larger than you may expect because this is a variable font. It has all different instances of Roboto built in a single file so it's all you would add to handle all of Roboto's instances and Open Type functionality.

Serve through a Specialized CDN

Serving content through a CDN has always been a key to improve performance. The edge servers are in diverse geographical locations and they will route users to the closest server to their location.

Serving fonts through a CDN is no different. Sites like Adobe Fonts (formally known as Typekit) and Google Fonts provide a faster experience downloading the fonts to your devices.

Combine the CDN service with preconnect or dns-fetch resource hints for an even faster experience.

Cache fonts using a service worker

Most of the time when we hear about Service Workers we hear about them in the context of Progressive Web Applications (PWAs) but they can also be used to improve your site's performance by caching assets in the browser.

Fonts will be the largest assets we cache with the service worker so we want to keep a few locally hosted fonts around and keep them for a while so we don't have to download them too often.

Using Workbox.js we can simplify the process of creating a font cache or even multiple caches for local and external fonts.

```
const fontHandler = workbox.strategies.cacheFirst({
  cacheName: "fonts-cache",
  plugins: [
    new workbox.expiration.Plugin({
      maxAgeSeconds: 30 * 24 * 60 * 60,
      maxEntries: 10
    })
  ]
});
```

When caching external fonts we want to do three things:

- Cache them for a long time (30 days in this case)
- Make sure we cache opaque responses
- Allow the browser to purge the cache if the origin's quota is exceeded

We do the last step because opaque responses are usually padded by browsers to prevent user information from leaking across domains and we want to make sure that browsers will handle full caches for the origin (your site) before the browser decides what to delete instead of asking yout.

```
const extFontHandler = workbox.strategies.staleWhileRevalidate({
  cacheName: "external-fonts",
  plugins: [
    new workbox.expiration.Plugin({
      maxAgeSeconds: 30 * 24 * 60 * 60
    }),
    new workbox.cacheableResponse.Plugin({
      statuses: [0, 200],
      // Automatically cleanup if quota is exceeded.
      purgeOnQuotaError: true
    })
  ]
});
```

The second part is to register the routes that will use our handlers. The first route defines a route for external fonts from Google fonts (from googleapis or gstatic) and uses the `extFontHandler` handler.

```
// Third party fonts
workbox.routing.registerRoute(
  /https:\/\/fonts\.(googleapis|gstatic)\.com/,
  args => {
    return extFontHandler.handle(args);
  }
);
```

The second route matches local fonts by extension for TTF, OTF, WOFF and WOFF2. This route uses the `fontHandler` handler.

```
// Fonts
workbox.routing.registerRoute(/.*\.(?:woff|woff2|ttf|otf)/, args => {
  return fontHandler.handle(args);
});
```

Use resource hints

If you're using a third party font service like Google Fonts or Typekit you should work on mitigating potential latency. Say you have a typical Google Font embed code in your <head>. You could minimize the amount of time it takes to connect with that server using the preconnect resource hint.

These hints will not load the resource but will do a DNS lookup for the host, TCP handshake, and optional TLS negotiation, all before the resource is actually requested.

```
<link rel="preconnect" href="https://fonts.googleapis.com/" crossorigin="
<link rel="preconnect" href="https://fonts.gstatic.com/" crossorigin="ano
```

A more widely compatible alternative to preconnect is dns-prefetch. It won't establish a connection to the server, but it will resolve the DNS for the specified host, which can still speed things up a bit:

```
<link rel="dns-prefetch" href="https://fonts.googleapis.com/">
<link rel="dns-prefetch" href="https://fonts.gstatic.com/">
```

Work to control font loading

Because of their size fonts tend to be some of the largest components of any web pages. According to the HTTP Archive, the sum of transfer size of all fonts (eot, ttf, woff, woff2, or otf requested by the page is 98KB for Desktop and 83.4KB for mobile.

There are several CSS and Javascript techniques to help browsers control and speed up font display and how it swaps when the web font is loaded.

The idea is to load the page as quickly as possible using fallback fonts and then swap the web font in when it's ready.

Use font-dispay

The `font-display` property of the `@font-face` rule allows the developer to better control how/when/if web fonts change the way the text looks. It is part of the CSS Fonts Module Level 4 specification and currently supported in most major desktop browsers (except Edge) and in Chrome for Android (see caniuse entry for more details).

Using the proerty, `@font-face` declarations now look like this:

```
@font-face {
  font-family: 'Open Sans'Open Sans'url("opensans.woff2") format("woff2")
        "woff2"url("opensans.woff") format("woff");
  font-display: swap;
}
```

The font display timeline is based on a timer that begins the moment the user agent attempts to use a given downloaded font face. The timeline is divided into the three periods which dictate the rendering behavior of any elements using the font face.

Font block period
: If the font face is not loaded, any element attempting to use it must render an invisible fallback font face. If the font face successfully loads during this period, it is used normally.

Font swap period
: If the font face is not loaded, any element attempting to use it must render a fallback font face. If the font face successfully loads during this period, it is used normally.

Font failure period
: If the font face is not loaded, the user agent treats it as a failed load causing normal font fallback.

Using the timeline above, we can now understand the possible values for `display-font`.

auto
: Whatever the user agent would normally do. This varies from browser to browser

block
: Gives the font face a short block period and an infinite swap period.

swap
: Gives the font face an extremely small block period and an infinite swap period.

fallback
: Gives the font face an extremely small block period and a short swap period.

optional
: Gives the font face an extremely small block period and no swap period.

I normaly use swap as the value for `font-display` as it gives me a quick render of the page and the correct fonts once they have downloaded. As with many things in fonts, test it and make sure that it does what you want it to do, your mileage may vary.

Use Font Face Observer

Font Face Observer is a font loader that allows you to work with fonts from multiple origins using a promise-based interface. It doesn't matter where your fonts come from: host them yourself, or use a web font service such as Google Fonts, Typekit, Fonts.com, and Webtype.

Font Face Observer doesn't replace `@font-face` declarations. You stil need to declare your fonts in your CSS and use `font-display` like so:

```
@font-face {
  font-family: 'NeueMontr'NeueMontreal'url('/fonts/NeueMontreal-Bold.woff2
        'woff2'url('/fonts/NeueMontreal-Bold.woff')('woff');
  font-display: swap;
}

@font-f'woff'@font-faceily: 'Fuji';
  src:  url('/fonts/F'Fuji'url('/fonts/Fuji-Light.woff2')       url('/font
```

```
  font-weight: normal;
  font-style: normal;
  font-display: swap;
}

@font-face {
  font'/fonts/Fuji-Light.woff'@font-facets/Fuji-Bold.woff2') format('woff2
  font-we'woff'url('/fonts/Fuji-Bold.woff') format('woff');
  font-weight: 700;
  font-style: normal;
  font-display: swap;
}
```

Then the page you want to use the fonts needs to load the Font Face Observer script, either locally:

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

Or from a CDN:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/fontfaceobserver/2.0.1
```

Then we create the script that will run the loader. It takes the following steps:

1. It creates new FontFaceObserver objects for each of the fonts that we want to load
2. It adds a class to the root element (`html`) to indicate that the fonts are loading
3. It uses `Promise.all` to load the fonts we defined in step 1
    1. If all the fonts load successfully we add the `fonts-loaded` class to the root element
    2. if any of the fonts fail to load then `Promise.all` will reject and the catch portion of the chain will add the `fonts-failed` class to the HTML element

```
//1
```

```
const NeueMontreal = new FontFaceObserver("Ne"NeueMontreal"const Fuji = ne
const FujiBold = new FontFaceObserver("Fuji", {
  weig"Fuji"00"
});

let html = document.documentElement;

// 2
html.classList.add("fonts-loading");

"fonts-loading"// 3
Promise.all([
    NeueMontreal.load(),
    Fuji.load(),
    FujiBold.load()
  ]).then(() => {
    // 4 successList.remove("fonts-loading");
    html.classList.add("fonts-loaded");
    console.log("All fonts have loaded.");
  })
  .catch(() => {
    // 4 failure
  "fonts-loading"// 4 failure
    html.classList.remove("fonts-loading");
    html.classList.add("fonts-failed");
    console.log("One or more fonts failed to load");
  });
```

Each class (`fonts-loaded` and `fonts-failed`) should match classes in your CSS that use web fonts and fallbacks as appropriate. Using different classes means that you don't have to wait for web font download to timeout.

Evaluate using the CSS font loading API

The CSS Font Loading Module Level 3 provides a programmatic way to handle font loading and handling of related events.

Even though the specification it's at the candidate recommendation stage, it's supported by most modern browsers (Edge is the exception) so I'm confident in

suggesting you evaluate it.

The script runs the following tasks

1. We define a `logLoaded` function to log sucessful font loads to the console
2. For each font we want to process we:
    1. Create a new `FontFace` object representing the font with the following attributes:
        1. Name
        2. URL
        3. An optional style object representing the basic characteristics (style, weight and stretch) of the font we're loading
    2. Add the font to the fonts stack
    3. Log the successful result using the `logLoaded` function
3. Using the `ready()` method as an example we make the element with class `.content` visible

```
//1
function logLoaded(fontFace) {
  console.log(fontFace.family, "lo"loaded successfully."}

//2
// These rules replace CSS @font-face declarations.
const NeueMontrealFontFace = new FontFace(
  "NeueMontreal",
  "url(/fonts/NeueMontreal-Bold.woff2)"
);
document.fonts.add(NeueMontrealFontFace);
NeueMontrealFontFace.loaded.then(logLoaded);

co"NeueMontreal"ce = new FontFace("Fuji",
      "url(/fonts/Fuji-Light.woff2)", {
  style: "normal",
  weight: "400"
});
document.fonts.add(fujiFontFace);
fujiFontFace.loaded.then(logLoaded);

const fujiBoldFontFace = new FontFace("Fuji",
```

```
      "url(/fonts/Fuji-Bold.woff2)", {
  style: "normal",
  weight: "700"
});
document.fonts.add(fujiBoldFontFace);
fujiBoldFontFace.loaded.then(logLoade"Fuji"//3
document.fonts.ready.then(function() {
  const content = document.getElementById("content");
  content.style.visibility = "visible";
});
```

Use variable fonts in browsers that support them

> In order to use variable fonts on your operating system, you need to make sure that it is up to date. Linux OSes need the latest Freetype version, and macOS prior to 10.13 (High Sierra) will not work with variable fonts.

Variable fonts are an evolution of the OpenType font specification that enables multiple variations of a typeface to be incorporated into a single file, rather than having a separate font file for every width, weight, or style; reducing the number of requests and, potentially, the file sizes for the font assets by downloading a single file. The drawback is that it provides all the variations for the given font and downloading it means you get all the variations ***whether you plan on using them or not.***

Subsetting fonts will reduce the number of characters but will not remove unused instances or any data other than glyphs.

To make these variable fonts with our current CSS we need to make some modifications. Using Roboto and its values as an example, the `@font-face` declaration looks like this:

```
@font-face: Roboto;
```

```
src:  url('/fonts/Roboto-min-VF.woff2') format('woff2'),
      'woff2'url('/fonts/Roboto-min-VF.woff') format('woff');
font-weight:  250 900;
font-width: 75 100;
font-style: -12 0;
```

We can then use values within the defined boundaries in our style sheets.

```
.my-class {
  font-weight: 450;
  font-style: -12;
}
```

We will not cover details about Variable Fonts, if you want a deeper referece, check MDN's Variable Fonts Guide.

However working with Variable fonts poses the following question:

When are variable fonts not the best option for your site/app?

Say, for example, that you're only using Roboto Regular and Bold in your application, and no Open Type features.

The variable font (compressed with WOFF2) is 978KB. Compressing individual weights of the font (regular and bold) using the same tool gives me a total of 135KB.

And even if you use the 4 basic font styles (regular, italic, bold and bold-italics), the WOFF2 fonts give you a combined weight of 270KB.

So, strictly from a performance point of view, variable fonts may not be your friend if you're not using the full feature set of a font.