



Faster fonts

This chapter covers

- Limiting the number of fonts through selection
- Rolling your own @font-face cascade
- Understanding the benefits of server compression for older font formats
- Limiting the size of fonts by subsetting
- Using the unicode-range CSS property to serve font subsets
- Managing the loading of fonts through JavaScript APIs

In the preceding chapter, you learned how to optimize images, but as it turns out, many other aspects of a page benefit from optimization as well. In this chapter, you'll explore yet another asset type commonly found in websites: fonts. Fonts can represent a significant portion of the payload of many websites, and the manner in which they're delivered is worth careful consideration.

The state of support for fonts has been somewhat fractured since their introduction into the developer's toolkit. Although support for the CSS @font-face property is ubiquitous, the font formats available for embedding have varying degrees of support.

The TrueType and Embedded OpenType formats enjoy wide support in browsers both antiquated and modern, so you may be tempted to call it a day and go with one of these. But these font formats aren't optimal for the web, because they're uncompressed. Newer formats such as WOFF and WOFF2 have a smaller footprint and are the most optimal for embedding. This doesn't mean that you shouldn't use older font formats. They *should* be part of a `@font-face` cascade, but only as last resorts for browsers that don't support newer and more optimal formats.

As you work your way through this chapter, you'll start with the basics and learn to select the fewest fonts and font variants necessary for a given page, and how to create an optimal `@font-face` cascade. You'll also explore how server compression can reduce the size of older font formats, and how to reduce the size of all font formats through subsetting. Finally, you'll take on the task of controlling the way fonts are displayed by using the CSS `font-display` property, as well as falling back to the native font-loading API in JavaScript, and then finally to the Font Face Observer library for older browsers. Let's begin!

7.1 Using fonts wisely

An optimal use of fonts begins with selection. Although font providers such as Google Fonts and Adobe Typekit do a lot to guide you in selecting fonts, at times you'll need to host fonts yourself—either because the font you need isn't available on these services, or your client has specific requirements that may prohibit you from using them.

Speaking of clients, your client from Legendary Tones is back. They've bought advertising for one of their more popular articles, and they want to spruce up that page with some nicer typefaces. In the course of this section, you'll choose the fonts and font variants you need for the site, convert them to the proper formats, and roll your own `@font-face` cascade. Before you start, you need to download the client's site and run it locally with these commands:

```
git clone https://github.com/webopt/ch7-fonts.git
cd ch7-fonts
npm install
node http.js
```

With the site downloaded and the web server running, point your browser to `http://localhost:8080` and you'll get started!

7.1.1 Selecting fonts and font variants

At a glance, the client's website design could be improved by adding fonts. The designer on the project has recommended a nice sans serif font named Open Sans. To help out, the designer placed the Open Sans font family in a subfolder of the `css` folder named `open-sans`. A glance into this folder reveals 10 styles. Clearly, you need to be choosy; otherwise, page-load times could be significantly increased.

Want to skip ahead?

If you get stuck and want to skip ahead (or you're impatient and want to see the results), you can use the `git` command. Type `git checkout -f fontface` and you'll see the end result of the work done in this section.

So how do you tease out what font variants you need? The first step is easy. The Open Sans font family comes in two styles: italic and normal. For this site's content, you need only the normal, nonitalicized variants, so this narrows the field to a slimmer selection of five fonts of varying weights, from Light to Extra Bold.

Five isn't terribly excessive, but you can weed out some of the unnecessary font weights. This requires talking with the designer to find out what the client needs. Fortunately, they've been pretty clear about their needs and have given you a small diagram showing you what font variants you should use. Figure 7.1 shows this diagram, which annotates all of the font variants (font weights, in this case).

As you can see, the variants are determined by their CSS `font-weight` property value. `font-weight` specifies how "heavy" the affected text should be. This specification can be made in presets such as `normal`, `bold`, `bolder`, or `lighter`, or through more-specific integer values in increments of 100, starting at the lightest value of 100, and ending at the heaviest value of 900. The default value for most elements is `normal`,



Figure 7.1 The client's content page with all of the font weights annotated

which is equivalent to a value of 400. Table 7.1 maps font-weight values with their corresponding Open Sans font variants and indicates whether you'll use them on the page.

Table 7.1 The available font variants in the Open Sans font family, their font-weight values, and whether they'll be used on the page

Font weight value	Font variant filename	Use on page?
300	OpenSans-Light.ttf	Yes
400	OpenSans-Regular.ttf	Yes
600	OpenSans-SemiBold.ttf	No
700	OpenSans-Bold.ttf	Yes
800	OpenSans-ExtraBold.ttf	No

Knowing which font variants you need, you can discard those you don't and use these three: OpenSans-Light.ttf, OpenSans-Regular.ttf, and OpenSans-Bold.ttf.

By being choosy and selecting only what you need, you're lightening the load for the user by serving the necessities. Fonts aren't always diminutive assets, so it behooves you to be selective. When you're finished, you can move on to writing your own @font-face cascade.

7.1.2 Rolling your own @font-face cascade

With your font variants identified and the corresponding font files selected, you can begin embedding them into the client's website. But before you can write your @font-face declarations, you need to convert the TrueType font files to the other formats that you need.

CONVERTING FONTS

Because you have only the TrueType (TTF) fonts for Open Sans available, you need to convert them to the three other formats you need. Table 7.2 lists those formats and their browser support.

Table 7.2 Font formats, along with their file extensions and browser support. Opera Mini doesn't support custom fonts.

Font format	Extension	Browser support
TrueType	ttf	All except for and IE8 and below
Embedded OpenType	eot	IE6+
WOFF	woff	All except for Android Browser 4.3 and below, and IE8 and below
WOFF2	woff2	Firefox 39+, Chrome 36+, Opera 23+, Android Browser 4.7+, Chrome and Firefox for Android, and Opera Mobile 36+

You can use various tools, available as web services or via download, for conversion. Conveniently enough, you can acquire some command-line utilities with `npm`. These are as follows:

- `ttf2eot`—Converts TTF to Embedded OpenType (EOT)
- `ttf2woff`—Converts TTF to WOFF
- `ttf2woff2`—Converts TTF to WOFF2

If fonts for your website are available only in OpenType (OTF) format, you can download the `otf2ttf` Node package to convert your OTF files to TTF prior to continuing. In the case of Open Sans, however, you're starting off with TTF, so you don't need this utility. To install these utilities globally so that you can use them anywhere on your system, you run this command:

```
npm install -g ttf2eot ttf2woff ttf2woff2
```

This could take a minute or so, depending on your internet connection and, but after it's finished, you'll be able to run these commands from any folder. After `npm` finishes, you can begin converting fonts.

Warning: Mind the licensing agreement!

The terms of use for fonts you want to use can vary from font to font, so you *need* to read the licensing agreements that come with them. Open Sans is free in every sense of the word and gives clear permissions about its use. Other font creators may require you to pay use rights. Even then, restrictions may exist on embedding. *Always* check the terms of use on the fonts you want to use and make sure that you're in compliance!

To convert the Open Sans fonts, go to the `css/open-sans` folder in your terminal, and start by generating EOT files for IE:

```
ttf2eot OpenSans-Light.ttf OpenSans-Light.eot
ttf2eot OpenSans-Regular.ttf OpenSans-Regular.eot
ttf2eot OpenSans-Bold.ttf OpenSans-Bold.eot
```

This should generate all of the EOT fonts you need. To generate the WOFF fonts with `ttf2woff`, the process and syntax are the same as with `ttf2eot`:

```
ttf2woff OpenSans-Light.ttf OpenSans-Light.woff
ttf2woff OpenSans-Regular.ttf OpenSans-Regular.woff
ttf2woff OpenSans-Bold.ttf OpenSans-Bold.woff
```

Finally, you'll create the WOFF2 files by using `tt2woff2`, which has a somewhat different syntax:

```
cat OpenSans-Light.ttf | ttf2woff2 >> OpenSans-Light.woff2
cat OpenSans-Regular.ttf | ttf2woff2 >> OpenSans-Regular.woff2
cat OpenSans-Bold.ttf | ttf2woff2 >> OpenSans-Bold.woff2
```

UNIX-like systems vs. Windows systems

In the preceding example, the `cat` command is used to output the contents of font files via the pipe operator to the `ttf2woff2` program. The equivalent of `cat` on a Windows system is `type`.

With these commands, you've generated all of the fonts you need for an optimal `@font-face` cascade. Let's move on to embedding these fonts!

BUILDING THE @FONT-FACE CASCADE

How you build the `@font-face` cascade is important. When done right, it hints to the browser which formats are available and provides the optimal format. For modern browsers, you can reap the benefits of highly compressed formats such as WOFF and WOFF2, and for older browsers, you can safely fall back to less-optimal EOT and TTF formats.

A caveat on SVG fonts

If you've had experience with embedding fonts, you might be wondering where SVG fonts fit in. The short answer is that they no longer do. SVG fonts are deprecated or in the process of deprecation in future releases of major browsers. It's best to avoid them altogether.

Let's get started with writing the `@font-face` code by opening `styles.css` in the `css` folder. The following listing shows the `@font-face` code for the first font you need, which is the regular font weight for Open Sans. You'll want to place this at the start of `styles.css`.

Listing 7.1 @font-face declaration for Open Sans Regular

```
@font-face{
  font-family: "Open Sans Regular";
  font-weight: 400;
  font-style: normal;
  src: local("Open Sans Regular"),
        local("OpenSans-Regular"),
        url("open-sans/OpenSans-Regular.woff2") format("woff2"),
        url("open-sans/OpenSans-Regular.woff") format("woff"),
        url("open-sans/OpenSans-Regular.eot") format("embedded-opentype"),
        url("open-sans/OpenSans-Regular.ttf") format("truetype ");
}
```

Weight of the typeface points to `font-weight: 400;`

Font family string of embedded font face points to `font-family: "Open Sans Regular";`

Font style of the typefaces points to `font-style: normal;`

WOFF version points to `url("open-sans/OpenSans-Regular.woff2") format("woff2"),`

WOFF2 version points to `url("open-sans/OpenSans-Regular.woff2") format("woff2"),`

EOT version points to `url("open-sans/OpenSans-Regular.eot") format("embedded-opentype"),`

TTF version points to `url("open-sans/OpenSans-Regular.ttf") format("truetype ");`

local() sources check for a font on user's system before downloading remote files points to the `local()` entries in the `src` property.

In this `@font-face` cascade, you're specifying the most optimal scenario to the least optimal one. Let's look at the `src` property: this property takes a comma-separated list of sources for the specified font. The sources are loaded in the order they are specified. You start with a `local()` declaration that checks for the font on the user's system. This is the most optimal outcome, because it saves the browser from having to download anything altogether.

If a `local` source isn't found, the browser downloads one out of a set of font formats that you converted earlier in this section. Which format is downloaded is based on the capability of the browser. You want to start off on the right foot, so you begin with the best format, which is the WOFF2 version. To achieve greater support, you need to fall back to increasingly less-optimal formats for less-capable browsers. Figure 7.2 details this process.

When the request for the WOFF2 font format fails, the browser will check for the next format in the list, which is the slightly less optimal WOFF version. Most browsers succeed by this point and load the WOFF version. Other browsers will fall back to the EOT or TTF files.

After the `@font-face` declaration is written, you'll want to modify the `font-family` property on the body selector to reference this font as the default for the document, like so:

```
font-family: "Open Sans Regular", Helvetica, Arial, sans-serif;
```

This property specifies a number of fonts in order of preference. Open Sans Regular is specified first, and is thus the preferred font. The next few are fallbacks in case the `@font-face` you're depending on doesn't load. After you've specified the font, you can reload the document in different browsers and notice that the font is now in use.

If you check the network tab in various browsers, you'll see that Firefox and Chrome use the WOFF2 file. Safari uses the WOFF file. Older browsers such as IE8 download the EOT file. If you install the TTF font files to your system, you'll notice

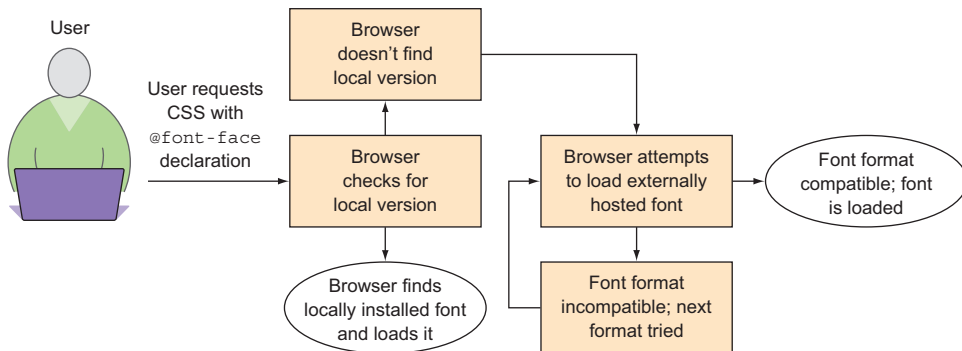


Figure 7.2 The process of a user's browser processing a `@font-face` cascade. The browser searches for a locally installed version (if specified,) and if it can't find one, it will iterate through all of the `@font-face src()` calls for various formats of the same font.

that no fonts at all download, and are instead referenced from your computer. This behavior can be circumvented by removing all `local()` sources, but this approach isn't optimal for production websites. Always be sure to test that your remote font files work properly by removing all `local()` source references, and add them before you deploy your site to production.

The last thing you need to do is to create `@font-faces` for the other two font weights, shown here.

Listing 7.2 @font-face declarations for remaining Open Sans font variants

```
@font-face{
  font-family: "Open Sans Light";
  font-weight: 300;
  font-style: normal;
  src: local("Open Sans Light"),
       local("OpenSans-Light"),
       url("open-sans/OpenSans-Light.woff2") format("woff2"),
       url("open-sans/OpenSans-Light.woff") format("woff"),
       url("open-sans/OpenSans-Light.eot") format("embedded-opentype"),
       url("open-sans/OpenSans-Light.ttf") format("truetype");
}

@font-face{
  font-family: "Open Sans Bold";
  font-weight: 700;
  font-style: normal;
  src: local("Open Sans Bold"),
       local("OpenSans-Bold"),
       url("open-sans/OpenSans-Bold.woff2") format("woff2"),
       url("open-sans/OpenSans-Bold.woff") format("woff"),
       url("open-sans/OpenSans-Bold.eot") format("embedded-opentype"),
       url("open-sans/OpenSans-Bold.ttf") format("truetype ");
}
```

After these `@font-faces` are written, you need to search `styles.css` for `font-weight` properties of 300 and 700. For selectors with `font-weight` properties of 700, add this rule:

```
font-family: "Open Sans Bold";
```

For selectors with `font-weight` properties of 300, add this rule:

```
font-family: "Open Sans Light";
```

Now the site should have the Open Sans font family displaying all text in the document in your varying font weights. Congratulations! You embedded fonts in a way that favored the smallest and best-performing font formats *first*, which lowers load times for your users. Even though older browsers receive less-optimal formats, they'll still display the custom typeface. Of course, these older formats aren't trivial to load. That's where server compression comes in!

7.2 Compressing EOT and TTF font formats

You may recall that your `@font-face` cascade starts off fine with high-performing formats such as WOFF2 and WOFF. But the two formats after that, although well-supported, are less optimal. The reason behind this is that the WOFF2 and WOFF formats are internally compressed. A compression algorithm is intrinsic to those formats, and server compression isn't necessary.

TTF and EOT font formats *aren't* compressed, so they're great candidates for server compression. Server compression can carry overhead for binary file types, but the process is worth it to speed the delivery of these less-optimized formats.

By default, these formats are compressed when you use the local Node web server with the `compression` module. But different web servers may or may not compress these formats by default, and may require further configuration. For instance, Apache web servers need to use `mod_deflate` to compress these files. This listing shows a portion of an Apache server configuration that specifies compression for TTF and EOT fonts.

Listing 7.3 Apache server configured to compress TTF and EOT fonts

```

<IfModule mime_module>
  AddType font/ttf .ttf
  AddType font/eot .eot
</IfModule>

<IfModule mod_deflate.c>
  AddOutputFilterByType DEFLATE font/ttf font/eot
</IfModule>
  
```

The diagram includes the following annotations with arrows pointing to the corresponding code sections:

- Add media type definition for TTF fonts.** (points to `AddType font/ttf .ttf`)
- Add media type definition for EOT fonts.** (points to `AddType font/eot .eot`)
- Check if mime_module is installed.** (points to the `<IfModule mime_module>` block)
- Check whether deflate module is installed.** (points to the `<IfModule mod_deflate.c>` block)
- Compress the .ttf and .eot files by their media types.** (points to `AddOutputFilterByType DEFLATE font/ttf font/eot`)

This is just one example of a web server being configured to compress fonts. Configuring other web servers to enable this same functionality requires research. The point of this section is to point out the performance gains in compressing these file types. The benefits of compressing these formats can be seen in figure 7.3, where the `OpenSans-Regular.ttf` and `OpenSans-Regular.eot` font files are compared before and after their compression.

There's a clear benefit in using server compression on these older formats. Compressing them achieves a file size similar to WOFF equivalents, so it's something of an equalizer for browsers that don't support WOFF. WOFF2 still beats out compressed TTF and EOT files, but again, not every browser supports WOFF2. The goal is to deliver the best result possible for *every* browser, and this is another method that gets you closer to achieving that goal.

For smaller assets, compression is efficient, but for larger ones such as TTF and EOT fonts, compression can take longer, resulting in a longer TTFB. Always be sure to test to see which scenario yields lower load times. For smaller files, compression may not be worth the processing time.

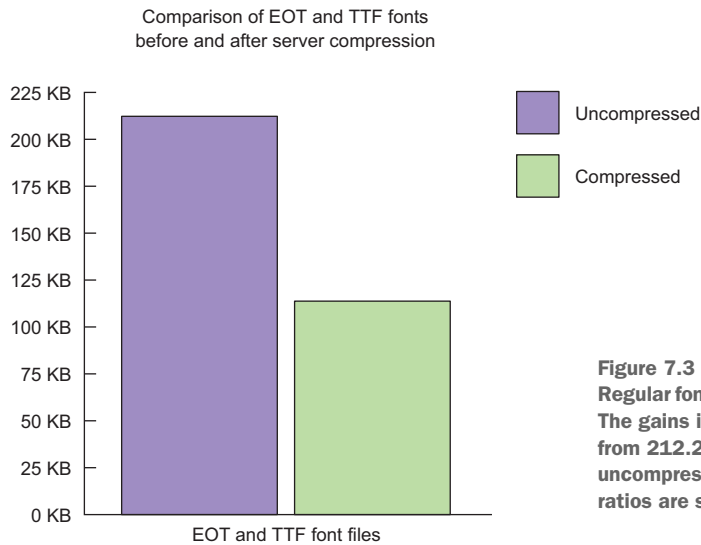


Figure 7.3 The size of the Open Sans Regular font before and after compression. The gains in this example are about 45%, from 212.26 KB to 113.76 KB, over the uncompressed versions. EOT compression ratios are similar.

7.3 Subsetting fonts

Your client is happy that the site looks spiffier, but is curious about whether there's a way to lighten the load that these fonts are adding. The unfortunate reality of adding fonts to any site is that more data is going to end up being transferred over the wire. By adding three fonts, you've added around 185 KB of extra weight on browsers that download the WOFF2 fonts. Less-capable browsers fall back to WOFF, and other versions that represent approximately 260 KB of extra data. That's a *lot*, and there must be a way to trim the fat.

Fortunately, you can use a technique to reduce font size: subsetting. *Subsetting* is the practice of selecting only the characters you need in a font file and discarding the rest. A practical application of this technique involves subsetting a font by language. For example, if a site's content is in English, Latin characters should suffice. If you've ever used Google Fonts, you've taken advantage of subsetting, because it's part of the service's settings dialog box after you choose a font, as shown in figure 7.4.

Though services like Google Fonts and Adobe Typekit offer subsetting, some scenarios prevent the use of third-party services, particularly if a site's design calls for fonts not found on font services or if a font requires a subscription fee. For these and any number of potential reasons, it's good to know how to subset fonts on your own so that you can be in charge of optimal font delivery for a website in your care. In this section, you'll learn how to manually subset fonts by using a command-line tool, as well as how to use the unicode-range CSS property to serve fonts for multilingual websites.

2. Choose the character sets you want:

☐ Greek (greek)
 ☐ Greek Extended (greek-ext)
 ☒ Latin (latin)

☐ Vietnamese (vietnamese)
 ☐ Cyrillic Extended (cyrillic-ext)

☐ Latin Extended (latin-ext)
 ☐ Cyrillic (cyrillic)

Figure 7.4 Google subsetting fonts by language

7.3.1 *Manually subsetting fonts*

Subsetting fonts can be done on the command line with a Python-based set of utilities named `fonttools`. In this section, you'll learn about Unicode ranges, install Python (if it is not preloaded), and use the `pip` package manager to install the `fonttools` library. You'll also use the `pyftsubset` command-line utility to generate subsets for your fonts.

UNDERSTANDING UNICODE RANGES

To understand the mechanism by which subsetting works, you need to know what Unicode is and how glyphs for various languages exist in predefined Unicode ranges.

If you've spent any time in web development, you've heard of Unicode, but maybe you don't know exactly what it is. *Unicode* is a standard that normalizes the way that characters for all languages are represented. More than 120,000 Unicode reservations exist for characters across various languages, and the standard is continually evolving to accommodate more.

The idea of Unicode isn't just to accommodate such a large range of characters; it's to reserve space for them in a consistent way when a Unicode character set is used. The best example of a widely used Unicode character set is UTF-8, which is the de facto standard used on the web. When a Unicode character set is used, the reserved space for a letter is the same in all documents. For example, a lowercase *p* is always located at a Unicode code point of U+0070. Figure 7.5 shows this code point among others in a table.

You need an understanding of Unicode characters because fonts use Unicode code points to reserve spaces for specific characters.

	000	001	002	003	004	005	006	007	
0	NUL 0000	DLE 0010	SP 0020	0 0030	@ 0040	P 0050	` 0060	p 0070	Glyph
1	SOH 0001	DC1 0011	! 0021	1 0031	A 0041	Q 0051	a 0061	q 0071	Unicode identifier
2	STX 0002	DC2 0012	" 0022	2 0032	B 0042	R 0052	b 0062	r 0072	
3	ETX 0003	DC3 0013	# 0023	3 0033	C 0043	S 0053	c 0063	s 0073	

Figure 7.5 A portion of a table of Unicode characters from unicode.org, showing glyphs and their code points. The lowercase *p* is identified by its Unicode code point of U+0070.

You also use Unicode code points to subset fonts. Some fonts contain far more characters than are necessary for most uses cases. If you write your content in English, it stands to reason that you don't need the Cyrillic characters that a font may provide. By subsetting, you export only the glyphs appropriate for a website's content. This results in smaller font file sizes, and as you know by now, smaller assets translate to lower page-load times.

The typical way of subsetting a font is by using a *Unicode range*. This range is expressed as two Unicode code points and includes all code points between them. A popular Unicode range is the Basic Latin range, which includes lower and uppercase characters from the English alphabet, numbers zero through nine, and a multitude of special symbols such as punctuation marks. This range is specified as U+0000 to U+007F. When this range is fed into a subsetting tool, it exports the specified range to a smaller file.

Finding other Unicode ranges

The Unicode Consortium's official site for the Unicode standard (unicode.org) contains an exhaustive listing of all the languages that the standard reserves space for. To find a Unicode range, go to <http://unicode.org/charts> and browse the listing. Click the language you're looking for, and the PDF chart for that language will have the range in the upper-left and upper-right corners of the document. For example, the range for Armenian characters is U+0530-058F.

Later in this section, you'll subset the Open Sans font to the Basic Latin Unicode range by using a command-line tool named `pyftsubset`, which is a part of the `fonttools` library. But before you can use this tool, you need to install it.

INSTALLING FONTTOOLS

Installing `fonttools` is easy. In chapter 3, you downloaded and installed Ruby so you could use the `gem` package manager to install the `uncss` utility. In this short section, you'll do something similar, only you'll be installing Python, which gives access to the `pip` package manager that you can use to install the `fonttools` package. This package is host to a command-line font subsetting utility named `pyftsubset`.

If you're a Mac user, Python comes preinstalled. Many distributions of Linux also have Python preinstalled. The easiest way to see whether Python is already installed on any system is to run the `python --version` command. If Python is installed, the version number will display on the screen, and you're good to go. The developer of `fonttools` states that the program requires Python 2.7, or 3.3 or later.

Windows users won't have the convenience of Python being preinstalled, but this is a minor obstacle. To install Python, go to <http://python.org/downloads> and get the installer. The installer will simplify the process for you, and requires nothing more than going through a series of steps.

After Python is installed, ensure that the `pip` package manager is available by typing `pip -V` at the command line. If you receive an error and don't see a version number, you

need to install pip. Because Python is available by this point, this is easily remedied by running the `easy_install pip` command. After it's finished, the pip installer will be available, and you can install the `fonttools` package by typing `pip install fonttools`.

After `fonttools` is installed, you can check whether the `pyftsubset` utility is available by entering `pyftsubset --help` at the command line. If your screen buffer fills up with help text, the utility is installed, and you're ready to start subsetting fonts!

SUBSETTING FONTS WITH PYFTSUBSET

Now that `fonttools` is installed and `pyftsubset` is working, it's time to get down to business. Because the content is in English, you want to subset the Basic Latin Unicode range. This range contains all the letters and numbers used in the English alphabet, as well as all the symbols you'll need, such as punctuation. When you look up the Basic Latin range on the official Unicode website, you find that the range is U+0000 to U+007F. This is the information you'll feed to the `pyftsubset` utility to generate your subsets.

To begin subsetting fonts with this utility, open a terminal window and traverse to the `css/open-sans` directory within the client website folder. In order to work, `pyftsubset` requires TTF, OTF, or WOFF files. For simplicity, you'll subset the original TTF files and use the converters from section 7.1 to reconvert the subsetting TTF fonts to the other formats that you need.

After you're in the correct folder, you'll start by subsetting the `OpenSans-Regular.ttf` font file with the following command:

```
pyftsubset OpenSans-Regular.ttf --unicodes=U+0000-007F --output-  
file=OpenSans-Regular-BasicLatin.ttf --name-IDs='*'
```

Quite a bit is going on in this command, so let's break it down. Figure 7.6 diagrams each of these options.

After a short wait, the program will finish and output `OpenSans-Regular-BasicLatin.ttf` as specified in the `--output-file` flag. If you look at the size of this file as compared to `OpenSans-Regular.ttf`, you'll observe that you've shrunk the file by about 90%, from 212.26 KB to 17.68 KB. This is a *huge* reduction in the font's overall size. What's more, this isn't even the optimal font format; Open Sans has a lot of characters because

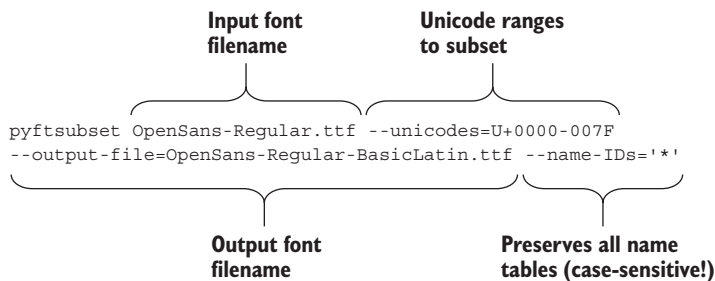


Figure 7.6 Subsetting a font with `pyftsubset`. The input file is specified first, followed by the Unicode range of characters you want to subset from the input font, and then the output filename. The last option is used to preserve all name table entries, which ensures better compatibility with the font converters.

of its broad support for many languages. You won't always get this kind of mileage from subsetting your fonts, but it can pay to take time to see what's possible.

Before you declare victory, you still have to convert this subsetting font to EOT, WOFF, and WOFF2 formats with these commands:

```
ttf2eot OpenSans-Regular-BasicLatin.ttf OpenSans-Regular-BasicLatin.eot
ttf2woff OpenSans-Regular-BasicLatin.ttf OpenSans-Regular-BasicLatin.woff
cat OpenSans-Regular-BasicLatin.ttf | ttf2woff2 >>
    OpenSans-Regular-BasicLatin.woff2
```

After you finish converting all fonts, repeat the same subsetting procedure by using `pyftsubset` for `OpenSans-Bold.ttf` and `OpenSans-Light.ttf`, and convert those files to their respective EOT, WOFF, and WOFF2 versions. Then you need to update the `@font-face` sources in `styles.css` to reference the new subsetting font files.

On special symbols

When subsetting a font, bear in mind the content of the site that the font is destined for. A site about coffee and coffee products may use words from other languages—such as *café*, which has an accented e character. The Basic Latin range lacks these characters, so you may want to take care to include the glyphs you may need down the road.

With this effort, you've managed to peel quite a bit off the fonts' file sizes. Depending on the font's format, you've slimmed your fonts down anywhere from 85% to 90% of their original size. This translates into a rather large improvement in page-load time, as you can see in figure 7.7.

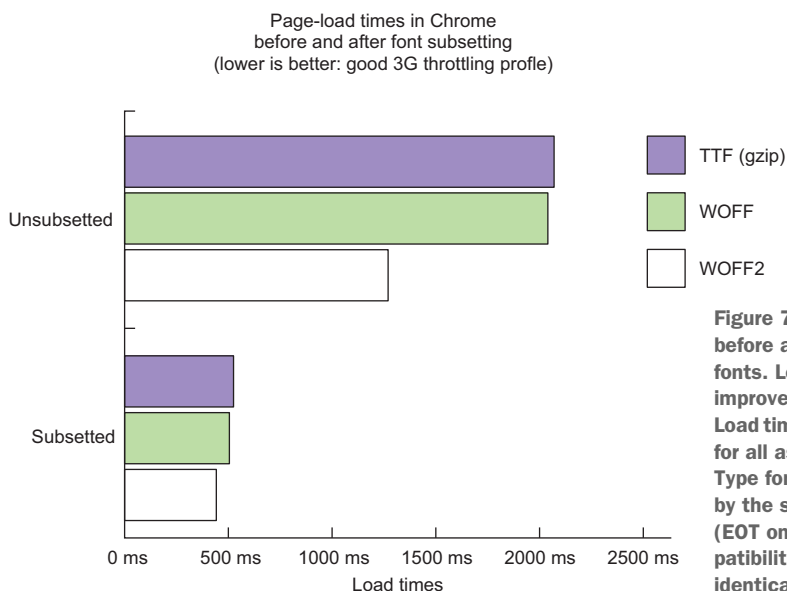


Figure 7.7 Load times before and after subsetting fonts. Load times are improved by well over 200%. Load times include load times for all assets on site. True-Type fonts were compressed by the server in these trials. (EOT omitted due to incompatibility; file sizes are nearly identical to TTF.)

By now, you're probably feeling unstoppable. The next challenge is subsetting by language with the `unicode-range` property in your `@font-face` cascade. With this property, you can target specific languages via Unicode ranges as you did with `pyftsubset`.

7.3.2 *Delivering font subsets by using the unicode-range property*

The client wants to present the content in more than one language. The site is, for whatever reason, particularly popular in some Eurasian countries, especially Russia.

This presents a bit of a challenge, because the client needs to get the content translated. But you can work while the content translators do their thing, because the client has given you Russian placeholder copy to work with.

The problem is that this site needs to be able to serve the Cyrillic characters that the Russian language uses. You can achieve this in one of two ways:

- You can serve up the entire unsubsetted font file so that all languages will have the characters they need at their disposal, no matter the circumstances.
- You can serve up only the subsets that are needed by that page.

Which way do you think is better? If you guessed option two, you're correct. As you saw in the previous section on manual subsetting, serving the unsubsetted font hinders performance, so you'll want to subset for other languages appropriately. In this case, you don't want to force English-speaking users to download the Cyrillic subset of the font.

This is where the `unicode-range` CSS property comes in. This property is specified within a `@font-face` definition, and its value is a range and/or set of Unicode code points in the same format as those you fed into the `pyftsubset` program. If the browser detects that the content of the page contains characters that fall into this range, it'll download the font. If it doesn't, it won't download the font.

This property doesn't have universal browser support, so if you intend to use it, you may need to have a fallback strategy. No well-maintained polyfills exist for `unicode-range` at this time, so fallbacks often require an alternate approach rather than a fallback. I'll demonstrate an alternate approach later in this section.

In this section, you'll learn how to use `pyftsubset` to generate a new font subset containing the Cyrillic characters you need. After you convert them, you'll then embed them in a new `@font-face` and use the `unicode-range` property to inform the browser of which characters the associated `@font-face` applies to. Then, you'll discuss a fallback method using JavaScript.

To get started in this subsetting exercise, you need to switch over to a new branch of code by using `git`. Type in the command `git checkout -f unicoderange`. The first thing you'll see is that there are two HTML files: the English version of the article that you've seen before (`index-en.html`), and a Russian version (`index-ru.html`) using Cyrillic characters. Let's get started!

GENERATING THE CYRILLIC FONT SUBSETS

Before you can use `unicode-range` to deliver the proper font subset to Russian viewers, you need to create the subset of those characters by using `pyftsubset`.

As you may suspect, the method for generating the Cyrillic subset with this program is much the same as it was when you generated the Basic Latin subset. The only difference is that you need a different Unicode point range to grab the characters.

Whereas the Basic Latin Unicode range is simple, the Cyrillic Unicode range is complex. It's three distinct comma-separated ranges that you pass to `pyftsubset`'s `--unicodes` option. I'll provide these ranges for you in this example, but they can be found on the Unicode website. To create the Cyrillic subsets for the `OpenSans-Regular.ttf` font file, type in this command from within the `css/open-sans` folder at the command line:

```
pyftsubset OpenSans-Regular.ttf --unicodes=U+0400-045F,U+0490-0491,U+04B0-04B1 --output-file=OpenSans-Regular-Cyrillic.ttf --name-IDs='*'
```

The only differences between this command and the one you used to generate the Basic Latin subset for Open Sans Regular are the Unicode ranges you pass to the `--unicodes` option and the output filename. After this finishes, you'll see that a new file by the name of `OpenSans-Regular-Cyrillic.ttf` will be in the folder. As before, you need to convert this font into the EOT, WOFF, and WOFF2 versions:

```
ttf2eot OpenSans-Regular-Cyrillic.ttf OpenSans-Regular-Cyrillic.eot
ttf2woff OpenSans-Regular-Cyrillic.ttf OpenSans-Regular-Cyrillic.woff
cat OpenSans-Regular-Cyrillic.ttf | ttf2woff2 >> OpenSans-Regular-Cyrillic.woff2
```

After these conversions finish, repeat the process of generating Cyrillic subsets for `OpenSans-Light.ttf` and `OpenSans-Bold.ttf` to `OpenSans-Light-Cyrillic.ttf` and `OpenSans-Bold-Cyrillic.ttf`, respectively. Then convert those files to the formats you need for their respective `@font-face` definitions. Now you're ready to learn about the `unicode-range` property and use it in your new Cyrillic `@font-faces`!

USING THE UNICODE-RANGE PROPERTY

Using the `unicode-range` property in CSS isn't much different from passing Unicode ranges to the `--unicodes` option when you subset fonts with `pyftsubset`. If you open `styles.css` from the client's website in a text editor and look at the `@font-face` declarations, you'll notice that `unicode-range` has already been used for the Basic Latin subsets:

```
unicode-range: U+0000-007F;
```

The format for this property is simple but flexible. It accepts any number of single Unicode code points, ranges, and/or wildcards. Variations of this property's use are shown here.

Listing 7.4 unicode-range values

```

/* Single value */
unicode-range: U+0026;

/* Range */
unicode-range: U+0000-007F;

/* Wildcard Range */
unicode-range: U+002?

/* Multiple Values */
unicode-range: U+0000-007F, U+0100, U+02??;

```

Single Unicode code point expressed as a singular value

Range of Unicode code points separated by a hyphen








Wildcard range of Unicode code points is specified using question marks.

Multiple values separated by commas

With a Unicode range specified for the Latin subset, it stands to reason that if you visit the Russian version of the page in your browser at <http://localhost:8080/index-ru.html>, the Basic Latin subset shouldn't load at all, correct? As you can see in figure 7.8, this isn't the case.

“Wait a minute! What gives?” might be your first thought, but the fact is that this version of the site *is* using characters from the Basic Latin subset you created. This font subset contains things that are common not just in English but in Russian, too—things like punctuation and numerical characters. For numerous reasons, characters in the Basic Latin Unicode range are common in many languages. So the `unicode-range` property in this instance is working exactly as it ought to: it's getting the font subset only when it's needed!

What you want to do is prevent the Cyrillic font subsets from being downloaded on pages that *don't* need them. To do this, you need to establish a new `@font-face` for each of the new Cyrillic font subsets with their own `unicode-range` value. The

Name	Method	Status
 index-ru.html	GET	200
 styles.css	GET	200
 logo.svg	GET	200
 ohm.svg	GET	200
 OpenSans-Light-BasicLatin.woff2	GET	200
 OpenSans-Regular-BasicLatin.woff2	GET	200
 OpenSans-Bold-BasicLatin.woff2	GET	200

Loaded fonts {

Figure 7.8 The Basic Latin font subsets are loaded on the Russian version of the page, despite having a `unicode-range` property set to use these fonts only for pages displaying characters from the Basic Latin subset.

following listing shows a `@font-face` declaration for the Cyrillic subset of Open Sans Regular that you'll add to `styles.css`.

Listing 7.5 `@font-face` for Open Sans Regular Cyrillic subset

```
@font-face{
  font-family: "Open Sans Regular";
  font-weight: 400;
  font-style: normal;
  src: local("Open Sans Regular"),
      local("OpenSans-Regular"),
      url("open-sans/OpenSans-Regular-Cyrillic.woff2") format("woff2"),
      url("open-sans/OpenSans-Regular-Cyrillic.woff") format("woff"),
      url("open-sans/OpenSans-Regular-Cyrillic.eot")
      ➤ format("embedded-opentype"),
      url("open-sans/OpenSans-Regular-Cyrillic.ttf") format("truetype");
  unicode-range: U+0400-045F,U+0490-0491,U+04B0-04B1;
}
```

Source formats for the Cyrillic font subset

Multiple character sets can be used within same font-family.

unicode-range the font applies to.

After adding this new font to `styles.css`, add the remaining `@font-face` declarations for the Cyrillic subsets of Open Sans Light and Open Sans Bold. When adding this, make sure to update the `font-family` and `local()` source names appropriately. They'll be the same as the values for the Basic Latin subsets of those font variants.

When you complete the remaining `@font-face`s, you'll be able to see how `unicode-range` affects font delivery. You have `index-ru.html` open, so open `index-en.html` in another tab and check the network utility in the Developer Tools for each page in Chrome. A comparison of the Network tab output for the English and Russian versions of the page can be seen in figure 7.9.

You'll notice that the Russian page pulls down the Cyrillic subsets with the Basic Latin ones, whereas the English page, not having any need for Cyrillic characters, heads the `unicode-range` property and ignores that subset.

This technique has utility for any multilingual website with languages that use different character ranges. Most western languages such as German, Spanish, and French

Name	Method	Status	Name	Method	Status
<input type="checkbox"/> OpenSans-Light-Cyrillic.woff2	GET	200	<input type="checkbox"/> OpenSans-Light-BasicLatin.woff2	GET	200
<input type="checkbox"/> OpenSans-Regular-Cyrillic.woff2	GET	200	<input type="checkbox"/> OpenSans-Regular-BasicLatin.woff2	GET	200
<input type="checkbox"/> OpenSans-Bold-Cyrillic.woff2	GET	200	<input type="checkbox"/> OpenSans-Bold-BasicLatin.woff2	GET	200
<input type="checkbox"/> OpenSans-Light-BasicLatin.woff2	GET	200			
<input type="checkbox"/> OpenSans-Bold-BasicLatin.woff2	GET	200			
<input type="checkbox"/> OpenSans-Regular-BasicLatin.woff2	GET	200			

Russian version

English version

Figure 7.9 The fonts downloaded by the Russian version of the page (left) as compared to the English version (right), even though they both use the same style sheet. The `unicode-range` property detects whether any characters in the document exist in the defined ranges, and if so, the related `@font-face` resource is served up.

do fine with a more inclusive Latin subset, but languages such as Greek and Russian benefit from subsetting because of their different alphabets. Asian languages especially benefit from this method, because Asian languages can have thousands of characters.

Not all browsers support this property, so it pays to think about how to fall back to methods that are more compatible with older browsers.

FALLBACKS FOR OLDER BROWSERS

Although `unicode-range` is a great feature, its overall support isn't universal. Although well supported in newer WebKit browsers and Firefox, others may not have support for it by the time you read this. These browsers will ignore the `unicode-range` property and download all of the font subsets found in a CSS file without discretion. Figure 7.10 shows Safari 9's behavior with the English version of the page; all fonts load on the page as though the `unicode-range` property never existed.

So what can you do for browsers that don't support `unicode-range`? One possible approach is to create broader subsets if the increased number of glyphs won't be too detrimental to page performance. Both versions of the page would still download the extra characters, but instead of six requests over three font variants, the weight would be spread across three requests over the same number of font variants.

This approach doesn't work for content in languages such as Japanese, in which the number of glyphs can push font file sizes into the massive category. Although developers who code for sites in these languages may expect to have a large amount of a site's payload dedicated to fonts, it's not okay to push these subsets onto users who don't need them. It's not a nice thing to do to your visitors. The solution, therefore, lies in JavaScript.

For multilingual sites, developers use the `<html>` tag's `lang` attribute to define the document's language. These language codes conform to the ISO 639-1 standard. In `index-ru.html`, this looks like `<html lang="ru">`. You can write a small bit of inline JavaScript that checks the language code in this tag. If the language code is what you're looking for, you load a separate, smaller style sheet that contains the `@font-face` declarations for the subsets that you want to defer loading.

To implement this for Russian content, you start by moving the Cyrillic `@font-face` definitions into a separate CSS file named `ru.css`. You then reference `ru.css` with a `<link>` tag containing a placeholder `data-href` attribute that stores its location, along with a `data-lang` attribute that stores the content language code it's intended for.







Name	Domain	Type	Method
 OpenSans-Regular-Cyrillic.woff	localhost	Font	GET
 OpenSans-Regular-BasicLatin.woff	localhost	Font	GET
 OpenSans-Light-Cyrillic.woff	localhost	Font	GET
 OpenSans-Light-BasicLatin.woff	localhost	Font	GET
 OpenSans-Bold-Cyrillic.woff	localhost	Font	GET
 OpenSans-Bold-BasicLatin.woff	localhost	Font	GET

Figure 7.10 Cyrillic subsets loading on the English version of the page, regardless of the `unicode-range` property. The behavior shown is in Safari.

This prevents the CSS from being loaded at all until it's evaluated by the following `<script>` block. If the script determines that the `<html>` `lang` attribute's value matches that of the `<link>` tag's `data-lang` attribute, it'll download and parse that style sheet immediately. This listing shows this mechanism in action.

Listing 7.6 Deferring loading of font subsets with JavaScript

```

<!doctype html>
<html lang="ru">
  <head>
    <title>Легендарные Тонизирует - Этого не случится.</title>
    <link rel="stylesheet" href="css/styles.css" type="text/css">
    <link rel="stylesheet" data-href="css/ru.css"
      data-lang="ru" type="text/css">
    <script>
      (function(document) {
        var documentLang = document.querySelector("html")
          .getAttribute("lang"),
            linkCollection = document
              .querySelectorAll("link[data-href]");
        for(var i = 0; i < linkCollection.length; i++){
          var linkLang = linkCollection[i]
            .getAttribute("data-lang"),
              linkHref = linkCollection[i]
                .getAttribute("data-href");
          if(documentLang === linkLang){
            linkCollection[i].setAttribute("href", linkHref);
          }
        }) (document);
      }
    </script>
    <noscript>
      <link rel="stylesheet" href="css/ru.css" type="text/css">
    </noscript>
  </head>

```

Annotations:

- lang attribute is set to a language code of "ru" (Russian).** (points to `<html lang="ru">`)
- @font-faces for the Cyrillic subsets are moved to another CSS file.** (points to `data-href="css/ru.css"`)
- <html> tag's lang attribute is saved to a variable.** (points to `document.querySelector("html")`)
- <link> tag collection is looped over.** (points to `linkCollection`)
- <link> tag's data-lang attribute is captured.** (points to `linkCollection[i].getAttribute("data-lang")`)
- data-lang attribute is checked to see if it matches the document language.** (points to `if(documentLang === linkLang)`)
- <link> tags with data-href attributes are saved to a variable.** (points to `linkCollection`)
- <link> tag's data-href attribute is captured.** (points to `linkCollection[i].getAttribute("data-href")`)
- Appropriate <link> tag's data-href attribute is changed to an href attribute.** (points to `linkCollection[i].setAttribute("href", linkHref)`)
- A <noscript> fallback to download the font subsets** (points to `<noscript>`)

Because this `<script>` block is near the beginning of the document, the browser will discover it almost immediately, so it's executed sooner. This keeps delays to a minimum.

The script also has the capability of handling multiple `<link>` tags that follow the `data-href` pattern. This gives the code the flexibility to contain references to as many `<link>` tags as necessary for additional font subsets. You could place this code in the `<head>` of every single page of a multilingual website, and only the CSS and font subsets you need for that page's language would be loaded.

You should consider users who may have JavaScript disabled. To cover this, you'll serve the font subset via a `<link>` tag nested in a `<noscript>` element. This fallback isn't optimal because it won't discriminate based on the `<html>` `lang` attribute's value, but it'll ensure that users get the font subset necessary for the page.

Name	Name
 index-en.html	 index-ru.html
 styles.css	 styles.css
 logo.svg	 ru.css
 ohm.svg	 logo.svg
 OpenSans-Regular-BasicLatin.woff	 ohm.svg
 OpenSans-Light-BasicLatin.woff	 OpenSans-Regular-Cyrillic.woff
 OpenSans-Bold-BasicLatin.woff	 OpenSans-Regular-BasicLatin.woff
	 OpenSans-Light-Cyrillic.woff
	 OpenSans-Light-BasicLatin.woff
	 OpenSans-Bold-Cyrillic.woff
	 OpenSans-Bold-BasicLatin.woff

English content page

Russian content page

Figure 7.11 The contents of the network tab in Safari on both the English (left) and Russian (right) versions of the content page, with your fallback script enabled on each page. The English version downloads only the fonts it needs, whereas the Russian version grabs the additional `ru.css` and the font subsets contained therein.

The effect on your client’s website is that you can have the same end result as with the `unicode-range` property, only in every browser. Figure 7.11 illustrates the effect of this script on both the Russian and English versions of the article as loaded in Safari.

Is this solution remotely as optimal as `unicode-range`? Nope! It simply illustrates that JavaScript solutions can be crafted if there’s a genuine concern. Simpler JavaScript solutions for simpler scenarios could be written. A server-side approach may make more sense to you as well. For example, you could store the language code in a cookie, and use a server-side language such as PHP to inject the `<link>` element for the font subset into the document based on a condition.

As time marches on, `unicode-range` will gain more support until it’s eventually preferable to allow older browsers less-optimal experiences. The idea is that you have options in case `unicode-range` *isn’t* one of them.

In the next and final section of this chapter, you’ll learn to control the way fonts are displayed via CSS and JavaScript mechanisms.

7.4 Optimizing the loading of fonts

Loading any asset on a website involves pitfalls that vary based on the asset’s type. For instance, loading CSS with the `<link>` tag blocks rendering until the style sheet is downloaded and parsed and the styles are applied to the document. `<script>` tags that reference external JavaScript files similarly block rendering of the page when they’re placed toward the top of the document.

Fonts are no different, and loading them causes no shortage of issues that can have ramifications on the readability of your site. In this section, you’ll learn about the visual anomalies that can occur as fonts load. You’ll then learn how to control the way fonts are displayed by using the `font-display` CSS property, and then fall back to using the

JavaScript-based font-loading API when `font-display` is unavailable. If neither method is available to the browser, you'll learn how to fall back to a third-party script to achieve the same results.

7.4.1 Understanding font-loading problems

The Legendary Tones owners have sent an email saying that, although they're happy with the way the fonts look, they're noticing that text on the page seems to take a while to render on slow connections. This is understandable, but the reality is that this is how some browsers work when it comes to downloading fonts. The phenomenon the client is referring to is called the *Flash of Invisible Text*.

Flash of Invisible Text (henceforth referred to as *FOIT*) is similar to the Flash of Unstyled Content (FOUC) anomaly, only instead of unstyled content, you're dealing with text being invisible until the document's fonts are fully loaded. It's noticeable on even fast connections if you pay close attention. As connection speed decreases and network latency increases, the problem becomes more evident. Mobile devices on slow mobile networks such as 2G and 3G are more susceptible to this phenomenon. You can see this problem in figure 7.12.

This seems like an annoying bug, but it's how browsers are designed to behave. Browsers wait to render text while downloading fonts in order to avoid an effect known as the *Flash of Unstyled Text (FOUT)*. FOUT is similar to FOUC talked about in chapter 3, except that instead of an unstyled page, text initially loads in a system typeface and then suddenly re-renders with the custom typeface applied. The browser will hide text for only so long while a font loads, and when this period of time is exceeded, the unstyled text shows up before the font has finished loading. After the font has loaded, the unstyled text is styled. This is shown in figure 7.13.

The browser's intentions are good, but if a connection stalls, the user could be left waiting for 3 seconds or longer to see text on the page. In browsers such as Safari, the content may never show if the request stalls. If the user aborts loading the page, or the font assets otherwise fail to load, the content may remain permanently invisible until

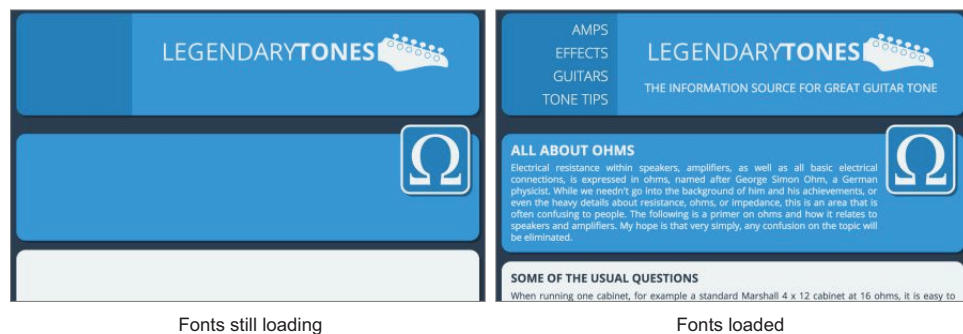


Figure 7.12 As a page loads embedded fonts, the text is initially invisible (left) until the fonts fully load, at which point the text styled in those font faces appears.

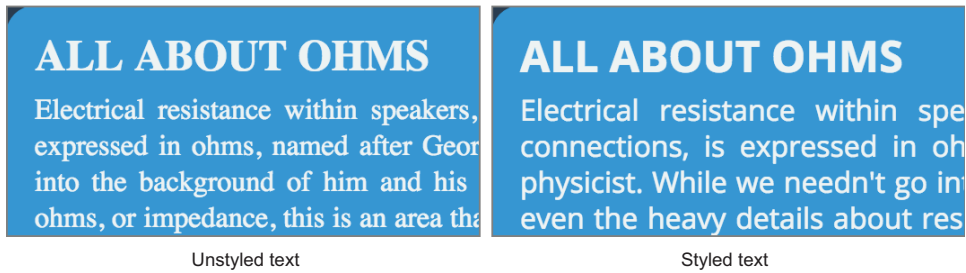


Figure 7.13 When the download time for a font is too long, the text will eventually become visible, but is unstyled because of the still-loading font resource (left). After all of the fonts load, the text will become styled (right). This is known as FOUT.

the page is refreshed. This holds true even if the developer of the website has specified fallbacks to system fonts in the `font-family` properties. Newer versions of Chrome try to mitigate this issue automatically, but it's not perfect, nor does every browser try to remedy the issue under the hood.

So what can you do? You embrace the FOUT on page load, and use the CSS `font-display` property. This property lets you ensure that your content will appear as soon as possible, and won't leave your users in a lurch with hidden text. Let's get started!

7.4.2 Using the CSS `font-display` property

The `font-display` property in CSS provides a convenient way to control the display of fonts with a minimum of effort. Though this approach is limited to Chrome browsers at the time of this writing, it's the best first resort in your plan to control the display of fonts. To get started, let's check out a new branch of the website code with `git`:

```
git checkout font-display
```

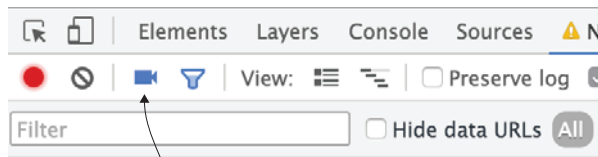
Want to skip ahead?

If you get stuck or want to skip ahead to see how the completed font-loading API code looks and behaves, you can do so by entering `git checkout -f font-display-complete` at the command line.

After the code has downloaded to your computer from GitHub, open `index.html` and `styles.css` in your text editor.

CONTROLLING HOW AND WHEN FONTS DISPLAY

To get started, open the Developer Tools in Chrome and change your network throttling profile to Regular 3G. You'll be able to see the FOIT effect quite easily. As a general rule, the slower the connection, the more noticeable this effect. To pinpoint the moment when fonts become visible on the screen, you can toggle the Capture Screenshots button in the Network tab, as shown in figure 7.14.



Capture Screenshots toggle

Figure 7.14 The toggle button to capture screenshots in Chrome Developer Tools

When this button is toggled and the page is reloaded, a roll of screenshots of the page load populates above the network request waterfall chart. With this, you can pinpoint the exact moment that fonts appear on the page. With the Regular 3G throttling profile selected, the text doesn't appear until about 875 milliseconds from the time the page begins to download. This is okay, but depending on connection speed and latency, this could fluctuate. The goal is to allow the user to see content as soon as possible.

Learn a bit more about font-display on the web!

You can learn a little bit more about the `font-display` property, including how to detect support for the property, by checking out an article I wrote for CSS-Tricks at <https://css-tricks.com/font-display-masses>.

One way to control this behavior is via the `font-display` property in CSS. Although not universally supported, this property gives you a great degree of control over the way fonts are displayed. This property is placed inside a `@font-face` declaration and accepts one of the following values:

- `auto`—The default value. In most browsers, this is analogous to `block`.
- `block`—Blocks the rendering of text until the associated font is loaded. This is the effect described in the previous section, and what you're trying to overcome.
- `swap`—Fallback text is shown first. After the font is loaded, the custom typeface is swapped in.
- `fallback`—A compromise between `auto` and `swap`. For a short time (roughly 100 ms), text is invisible. If this elapses and the font isn't yet loaded, the fallback text appears. After the font is loaded, the custom typeface is swapped in.
- `optional`—Exactly like `fallback`, except that the browser is given more latitude to decide whether a font is downloaded or applied. This setting kicks in when the user's internet connection is sufficiently slow. This setting is particularly useful for sites where custom typefaces are considered entirely optional.

On the *Legendary Tones* article page, you'll use a `font-display` value of `swap`. To set this property, open `styles.css` in the `css` folder and locate the `@font-face` declarations at the top of the file. Inside the first `@font-face` declaration, add the `font-display` property.

Listing 7.7 Using the font-display property

```
@font-face{
  font-family: "Open Sans Light";
  font-weight: 300;
  font-style: normal;
  src: local("Open Sans Light"),
       local("OpenSans-Light"),
       url("open-sans/OpenSans-Light-BasicLatin.woff2") format("woff2"),
       url("open-sans/OpenSans-Light-BasicLatin.woff") format("woff"),
       url("open-sans/OpenSans-Light-BasicLatin.eot")
         format("embedded-opentype"),
       url("open-sans/OpenSans-Light-BasicLatin.ttf") format("truetype");
  font-display: swap;
}
```

font-display used inside an @font-face declaration

With this one property in place, reload the page on a slower network-throttling profile and you'll see that the text displays progressively without being hidden by the browser.

This setting represents the easiest possible solution for controlling the rendering of fonts when you have control over the CSS that delivers them, but it doesn't have wide support in browsers, nor does it allow you to control the way that fonts are displayed when they're referenced from third-party providers such as Typekit or Google Fonts. That's when you can fall back to a more widely supported JavaScript solution, known as the font-loading API.

7.4.3 Using the font-loading API

The *font-loading API* is a JavaScript-based tool that controls how fonts are loaded. Its open-ended nature gives you a lot of latitude in determining how to apply typefaces to a document, whether they're hosted on your own server or with a font provider such as Google Fonts. To use the font-display CSS property discussed in the previous section, you need to have control over the CSS that serves fonts. This is a luxury you don't have when you use third-party font providers. The font-loading API gives you a similar ability to control the display of fonts regardless of their origin, but through JavaScript rather than CSS.

Before you get started, you need to use `git` to switch to a new branch of code. Go into your terminal window and type `git checkout -f font-loader-api`. When this is complete, you're ready to go.

Want to skip ahead?

If you want to skip ahead and see the work at the end of this section, you can do so by typing `git checkout -f font-loader-api-complete`.

To get started, look in `styles.css` for `font-family` definitions that use custom typefaces. For this site, you have three font variants: Open Sans Light, Open Sans Regular, and

Open Sans Bold. Table 7.3 lists these font-family definitions and the selectors they apply to.

Table 7.3 Embedded fonts' font-family property values and their associated CSS selectors

font-family	Associated CSS selectors
Open Sans Light	.navItem a
Open Sans Regular	body
Open Sans Bold	.articleTitle .sectionHeader

You'll use this information to do two things. The first is to replace the font-family properties for all of these with system fonts. For this website, you'll use the following property and value for the associated selectors in table 7.3:

```
font-family: "Helvetica", "Arial", sans-serif;
```

This removes the Open Sans font families from the page, which allows the content to be seen immediately, because these fonts aren't downloaded from the web server.

Second, you nest these selectors under a class that you'll put on the `<html>` element when the fonts have been loaded. But before you write the font-loading script, you'll write the CSS for applying the Open Sans fonts after this class is applied to the `<html>` element. This is shown next.

Listing 7.8 Controlling font display by using the fonts-loaded class

```
.fonts-loaded body{
    font-family: "Open Sans Regular";
}

.fonts-loaded .navItem a{
    font-family: "Open Sans Light";
}

.fonts-loaded .articleTitle,
.fonts-loaded .sectionHeader{
    font-family: "Open Sans Bold";
}
```

By placing this small snippet of CSS at the end of `styles.css`, you can control *when* you apply the typefaces you're loading with the font-loading API. Because you've specified the system fonts as the initial font set, the unstyled text will be immediately visible when the page first renders, and the custom typefaces will be applied after they've loaded.

Open `index.html` in your text editor to start writing your font-loading script. After the `<link>` tag that imports `styles.css` (which imports our fonts,) add the code in the following listing.

Listing 7.9 Using the font-loading API

```

(function (document) {
  if (document.fonts) {
    document.fonts.load("1em Open Sans Light");
    document.fonts.load("1em Open Sans Regular");
    document.fonts.load("1em Open Sans Bold");

    document.fonts.ready.then(function (fontFaceSet) {
      document.documentElement.className += " fonts-loaded";
    });
  } else {
    document.documentElement.className += " fonts-loaded";
  }
})(document);

```

Font-loading API uses the `load()` method to load the fonts.

Check for the presence of the font-loading API.

ready.then method runs when all the specified fonts have loaded.

fonts-loaded class added to the `<html>` element.

If the font-loading API is unavailable, add the fonts-loaded class to the `<html>` element.

Because you're managing three font variants, you initiate a separate call to load each typeface. The core property of the font-loading API that you're using to achieve this is the `font` object's `load` method. Rather than using the API to explicitly load a font by its URL, you rely on CSS to define `@font-faces` for the document. But just because the `@font-faces` are defined doesn't mean that the browser downloads those fonts. Browser behavior is well optimized, and modern browsers will inspect the document to see whether any defined `@font-faces` are in use. If they are, they'll be downloaded, but because you initially set all of your `font-family` values to use system fonts, none of the font variants are downloaded until you tell the browser to do so via the `load()` method shown in listing 7.9.

When all of the fonts have loaded, the `fonts-loaded` class is added to the `<html>` element. This defeats the browser's initial FOIT, allowing the content to be read as soon as the document is loaded and the CSS is applied. *Then* the fonts are applied to the document when they're available. This ensures that no matter what may happen on the user's end, the text will be visible as soon as possible, and if a font fails to load, the text will remain that way.

One drawback of this method is that it *does* cause a repainting of text elements on the page, but the increased accessibility is worth the trade-off. If you choose system fonts that are similar to custom typefaces, document reflow can be minimized.

OPTIMIZING FOR REPEAT VISITORS

Our solution works great for first-time visitors, but you need to optimize for repeat visits when the fonts are already in the user's browser cache. With the code as it is now, the FOUT occurs on subsequent page visits even with the font in the cache. You can overcome this by using a cookie and modifying your font-loading code slightly.

Let's modify two parts of the code you've written. On the line where you check for the font-loading API, you add a condition to check for the presence of a cookie:

```
if (document.fonts && document.cookie.indexOf("fonts-loaded") !== -1) {
```

This change adds a check for a cookie you'll define later. This cookie's name is `fonts-loaded` and it has no particular value. You check for its existence by using the `indexOf` string method, which returns (somewhat unintuitively) a value of `-1` if the search string isn't found. This ensures that the font-loading code you've written runs only if the font-loading API is available *and* a `fonts-loaded` cookie hasn't been set.

But now you need to *set* that cookie somewhere. To do that, you add this bit of code after the line where you add the `fonts-loaded` class to the `<html>` element:

```
document.cookie = "fonts-loaded=";
```

This adds an empty cookie by the name of `fonts-loaded` for the current domain. When this cookie is set, and the user navigates to subsequent pages, the font-loading code doesn't run again. The `else` condition therefore takes effect immediately and adds the `fonts-loaded` class to the `<html>` element.

This reintroduces the FOIT, but the risks of the effect are now mitigated because the fonts are in the user's browser cache. This is okay so long as you can be assured that the fonts will load. Now that the fonts are in the cache, the assurance can be made that the effect won't block the user from ever seeing the content on the page.

JavaScript is a fine way to check for the cookie and apply the `fonts-loaded` class. If you *really* want to be speedy about it, you could use a back-end language (for example, PHP) to modify the document so that the `fonts-loaded` class is on the `<html>` element when the content is sent by the server. You remove the `else` condition that adds the class in the JavaScript, and modify the output by checking for the cookie on the back end. Here's how this is done in PHP.

Listing 7.10 Conditionally adding the `fonts-loaded` class via PHP

```
<?php if(isset($_COOKIE["fonts-loaded"])){
    ?><html class="fonts-loaded">
<?php }
else {
    ?><html>
<?php } ?>
```

Check if the `fonts-loaded` cookie is set via the `isset()` function.

If the cookie is set, add the `fonts-loaded` class on the `<html>` element.

If the cookie isn't set, don't modify the `<html>` element.

By using a back-end language to modify the response, you're changing the `<html>` element before it's sent to the client. That said, the JavaScript solution is serviceable, so both approaches are reasonable solutions. It all depends on the tools you have at your disposal, your skill set, and the time available to you.

ACCOMMODATING USERS WITH JAVASCRIPT DISABLED

As always, it comes back to users with JavaScript disabled. Because of the way you've developed this solution, the `@font-faces` you've specified will never take effect because the font-loading scripts never run and apply the `fonts-loaded` class to the document. As a result, the content will be displayed using the system fonts you specified as the first to appear.

If you or your organization doesn't care that this small segment of users never gets to see your fancy new font faces in action, feel free to call it a day. But you or your organization may well take umbrage with this, so let's go over a quick fix that involves our old friend the `<noscript>` tag. You can use `<noscript>` to trigger the default browser-loading behavior by nesting an inline `<style>` tag that applies the Open Sans font families as the default.

Listing 7.11 `<noscript>` alternative to JavaScript font loading

```

<noscript>
  <style>
    body{
      font-family: "Open Sans Regular", "Helvetica", "Arial", sans-serif;
    }

    .navItem a{
      font-family: "Open Sans Light", "Helvetica", "Arial", sans-serif;
    }

    .articleTitle,
    .sectionHeader{
      font-family: "Open Sans Bold", "Helvetica", "Arial", sans-serif;
    }
  </style>
</noscript>

```

By adding this little bit of inline CSS, you're returning the user without JavaScript to the browser's default font-loading behavior. This means that although they won't be able to reap the benefits of the font-loading API, they'll at least be provided with a base level of functionality. Continuing on, you'll learn about the Font Face Observer library to polyfill what the font-loading API provides.

7.4.4 Using Font Face Observer as a fallback

The unfortunate reality is that the font-loading API isn't yet universally supported. It has strong support in modern browsers, but some browsers (for example, IE) are lacking. Your client would appreciate it if you could make sure that more browsers receive an optimal font-loading experience. This is where a polyfill such as Font Face Observer comes in.

Font Face Observer (<https://github.com/bramstein/fontfaceobserver>) is a font-loading library by Danish developer Bram Stein. Although it's not a direct polyfill in the sense that you can drop it into a page and have your existing font-loading API code work without a hitch, it gives the developer similar ability to manage font loading.

In this section, you'll write a script that kicks in when the font-loading API isn't available and that loads two external scripts: the Font Face Observer script, and a

script that loads the fonts via Font Face Observer. To get started, you need to download new code from GitHub. Type `git checkout -f fontface-observer`, and after the code has downloaded, you'll be ready to start!

CONDITIONALLY LOADING THE EXTERNAL SCRIPTS

After downloading the new branch with `git`, you'll notice a `js` folder containing two scripts: `fontfaceobserver.min.js`, which is the minified Font Face Observer library, and `fontloading.js`, which contains an empty closure where you'll place the alternative font-loading behavior. Because you don't want to invoke the overhead of the Font Face Observer script in all browsers, you want to load it and the script with your fallback-loading behavior only when the font-loading API isn't available. To do this, you add the code in the following listing between the initial `if` conditional that checks for the `document.fonts` object and the `fonts-loaded` cookie, and the `else` conditional that follows it.

Listing 7.12 Conditionally loading Font Face Observer and font-loading scripts

```

else if(!document.fonts && document.cookie.indexOf("fonts-loaded") === -1){
    var fontFaceObserverScript = document.createElement("script"),
        fontLoadingScript = document.createElement("script");

    fontFaceObserverScript.src = "js/fontfaceobserver.min.js";
    fontLoadingScript.src = "js/fontloading.js";
    fontFaceObserverScript.defer = "defer";
    fontLoadingScript.defer = "defer";

    document.head.appendChild(fontFaceObserverScript);
    document.head.appendChild(fontLoadingScript);
}

```

Check if the font-loading API is unavailable, and if the fonts-loaded cookie hasn't been set.

Set the src attributes to the locations of both scripts.

Set the defer attribute to defer script parsing.

Create new <script> elements for Font Face Observer and the font loading script.

Append <script> elements to the end of the <head> tag.

The preceding code is simple. If the font-loading API isn't available *and* the fonts-loaded cookie hasn't been set, you then create new `<script>` elements for both Font Face Observer and the font-loading script, and set their `src` attributes to their respective locations. To ensure that they don't block page rendering, you set the `defer` attribute for both. To set everything up, you instruct the browser to load these scripts by appending them to the end of the `<head>` element.

WRITING THE FONT-LOADING BEHAVIOR

Open `js/fontloading.js` in your text editor and you'll notice that the content of this file is an empty JavaScript closure. Starting at line 2, add the contents of this listing to the file.

Listing 7.13 Using Font Face Observer to control the loading of fonts

```

document.onreadystatechange = function(){
    var openSansLight = new FontFaceObserver("Open Sans Light"),
        openSansRegular = new FontFaceObserver("Open Sans Regular"),
        openSansBold = new FontFaceObserver("Open Sans Bold");

    Promise.all([openSansLight.load(),
                 openSansRegular.load(),
                 openSansBold.load()]).then(function(){
        document.documentElement.className += " fonts-loaded";
        document.cookie = "fonts-loaded=";
    });
};

```

A promise that waits for all the fonts to be loaded.

Wait for the DOM to be ready.

Specify the font sources to be used in this document.

fonts-loaded class added to the <html> element, rendering custom fonts.

fonts-loaded cookie is set for subsequent page loads since the fonts will be cached.

Font Face Observer's syntax is similar to that of the font-loading API, but with slight differences. You define a `FontFaceObserver` object for each font variant you want to load. Then, through a JavaScript promise, you wait until all fonts have loaded. After the fonts have loaded, you apply the `fonts-loaded` class to the `<html>` element and set the `fonts-loaded` cookie. This allows you to reuse the mechanism by which you control the display of your fonts that you used in the font-loading API.

The result of this effort is an effective and widely compatible method that uses a native API where available, but then falls back to a capable polyfill. With this code in place, your client is happy with the font rendering, and as you know, a happy client is the only kind that you want.

7.5 Summary

In this chapter, you learned the following font optimization and delivery techniques:

- You can proactively lighten page weight by selecting only the font variants you need. Although it seems like common sense, it pays to audit your font selections. Doing so can improve your site's load times.
- Building an optimal `@font-face` cascade can help your site's performance by preferring locally installed fonts first, and then falling back to a set of the most optimal formats to the least optimal.
- You can compensate somewhat for the shortcomings of the TTF and EOT formats by compressing them on the server.
- Subsetting fonts can reduce the size of font files by limiting them to only the characters you need for the language of your site's content.
- Using the `unicode-range` property in modern browsers can assist you in using only the necessary font subsets as per your site's content language.
- If you need to selectively serve font subsets, you can write a script that can be used to serve subsets when `unicode-range` isn't an option.

- You can control how fonts are displayed by using the `font-display` property in CSS. Failing that, you can use the font-loading API to control how fonts display when `font-display` isn't available, or if you don't have control of the CSS that serves fonts, as in the case of third-party font providers such as Google Fonts or Typekit.
- If the font-loading API isn't available, you still have the ability to control how fonts are loaded and displayed through the use of the third-party Font Face Observer library.

Now that you're comfortable with these techniques, you can go forward in your web projects and apply them for the benefit of your clients (with your team's blessing, of course). In the next chapter, you'll learn how to optimize your application's JavaScript through techniques such as controlling the loading behavior of `<script>` elements, using high-performance native JavaScript APIs, working with leaner alternatives to jQuery, and more.