

Web Font Optimization



By Ilya Grigorik

Ilya is a Developer Advocate and Web Perf Guru

Typography is fundamental to good design, branding, readability, and accessibility. Webfonts enable all of the above and more: the text is selectable, searchable, zoomable, and high-DPI friendly, providing consistent and sharp text rendering regardless of the screen size and resolution. Webfonts are critical to good design, UX, and performance.

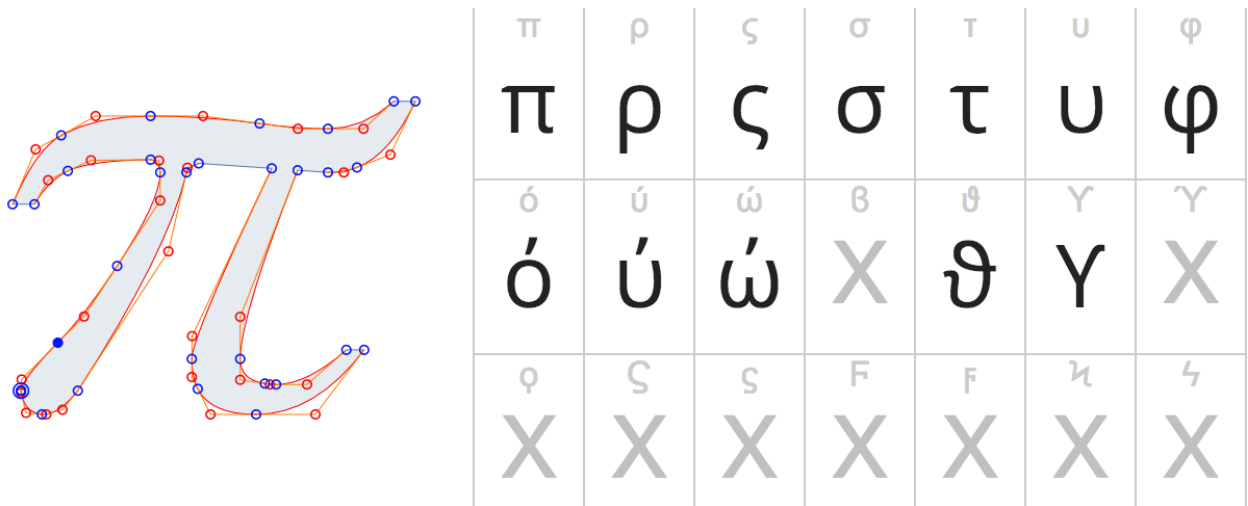
Webfont optimization is a critical piece of the overall performance strategy. Each font is an additional resource, and some fonts may block rendering of the text, but just because the page is using webfonts doesn't mean that it has to render slower. On the contrary, optimized fonts, combined with a judicious strategy for how they are loaded and applied on the page, can help reduce the total page size and improve page rendering times.

Anatomy of a webfont

TL;DR

- Unicode fonts can contain thousands of glyphs.
- There are four font formats: WOFF2, WOFF, EOT, and TTF.
- Some font formats require the use of compression.

A *webfont* is a collection of glyphs, and each glyph is a vector shape that describes a letter or symbol. As a result, two simple variables determine the size of a particular font file: the complexity of the vector paths of each glyph and the number of glyphs in a particular font. For example, Open Sans, which is one of the most popular webfonts, contains 897 glyphs, which include Latin, Greek, and Cyrillic characters.



When picking a font, it's important to consider which character sets are supported. If you need to localize your page content to multiple languages, you should use a font that can deliver a consistent look and experience to your users. For example, [Google's Noto font family](#). [\[7\]](#) aims to support all the world's languages. Note, however, that the total size of Noto, with all languages included, results in a 1.1GB+ ZIP download.

Clearly, using fonts on the web requires careful engineering to ensure that the typography doesn't impede performance. Thankfully, the web platform provides all the necessary primitives, and the rest of this guide provides a hands-on look at how to get the best of both worlds.

Webfont formats

Today there are four font container formats in use on the web: [EOT](#), [TTF](#), [WOFF](#), and [WOFF2](#) [\[8\]](#). Unfortunately, despite the wide range of choices, there isn't a single universal format that works across all old and new browsers: EOT is [IE only](#), TTF has [partial IE support](#), WOFF enjoys the widest support but is [not available in some older browsers](#), and WOFF 2.0 support is a [work in progress for many browsers](#).

So, where does that leave us? There isn't a single format that works in all browsers, which means that we need to deliver multiple formats to provide a consistent experience:

- Serve WOFF 2.0 variant to browsers that support it.
- Serve WOFF variant to the majority of browsers.
- Serve TTF variant to old Android (below 4.4) browsers.
- Serve EOT variant to old IE (below IE9) browsers.

Note: There's technically another container format, the [SVG font container](#), but IE and Firefox never supported it, and it is now deprecated in Chrome. As such, it's of limited use and it's intentionally omitted it in this guide.

Reducing font size with compression

A font is a collection of glyphs, each of which is a set of paths describing the letter form. The individual glyphs are different, but they contain a lot of similar information that can be compressed with GZIP or a compatible compressor:

- EOT and TTF formats are not compressed by default. Ensure that your servers are configured to apply [GZIP compression](#) when delivering these formats.
- WOFF has built-in compression. Ensure that your WOFF compressor is using optimal compression settings.
- WOFF2 uses custom preprocessing and compression algorithms to deliver ~30% file-size reduction over other formats. For more information, see [the WOFF 2.0 evaluation report](#) [↗](#).

Finally, it's worth noting that some font formats contain additional metadata, such as [font hinting](#) and [kerning](#) information that may not be necessary on some platforms, which allows for further file-size optimization. Consult your font compressor for available optimization options, and if you take this route, ensure that you have the appropriate infrastructure to test and deliver these optimized fonts to each browser. For example, [Google Fonts](#) maintains 30+ optimized variants for each font and automatically detects and delivers the optimal variant for each platform and browser.

Note: Consider using [Zopfli compression](#) for the EOT, TTF, and WOFF formats. Zopfli is a zlib compatible compressor that delivers ~5% file-size reduction over gzip.

Defining a font family with @font-face

TL;DR

- Use the `format()` hint to specify multiple font formats.
- Subset large Unicode fonts to improve performance. Use Unicode-range subsetting and provide a manual subsetting fallback for older browsers.

- Reduce the number of stylistic font variants to improve the page- and text-rendering performance.

The `@font-face` CSS at-rule allows you to define the location of a particular font resource, its style characteristics, and the Unicode codepoints for which it should be used. A combination of such `@font-face` declarations can be used to construct a "font family," which the browser will use to evaluate which font resources need to be downloaded and applied to the current page.

Format selection

Each `@font-face` declaration provides the name of the font family, which acts as a logical group of multiple declarations, font properties such as style, weight, and stretch, and the src descriptor, which specifies a prioritized list of locations for the font resource.

```
@font-face {  
  font-family: 'Awesome Font';  
  font-style: normal;  
  font-weight: 400;  
  src: local('Awesome Font'),  
       url('/fonts/awesome.woff2') format('woff2'),  
       url('/fonts/awesome.woff') format('woff'),  
       url('/fonts/awesome.ttf') format('truetype'),  
       url('/fonts/awesome.eot') format('embedded-opentype');  
}
```

```
@font-face {  
  font-family: 'Awesome Font';  
  font-style: italic;  
  font-weight: 400;  
  src: local('Awesome Font Italic'),  
       url('/fonts/awesome-i.woff2') format('woff2'),  
       url('/fonts/awesome-i.woff') format('woff'),  
       url('/fonts/awesome-i.ttf') format('truetype'),  
       url('/fonts/awesome-i.eot') format('embedded-opentype');  
}
```

First, note that the above examples define a single *Awesome Font* family with two styles (normal and *italic*), each of which points to a different set of font resources. In turn, each `src` descriptor contains a prioritized, comma-separated list of resource variants:

- The `local()` directive allows you to reference, load, and use locally installed fonts.
- The `url()` directive allows you to load external fonts, and are allowed to contain an optional `format()` hint indicating the format of the font referenced by the provided

URL.

Note: Unless you're referencing one of the default system fonts, it is rare for the user to have it locally installed, especially on mobile devices, where it is effectively impossible to "install" additional fonts. You should always start with a `local()` entry "just in case," and then provide a list of `url()` entries.

When the browser determines that the font is needed, it iterates through the provided resource list in the specified order and tries to load the appropriate resource. For example, following the example above:

1. The browser performs page layout and determines which font variants are required to render the specified text on the page.
2. For each required font, the browser checks if the font is available locally.
3. If the font is not available locally, the browser iterates over external definitions:
 - If a format hint is present, the browser checks if it supports the hint before initiating the download. If the browser doesn't support the hint, the browser advances to the next one.
 - If no format hint is present, the browser downloads the resource.

The combination of local and external directives with appropriate format hints allows you to specify all of the available font formats and let the browser handle the rest. The browser determines which resources are required and selects the optimal format.

Note: The order in which the font variants are specified matters. The browser picks the first format it supports. Therefore, if you want the newer browsers to use WOFF2, then you should place the WOFF2 declaration above WOFF, and so on.

Unicode-range subsetting

In addition to font properties such as style, weight, and stretch, the `@font-face` rule allows us to define a set of Unicode codepoints supported by each resource. This enables us to split a large Unicode font into smaller subsets (for example, Latin, Cyrillic, and Greek subsets) and only download the glyphs required to render the text on a particular page.

The unicode-range descriptor allows you to specify a comma-delimited list of range values, each of which can be in one of three different forms:

- Single codepoint (for example, `U+416`)

- Interval range (for example, U+400–4ff): indicates the start and end codepoints of a range
- Wildcard range (for example, U+4??): ? characters indicate any hexadecimal digit

For example, you can split your *Awesome Font* family into Latin and Japanese subsets, each of which the browser downloads on an as-needed basis:

```
@font-face {  
  font-family: 'Awesome Font';  
  font-style: normal;  
  font-weight: 400;  
  src: local('Awesome Font'),  
        url('/fonts/awesome-l.woff2') format('woff2'),  
        url('/fonts/awesome-l.woff') format('woff'),  
        url('/fonts/awesome-l.ttf') format('truetype'),  
        url('/fonts/awesome-l.eot') format('embedded-opentype');  
  unicode-range: U+000-5FF; /* Latin glyphs */  
}  
  
@font-face {  
  font-family: 'Awesome Font';  
  font-style: normal;  
  font-weight: 400;  
  src: local('Awesome Font'),  
        url('/fonts/awesome-jp.woff2') format('woff2'),  
        url('/fonts/awesome-jp.woff') format('woff'),  
        url('/fonts/awesome-jp.ttf') format('truetype'),  
        url('/fonts/awesome-jp.eot') format('embedded-opentype');  
  unicode-range: U+3000-9FFF, U+ff??; /* Japanese glyphs */  
}
```

Note: Unicode-range subsetting is particularly important for Asian languages, where the number of glyphs is much larger than in Western languages and a typical "full" font is often measured in megabytes instead of tens of kilobytes.

The use of Unicode range subsets and separate files for each stylistic variant of the font allows you to define a composite font family that is both faster and more efficient to download. Visitors only download the variants and subsets they need, and they aren't forced to download subsets that they may never see or use on the page.

That said, there's one small issue with unicode-range: not all browser support it yet. Some browsers simply ignore the unicode-range hint and download all variants, while others may not process the @font-face declaration at all. To address this, you need to fall back to "manual subsetting" for older browsers.

Because old browsers are not smart enough to select only the necessary subsets and cannot construct a composite font, you have to fall back to providing a single font resource that contains all the necessary subsets and hide the rest from the browser. For example, if the page is only using Latin characters, then you can strip other glyphs and serve that particular subset as a standalone resource.

1. How do you determine which subsets are needed?

- If the browser supports unicode-range subsetting, then it will automatically select the right subset. The page just needs to provide the subset files and specify appropriate unicode-ranges in the `@font-face` rules.
- If the browser doesn't support unicode-range subsetting, then the page needs to hide all unnecessary subsets; that is, the developer must specify the required subsets.

2. How do you generate font subsets?

- Use the open-source [pyftsubset tool](#) [↗](#) to subset and optimize your fonts.
- Some font services allow manual subsetting via custom query parameters, which you can use to manually specify the required subset for your page. Consult the documentation from your font provider.

Font selection and synthesis

Each font family is composed of multiple stylistic variants (regular, bold, italic) and multiple weights for each style, each of which, in turn, may contain very different glyph shapes—for example, different spacing, sizing, or a different shape altogether.



For example, the above diagram illustrates a font family that offers three different bold weights: 400 (regular), 700 (bold), and 900 (extra bold). All other in-between variants (indicated in gray) are automatically mapped to the closest variant by the browser.

When a weight is specified for which no face exists, a face with a nearby weight is used. In general, bold weights map to faces with heavier weights and light weights map to faces with lighter weights.

CSS3 font matching algorithm

Similar logic applies to *italic* variants. The font designer controls which variants they will produce, and you control which variants you'll use on the page. Because each variant is a separate download, it's a good idea to keep the number of variants small. For example, you can define two bold variants for the *Awesome Font* family:

```
@font-face {  
  font-family: 'Awesome Font';  
  font-style: normal;  
  font-weight: 400;  
  src: local('Awesome Font'),  
        url('/fonts/awesome-l.woff2') format('woff2'),  
        url('/fonts/awesome-l.woff') format('woff'),  
        url('/fonts/awesome-l.ttf') format('truetype'),  
        url('/fonts/awesome-l.eot') format('embedded-opentype');  
  unicode-range: U+000-5FF; /* Latin glyphs */  
}  
  
@font-face {  
  font-family: 'Awesome Font';  
  font-style: normal;  
  font-weight: 700;  
  src: local('Awesome Font'),  
        url('/fonts/awesome-l-700.woff2') format('woff2'),  
        url('/fonts/awesome-l-700.woff') format('woff'),  
        url('/fonts/awesome-l-700.ttf') format('truetype'),  
        url('/fonts/awesome-l-700.eot') format('embedded-opentype');  
  unicode-range: U+000-5FF; /* Latin glyphs */  
}
```

The above example declares the *Awesome Font* family that is composed of two resources that cover the same set of Latin glyphs (U+000-5FF) but offer two different "weights": normal (400) and bold (700). However, what happens if one of your CSS rules specifies a different font weight, or sets the font-style property to italic?

- If an exact font match isn't available, the browser substitutes the closest match.
- If no stylistic match is found (for example, no italic variants were declared in the example above), then the browser synthesizes its own font variant.

Style	Actual font	Synthesized
300	Lorem ipsum dolor fit amet...	Lorem ipsum dolor fit amet...
300, italic	<i>Lorem ipsum dolor fit amet...</i>	<i>Lorem ipsum dolor fit amet...</i>
400	Lorem ipsum dolor fit amet...	Lorem ipsum dolor fit amet...
400, italic	<i>Lorem ipsum dolor fit amet...</i>	<i>Lorem ipsum dolor fit amet...</i>
600	Lorem ipsum dolor fit amet...	Lorem ipsum dolor fit amet...
600, italic	<i>Lorem ipsum dolor fit amet...</i>	<i>Lorem ipsum dolor fit amet...</i>
700	Lorem ipsum dolor fit amet...	Lorem ipsum dolor fit amet...
700, italic	<i>Lorem ipsum dolor fit amet...</i>	<i>Lorem ipsum dolor fit amet...</i>
800	Lorem ipsum dolor fit amet...	Lorem ipsum dolor fit amet...
800, italic	<i>Lorem ipsum dolor fit amet...</i>	<i>Lorem ipsum dolor fit amet...</i>

Warning: Authors should also be aware that synthesized approaches may not be suitable for scripts like Cyrillic, where italic forms are very different in shape. For proper fidelity in those scripts, use an actual italic font.

The example above illustrates the difference between the actual vs. synthesized font results for Open Sans. All synthesized variants are generated from a single 400-weight font. As you can see, there's a noticeable difference in the results. The details of how to generate the bold and oblique variants are not specified. Therefore, the results vary from browser to browser, and are highly dependent on the font.

Note: For best consistency and visual results, don't rely on font synthesis. Instead, minimize the number of used font variants and specify their locations, such that the browser can download them when they are used on the page. That said, in some cases a synthesized variant [may be a viable option](#), but be cautious in using synthesized variants.

Optimizing loading and rendering

TL;DR

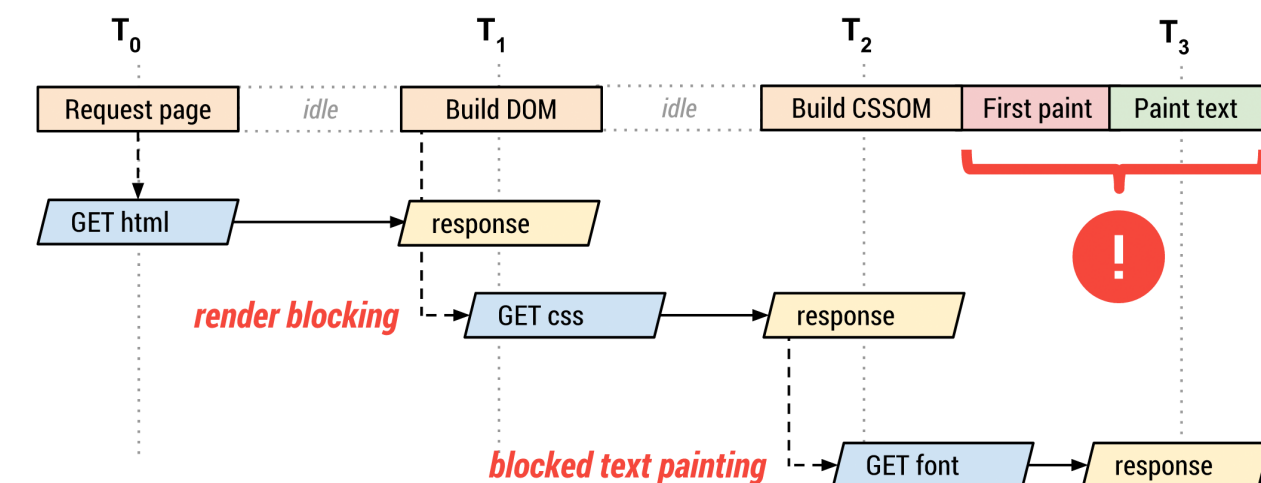
- By default, font requests are delayed until the render tree is constructed, which can result in delayed text rendering.
- `<link rel="preload">`, the CSS `font-display` property, and the Font Loading API provide the hooks needed to implementing custom font loading and rendering strategies, overriding the default behavior.

A "full" webfont that includes all stylistic variants, which you may not need, plus all the glyphs, which may go unused, can easily result in a multi-megabyte download. To address this, the `@font-face` CSS rule is specifically designed to allow you to split the font family into a collection of resources: unicode subsets, distinct style variants, and so on.

Given these declarations, the browser figures out the required subsets and variants and downloads the minimal set required to render the text, which is very convenient. However, if you're not careful, it can also create a performance bottleneck in the critical rendering path and delay text rendering.

The default behavior

Lazy loading of fonts carries an important hidden implication that may delay text rendering: the browser must construct the render tree, which is dependent on the DOM and CSSOM trees, before it knows which font resources it needs in order to render the text. As a result, font requests are delayed well after other critical resources, and the browser may be blocked from rendering text until the resource is fetched.



1. The browser requests the HTML document.
2. The browser begins parsing the HTML response and constructing the DOM.
3. The browser discovers CSS, JS, and other resources and dispatches requests.
4. The browser constructs the CSSOM after all of the CSS content is received and combines it with the DOM tree to construct the render tree.
 - Font requests are dispatched after the render tree indicates which font variants are needed to render the specified text on the page.
5. The browser performs layout and paints content to the screen.
 - If the font is not yet available, the browser may not render any text pixels.

- After the font is available, the browser paints the text pixels.

The "race" between the first paint of page content, which can be done shortly after the render tree is built, and the request for the font resource is what creates the "blank text problem" where the browser might render page layout but omits any text.

The next section describes a number of options for customizing this default behavior.

Preload your Webfont resources

If there's a high probability that your page will need a specific Webfont hosted at a URL you know in advance, you can take advantage of a new web platform feature: <link rel="preload">.

It allows you to include an element in your HTML, usually as part of the <head>, that will trigger a request for the Webfont early in the critical rendering path, without having to wait for the CSSOM to be created.

<link rel="preload"> serves as a "hint" to the browser that a given resource is going to be needed soon, but it doesn't tell the browser *how* to use it. You need to use preload in conjunction with an appropriate CSS @font-face definition in order to instruct the browser what to do with a given Webfont URL.

```
<head>
  <!-- Other tags... -->
  <link rel="preload" href="/fonts/awesome-l.woff2" as="font">
</head>
```

```
@font-face {
  font-family: 'Awesome Font';
  font-style: normal;
  font-weight: 400;
  src: local('Awesome Font'),
        url('/fonts/awesome-l.woff2') format('woff2'), /* will be preloaded */
        url('/fonts/awesome-l.woff') format('woff'),
        url('/fonts/awesome-l.ttf') format('truetype'),
        url('/fonts/awesome-l.eot') format('embedded-opentype');
  unicode-range: U+000-5FF; /* Latin glyphs */
}
```

Not all browsers support <link rel="preload">, and in those browsers, <link rel="preload"> will just be ignored. But every browser that supports preloading also supports WOFF2, so that's always the format that you should preload.

Caution: Using `<link rel="preload">` will make an unconditional, high-priority request for the Webfont's URL, regardless of whether it actually ends up being needed on the page. If there's a reasonable chance that the remote copy of the Webfont won't be needed—for instance, because the `@font-face` definition includes a `local()` entry for a common font like Roboto—then using `<link rel="preload">` will result in a wasted request. Some browsers will display a warning in their Developer Tools Console when a resource is preloaded but not actually used.

Customize the text rendering delay

While preloading makes it more likely that a Webfont will be available when a page's content is rendered, it offers no guarantees. You still need to consider how browsers behave when rendering text that uses a `font-family` which is not yet available.

Browser behaviors

The "race" between the first paint of page content, which can be done shortly after the render tree is built, and the request for the font resource is what creates the "blank text problem" where the browser might render page layout but omits any text. Most browsers implement a maximum timeout that they'll wait for a Webfont to download, after which a fallback font will be used. Unfortunately, browsers differ on implementation:

Browser	Timeout	Fallback	Swap
Chrome 35+	3 seconds	Yes	Yes
Opera	3 seconds	Yes	Yes
Firefox	3 seconds	Yes	Yes
Internet Explorer	0 seconds	Yes	Yes
Safari	No timeout	N/A	N/A

- Chrome and Firefox have a three second timeout after which the text is shown with the fallback font. If the font manages to download, then eventually a swap occurs and the text is re-rendered with the intended font.
- Internet Explorer has a zero second timeout which results in immediate text rendering. If the requested font is not yet available, a fallback is used, and text is re-rendered later once the requested font becomes available.

- Safari has no timeout behavior (or at least nothing beyond a baseline network timeout).

To ensure consistency moving forward, the CSS Working Group has proposed a new `@font-face` descriptor, `font-display`, and a corresponding property for controlling how a downloadable font renders before it is loaded.

The font display timeline

Similar to the existing font timeout behaviors that some browsers implement today, `font-display` segments the lifetime of a font download into three major periods:

1. The first period is the **font block period**. During this period, if the font face is not loaded, any element attempting to use it must instead render with an invisible fallback font face. If the font face successfully loads during the block period, the font face is then used normally.
2. The **font swap period** occurs immediately after the font block period. During this period, if the font face is not loaded, any element attempting to use it must instead render with a fallback font face. If the font face successfully loads during the swap period, the font face is then used normally.
3. The **font failure period** occurs immediately after the font swap period. If the font face is not yet loaded when this period starts, it's marked as a failed load, causing normal font fallback. Otherwise, the font face is used normally.

Understanding these periods means you can use `font-display` to decide how your font should render depending on whether or when it was downloaded.

Using font-display

To work with the `font-display` property, add it your `@font-face` rules:

```
@font-face {  
  font-family: 'Awesome Font';  
  font-style: normal;  
  font-weight: 400;  
  font-display: auto; /* or block, swap, fallback, optional */  
  src: local('Awesome Font'),  
        url('/fonts/awesome-l.woff2') format('woff2'), /* will be preloaded */  
        url('/fonts/awesome-l.woff') format('woff'),  
        url('/fonts/awesome-l.ttf') format('truetype'),  
        url('/fonts/awesome-l.eot') format('embedded-opentype');  
  unicode-range: U+000-5FF; /* Latin glyphs */  
}
```



`font-display` currently supports the following range of values: `auto` | `block` | `swap` | `fallback` | `optional`.

- **auto** uses whatever font display strategy the user-agent uses. Most browsers currently have a default strategy similar to **block**.
- **block** gives the font face a short block period (3s is recommended in most cases) and an infinite swap period. In other words, the browser draws "invisible" text at first if the font is not loaded, but swaps the font face in as soon as it loads. To do this the browser creates an anonymous font face with metrics similar to the selected font but with all glyphs containing no "ink." This value should only be used if rendering text in a particular typeface is required for the page to be usable.
- **swap** gives the font face a zero second block period and an infinite swap period. This means the browser draws text immediately with a fallback if the font face isn't loaded, but swaps the font face in as soon as it loads. Similar to **block**, this value should only be used when rendering text in a particular font is important for the page, but rendering in any font will still get a correct message across. Logo text is a good candidate for **swap** since displaying a company's name using a reasonable fallback will get the message across but you'd eventually use the official typeface.
- **fallback** gives the font face an extremely small block period (100ms or less is recommended in most cases) and a short swap period (three seconds is recommended in most cases). In other words, the font face is rendered with a fallback at first if it's not loaded, but the font is swapped as soon as it loads. However, if too much time passes, the fallback will be used for the rest of the page's lifetime. **fallback** is a good candidate for things like body text where you'd like the user to start reading as soon as possible and don't want to disturb their experience by shifting text around as a new font loads in.
- **optional** gives the font face an extremely small block period (100ms or less is recommended in most cases) and a zero second swap period. Similar to **fallback**, this is a good choice for when the downloading font is more of a "nice to have" but not critical to the experience. The **optional** value leaves it up to the browser to decide whether to initiate the font download, which it may choose not to do or it may do it as a low priority depending on what it thinks would be best for the user. This can be beneficial in situations where the user is on a weak connection and pulling down a font may not be the best use of resources.

`font-display` is gaining adoption in many modern browsers. You can look forward to consistency in browser behavior as it becomes widely implemented.

The Font Loading API

Used together, `<link rel="preload">` and the CSS `font-display` give developers a great deal of control over font loading and rendering, without adding in much overhead. But if you need additional customizations, and are willing to incur with the overhead introduced by running JavaScript, there is another option.

The Font Loading API provides a scripting interface to define and manipulate CSS font faces, track their download progress, and override their default lazyload behavior. For example, if you're sure that a particular font variant is required, you can define it and tell the browser to initiate an immediate fetch of the font resource:

```
var font = new FontFace("Awesome Font", "url(/fonts/awesome.woff2)", {  
  style: 'normal', unicodeRange: 'U+000-5FF', weight: '400'  
});  
  
// don't wait for the render tree, initiate an immediate fetch!  
font.load().then(function() {  
  // apply the font (which may re-render text and cause a page reflow)  
  // after the font has finished downloading  
  document.fonts.add(font);  
  document.body.style.fontFamily = "Awesome Font, serif";  
  
  // OR... by default the content is hidden,  
  // and it's rendered after the font is available  
  var content = document.getElementById("content");  
  content.style.visibility = "visible";  
  
  // OR... apply your own render strategy here...  
});
```



Further, because you can check the font status (via the `check()` method and track its download progress, you can also define a custom strategy for rendering text on your pages:

- You can hold all text rendering until the font is available.
- You can implement a custom timeout for each font.
- You can use the fallback font to unblock rendering and inject a new style that uses the desired font after the font is available.

Best of all, you can also mix and match the above strategies for different content on the page. For example, you can delay text rendering on some sections until the font is available, use a fallback font, and then re-render after the font download has finished, specify different timeouts, and so on.

Note: The Font Loading API is still [under development in some browsers](#). Consider using the [FontLoader polyfill](#) or the [webfontloader library](#) to deliver similar functionality, albeit with even more overhead from an additional JavaScript dependency.

Proper caching is a must

Font resources are, typically, static resources that don't see frequent updates. As a result, they are ideally suited for a long max-age expiry - ensure that you specify both a conditional ETag header, and an optimal Cache-Control policy for all font resources.

If your web application uses a service worker, serving font resources with a cache-first strategy is appropriate for most use cases.

You should not store fonts using localStorage or IndexedDB; each of those has its own set of performance issues. The browser's HTTP cache provides the best and most robust mechanism to deliver font resources to the browser.

Optimization checklist

Contrary to popular belief, the use of webfonts doesn't need to delay page rendering or have a negative impact on other performance metrics. The well-optimized use of fonts can deliver a much better overall user experience: great branding, improved readability, usability, and searchability, all while delivering a scalable multi-resolution solution that adapts well to all screen formats and resolutions. Don't be afraid to use webfonts.

That said, a naive implementation may incur large downloads and unnecessary delays. You need to help the browser by optimizing the font assets themselves and how they are fetched and used on your pages.

- **Audit and monitor your font use:** don't use too many fonts on your pages, and, for each font, minimize the number of used variants. This helps produce a more consistent and a faster experience for your users.
- **Subset your font resources:** many fonts can be subset, or split into multiple unicode-ranges to deliver just the glyphs that a particular page requires. This reduces the file size and improves the download speed of the resource. However, when defining the subsets, be careful to optimize for font re-use. For example, don't download a different but overlapping set of characters on each page. A good practice is to subset based on script: for example, Latin, Cyrillic, and so on.

- **Deliver optimized font formats to each browser:** provide each font in WOFF2, WOFF, EOT, and TTF formats. Make sure to apply GZIP compression to the EOT and TTF formats, because they are not compressed by default.
- **Give precedence to `local()` in your src list:** listing `local('Font Name')` first in your src list ensures that HTTP requests aren't made for fonts that are already installed.
- **Customize font loading and rendering using `<link rel="preload">`, `font-display`, or the Font Loading API:** default lazyloading behavior may result in delayed text rendering. These web platform features allow you to override this behavior for particular fonts, and to specify custom rendering and timeout strategies for different content on the page.
- **Specify revalidation and optimal caching policies:** fonts are static resources that are infrequently updated. Make sure that your servers provide a long-lived max-age timestamp and a revalidation token to allow for efficient font reuse between different pages. If using a service worker, a cache-first strategy is appropriate.

This article contains contributions from [Monica Dinculescu](#), [Rob Dodson](#), and Jeff Posnick.

[Previous](#)

← [Loading Third-Party JavaScript](#)

[Next](#)

[HTTP Caching](#) →

Except as otherwise noted, the content of this page is licensed under the [Creative Commons Attribution 3.0 License](#), and code samples are licensed under the [Apache 2.0 License](#). For details, see our [Site Policies](#). Java is a registered trademark of Oracle and/or its affiliates.

Last updated July 2, 2018.