

---

## fontkit

Fontkit is an advanced font engine for Node and the browser, used by PDFKit. It supports many font formats, advanced glyph substitution and layout features, glyph path extraction, color emoji glyphs, font subsetting, and more.

### Features

- Supports TrueType (.ttf), OpenType (.otf), WOFF, WOFF2, TrueType Collection (.ttc), and Datafork TrueType (.dfont) font files
- Supports mapping characters to glyphs, including support for ligatures and other advanced substitutions (see below)
- Supports reading glyph metrics and laying out glyphs, including support for kerning and other advanced layout features (see below)
- Advanced OpenType features including glyph substitution (GSUB) and positioning (GPOS)
- Apple Advanced Typography (AAT) glyph substitution features (morx table)
- Support for getting glyph vector paths and converting them to SVG paths, or rendering them to a graphics context
- Supports TrueType (glyf) and PostScript (CFF) outlines
- Support for color glyphs (e.g. emoji), including Apple's SBIX table, and Microsoft's COLR table
- Support for AAT variation glyphs, allowing for nearly infinite design control over weight, width, and other axes.
- Font subsetting support - create a new font including only the specified glyphs

### Installation

```
1 npm install fontkit
```

### Example

```
1 var fontkit = require('fontkit');
2
3 // open a font synchronously
4 var font = fontkit.openSync('font.ttf');
5
6 // layout a string, using default shaping features.
7 // returns a GlyphRun, describing glyphs and positions.
8 var run = font.layout('hello world!');
9
```

---

```
10 // get an SVG path for a glyph
11 var svg = run.glyphs[0].path.toSVG();
12
13 // create a font subset
14 var subset = font.createSubset();
15 run.glyphs.forEach(function(glyph) {
16     subset.includeGlyph(glyph);
17 });
18
19 let buffer = subset.encode();
```

## API

### **fontkit.open(filename, postscriptName = null)**

Opens a font file asynchronously, and returns a [Promise](#) with a font object. For collection fonts (such as TrueType collection files), you can pass a [postscriptName](#) to get that font out of the collection instead of a collection object.

### **fontkit.openSync(filename, postscriptName = null)**

Opens a font file synchronously, and returns a font object. For collection fonts (such as TrueType collection files), you can pass a [postscriptName](#) to get that font out of the collection instead of a collection object.

### **fontkit.create(buffer, postscriptName = null)**

Returns a font object for the given buffer. For collection fonts (such as TrueType collection files), you can pass a [postscriptName](#) to get that font out of the collection instead of a collection object.

## Font objects

There are several different types of font objects that are returned by fontkit depending on the font format. They all inherit from the [TTFFont](#) class and have the same public API, described below.

### Metadata properties

The following properties are strings (or null if the font does not contain strings for them) describing the font, as specified by the font creator.

- 
- `postscriptName`
  - `fullName`
  - `familyName`
  - `subfamilyName`
  - `copyright`
  - `version`

## Metrics

The following properties describe the general metrics of the font. See [here](#) for a good overview of how all of these properties relate to one another.

- `unitsPerEm` - the size of the font's internal coordinate grid
- `ascent` - the font's ascender
- `descent` - the font's descender
- `lineGap` - the amount of space that should be included between lines
- `underlinePosition` - the offset from the normal underline position that should be used
- `underlineThickness` - the weight of the underline that should be used
- `italicAngle` - if this is an italic font, the angle the cursor should be drawn at to match the font design
- `capHeight` - the height of capital letters above the baseline. See [here](#) for more details.
- `xHeight` - the height of lower case letters. See [here](#) for more details.
- `bbox` - the font's bounding box, i.e. the box that encloses all glyphs in the font

## Other properties

- `numGlyphs` - the number of glyphs in the font
- `characterSet` - an array of all of the unicode code points supported by the font
- `availableFeatures` - an array of all OpenType feature tags (or mapped AAT tags) supported by the font (see below for a description of this)

## Character to glyph mapping

Fontkit includes several methods for character to glyph mapping, including support for advanced OpenType and AAT substitutions.

**`font.glyphForCodePoint(codePoint)`** Maps a single unicode code point (number) to a Glyph object. Does not perform any advanced substitutions (there is no context to do so).

---

**font.hasGlyphForCodePoint(codePoint)** Returns whether there is glyph in the font for the given unicode code point.

**font.glyphsForString(string)** This method returns an array of Glyph objects for the given string. This is only a one-to-one mapping from characters to glyphs. For most uses, you should use [font.layout](#) (described below), which provides a much more advanced mapping supporting AAT and OpenType shaping.

### Glyph metrics and layout

Fontkit includes several methods for accessing glyph metrics and performing layout, including support for kerning and other advanced OpenType positioning adjustments.

**font.widthOfGlyph(glyph\_id)** Returns the advance width (described above) for a single glyph id.

**font.layout(string, features = [] | {})** This method returns a [GlyphRun](#) object, which includes an array of [Glyphs](#) and [GlyphPositions](#) for the given string. [Glyph](#) objects are described below. [GlyphPosition](#) objects include 4 properties: [xAdvance](#), [yAdvance](#), [xOffset](#), and [yOffset](#).

The [features](#) parameter is either an array of OpenType feature tags to be applied in addition to the default set, or an object mapping OpenType features to a boolean enabling or disabling each. If this is an AAT font, the OpenType feature tags are mapped to AAT features.

### Variation fonts

Fontkit has support for AAT variation fonts, where glyphs can adjust their shape according to user defined settings along various axes including weight, width, and slant. Font designers specify the minimum, default, and maximum values for each axis they support, and allow the user fine grained control over the rendered text.

**font.variationAxes** Returns an object describing the available variation axes. Keys are 4 letter axis tags, and values include [name](#), [min](#), [default](#), and [max](#) properties for the axis.

---

**font.namedVariations** The font designer may have picked out some variations that they think look particularly good, for example a light, regular, and bold weight which would traditionally be separate fonts. This property returns an object describing these named variation instances that the designer has specified. Keys are variation names, and values are objects with axis settings.

**font.getVariation(variation)** Returns a new font object representing this variation, from which you can get glyphs and perform layout as normal. The `variation` parameter can either be a variation settings object or a string variation name. Variation settings objects have axis names as keys, and numbers as values (should be in the range specified by `font.variationAxes`).

### Other methods

**font.getGlyph(glyph\_id, codePoints = [])** Returns a glyph object for the given glyph id. You can pass the array of code points this glyph represents for your use later, and it will be stored in the glyph object.

**font.createSubset()** Returns a Subset object for this font, described below.

### Font Collection objects

For font collection files that contain multiple fonts in a single file, such as TrueType Collection (.ttc) and Datafork TrueType (.dfont) files, a font collection object can be returned by Fontkit.

**collection.getFont(postscriptName)**

Gets a font from the collection by its postscript name. Returns a Font object, described above.

**collection.fonts**

This property is a lazily-loaded array of all of the fonts in the collection.

### Glyph objects

Glyph objects represent a glyph in the font. They have various properties for accessing metrics and the actual vector path the glyph represents, and methods for rendering the glyph to a graphics context.

---

You do not create glyph objects directly. They are created by various methods on the font object, described above. There are several subclasses of the base `Glyph` class internally that may be returned depending on the font format, but they all include the following API.

## Properties

- `id` - the glyph id in the font
- `name` - the glyph name in the font
- `codePoints` - an array of unicode code points that are represented by this glyph. There can be multiple code points in the case of ligatures and other glyphs that represent multiple visual characters.
- `path` - a vector Path object representing the glyph
- `bbox` - the glyph's bounding box, i.e. the rectangle that encloses the glyph outline as tightly as possible.
- `cbox` - the glyph's control box. This is often the same as the bounding box, but is faster to compute. Because of the way bezier curves are defined, some of the control points can be outside of the bounding box. Where `bbox` takes this into account, `cbox` does not. Thus, `cbox` is less accurate, but faster to compute. See here for a more detailed description.
- `advanceWidth` - the glyph's advance width.

## `glyph.render(ctx, size)`

Renders the glyph to the given graphics context, at the specified font size.

## Color glyphs (e.g. emoji)

Fontkit has support for several different color emoji font formats. Currently, these include Apple's SBIX table (as used by the "Apple Color Emoji" font), and Microsoft's COLR table (supported by Windows 8.1). Here is an overview of the various color font formats out there.

**`glyph.getImageForSize(size)`** For SBIX glyphs, which are bitmap based, this returns an object containing some properties about the image, along with the image data itself (usually PNG).

**`glyph.layers`** For COLR glyphs, which are vector based, this returns an array of objects representing the glyphs and colors for each layer in render order.

---

## Path objects

Path objects are returned by glyphs and represent the actual vector outlines for each glyph in the font. Paths can be converted to SVG path data strings, or to functions that can be applied to render the path to a graphics context.

### **path.moveTo(x, y)**

Moves the virtual pen to the given x, y coordinates.

### **path.lineTo(x, y)**

Adds a line to the path from the current point to the given x, y coordinates.

### **path.quadraticCurveTo(cpx, cpy, x, y)**

Adds a quadratic curve to the path from the current point to the given x, y coordinates using cpx, cpy as a control point.

### **path.bezierCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)**

Adds a bezier curve to the path from the current point to the given x, y coordinates using cp1x, cp1y and cp2x, cp2y as control points.

### **path.closePath()**

Closes the current sub-path by drawing a straight line back to the starting point.

### **path.toFunction()**

Compiles the path to a JavaScript function that can be applied with a graphics context in order to render the path.

### **path.toSVG()**

Converts the path to an SVG path data string.

---

### **path.bbox**

This property represents the path's bounding box, i.e. the smallest rectangle that contains the entire path shape. This is the exact bounding box, taking into account control points that may be outside the visible shape.

### **path.cbox**

This property represents the path's control box. It is like the bounding box, but it includes all points of the path, including control points of bezier segments. It is much faster to compute than the real bounding box, but less accurate if there are control points outside of the visible shape.

## **Subsets**

Fontkit can perform font subsetting, i.e. the process of creating a new font from an existing font where only the specified glyphs are included. This is useful to reduce the size of large fonts, such as in PDF generation or for web use.

Currently, subsets produce minimal fonts designed for PDF embedding that may not work as standalone files. They have no cmap tables and other essential tables for standalone use. This limitation will be removed in the future.

You create a Subset object by calling `font.createSubset()`, described above. The API on Subset objects is as follows.

### **subset.includeGlyph(glyph)**

Includes the given glyph object or glyph ID in the subset.

### **subset.encode()**

Returns a `Uint8Array` containing the encoded font file.

## **License**

MIT