PDFKit Guide

By Devon Govett

Version 0.15.2

Getting Started with PDFKit

Installation

Installation uses the npm package manager. Just type the following command after installing npm.

```
npm install pdfkit
```

Creating a document

Creating a PDFKit document is quite simple. Just require the **`pdfkit`** module in your JavaScript source file and create an instance of the **`PDFDocument`** class.

```
const PDFDocument = require('pdfkit');
const doc = new PDFDocument();
```

`PDFDocument` instances are readable Node streams. They don't get saved anywhere automatically, but you can call the **`pipe`** method to send the output of the PDF document to another writable Node stream as it is being written. When you're done with your document, call the **`end`** method to finalize it. Here is an example showing how to pipe to a file or an HTTP response.

```
doc.pipe(fs.createWriteStream('/path/to/file.pdf')); // write to PDF
doc.pipe(res);                                       // HTTP response

// add stuff to PDF here using methods described below...

// finalize the PDF and end the stream
doc.end();
```

The **`write`** and **`output`** methods found in PDFKit before version 0.5 are now deprecated.

Using PDFKit in the browser

PDFKit can be used in the browser as well as in Node! There are two ways to use PDFKit in the browser. The first is to create an app using an module bundler like Browserify or Webpack. The second is to create a standalone pdfkit script as explained here.

Using PDFKit in the browser is exactly the same as using it in Node, except you'll want to pipe the output to a destination supported in the browser, such as a Blob. Blobs can be used to generate a URL to allow display of generated PDFs directly in the browser via an **`iframe`**, or they can be used to upload the PDF to a server, or trigger a download in the user's browser.

To get a Blob from a **`PDFDocument`**, you should pipe it to a blob-stream, which is a module that generates a Blob from any Node-style stream. The following example uses Browserify to load **`PDFKit`** and **`blob-stream`**, but if you're not using Browserify, you can load them in whatever way you'd like (e.g. script tags).

```
// require dependencies
const PDFDocument = require('pdfkit');
const blobStream  = require('blob-stream');

// create a document the same way as above
const doc = new PDFDocument;

// pipe the document to a blob
const stream = doc.pipe(blobStream());

// add your content to the document here, as usual

// get a blob when you're done
doc.end();
stream.on('finish', function() {
  // get a blob you can do whatever you like with
  const blob = stream.toBlob('application/pdf');

  // or get a blob URL for display in the browser
  const url = stream.toBlobURL('application/pdf');
  iframe.src = url;
});
```

You can see an interactive in-browser demo of PDFKit here.

Note that in order to Browserify a project using PDFKit, you need to install the **`brfs`** module with npm, which is used to load built-in font data into the package. It is listed as a **`devDependencies`** in PDFKit's **`package.json`**, so it isn't installed by default for Node users. If you forget to install it, Browserify will print an error message.

Adding pages

The first page of a PDFKit document is added for you automatically when you create the document unless you provide **`autoFirstPage: false`**. Subsequent pages must be added by you. Luckily, it is quite simple!

```
doc.addPage()
```

To add some content every time a page is created, either by calling **`addPage()`** or automatically, you can use the **`pageAdded`** event.

```
doc.on('pageAdded', () => doc.text("Page Title"));
```

You can also set some options for the page, such as its size and orientation.

The **`layout`** property can be either **`portrait`** (the default) or **`landscape`**. The **`size`** property can be either an array specifying **`[width, height]`** in PDF points (72 per inch), or a string specifying a predefined size. A list of the predefined paper sizes can be seen here. The default is **`letter`**.

Passing a page options object to the **`PDFDocument`** constructor will set the default paper size and layout for every page in the document, which is then overridden by individual options passed to the **`addPage`** method.

You can set the page margins in two ways. The first is by setting the **`margin`** property (singular) to a number, which applies that margin to all edges. The other way is to set the **`margins`** property (plural) to an object with **`top`**, **`bottom`**, **`left`**, and **`right`** values. The default is a 1 inch (72 point) margin on all sides.

For example:

```
// Add a 50 point margin on all sides
doc.addPage({
  margin: 50});


// Add different margins on each side
doc.addPage({
  margins: {
    top: 50,
    bottom: 50,
    left: 72,
    right: 72
  }
});
```

Switching to previous pages

PDFKit normally flushes pages to the output file immediately when a new page is created, making it impossible to jump back and add content to previous pages. This is normally not an issue, but in some circumstances it can be useful to add content to pages after the whole document, or a part of the document, has been created already. Examples include adding page numbers, or filling in other parts of information you don't have until the rest of the document has been created.

PDFKit has a **`bufferPages`** option in versions v0.7.0 and later that allows you to control when pages are flushed to the output file yourself rather than letting PDFKit handle that for you. To use it, just pass **`bufferPages: true`** as an option to the **`PDFDocument`** constructor. Then, you can call **`doc.switchToPage(pageNumber)`** to switch to a previous page (page numbers start at 0).

When you're ready to flush the buffered pages to the output file, call **`flushPages`**. This method is automatically called by **`doc.end()`**, so if you just want to buffer all pages in the document, you never need to call it. Finally, there is a **`bufferedPageRange`** method, which returns the range of pages that are currently buffered. Here is a small example that shows how you might add page numbers to a document.

```
// create a document, and enable bufferPages mode
let i;
let end;
const doc = new PDFDocument({
  bufferPages: true});

// add some content...
doc.addPage();
// ...
doc.addPage();

// see the range of buffered pages
const range = doc.bufferedPageRange(); // => { start: 0, count: 2 }

for (i = range.start, end = range.start + range.count, range.start <= end; i < end; i+
+) {
  doc.switchToPage(i);
  doc.text(`Page ${i + 1} of ${range.count}`);
}

// manually flush pages that have been buffered
doc.flushPages();

// or, if you are at the end of the document anyway,
// doc.end() will call it for you automatically.
doc.end();
```

Setting default font

The default font is 'Helvetica'. It can be configured by passing **`font`** option

```
// use Courier font by default
const doc = new PDFDocument({font: 'Courier'});
```

Setting document metadata

PDF documents can have various metadata associated with them, such as the title, or author of the document. You can add that information by adding it to the **`doc.info`** object, or by passing an info object into the document at creation time.

Here is a list of all of the properties you can add to the document metadata. According to the PDF spec, each property must have its first letter capitalized.

`Title` – the title of the document

`Author` – the name of the author

`Subject` – the subject of the document

`Keywords` – keywords associated with the document

`CreationDate` – the date the document was created (added automatically by PDFKit)

`ModDate` – the date the document was last modified

Encryption and Access Privileges

PDF specification allow you to encrypt the PDF file and require a password when opening the file, and/or set permissions of what users can do with the PDF file. PDFKit implements standard security handler in PDF version 1.3 (40-bit RC4), version 1.4 (128-bit RC4), PDF version 1.7 (128-bit AES), and PDF version 1.7 ExtensionLevel 3 (256-bit AES).

To enable encryption, provide a user password when creating the **`PDFDocument`** in **`options`** object. The PDF file will be encrypted when a user password is provided, and users will be prompted to enter the password to decrypt the file when opening it.

`userPassword` - the user password (string value)

To set access privileges for the PDF file, you need to provide an owner password and permission settings in the **`option`** object when creating **`PDFDocument`**. By default, all operations are disallowed. You need to explicitly allow certain operations.

`ownerPassword` - the owner password (string value)

`permissions` - the object specifying PDF file permissions

Following settings are allowed in **`permissions`** object:

`printing` - whether printing is allowed. Specify **`"lowResolution"`** to allow degraded printing, or **`"highResolution"`** to allow printing with high resolution

`modifying` - whether modifying the file is allowed. Specify **`true`** to allow modifying document content

`copying` - whether copying text or graphics is allowed. Specify **`true`** to allow copying

`annotating` - whether annotating, form filling is allowed. Specify **`true`** to allow annotating and form filling

`fillingForms` - whether form filling and signing is allowed. Specify **`true`** to allow filling in form fields and signing

`contentAccessibility` - whether copying text for accessibility is allowed. Specify **`true`** to allow copying for accessibility

`documentAssembly` - whether assembling document is allowed. Specify **`true`** to allow document assembly

You can specify either user password, owner password or both passwords. Behavior differs according to passwords you provides:

When only user password is provided, users with user password are able to decrypt the file and have full access to the document.

When only owner password is provided, users are able to decrypt and open the document without providing any password, but the access is limited to those operations explicitly permitted. Users with owner password have full access to the document.

When both passwords are provided, users with user password are able to decrypt the file but only have limited access to the file according to permission settings. Users with owner

password have full access to the document.

Note that PDF file itself cannot enforce access privileges. When file is decrypted, PDF viewer applications have full access to the file content, and it is up to viewer applications to respect permission settings.

To choose encryption method, you need to specify PDF version. PDFKit will choose best encryption method available in the PDF version you specified.

`pdfVersion` - a string value specifying PDF file version

Available options includes:

`1.3` - PDF version 1.3 (default), 40-bit RC4 is used

`1.4` - PDF version 1.4, 128-bit RC4 is used

`1.5` - PDF version 1.5, 128-bit RC4 is used

`1.6` - PDF version 1.6, 128-bit AES is used

`1.7` - PDF version 1.7, 128-bit AES is used

`1.7ext3` - PDF version 1.7 ExtensionLevel 3, 256-bit AES is used

When using PDF version 1.7 ExtensionLevel 3, password is truncated to 127 bytes of its UTF-8 representation. In older versions, password is truncated to 32 bytes, and only Latin-1 characters are allowed.

PDF/A

PDF/A is a standard (ISO 19005-1:2005) which defines rules for electornic documents intended for long-term archiving. The restrictions on PDF/A documents are:

Cannot be encrypted

Fonts must be embedded

No JavaScript

No audio content

No video content

Addition of XMP metadata

Must define color spaces

Currently, PDFKit aims to support PDF/A-1b, PDF/A-2b, PDF/A-3b and PDF/A-1a, PDF/A-2a, PDF/A-3a standards, also known as level B conformance and level A conformance, respectively.

In order to create PDF/A documents, set **`subset`** to either **`PDF/A-1`** or **`PDF/A-1b`** for level B (basic) conformance, or **`PDF/A-1a`** for level A (accessible) conformance when creating the **`PDFDocument`** in **`options`** object.

Similary, use **`PDF/A-2`** or **`PDF/A-2b`** for PDF/A-2 level B conformance and **`PDF/A-2a`** for PDF/A-2 level A conformance. **`PDF/A-3`** or **`PDF/A-3b`** can be used for PDF/A-3 level B conformance and **`PDF/A-3a`** for PDF/A-3 level A conformance.

Futhermore, you will need to specify the other options relevant to the PDF/A subset you wish to use, for PDF/A-1 being:

`pdfVersion` set to at least **`1.4`**

`tagged` set to **`true`** for PDF/A-1a

For PDF/A-2 and PDF/A-3, the **`pdfVersion`** needs to be set to at least **`1.7`** and **`tagged`** needs to be **`true`** for level A conformance.

In order to verify the generated document for PDF/A and its subsets conformance, veraPDF is an excellent open source validator.

Please note that PDF/A requires fonts to be embedded, as such the standard fonts PDFKit comes with cannot be used because they are in AFM format, which only provides neccessary metrics, without the font data. You should use **`registerFont()`** and use embeddable fonts such as **`ttf`**.

Adding content

Once you've created a **`PDFDocument`** instance, you can add content to the document. Check out the other sections described in this document to learn about each type of content you can add.

That's the basics! Now let's move on to PDFKit's powerful vector graphics abilities.

Paper Sizes

When creating a new document or adding a new page to your current document, PDFKit allows you to set the page dimensions. To improve convenience, PDFKit has a number of predefined page sizes. These sizes are based on the most commonly used standard page sizes.

Predefined Page Sizes

The following predefined sizes are based on the ISO (International) standards. All the dimensions in brackets are in PostScript points.

A-series

A0 (2383.94 x 3370.39)

A1 (1683.78 x 2383.94)

A2 (1190.55 x 1683.78)

A3 (841.89 x 1190.55)

A4 (595.28 x 841.89)

A5 (419.53 x 595.28)

A6 (297.64 x 419.53)

A7 (209.76 x 297.64)

A8 (147.40 x 209.76)

A9 (104.88 x 147.40)

A10 (73.70 x 104.88)

B-series

B0 (2834.65 x 4008.19)

B1 (2004.09 x 2834.65)

B2 (1417.32 x 2004.09)

B3 (1000.63 x 1417.32)

B4 (708.66 x 1000.63)

B5 (498.90 x 708.66)

B6 (354.33 x 498.90)

B7 (249.45 x 354.33)

B8 (175.75 x 249.45)

B9 (124.72 x 175.75)

B10 (87.87 x 124.72)

C-series

C0 (2599.37 x 3676.54)

C1 (1836.85 x 2599.37)

C2 (1298.27 x 1836.85)

C3 (918.43 x 1298.27)

C4 (649.13 x 918.43)

C5 (459.21 x 649.13)

C6 (323.15 x 459.21)

C7 (229.61 x 323.15)

C8 (161.57 x 229.61)

C9 (113.39 x 161.57)

C10 (79.37 x 113.39)

RA-series

RA0 (2437.80 x 3458.27)

RA1 (1729.13 x 2437.80)

RA2 (1218.90 x 1729.13)

RA3 (864.57 x 1218.90)

RA4 (609.45 x 864.57)

SRA-series

SRA0 (2551.18 x 3628.35)

SRA1 (1814.17 x 2551.18)

SRA2 (1275.59 x 1814.17)

SRA3 (907.09 x 1275.59)

SRA4 (637.80 x 907.09)

The following predefined sizes are based on the common paper sizes used mainly in the United States of America and Canada. The dimensions in brackets are in PostScript points.

EXECUTIVE (521.86 x 756.00)

LEGAL (612.00 x 1008.00)

LETTER (612.00 X 792.00)

TABLOID (792.00 X 1224.00)

PDFKit supports also the following paper sizes. The dimensions in brackets are in PostScript points.

4A0 (4767.89 x 6740.79)

2A0 (3370.39 x 4767.87)

FOLIO (612.00 X 936.00)

Setting the page size

In order to use the predefined sizes, the name of the size (as named in the lists above) should be passed to either the **`PDFDocument`** constructor or the **`addPage()`** function in the **`size`** property of the **`options`** object, as shown in the example below, using **`A7`** as the preferred size.

```
// Passing size to the constructor
const doc = new PDFDocument({size: 'A7'});

// Passing size to the addPage function
doc.addPage({size: 'A7'});
```

Vector Graphics in PDFKit

An introduction to vector graphics

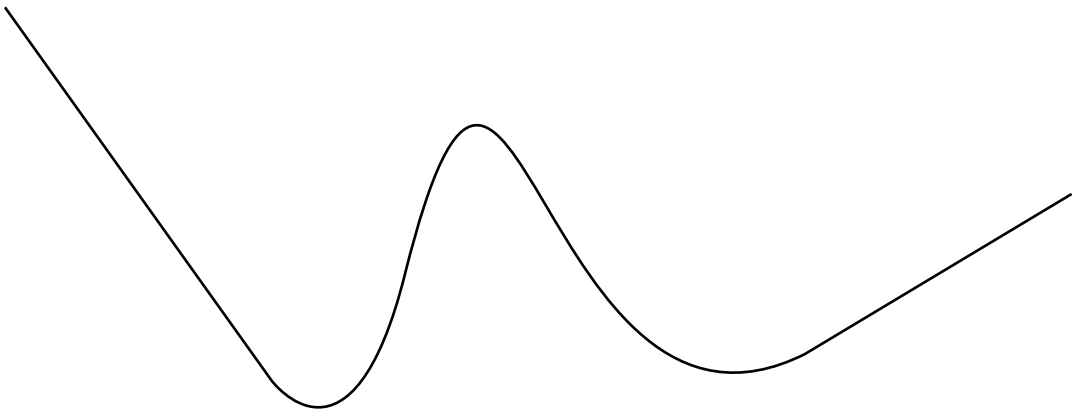
Unlike images which are defined by pixels, vector graphics are defined through a series of drawing commands. This makes vector graphics scalable to any size without a reduction in quality (pixelization). The PDF format was designed with vector graphics in mind, so creating vector drawings is very easy. The PDFKit vector graphics APIs are very similar to that of the HTML5 canvas element, so if you are familiar at all with that API, you will find PDFKit easy to pick up.

Creating basic shapes

Shapes are defined by a series of lines and curves. **`lineTo`**, **`bezierCurveTo`** and **`quadraticCurveTo`** all draw from the current point (which you can set with **`moveTo`**) to the specified point (always the last two arguments). Bezier curves use two control points and quadratic curves use just one. Here is an example that illustrates defining a path.

```
doc.moveTo(0, 20)                                 // set the current point
   .lineTo(100, 160)                              // draw a line
   .quadraticCurveTo(130, 200, 150, 120)          // draw a quadratic curve
   .bezierCurveTo(190, -40, 200, 200, 300, 150) // draw a bezier curve
   .lineTo(400, 90)                               // draw another line
   .stroke();                                     // stroke the path
```

The output of this example looks like this:

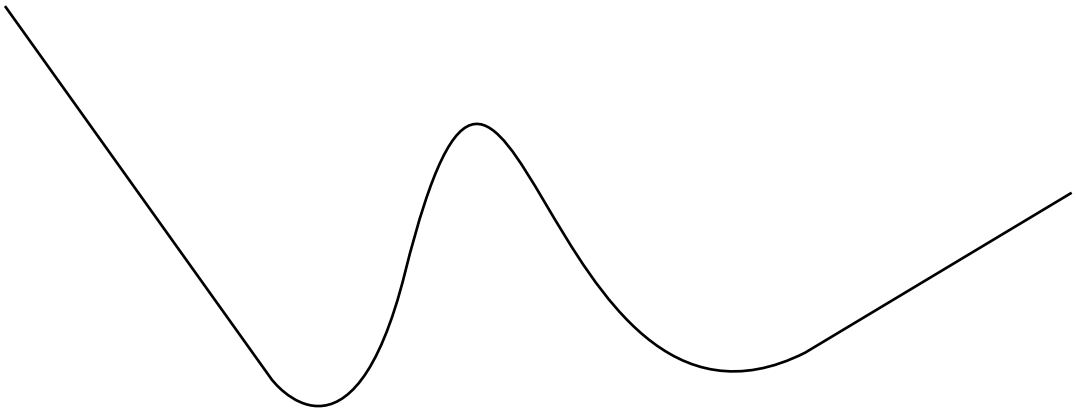


One thing to notice about this example is the use of method chaining. All methods in PDFKit are chainable, meaning that you can call one method right after the other without referencing the **`doc`** variable again. Of course, this is an option, so if you don't like how the code looks when chained, you don't have to write it that way.

SVG paths

PDFKit includes an SVG path parser, so you can include paths written in the SVG path syntax in your PDF documents. This makes it simple to include vector graphics elements produced in many popular editors such as Inkscape or Adobe Illustrator. The previous example could also be written using the SVG path syntax like this.

```
doc.path('M 0,20 L 100,160 Q 130,200 150,120 C 190,-40 200,200 300,150 L 400,90')
   .stroke()
```



The PDFKit SVG parser supports all of the command types supported by SVG, so any valid SVG path you throw at it should work as expected.

Shape helpers

PDFKit also includes some helpers that make defining common shapes much easier. Here is a list of the helpers.

`rect(x, y, width, height)`

`roundedRect(x, y, width, height, cornerRadius)`

`ellipse(centerX, centerY, radiusX, radiusY = radiusX)`

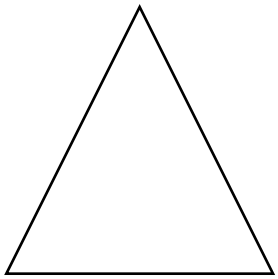
`circle(centerX, centerY, radius)`

`polygon(points...)`

The last one, **`polygon`**, allows you to pass in a list of points (arrays of x,y pairs), and it will create the shape by moving to the first point, and then drawing lines to each consecutive point. Here is how you'd draw a triangle with the polygon helper.

```
doc.polygon([100, 0], [50, 100], [150, 100]);
doc.stroke();
```

The output of this example looks like this:



Fill and stroke styles

So far we have only been stroking our paths, but you can also fill them with the **`fill`** method, and both fill and stroke the same path with the **`fillAndStroke`** method. Note that calling **`fill`** and then **`stroke`** consecutively will not work because of a limitation in the PDF spec. Use the **`fillAndStroke`** method if you want to accomplish both operations on the same path.

In order to make our drawings interesting, we really need to give them some style. PDFKit has many methods designed to do just that.

`lineWidth`

`lineCap`

`lineJoin`

`miterLimit`

`dash`

`fillColor`

`strokeColor`

`opacity`

`fillOpacity`

`strokeOpacity`

Some of these are pretty self explanatory, but let's go through a few of them.

Line cap and line join

The **`lineCap`** and **`lineJoin`** properties accept constants describing what they should do. This is best illustrated by an example.

```
// these examples are easier to see with a large line width
doc.lineWidth(25);

// line cap settings
doc.lineCap('butt')
   .moveTo(50, 20)
   .lineTo(100, 20)
   .stroke();

doc.lineCap('round')
   .moveTo(150, 20)
   .lineTo(200, 20)
   .stroke();

// square line cap shown with a circle instead of a line so you can see it
doc.lineCap('square')
   .moveTo(250, 20)
   .circle(275, 30, 15)
   .stroke();

// line join settings
doc.lineJoin('miter')
   .rect(50, 100, 50, 50)
   .stroke();

doc.lineJoin('round')
   .rect(150, 100, 50, 50)
   .stroke();

doc.lineJoin('bevel')
   .rect(250, 100, 50, 50)
   .stroke();
```

The output of this example looks like this.

Dashed lines

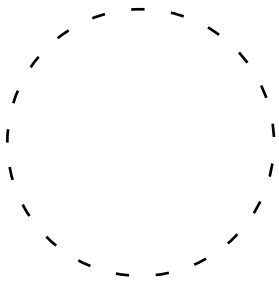
The **dash** method allows you to create non-continuous dashed lines. It takes a length specifying how long each dash should be, as well as an optional hash describing the additional properties **space** and **phase**. Lengths must be positive numbers; **dash** will throw if passed invalid lengths.

The **space** option defines the length of the space between each dash, and the **phase** option defines the starting point of the sequence of dashes. By default the **space** attribute is equal to the **length** and the **phase** attribute is set to **0**. You can use the **undash** method to make the line solid again.

The following example draws a circle with a dashed line where the space between the dashes is double the length of each dash.

```
doc.circle(100, 50, 50)
   .dash(5, {space: 10})
   .stroke();
```

The output of this example looks like this:



Color

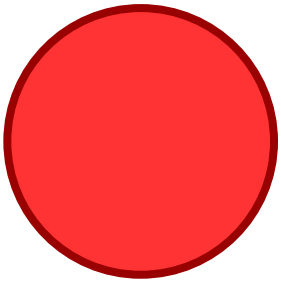
What is a drawing without color? PDFKit makes it simple to set the fill and stroke color and opacity. You can pass an array specifying an RGB or CMYK color, a hex color string, or use any of the named CSS colors.

The **`fillColor`** and **`strokeColor`** methods accept an optional second argument as a shortcut for setting the **`fillOpacity`** and **`strokeOpacity`**. Finally, the **`opacity`** method is a convenience method that sets both the fill and stroke opacity to the same value.

The **`fill`** and **`stroke`** methods also accept a color as an argument so that you don't have to call **`fillColor`** or **`strokeColor`** beforehand. The **`fillAndStroke`** method accepts both fill and stroke colors as arguments.

```
doc.circle(100, 50, 50)
   .lineWidth(3)
   .fillOpacity(0.8)
   .fillAndStroke("red", "#900")
```

This example produces the following output:



Gradients

PDFKit also supports gradient fills. Gradients can be used just like color fills, and are applied with the same methods (e.g. **`fillColor`**, or just **`fill`**). Before you can apply a gradient with these methods, however, you must create a gradient object.

There are two types of gradients: linear and radial. They are created by the **`linearGradient`** and **`radialGradient`** methods. Their function signatures are listed below:

`linearGradient(x1, y1, x2, y2)` – **`x1,y1`** is the start point, **`x2,y2`** is the end point

`radialGradient(x1, y1, r1, x2, y2, r2)` – **`r1`** is the inner radius, **`r2`** is the outer radius

Once you have a gradient object, you need to create color stops at points along that gradient. Stops are defined at percentage values (0 to 1), and take a color value (any usable by the fillColor method), and an optional opacity.

You can see both linear and radial gradients in the following example:

```
// Create a linear gradient
let grad = doc.linearGradient(50, 0, 150, 100);
grad.stop(0, 'green')
    .stop(1, 'red');

doc.rect(50, 0, 100, 100);
doc.fill(grad);

// Create a radial gradient
grad = doc.radialGradient(300, 50, 0, 300, 50, 50);
grad.stop(0, 'orange', 0)
    .stop(1, 'orange', 1);

doc.circle(300, 50, 50);
doc.fill(grad);
```

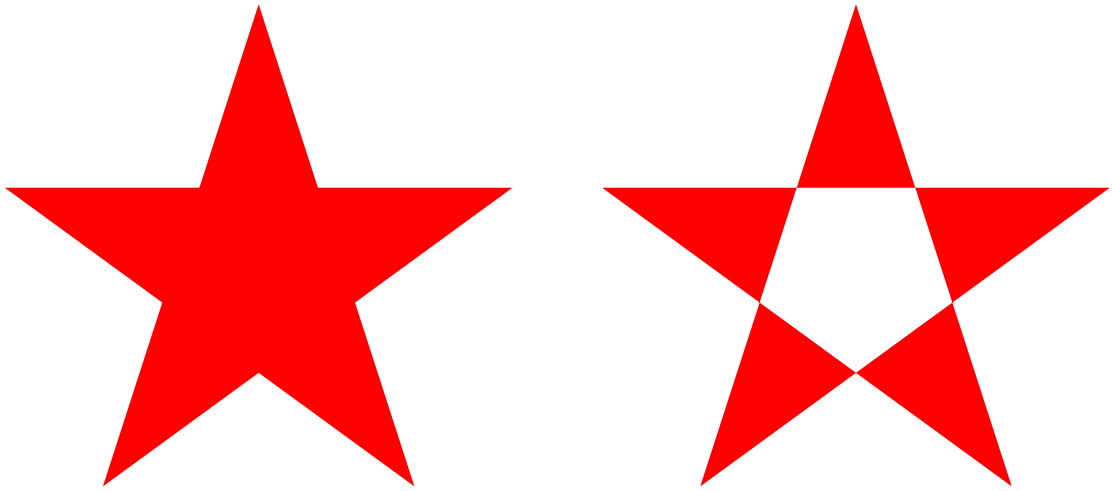
Here is the output from the this example:

Winding rules

Winding rules define how a path is filled and are best illustrated by an example. The winding rule is an optional attribute to the **`fill`** and **`fillAndStroke`** methods, and there are two values to choose from: **`non-zero`** and **`even-odd`**.

```
// Initial setup
doc.fillColor('red')
   .translate(-100, -50)
   .scale(0.8);

// Draw the path with the non-zero winding rule
doc.path('M 250,75 L 323,301 131,161 369,161 177,301 z')
   .fill('non-zero');

// Draw the path with the even-odd winding rule
doc.translate(280, 0)
   .path('M 250,75 L 323,301 131,161 369,161 177,301 z')
   .fill('even-odd');
```

You'll notice that I used the **`scale`** and **`translate`** transformations in this example. We'll cover those in a minute. The output of this example, with some added labels, is below.



Saving and restoring the graphics stack

Once you start producing more complex vector drawings, you will want to be able to save and restore the state of the graphics context. The graphics state is basically a snapshot of all the styles and transformations (see below) that have been applied, and many states can be created and stored on a stack. Every time the **`save`** method is called, the current graphics state is pushed onto the stack, and when you call **`restore`**, the last state on the stack is applied to the context again. This way, you can save the state, change some styles, and then restore it to how it was before you made those changes.

Transformations

Transformations allow you to modify the look of a drawing without modifying the drawing itself. There are three types of transformations available, as well as a method for setting the transformation matrix yourself. They are **`translate`**, **`rotate`** and **`scale`**.

The **`translate`** transformation takes two arguments, **`x`** and **`y`**, and effectively moves the origin of the page which is (0, 0) by default, to the left and right **`x`** and **`y`** units.

The **`rotate`** transformation takes an angle and optionally, an object with an **`origin`** property. It rotates the document **`angle`** degrees around the passed **`origin`** or by default, around the origin (top left corner) of the page.

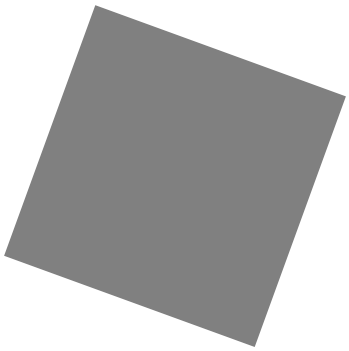
The **`scale`** transformation takes a scale factor and an optional **`origin`** passed in an options hash as with the **`rotate`** transformation. It is used to increase or decrease the size of the units in the drawing, or change its size. For example, applying a scale of **`0.5`** would make the drawing appear at half size, and a scale of **`2`** would make it appear twice as large.

If you are feeling particularly smart, you can modify the transformation matrix yourself using the **`transform`** method.

We used the **`scale`** and **`translate`** transformations above, so here is an example of using the **`rotate`** transformation. We'll set the origin of the rotation to the center of the rectangle.

```
doc.rotate(20, {origin: [150, 70]})
   .rect(100, 20, 100, 100)
   .fill('gray');
```

This example produces the following effect.

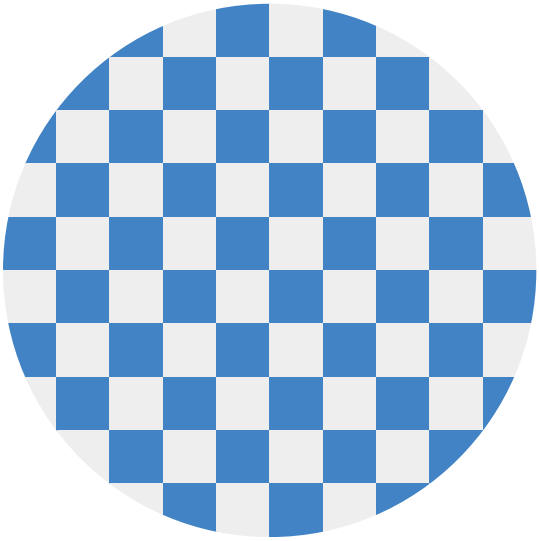


Clipping

A clipping path is a path defined using the normal path creation methods, but instead of being filled or stroked, it becomes a mask that hides unwanted parts of the drawing. Everything falling inside the clipping path after it is created is visible, and everything outside the path is invisible. Here is an example that clips a checkerboard pattern to the shape of a circle.

```
// Create a clipping path
doc.circle(100, 100, 100)
   .clip();

// Draw a checkerboard pattern
for (let row = 0; row < 10; row++) {
  for (let col = 0; col < 10; col++) {
    const color = (col % 2) - (row % 2) ? '#eee' : '#4183C4';
    doc.rect(row * 20, col * 20, 20, 20)
       .fill(color);
  }
}
```

The result of this example is the following:



If you want to "unclip", you can use the **`save`** method before the clipping, and then use **`restore`** to retrieve access to the whole page.

That's it for vector graphics in PDFKit. Now let's move on to learning about PDFKit's text support!

Text in PDFKit

The basics

PDFKit makes adding text to documents quite simple, and includes many options to customize the display of the output. Adding text to a document is as simple as calling the **`text`** method.

```
doc.text('Hello world!')
```

Internally, PDFKit keeps track of the current X and Y position of text as it is added to the document. This way, subsequent calls to the **`text`** method will automatically appear as new lines below the previous line. However, you can modify the position of text by passing X and Y coordinates to the **`text`** method after the text itself.

```
doc.text('Hello world!', 100, 100)
```

If you want to move down or up by lines, just call the **`moveDown`** or **`moveUp`** method with the number of lines you'd like to move (1 by default).

Line wrapping and justification

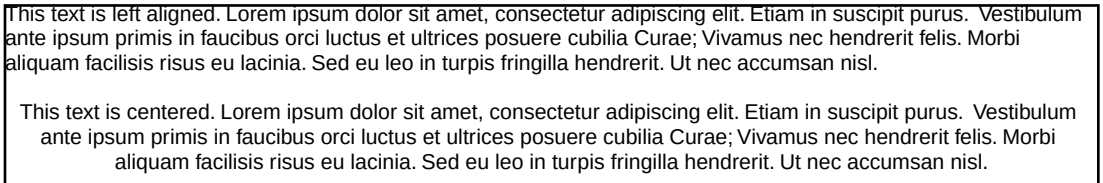
PDFKit includes support for line wrapping out of the box! If no options are given, text is automatically wrapped within the page margins and placed in the document flow below any previous text, or at the top of the page. PDFKit automatically inserts new pages as necessary so you don't have to worry about doing that for long pieces of text. PDFKit can also automatically wrap text into multiple columns.

The text will automatically wrap unless you set the **`lineBreak`** option to **`false`**. By default it will wrap to the page margin, but the **`width`** option allows you to set a different width the text should be wrapped to. If you set the **`height`** option, the text will be clipped to the number of lines that can fit in that height.

When line wrapping is enabled, you can choose a text justification. There are four options: **`left`** (the default), **`center`**, **`right`**, and **`justify`**. They work just as they do in your favorite word processor, but here is an example showing their use in a text box.

```
const lorem = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam in
suscipit purus.  Vestibulum ante ipsum primis in faucibus orci luctus et ultrices
posuere cubilia Curae; Vivamus nec hendrerit felis. Morbi aliquam facilisis risus eu
lacinia. Sed eu leo in turpis fringilla hendrerit. Ut nec accumsan nisl.';

doc.fontSize(8);
doc.text(`This text is left aligned. ${lorem}`, {
  width: 410,
  align: 'left'
}
);

doc.moveDown();
doc.text(`This text is centered. ${lorem}`, {
  width: 410,
  align: 'center'
}
);

doc.moveDown();
doc.text(`This text is right aligned. ${lorem}`, {
  width: 410,
  align: 'right'
}
);

doc.moveDown();
doc.text(`This text is justified. ${lorem}`, {
  width: 410,
  align: 'justify'
}
);

// draw bounding rectangle
doc.rect(doc.x, 0, 410, doc.y).stroke();
```

The output of this example, looks like this:



Text styling

PDFKit has many options for controlling the look of text added to PDF documents, which can be passed to the **`text`** method. They are enumerated below.

`lineBreak` - set to **`false`** to disable line wrapping all together

`width` - the width that text should be wrapped to (by default, the page width minus the left and right margin)

`height` - the maximum height that text should be clipped to

`ellipsis` - the character to display at the end of the text when it is too long. Set to **`true`** to use the default character.

`columns` - the number of columns to flow the text into

`columnGap` - the amount of space between each column (1/4 inch by default)

`indent` - the amount in PDF points (72 per inch) to indent each paragraph of text

`paragraphGap` - the amount of space between each paragraph of text

`lineGap` - the amount of space between each line of text

`wordSpacing` - the amount of space between each word in the text

`characterSpacing` - the amount of space between each character in the text

`fill` - whether to fill the text (**`true`** by default)

`stroke` - whether to stroke the text

`link` - a URL to link this text to (shortcut to create an annotation)

`goTo` - go to anchor (shortcut to create an annotation)

`destination` - create anchor to this text

`underline` - whether to underline the text

`strike` - whether to strike out the text

`oblique` - whether to slant the text (angle in degrees or **`true`**)

`baseline` - the vertical alignment of the text with respect to its insertion point (values as canvas textBaseline)

`continued` - whether the text segment will be followed immediately by another segment. Useful for changing styling in the middle of a paragraph.

`features` - an array of OpenType feature tags to apply. If not provided, a set of defaults is used.

Additionally, the fill and stroke color and opacity methods described in the vector graphics section are applied to text content as well.

Here is an example combining some of the options above, wrapping a piece of text into three columns, in a specified width and height.

```
const lorem = 'Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam in
suscipit purus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices
posuere cubilia Curae; Vivamus nec hendrerit felis. Morbi aliquam facilisis risus eu
lacinia. Sed eu leo in turpis fringilla hendrerit. Ut nec accumsan nisl. Suspendisse
rhoncus nisl posuere tortor tempus et dapibus elit porta. Cras leo neque, elementum a
rhoncus ut, vestibulum non nibh. Phasellus pretium justo turpis. Etiam vulputate,
odio vitae tincidunt ultricies, eros odio dapibus nisi, ut tincidunt lacus arcu eu
elit. Aenean velit erat, vehicula eget lacinia ut, dignissim non tellus. Aliquam nec
lacus mi, sed vestibulum nunc. Suspendisse potenti. Curabitur vitae sem turpis.
Vestibulum sed neque eget dolor dapibus porttitor at sit amet sem. Fusce a turpis
lorem. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere
cubilia Curae;';

doc.text(lorem, {
  columns: 3,
  columnGap: 15,
  height: 100,
  width: 465,
  align: 'justify'
});
```

The output looks like this:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam in suscipit purus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Vivamus nec hendrerit felis. Morbi aliquam facilisis risus eu lacinia. Sed eu leo in turpis fringilla hendrerit. Ut nec accumsan nisl. Suspendisse rhoncus nisl posuere tortor tempus et dapibus elit porta. Cras leo neque, elementum a rhoncus ut, vestibulum non nibh. Phasellus pretium justo turpis. Etiam vulputate, odio vitae tincidunt ultricies, eros odio dapibus nisi, ut tincidunt lacus arcu eu elit. Aenean velit erat, vehicula eget lacinia ut, dignissim non tellus. Aliquam nec lacus mi, sed vestibulum nunc. Suspendisse potenti. Curabitur vitae sem turpis. Vestibulum sed neque eget dolor dapibus porttitor at sit amet sem. Fusce a turpis lorem. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

Text measurements

If you're working with documents that require precise layout, you may need to know the size of a piece of text. PDFKit has two methods to achieve this: **`widthOfString(text, options)`** and **`heightOfString(text, options)`**. Both methods use the same options described in the Text styling section, and take into account the eventual line wrapping.

Lists

The **`list`** method creates a bulleted list. It accepts as arguments an array of strings, and the optional **`x`**, **`y`** position. You can create complex multilevel lists by using nested arrays. Lists use the following additional options:

`bulletRadius`

`textIndent`

`bulletIndent`

Rich Text

As mentioned above, PDFKit supports a simple form of rich text via the **`continued`** option. When set to true, PDFKit will retain the text wrapping state between **`text`** calls. This way, when you call text again after changing the text styles, the wrapping will continue right where it left off.

The options given to the first **`text`** call are also retained for subsequent calls after a **`continued`** one, but of course you can override them. In the following example, the **`width`** option from the first **`text`** call is retained by the second call.

```
doc.fillColor('green')
   .text(lorem.slice(0, 500), {
     width: 465,
     continued: true
   }).fillColor('red')
   .text(lorem.slice(500));
```

Here is the output:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam in suscipit purus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Vivamus nec hendrerit felis. Morbi aliquam facilisis risus eu lacinia. Sed eu leo in turpis fringilla hendrerit. Ut nec accumsan nisl. Suspendisse rhoncus nisl posuere tortor tempus et dapibus elit porta. Cras leo neque, elementum a rhoncus ut, vestibulum non nibh. Phasellus pretium justo turpis. Etiam vulputate, odio vitae tincidunt ultricies, eros odio dapibus nisi, ut tincidunt lacus arcu eu elit. Aenean velit erat, vehicula eget lacinia ut, dignissim non tellus. Aliquam nec lacus mi, sed vestibulum nunc. Suspendisse potenti. Curabitur vitae sem turpis. Vestibulum sed neque eget dolor dapibus porttitor at sit amet sem. Fusce a turpis lorem. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae;

To cancel a link in rich text set the **`link`** option to **`null`**.

```
doc.fillColor('red')
   .text(lorem.slice(0, 199), {
     width: 465,
     continued: true
   })
   .fillColor('blue')
   .text(lorem.slice(199, 282), {
     link: 'http://www.example.com',
     continued: true
   })
   .fillColor('green')
   .text(lorem.slice(182, 400), {
     link: null
   });
```

Here is the output:

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Etiam in suscipit purus. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Vivamus nec hendrerit felis. Morbi aliquam facilisis risus eu lacinia. Sed eu leo in turpis fringilla hendrerit.hendrerit felis. Morbi aliquam facilisis risus eu lacinia. Sed eu leo in turpis fringilla hendrerit. Ut nec accumsan nisl. Suspendisse rhoncus nisl posuere tortor tempus et dapibus elit porta. Cras leo neque, elementum

Fonts

The PDF format defines 14 standard fonts that can be used in PDF documents. PDFKit supports each of them out of the box. Besides Symbol and Zapf Dingbats this includes 4 styles (regular, bold, italic/oblique, bold+italic) of Helvetica, Courier, and Times. To switch between standard fonts, call the **`font`** method with the corresponding Label:

`'Courier'`

`'Courier-Bold'`

`'Courier-Oblique'`

`'Courier-BoldOblique'`

`'Helvetica'`

`'Helvetica-Bold'`

`'Helvetica-Oblique'`

`'Helvetica-BoldOblique'`

`'Symbol'`

`'Times-Roman'`

`'Times-Bold'`

`'Times-Italic'`

`'Times-BoldItalic'`

`'ZapfDingbats'`

The PDF format also allows fonts to be embedded right in the document. PDFKit supports embedding TrueType (**`.ttf`**), OpenType (**`.otf`**), WOFF, WOFF2, TrueType Collection (**`.ttc`**), and Datafork TrueType (**`.dfont`**) fonts.

To change the font used to render text, just call the **`font`** method. If you are using a standard PDF font, just pass the name to the **`font`** method. Otherwise, pass the path to the font file, or a **`Buffer`** containing the font data. If the font is a collection font (**`.ttc`** and **`.dfont`** files), meaning that it contains multiple styles in the same file, you should pass the name of the style to be extracted from the collection.

Here is an example showing how to set the font in each case.

```
// Set the font size
doc.fontSize(18);

// Using a standard PDF font
doc.font('Times-Roman')
   .text('Hello from Times Roman!')
   .moveDown(0.5);

// Using a TrueType font (.ttf)
doc.font('fonts/GoodDog.ttf')
   .text('This is Good Dog!')
   .moveDown(0.5);
```

```
// Using a collection font (.ttc or .dfont)
doc.font('fonts/Chalkboard.ttc', 'Chalkboard-Bold')
   .text('This is Chalkboard, not Comic Sans.');
```

The output of this example looks like this:

Another nice feature of the PDFKit font support, is the ability to register a font file under a name for use later rather than entering the path to the font every time you want to use it.

```
// Register a font
doc.registerFont('Heading Font', 'fonts/Chalkboard.ttc', 'Chalkboard-Bold');

// Use the font later
doc.font('Heading Font')
   .text('This is a heading.');
```

That's about all there is too it for text in PDFKit. Let's move on now to images.

Images in PDFKit

Adding images to PDFKit documents is an easy task. Just pass an image path, buffer, or data uri with base64 encoded data to the **`image`** method along with some optional arguments. PDFKit supports the JPEG and PNG formats. If an X and Y position are not provided, the image is rendered at the current point in the text flow (below the last line of text). Otherwise, it is positioned absolutely at the specified point. The image will be scaled according to the following options.

Neither **`width`** or **`height`** provided - image is rendered at full size

`width` provided but not **`height`** - image is scaled proportionally to fit in the provided **`width`**

`height` provided but not **`width`** - image is scaled proportionally to fit in the provided **`height`**

Both **`width`** and **`height`** provided - image is stretched to the dimensions provided

`scale` factor provided - image is scaled proportionally by the provided scale factor

`fit` array provided - image is scaled proportionally to fit within the passed width and height

`cover` array provided - image is scaled proportionally to completely cover the rectangle defined by the passed width and height

`link` - a URL to link this image to (shortcut to create an annotation)

`goTo` - go to anchor (shortcut to create an annotation)

`destination` - create anchor to this image

`ignoreOrientation` - (true/false) ignore JPEG EXIF orientation. By default, images with JPEG EXIF orientation are properly rotated and/or flipped. Defaults to **`false`**, unless **`ignoreOrientation`** option set to **`true`** when creating the **`PDFDocument`** object (e.g. **`new PDFDocument({ignoreOrientation: true})`**)

When a **`fit`** or **`cover`** array is provided, PDFKit accepts these additional options:

`align` - horizontally align the image, the possible values are **`'left'`**, **`'center'`** and **`'right'`**

`valign` - vertically align the image, the possible values are **`'top'`**, **`'center'`** and **`'bottom'`**

Here is an example showing some of these options.

```
// Scale proprotionally to the specified width
doc.image('images/test.jpeg', 0, 15, {width: 300})
   .text('Proportional to width', 0, 0);

// Fit the image within the dimensions
doc.image('images/test.jpeg', 320, 15, {fit: [100, 100]})
   .rect(320, 15, 100, 100)
   .stroke()
   .text('Fit', 320, 0);

// Stretch the image
doc.image('images/test.jpeg', 320, 145, {width: 200, height: 100})
   .text('Stretch', 320, 130);
```

```
// Scale the image
doc.image('images/test.jpeg', 320, 280, {scale: 0.25})
   .text('Scale', 320, 265);

// Fit the image in the dimensions, and center it both horizontally and vertically
doc.image('images/test.jpeg', 430, 15, {fit: [100, 100], align: 'center', valign:
'center'})
   .rect(430, 15, 100, 100).stroke()
   .text('Centered', 430, 0);
```

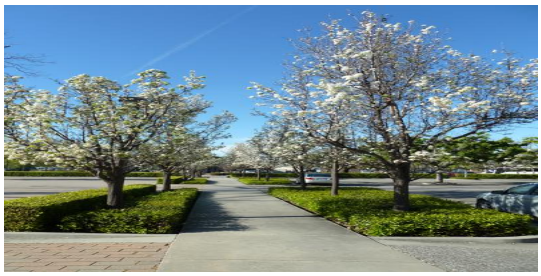
This example produces the following output:

Proportional to width

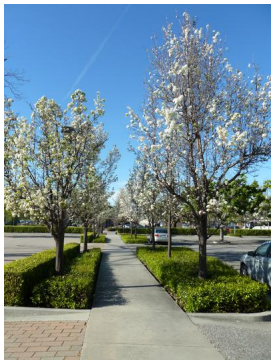
Fit

Centered

Stretch



Scale



That is all there is to adding images to your PDF documents with PDFKit. Now let's look at adding outlines.

Outlines in PDFKit

Outlines are the heirachical bookmarks that display in some PDF readers. Currently only page bookmarks are supported, but more may be added in the future. They are simple to add and only require a single method:

`addItem(title, options)`

Here is an example of adding a bookmark with a single child bookmark.

```
// Get a reference to the Outline root
const { outline } = doc;

// Add a top-level bookmark
const top = outline.addItem('Top Level');

// Add a sub-section
top.addItem('Sub-section');
```

Options

The **`options`** parameter currently only has one property: **`expanded`**. If this value is set to **`true`** then all of that section's children will be visible by default. This value defaults to **`false`**.

In this example the 'Top Level' section will be expanded to show 'Sub-section'.

```
// Add a top-level bookmark
const top = outline.addItem('Top Level', { expanded: true });

// Add a sub-section
top.addItem('Sub-section');
```

Annotations in PDFKit

Annotations are interactive features of the PDF format, and they make it possible to include things like links and attached notes, or to highlight, underline or strikeout portions of text. Annotations are added using the various helper methods, and each type of annotation is defined by a rectangle and some other properties. Here is a list of the available annotation methods:

```
note(x, y, width, height, contents, options)
link(x, y, width, height, url, options)
goTo(x, y, w, h, name, options)
highlight(x, y, width, height, options)
underline(x, y, width, height, options)
strike(x, y, width, height, options)
lineAnnotation(x1, y1, x2, y2, options)
rectAnnotation(x, y, width, height, options)
ellipseAnnotation(x, y, width, height, options)
textAnnotation(x, y, width, height, text, options)
fileAnnotation(x, y, width, height, file, options)
```

Many of the annotations have a **`color`** option that you can specify. You can use an array of RGB values, a hex color, or a named CSS color value for that option.

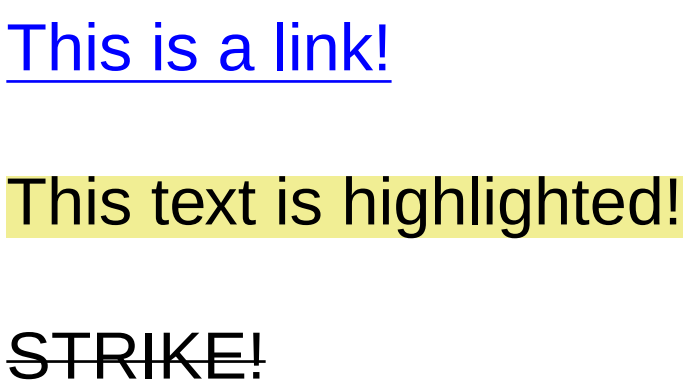
If you are adding an annotation to a piece of text, such as a link or underline, you will need to know the width and height of the text in order to create the required rectangle for the annotation. There are two methods that you can use to do that. To get the width of any piece of text in the current font, just call the **`widthOfString`** method with the string you want to measure. To get the line height in the current font, just call the **`currentLineHeight`** method.

You must remember that annotations have a stacking order. If you are putting more than one annotation on a single area and one of those annotations is a link, make sure that the link is the last one you add, otherwise it will be covered by another annotation and the user won't be able to click it.

Here is an example that uses a few of the annotation types.

```
// Add the link text
doc.fontSize(25)
   .fillColor('blue')
   .text('This is a link!', 20, 0);

// Measure the text
const width = doc.widthOfString('This is a link!');
const height = doc.currentLineHeight();

// Add the underline and link annotations
doc.underline(20, 0, width, height, {color: 'blue'})
   .link(20, 0, width, height, 'http://google.com/');

// Create the highlighted text
doc.moveDown()
   .fillColor('black')
   .highlight(20, doc.y, doc.widthOfString('This text is highlighted!'), height)
   .text('This text is highlighted!');

// Create the crossed out text
doc.moveDown()
   .strike(20, doc.y, doc.widthOfString('STRIKE!'), height)
   .text('STRIKE!');

// Adding go to as annotation
doc.goTo(20, doc.y, 10, 20, 'LINK', {});
```

The output of this example looks like this.



Annotations are currently not the easiest things to add to PDF documents, but that is the fault of the PDF spec itself. Calculating a rectangle manually isn't fun, but PDFKit makes it easier for a few common annotations applied to text, including links, underlines, and strikes. Here's an example showing two of them:

```
doc.fontSize(20)
   .fillColor('red')
   .text('Another link!', 20, 0, {
     link: 'http://apple.com/',
```

```
    underline: true
  }
);
```

The output is as you'd expect:

Forms in PDFKit

Forms are an interactive feature of the PDF format. Forms make it possible to add form annotations such as text fields, combo boxes, buttons and actions. Before addings forms to a PDF you must call the document **`initForm()`** method.

`initForm()` - Must be called before adding a form annotation to the document.

```
javascript doc.font('Helvetica'); // establishes the default form field font
doc.initForm();
```

Form Annotation Methods

Form annotations are added using the following document methods.

formText(name, x, y, width, height, options)

formPushButton(name, x, y, width, height, name, options)

formCombo(name, x, y, width, height, options)

formList(name, x, y, width, height, options)

The above methods call the **formAnnotation** method with type set to one of **text**, **pushButton**, **radioButton**, **combo** or **list**.

formAnnotation(name, type, x, y, width, height, options)

name Parameter

Form annotations are each given a **name** that is used for identification. Field names are hierarchical using a period ('.') as a separaror (e.g. shipping.address.street). More than one form field can have the same name. When this happens, the fields will have the same value. There is more information on **name** in the Field Names section below.

options Parameter

Common Options

Form Annotation **options** that are common across all form annotation types are:

required [boolean] - The field must have a value by the time the form is submitted.

noExport [boolean] - The field will not be exported if a form is submitted.

readOnly [boolean] - The user may not change the value of the field, and the field will not respond to mouse clicks. This is useful for fields that have computed values.

value [number|string] - The field's value.

defaultValue [number|string] - The default value to which the field reverts if a reset-form action is executed.

Some form annotations have **color** options. You can use an array of RGB values, a hex color, or a named CSS color value for that option.

backgroundColor - field background color

borderColor - field border color

Text Field Options

align [string] - Sets the alignment to **left**, **center** or **right**.

multiline [boolean] - Allows the field to have multiple lines of text.

password [boolean] - The text will be masked (e.g. with asterisks).

noSpell [boolean] - If set, text entered in the field is not spell-checked

format [object] - See the section on Text Field Formatting below.

fontSize [number] - Sets the fontSize (default or 0 means auto sizing)

```
js doc.formText('leaf2', 10, 60, 200, 40, { multiline: true, align: 'right',
format: { type: 'date', params: 'm/d' } });
```

Combo and List Field Options

sort [boolean] - The field options will be sorted alphabetically.

edit [boolean] - (combo only) Allow the user to enter a value in the field.

multiSelect [boolean] - Allow more than one choice to be selected.

noSpell [boolean] - (combo only) If set and **edit** is true, text entered in the field is not spell-checked.

select [array] - Array of choices to display in the combo or list form field.

```
js opts = { select: ['', 'github', 'bitbucket', 'gitlab'], value: '',
defaultValue: '', align: 'left' }; doc.formCombo('ch1', 10, y, 100, 20,
opts);
```

Button Field Options

label [string] - Sets the label text. You can also set an icon, but for this you will need to 'expert-up' and dig deeper into the PDF Reference manual.

```
js var opts = { backgroundColor: 'yellow', label: 'Test Button' };
doc.formPushButton('btn1', 10, 200, 100, 30, opts);
```

Text Field Formatting

When needing to format the text value of a Form Annotation, the following **`options`** are available. This will cause predefined document JavaScript actions to automatically format the text. Refer to the section Formatting scripts in Acrobat Forms Plugin of the Acrobat SDK documentation for more information.

Add a format dictionary to **`options`**. The dictionary must contain a **`type`** attribute.

`format` - generic object

`format.type` - value must be one of **`date`**, **`time`**, **`percent`**, **`number`**, **`zip`**, **`zipPlus4`**, **`phone`** or **`ssn`**.

When **`type`** is **`date`**, **`time`**, **`percent`** or **`number`** the format dictionary must contain additional parameters as described below.

Date format

`format.param` (string) - specifies the value and display format and can include:**`d`** - single digit day of month

`dd` - double digit day of month

`m` - month digit

`mm` - month double digit

`mmm` - abbreviated month name

`mmmm` - full month name

`yy` - two digit year

`yyyy` - four digit year

`hh` - hour for 12 hour clock

`HH` - hour for 24 hour clock

`MM` - two digit minute

`tt` - am or pm

```
js // Date text field formatting doc.formText('field.date', 10, 60, 200, 40,
{ align: 'center', format: { type: 'date', param: 'mmmm d, yyyy' } });
```

Time format

`format.param` - value must be a number between 0 and 3, representing the formats "14:30", "2:30 PM", "14:30:15" and "2:30:15 PM".

```
js // Time text field formatting doc.formText('field.time', 10, 60, 200, 40,
{ align: 'center', format: { type: 'time', param: 2 } });
```

Number and percent format

format.nDec [number] - the number of places after the decimal point

format.sepComma [boolean] - display a comma separator, otherwise do not display a separator.

format.negStyle string

- the value must be one of **MinusBlack** , **Red**, **ParensBlack**, **ParensRed**

format.currency string

- a currency symbol to display

format.currencyPrepend boolean

- set to true to prepend the currency symbol

```
js // Currency text field formatting doc.formText('leaf2', 10, 60, 200, 40,
{ multiline: true, align: 'right', format: { type: 'number', nDec: 2,
sepComma: true, negStyle: 'ParensRed', currency: '$', currencyPrepend:
true } });
```

Field Names

Form Annotations are, by default, added to the root of the PDF document. A PDF form is organized in a name heirarchy, for example shipping.address.street. Capture this heirarchy either by setting the **`name`** of each form annotation with the full hierarchical name (e.g. shipping.address.street) or by creating a hierarchy of form fields and form annotations and refering to a form field or form annotations parent using **`options.parent`**.

A form field is an invisible node in the PDF form and is created using the document **`formField`** method. A form field must include the node's **`name`** (e.g. shipping) and may include other information such as the default font that is to be used by all child form annotations.

Using the **`formField`** method you might create a shipping field that is added to the root of the document, an address field that refers to the shipping field as it's parent, and a street Form Annotation that would refer to the address field as it's parent.

Create form fields using the document method:

`formField( name, options )` - returns a reference to the field

-- Example PDF using field hierarchy, three text fields and a push button --

`javascript doc.font('Helvetica'); // establishes the default font doc.initForm();

let rootField = doc.formField('rootField'); let child1Field = doc.formField('child1Field', { parent: rootField }); let child2Field = doc.formField('child2Field', { parent: rootField });

// Add text form annotation 'rootField.child1Field.leaf1' doc.formText('leaf1', 10, 10, 200, 40, { parent: child1Field, multiline: true }); // Add text form annotation 'rootField.child1Field.leaf2' doc.formText('leaf2', 10, 60, 200, 40, { parent: child1Field, multiline: true }); // Add text form annotation 'rootField.child2Field.leaf1' doc.formText('leaf1', 10, 110, 200, 80, { parent: child2Field, multiline: true });

// Add push button form annotation 'btn1' var opts = { backgroundColor: 'yellow', label: 'Test Button' }; doc.formPushButton('btn1', 10, 200, 100, 30, opts);
`

The output of this example looks like this.

Advanced Form Field Use

Forms can be quite complicated and your needs will likely grow to sometimes need to directly specify the attributes that will go into the Form Annotation or Field dictionaries. Consult the PDF Reference and set these attributes in the **`options`** object. Any options that are not listed above will be added directly to the corresponding PDF Object.

Font

The font used for a Form Annotation is set using the **`document.font`** method. Yes that's the same method as is used when setting the text font.

The **`font`** method must be called before **`initForm`** and may be called before **`formField`** or any of the form annotation methods.

```
js doc.font('Courier'); doc.formText('myfield', 10, 10, 200, 20);
```

Named JavaScript

In support of Form Annotations that execute JavaScript in PDF, you may use the following document method:

`addNamedJavaScript( name, string )`

Limitations

It is recommended that you test your PDF form documents across all platforms and viewers that you wish to support.

Form Field Appearances

Form elements must each have an appearance set using the **`AP`** attribute of the annotation. If this attribute is not set, the form element's value may not be visible. Because appearances can be complex to generate, Adobe Acrobat has an option to build these apperances from form values and Form Annotation attributes when a PDF is first opened. To do this PDFKit always sets the Form dictionary's **`NeedAppearances`** attribute to true. This could mean that the PDF will be dirty upon open, meaning it will need to be saved.

The **`NeedAppearances`** flag may not be honored by all PDF viewers.

Some form documents may not need to generate appearances. This may be the case for text Form Annotations that initially have no value. This is not true for push button widget annotations. Please test

Document JavaScript

Many PDF Viewers, aside from Adobe Acrobat Reader, do not implement document JavaScript. Even Adobe Readers may not implement document JavaScript where it is not permitted by a device's app store terms of service (e.g. iOS devices).

Radio and Checkboxes

Support for radio and checkboxes requires a more advanced attention to their rendered appearances and are not supported in this initial forms release.

Destinations

Anchor may specify a destination by **`addNamedDestination(name, ...args)`**, which consists of a page, the location of the display window on that page, and the zoom factor to use when displaying that page.

Examples of creating anchor:

```
// Insert anchor for current page
doc.addNamedDestination('LINK');

// Insert anchor for current page with only horizontal magnified to fit where
vertical top is 100
doc.addNamedDestination('LINK', 'FitH', 100);

// Insert anchor to display a portion of the current page, 1/2 inch in from the top
and left and zoomed 50%
doc.addNamedDestination('LINK', 'XYZ', 36, 36, 50);

// Insert anchor for this text
doc.text('End of paragraph', { destination: 'ENDP' });
```

Examples of go to link to anchor:

```
// Go to annotation
doc.goTo(10, 10, 100, 20, 'LINK')

// Go to annotation for this text
doc.text('Another goto', 20, 0, {
  goTo: 'ENDP',
  underline: true
});
```

Attachments in PDFKit

Embedded Files

Embedded files make it possible to embed any external file into a PDF. Adding an embedded file is as simple as calling the **`file`** method and specifying a filepath.

```
doc.file(path.join(__dirname, 'example.txt'))
```

It is also possible to embed data directly as a Buffer, ArrayBuffer or base64 encoded string. If you are embedding data, it is recommended you also specify a filename like this:

```
doc.file(Buffer.from('this will be a text file'), { name: 'example.txt' })
```

When embedding a data URL, the **`type`** option will be set to the data URL's MIME type automatically:

```
doc.file('data:text/plain;base64,YmFzZTY0IHN0cmluZw==', { name: 'base64.txt' })
```

There are a few other options for **`doc.file`**:

`name` - specify the embedded file's name

`type` - specify the embedded file's subtype as a MIME-Type

`description` - add descriptive text for the embedded file

`hidden` - if true, do not show file in the list of embedded files

`creationDate` - override the date and time the file was created

`modifiedDate` - override the date and time the file was last updated

`relationship` - relationship between the PDF document and its attached file. Can be 'Alternative', 'Data', 'Source', 'Supplement' or 'Unspecified'.

If you are attaching a file from your file system, creationDate and modifiedDate will be set to the source file's creationDate and modifiedDate.

Setting the **`hidden`** option prevents this file from showing up in the pdf viewer's attachment panel. While this may not be very useful for embedded files, it is absolutely necessary for file annotations, to prevent them from showing up twice in the attachment panel.

File Annotations

A file annotation contains a reference to an embedded file that can be placed anywhere in the document. File annotations show up in your reader's annotation panel as well as the attachment panel.

In order to add a file annotation, you should first read the chapter on annotations. Like other annotations, you specify position and size with **`x`**, **`y`**, **`width`** and **`height`**, unlike other annotations you must also specify a file object. The file object may contain the same options as **`doc.file`** in the previous section with the addition of the source file or buffered data in **`src`** .

Here is an example of adding a file annotation:

```
const file = {
  src: path.join(__dirname, 'example.txt'),
  name: 'example.txt',
  description: 'file annotation description'
}
const options = { Name: 'Paperclip' }

doc.fileAnnotation(100, 100, 100, 100, file, options)
```

The annotation's appearance may be changed by setting the **`Name`** option to one of the three predefined icons **`GraphPush`**, **`Paperclip`** or **`Push`** (default value).

Accessibility

Accessible PDFs are usable by visually impaired users who rely on screen readers/text-to-speech engines/vocalisation.

The two main tasks required to create accessible PDFs are marking content and defining the document's logical structure. These are detailed in the following sections.

Some other simpler tasks are also required.

This checklist covers everything that is required to create a conformant PDF/UA (PDF for Universal Accessibility) document (which is an extension of Tagged PDF):

Pass the option **`pdfVersion: '1.5'`** (or a higher version) when creating your **`PDFDocument`** (depending on the features you use, you may only need 1.4; refer to the PDF reference for details).

Pass the option **`subset: 'PDF/UA'`** when creating your **`PDFDocument`** (if you wish the PDF to be identified as PDF/UA-1).

Pass the option **`tagged: true`** when creating your **`PDFDocument`** (technically, this sets the **`Marked`** property in the **`Markings`** dictionary to **`true`** in the PDF).

Provide a **`Title`** in the **`info`** option, and pass **`displayTitle: true`** when creating your **`PDFDocument`**.

Specify natural language in the document options and/or logical structure and/or non-structure marked **`Span`** content.

Add logical structure with all significant content included.

Include accessibility information (such as alternative text, actual text, etc.) in the logical structure and/or non-structure marked **`Span`** content.

Include all spaces which separate words/sentences/etc. in your marked structure content, even at the ends of lines, paragraphs, etc.. I.e. don't do **`doc.text("Hello, world!")`** but instead do **`doc.text("Hello, world! ")`**.

Mark all non-structure content as artifacts.

As well as creating the logical structure, write objects to the PDF in the natural "reading order".

Do not convey information solely using visuals (such as colour, contrast or position on the page).

No flickering or flashing content.

Marked Content

Marked content sequences are foundational to creating accessible PDFs.

All marked content sequences are associated with a registered tag, such as 'Span'.

Example of marking content:

```
// Mark some text as a "Span"
doc.markContent('Span');
doc.text('Hello, world! ');
doc.endMarkedContent();
```

Marked content is automatically ended when a page is ended, and if a new page is automatically added by text wrapping, marking is automatically begun again on the new page.

Tags to use are listed in a later section.

Marked Content Options

When marking content, you can provide options (take care to use correct capitalisation):

`type` – used for artifact content; may be **`Pagination`** (e.g. headers and footers), **`Layout`** (e.g. rules and backgrounds) or **`Page`** (cut marks etc.)

`bbox` – bounding box for artifact content: **`[left, top, right, bottom]`** in default coordinates

`attached` – used for **`Pagination`** artifact content, array of one or more strings: **`Top`**, **`Bottom`**, **`Left`**, **`Right`**

`lang` – used for **`Span`** content: human language code (e.g. **`en-AU`**) which overrides default document language, and any enclosing structure element language

`alt` – used for **`Span`** content: alternative text for an image or other visual content

`expanded` – used for **`Span`** content: the expanded form of an abbreviation or acronym

`actual` – used for **`Span`** content: the actual text the content represents (e.g. if it is rendered as vector graphics)

It is advisable not to use **`Span`** content for specifying alternative text, expanded form, or actual text, especially if there is a possibility of the content automatically wrapping, which would result in the text appearing twice. Set these options on an associated structure element instead.

Logical Structure

Logical structures defines the reading order of a document, and can provide alternative text for images and other visual content.

To define logical structure, you need to mark the structure content, keep a reference to it, then incorporate it into a structure tree.

So far, PDFKit only supports marked content in the logical structure, not annotations, forms, or anything else.

Example of marking structure content:

```
// Mark some text as a paragraph ("P"); the tag should match the intended structure element's type
const myStructContent = doc.markStructureContent('P');
doc.text('Hello, world! ');
doc.endMarkedContent();
```

Example of the simplest of structure trees:

```
// Add a single structure element which includes the structure content to the document's structure
doc.addStructure(doc.struct('P', myStructContent));
```

Tags/element types to use are listed in a later section.

Note that to be conformant to Tagged PDF, all content not part of the logical structure should be marked as **`Artifact`**.

Automatic Ending of Structure Content and Artifacts

Structure content does not nest, and is mutually exclusive with artifact content; marking structure or artifact content will automatically end current marking of structure or artifact content (and any descendent marking):

```
// Mark multiple paragraphs without needing to close them
doc.markContent('Artifact', { type: "Layout" });
doc.rect(x1, y1, w1, h1);
const myStructContent = doc.markStructureContent('P');
doc.text('Hello, world! ');
doc.markContent('Artifact', { type: "Layout" });
doc.rect(x2, y2, w2, h2);
const myStructContent = doc.markStructureContent('P');
doc.markContent('Span');
doc.text('Bonjour, tout le monde! ');
doc.markContent('Artifact', { type: "Layout" });
doc.rect(x3, y3, w3, h3);
const myStructContent = doc.markStructureContent('P');
doc.text('Hello again! ');
```

Complex Structure

Multiple elements may be added directly to the document, or to structure elements, and may nest:

```
// Create nested structure elements
const section1 = doc.struct('Sect', [
    doc.struct('P', [
        someTextStructureContent,
        doc.struct('Link', someLinkStructureContent),
        moreTextStructureContent
    ])
]);
const section2 = doc.struct('Sect', secondSectionStructureContent);

// Add them to the document's structure
doc.addStructure(section1).addStructure(section2);
```

Incremental Construction of Structure

Structure can be built incrementally. Elements can optionally be (recursively) ended once you have finished adding to them, allowing them to be flushed out as soon as possible:

```
// Begin a new section and add it to the document's structure
const mySection = doc.struct('Sect');
doc.addStructure(mySection);

// Create a new paragraph and add it to the section
const myParagraph = doc.struct('P');
mySection.add(myParagraph);

// Add content, both to the page, and the paragraph
const myParagraphContent = doc.markStructureContent('P');
myParagraph.add(myParagraphContent);
doc.text('Hello, world! ');

// End the paragraph, allowing it to be flushed out, freeing memory
myParagraph.end();
```

Note that if you provide children when creating a structure element (i.e. providing them to **`doc.struct()`** rather than using **`structElem.add()`**) then **`structElem.end()`** is called automatically. You therefore cannot add additional children with **`structElem.add()`**, i.e. you cannot mix atomic and incremental styles for the same structure element.

For an element to be flushed out, it must:

be ended,

have been added to its parent, and

if it has content defined through closures (see next section), be attached to the document's structure (through its ancestors)

When you call **`doc.end()`**, the document's structure is recursively ended, resulting in all elements being flushed out. If you created elements but forgot to add them to the document's structure, they will not be flushed, but the PDF stream will wait for them to be flushed before ending, causing your application to hang. Make sure if you create any

elements, you add them to a parent, so ultimately all elements are attached to the document. It's best to add elements to their parents as you go.

Shortcut for Elements Containing Only Marked Content

The common case where a structure element contains only content marked with a tag matching the structure element type can be achieved by using a closure:

```
doc.addStructure(doc.struct('P', () => {
    doc.text('Hello, world! ');
}));
```

This is equivalent to:

```
const myStruct = doc.struct('P');
doc.addStructure(myStruct);
const myStructContent = doc.markStructureContent('P');
doc.text('Hello, world! ');
doc.endMarkedContent();
myStruct.add(myStructContent);
myStruct.end();
```

Note that the content is marked and the closure is executed if/when the element is attached to the document's structure. This means that you can do something like this:

```
const myParagraph = doc.struct('P', [
    () => { doc.text("Please see ", { continued: true }); },
    doc.struct('Link', () => {
        doc.text("something", { link: "http://www.example.com/", continued: true });
    }),
    () => { doc.text(" for details. ", { link: null }); }
]);
```

and no content will be added to the page until/unless something like this is done:

```
doc.addStructure(section1);
section1.add(myParagraph); // Content is added now
```

or alternatively:

```
section1.add(myParagraph);
doc.addStructure(section1); // Content is added now
```

This is important because otherwise when the **`Link`** element is constructed, its content will be added to the page, and then the list containing the link element will be passed to the construct the **`P`** element, and only during the construction of the **`P`** element will the other **`P`** content be added to the page, resulting in page content being out of order. It's best to add elements to their parents as you go.

Structure Element Options

When creating a structure element, you can provide options:

`title` - title of the structure element (e.g. "Chapter 1")

`lang` – human language code (e.g. **`en-AU`**) which overrides default document language

`alt` – alternative text for an image or other visual content

`expanded` – the expanded form of an abbreviation or acronym

`actual` – the actual text the content represents (e.g. if it is rendered as vector graphics)

Example of a structure tree with options specified:

```
const titlePage = doc.struct('Sect', {
    title: 'Title Page'
}, [
    doc.struct('H', [
        doc.struct('Span', {
            expanded: 'Portable Document Format for Universal Accessibility',
            actual: 'PDF/UA'
        }, [
            pdfUAStructureContent
        ]),
        doc.struct('Span', {
            actual: 'in a Nutshell'
        }, [
            inANutshellStructureContent
        ]),
    ]),
    doc.struct('Figure', {
        alt: 'photo of a concrete path with tactile paving'
    }, [
        photoStructureContent
    ])
]);
```

Automatic Marking and Structure Construction for Text

The **`text()`** method accepts a **`structParent`** option which you can use to specify a structure element to add each paragraph to. It will mark each paragraph of content, create a structure element for it, and then add it to the parent element you provided. It will use the **`P`** type, unless you specify a different type with a **`structType`** option.

Example of creating structure automatically with **`text()`**:

```
// Create a section, add it to the document structure, then add paragraphs to it
const section = doc.struct('Sect');
doc.addStructure(section);
doc.text("Foo. \nBar. ", { structParent: section });
```

This is equivalent to:

```
const section = doc.struct('Sect');
doc.addStructure(section);
section.add(doc.struct('P', () => { doc.text("Foo. "); }));
section.add(doc.struct('P', () => { doc.text("Bar. "); }));
```

The **`list()`** method also accepts a **`structParent`** option. By default, it add list items (type **`LI`**) to the parent, each of which contains a label (type **`Lbl`**, which holds the bullet, number, or

letter) and a body (type **`LBody`**, which holds the actual item content). You can override the default types with a **`structTypes`** option, which is a list: **`[ itemType, labelType, bodyType ]`**. You can make any of the types **`null`** to omit that part of the structure (i.e. to add labels and bodies directly to the parent, and/or to collapse the label and body into a single element).

Example of creating structure automatically with **`list()`**:

```
// Create a list, add it to the structure tree, then add items to it
const list = doc.struct('List');
someElement.add(list);
doc.list(["Foo. ", "Bar. "], { structParent: list });
```

Tags and Structure Element Types

Here are the tags and structure element types which are defined in Tagged PDF. You must ensure you give them with the correct capitalisation.

Tagged PDF also supports custom types which map to standard types, but PDFKit does not have support for this.

Non-structure tags:

`Artifact` - used to mark all content not part of the logical structure

`ReversedChars` - every string of text has characters in reverse order for technical reasons (due to how fonts work for right-to-left languages); strings may have spaces at the beginning or end to separate words, but may not have spaces in the middle

"Grouping" elements:

`Document` - whole document; must be used if there are multiple parts or articles

`Part` - part of a document

`Art` - article

`Sect` - section; may nest

`Div` - generic division

`BlockQuote` - block quotation

`Caption` - describing a figure or table

`TOC` - table of contents, may be nested, and may be used for lists of figures, tables, etc.

`TOCI` - table of contents (leaf) item

`Index` - index (text with accompanying **`Reference`** content)

`NonStruct` - non-structural grouping element (element itself not intended to be exported to other formats like HTML, but 'transparent' to its content which is processed normally)

`Private` - content only meaningful to the creator (element and its content not intended to be exported to other formats like HTML)

"Block" elements:

`H` - heading (first element in a section, etc.)

`H1` to **`H6`** - heading of a particular level intended for use only if nesting sections is not possible for some reason

`P` - paragraph

`L` - list; should include optional **`Caption`**, and list items

`LI` - list item; should contain **`Lbl`** and/or **`LBody`**

`Lbl` - label (bullet, number, or "dictionary headword")

`LBody` - list body (item text, or "dictionary definition"); may have nested lists or other blocks

"Table" elements:

Table – table; should either contain **TR**, or **THead**, **TBody** and/or **TFoot**

TR – table row

TH – table heading cell

TD – table data cell

THead – table header row group

TBody – table body row group; may have more than one per table

TFoot – table footer row group

"Inline" elements:

Span – generic inline content

Quote – inline quotation

Note – e.g. footnote; may have a **Lbl** (see "block" elements)

Reference – content in a document that refers to other content (e.g. page number in an index)

BibEntry – bibliography entry; may have a **Lbl** (see "block" elements)

Code – code

Link – hyperlink; should contain a link annotation

Annot – annotation (other than a link)

Ruby – Chinese/Japanese pronunciation/explanation

RB – Ruby base text

RT – Ruby annotation text

RP – Ruby punctuation

Warichu – Japanese/Chinese longer description

WT – Warichu text

WP – Warichu punctuation

"Illustration" elements (should have **alt** and/or **actualtext** set):

Figure – figure

Formula – formula

Form – form widget

You made it!

That's all there is to creating PDF documents in PDFKit. It's really quite simple to create beautiful multi-page printable documents using Node.js!

This guide was generated from Markdown files using a PDFKit generation script. The examples are actually run to generate the output shown inline. The script generates both the website and the PDF guide, and can be found on Github. Check it out if you want to see an example of a slightly more complicated renderer using a parser for Markdown and a syntax highlighter.

If you have any questions about what you've learned in this guide, please don't hesitate to ask the author or post an issue on Github. Enjoy!