PDFKit Guide

By Devon Govett

Version 0.9.0

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 CoffeeScript or 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

As of version 0.6, 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 use Browserify, which is a Node module packager for the browser with the familiar **`require`** syntax. The second is to use a prebuilt version of PDFKit, which you can download from Github.

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 **`devDependency`** 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 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.

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.

Vector Graphics in PDFKit

An introduction to vector graphics

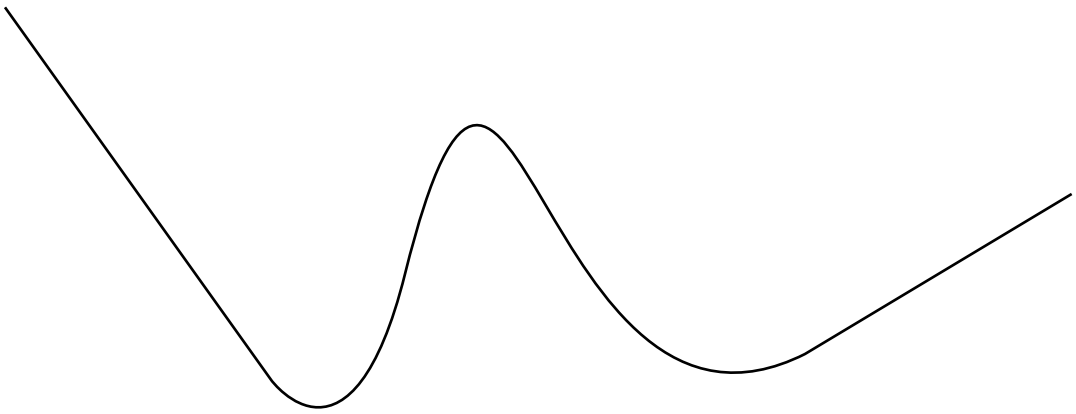
Unlike images which are defined by pixels, vector graphics are defined through a series of drawing commands. This makes vector graphics scalable to any size without a reduction in quality (pixelization). The PDF format was designed with vector graphics in mind, so creating vector drawings is very easy. 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:

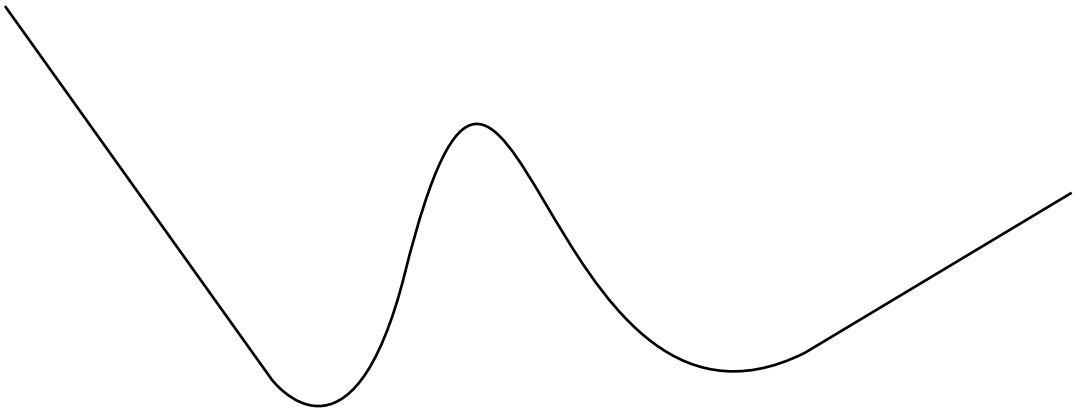


One thing to notice about this example is the use of method chaining. All methods in PDFKit are chainable, meaning that you can call one method right after the other without referencing the **`doc`** variable again. Of course, this is an option, so if you don't like how the code looks when chained, you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

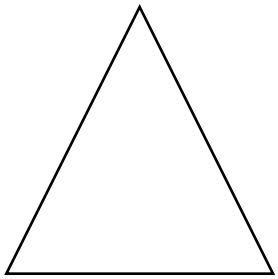
`circle(centerX, centerY, radius)`

`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

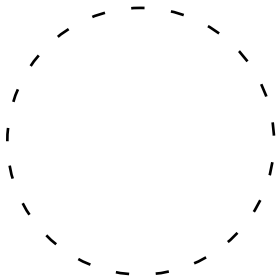
The **dash** method allows you to create non-continuous dashed lines. It takes a length specifying how long each dash should be, as well as an optional hash describing the additional properties **space** and **phase**.

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

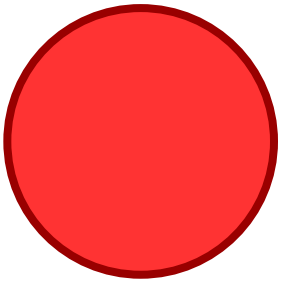
What is a drawing without color? PDFKit makes it simple to set the fill and stroke color and opacity. You can pass an array specifying an RGB or CMYK color, a hex color string, or use any of the named CSS colors.

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);
```

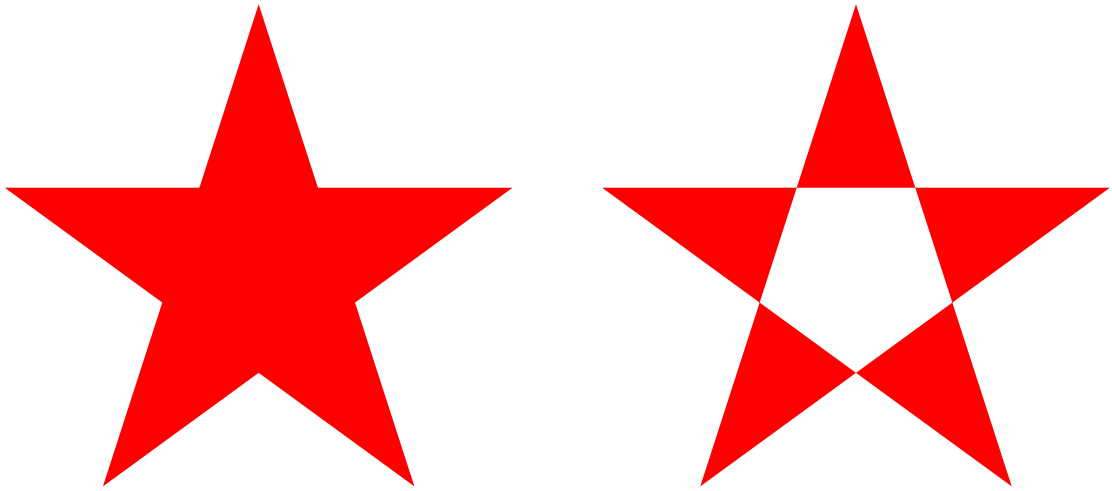
Here is the output from the this example:

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.

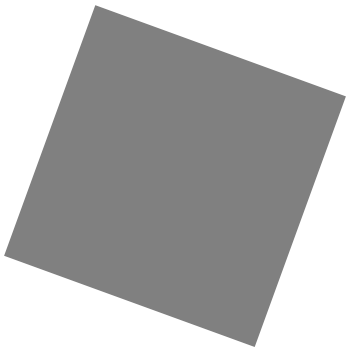
The **`scale`** transformation takes a scale factor and an optional **`origin`** passed in an options hash as with the **`rotate`** transformation. It is used to increase or decrease the size of the units in the drawing, or change its size. For example, applying a scale of **`0.5`** would make the drawing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

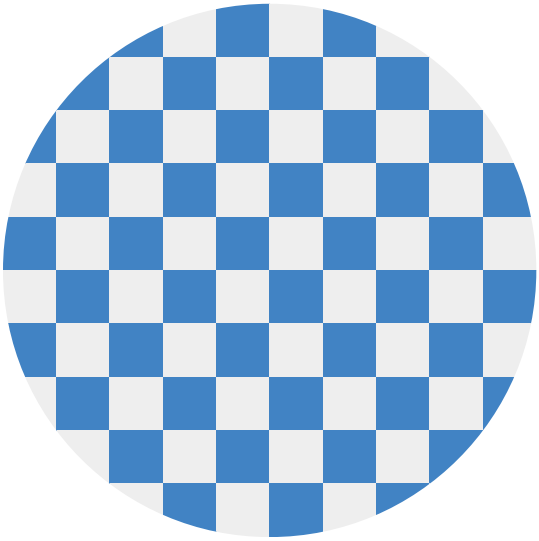


Clipping

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

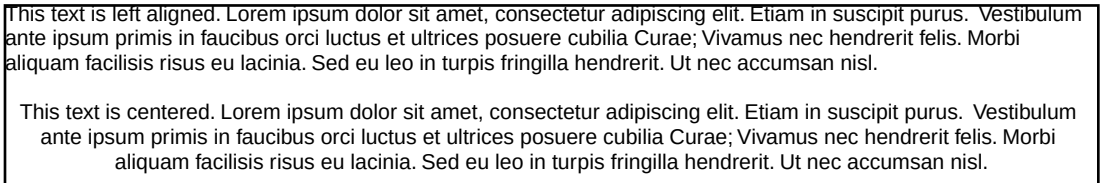
PDFKit includes support for line wrapping out of the box! If no options are given, text is automatically wrapped within the page margins and placed in the document flow below any previous text, or at the top of the page. PDFKit automatically inserts new pages as necessary so you don't have to worry about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

`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;

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

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);
```

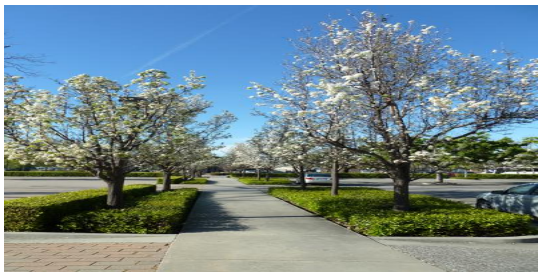
This example produces the following output:

Proportional to width

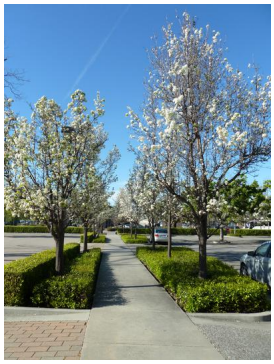
Fit

Centered

Stretch



Scale



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

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)
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)
```

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.

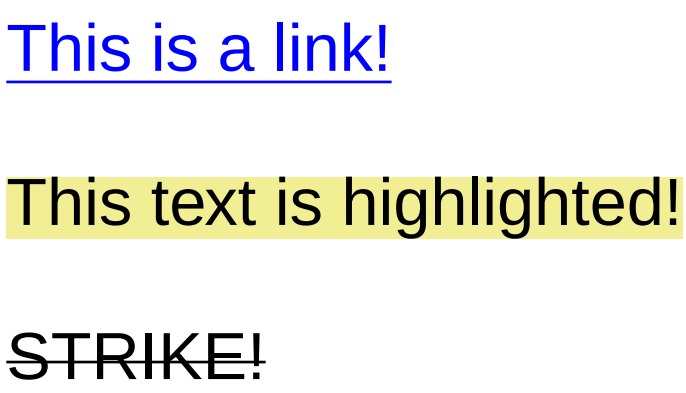
If you are adding an annotation to a piece of text, such as a link or underline, you will need to know the width and height of the text in order to create the required rectangle for the annotation. There are two methods that you can use to do that. To get the width of any piece of text in the current 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!');
```

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:

Another link!

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/Literate CoffeeScript 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!