



Sign Up

Sign In

Search packages

Search

pdf-lib TS

1.17.1 • Public • Published 2 years ago

 [Readme](#)

 [Code](#) Beta

 [4 Dependencies](#)

 [373 Dependents](#)

 [84 Versions](#)



Create and modify PDF documents in any JavaScript environment.

Designed to work in any modern JavaScript runtime. Tested in Node, Browser, Deno, and React Native environments.

Table of Contents

- [Features](#)
- [Motivation](#)
- [Usage Examples](#)
 - [Create Document](#)
 - [Modify Document](#)
 - [Create Form](#)
 - [Fill Form](#)
 - [Flatten Form](#)
 - [Copy Pages](#)
 - [Embed PNG and JPEG Images](#)
 - [Embed PDF Pages](#)
 - [Embed Font and Measure Text](#)
 - [Add Attachments](#)
 - [Set Document Metadata](#)
 - [Read Document Metadata](#)
 - [Set Viewer Preferences](#)
 - [Read Viewer Preferences](#)
 - [Draw SVG Paths](#)
- [Deno Usage](#)
- [Complete Examples](#)
- [Installation](#)
- [Documentation](#)
- [Fonts and Unicode](#)
- [Creating and Filling Forms](#)
- [Limitations](#)
- [Help and Discussion](#)
- [Encryption Handling](#)
- [Migrating to v1.0.0](#)
- [Contributing](#)
- [Maintainership](#)

- [Tutorials and Cool Stuff](#)
- [Prior Art](#)
- [Git History Rewrite](#)
- [License](#)

Features

- Create new PDFs
- Modify existing PDFs
- Create forms
- Fill forms
- Flatten forms
- Add Pages
- Insert Pages
- Remove Pages
- Copy pages between PDFs
- Draw Text
- Draw Images
- Draw PDF Pages
- Draw Vector Graphics
- Draw SVG Paths
- Measure width and height of text
- Embed Fonts (supports UTF-8 and UTF-16 character sets)
- Set document metadata
- Read document metadata
- Set viewer preferences
- Read viewer preferences
- Add attachments

Motivation

`pdf-lib` was created to address the JavaScript ecosystem's lack of robust support for PDF manipulation (especially for PDF *modification*).

Two of `pdf-lib` 's distinguishing features are:

1. Supporting modification (editing) of existing documents.

2. Working in all JavaScript environments - not just in Node or the Browser.

There are **other** good open source JavaScript PDF libraries available. However, most of them can only *create* documents, they cannot *modify* existing ones. And many of them only work in particular environments.

Usage Examples

Create Document

*This example produces **this PDF**.*

Try the JSFiddle demo

```
import { PDFDocument, StandardFonts, rgb } from 'pdf-lib'

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Embed the Times Roman font
const timesRomanFont = await pdfDoc.embedFont(StandardFonts.TimesRoman)

// Add a blank page to the document
const page = pdfDoc.addPage()

// Get the width and height of the page
const { width, height } = page.getSize()

// Draw a string of text toward the top of the page
const fontSize = 30
page.drawText('Creating PDFs in JavaScript is awesome!', {
  x: 50,
  y: height - 4 * fontSize,
  size: fontSize,
  font: timesRomanFont,
  color: rgb(0, 0.53, 0.71),
})
```

```
// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Modify Document

*This example produces **this PDF*** (when **this PDF** is used for the `existingPdfBytes` variable).

Try the JSFiddle demo

```
import { degrees, PDFDocument, rgb, StandardFonts } from 'pdf-lib';

// This should be a Uint8Array or ArrayBuffer
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const existingPdfBytes = ...

// Load a PDFDocument from the existing PDF bytes
const pdfDoc = await PDFDocument.load(existingPdfBytes)

// Embed the Helvetica font
const helveticaFont = await pdfDoc.embedFont(StandardFonts.Helvetica)

// Get the first page of the document
const pages = pdfDoc.getPages()
const firstPage = pages[0]

// Get the width and height of the first page
const { width, height } = firstPage.getSize()

// Draw a string of text diagonally across the first page
firstPage.drawText('This text was added with JavaScript!', {
  x: 5,
```

```

    y: height / 2 + 300,
    size: 50,
    font: helveticaFont,
    color: rgb(0.95, 0.1, 0.1),
    rotate: degrees(-45),
  })

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>

```

Create Form

This example produces *this PDF*.

Try the JSFiddle demo

See also **Creating and Filling Forms**

```

import { PDFDocument } from 'pdf-lib'

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Add a blank page to the document
const page = pdfDoc.addPage([550, 750])

// Get the form so we can add fields to it
const form = pdfDoc.getForm()

// Add the superhero text field and description
page.drawText('Enter your favorite superhero:', { x: 50, y: 700, size:

```

```
const superheroField = form.createTextField('favorite.superhero')
superheroField.setText('One Punch Man')
superheroField.addToPage(page, { x: 55, y: 640 })

// Add the rocket radio group, labels, and description
page.drawText('Select your favorite rocket:', { x: 50, y: 600, size: 20 })

page.drawText('Falcon Heavy', { x: 120, y: 560, size: 18 })
page.drawText('Saturn IV', { x: 120, y: 500, size: 18 })
page.drawText('Delta IV Heavy', { x: 340, y: 560, size: 18 })
page.drawText('Space Launch System', { x: 340, y: 500, size: 18 })

const rocketField = form.createRadioGroup('favorite.rocket')
rocketField.addOptionToPage('Falcon Heavy', page, { x: 55, y: 540 })
rocketField.addOptionToPage('Saturn IV', page, { x: 55, y: 480 })
rocketField.addOptionToPage('Delta IV Heavy', page, { x: 275, y: 540 })
rocketField.addOptionToPage('Space Launch System', page, { x: 275, y: 480 })
rocketField.select('Saturn IV')

// Add the gundam check boxes, labels, and description
page.drawText('Select your favorite gundams:', { x: 50, y: 440, size: 20 })

page.drawText('Exia', { x: 120, y: 400, size: 18 })
page.drawText('Kyrios', { x: 120, y: 340, size: 18 })
page.drawText('Virtue', { x: 340, y: 400, size: 18 })
page.drawText('Dynes', { x: 340, y: 340, size: 18 })

const exiaField = form.createCheckBox('gundam.exia')
const kyriosField = form.createCheckBox('gundam.kyrios')
const virtueField = form.createCheckBox('gundam.virtue')
const dynesField = form.createCheckBox('gundam.dynes')

exiaField.addToPage(page, { x: 55, y: 380 })
kyriosField.addToPage(page, { x: 55, y: 320 })
virtueField.addToPage(page, { x: 275, y: 380 })
```

```
dynamesField.addToPage(page, { x: 275, y: 320 })

exiaField.check()
dynamesField.check()

// Add the planet dropdown and description
page.drawText('Select your favorite planet*:', { x: 50, y: 280, size: 2

const planetsField = form.createDropdown('favorite.planet')
planetsField.addOptions(['Venus', 'Earth', 'Mars', 'Pluto'])
planetsField.select('Pluto')
planetsField.addToPage(page, { x: 55, y: 220 })

// Add the person option list and description
page.drawText('Select your favorite person:', { x: 50, y: 180, size: 18

const personField = form.createOptionList('favorite.person')
personField.addOptions([
  'Julius Caesar',
  'Ada Lovelace',
  'Cleopatra',
  'Aaron Burr',
  'Mark Antony',
])
personField.select('Ada Lovelace')
personField.addToPage(page, { x: 55, y: 70 })

// Just saying...
page.drawText(`* Pluto should be a planet too!`, { x: 15, y: 15, size:

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
```



```
// • Downloaded from the browser
// • Rendered in an <iframe>
```

Fill Form

*This example produces **this PDF*** (when **this PDF** is used for the `formPdfBytes` variable, **this image** is used for the `marioImageBytes` variable, and **this image** is used for the `emblemImageBytes` variable).

Try the JSFiddle demo

See also **Creating and Filling Forms**

```
import { PDFDocument } from 'pdf-lib'

// These should be Uint8Arrays or ArrayBuffers
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const formPdfBytes = ...
const marioImageBytes = ...
const emblemImageBytes = ...

// Load a PDF with form fields
const pdfDoc = await PDFDocument.load(formPdfBytes)

// Embed the Mario and emblem images
const marioImage = await pdfDoc.embedPng(marioImageBytes)
const emblemImage = await pdfDoc.embedPng(emblemImageBytes)

// Get the form containing all the fields
const form = pdfDoc.getForm()

// Get all fields in the PDF by their names
const nameField = form.getTextField('CharacterName 2')
const ageField = form.getTextField('Age')
const heightField = form.getTextField('Height')
```

```
const weightField = form.getTextField('Weight')
const eyesField = form.getTextField('Eyes')
const skinField = form.getTextField('Skin')
const hairField = form.getTextField('Hair')

const alliesField = form.getTextField('Allies')
const factionField = form.getTextField('FactionName')
const backstoryField = form.getTextField('Backstory')
const traitsField = form.getTextField('Feat+Traits')
const treasureField = form.getTextField('Treasure')

const characterImageField = form.getButton('CHARACTER IMAGE')
const factionImageField = form.getTextField('Faction Symbol Image')

// Fill in the basic info fields
nameField.setText('Mario')
ageField.setText('24 years')
heightField.setText(`5' 1"`)
weightField.setText('196 lbs')
eyesField.setText('blue')
skinField.setText('white')
hairField.setText('brown')

// Fill the character image field with our Mario image
characterImageField.setImage(marioImage)

// Fill in the allies field
alliesField.setText(
  [
    `Allies:`,
    ` • Princess Daisy`,
    ` • Princess Peach`,
    ` • Rosalina`,
    ` • Geno`,
    ` • Luigi`,
    ` • Donkey Kong`,
```

```

    `
      • Yoshi`,
    `
      • Diddy Kong`,
    ``
  ],
  `Organizations:`,
  `
    • Italian Plumbers Association`,
  ].join('\n'),
)

// Fill in the faction name field
factionField.setText(`Mario's Emblem`)

// Fill the faction image field with our emblem image
factionImageField.setImage(emblemImage)

// Fill in the backstory field
backstoryField.setText(
  `Mario is a fictional character in the Mario video game franchise, ow
)

// Fill in the traits field
traitsField.setText(
  [
    `Mario can use three basic three power-ups:`,
    `
      • the Super Mushroom, which causes Mario to grow larger`,
    `
      • the Fire Flower, which allows Mario to throw fireballs`,
    `
      • the Starman, which gives Mario temporary invincibility`,
  ].join('\n'),
)

// Fill in the treasure field
treasureField.setText(['• Gold coins', '• Treasure chests'].join('\n'))

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:

```

- // • Written to a file in Node
- // • Downloaded from the browser
- // • Rendered in an <iframe>

Flatten Form

This example produces *this PDF* (when **this PDF** is used for the `formPdfBytes` variable).

Try the JSFiddle demo

```
import { PDFDocument } from 'pdf-lib'

// This should be a Uint8Array or ArrayBuffer
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const formPdfBytes = ...

// Load a PDF with form fields
const pdfDoc = await PDFDocument.load(formPdfBytes)

// Get the form containing all the fields
const form = pdfDoc.getForm()

// Fill the form's fields
form.getTextField('Text1').setText('Some Text');

form.getRadioGroup('Group2').select('Choice1');
form.getRadioGroup('Group3').select('Choice3');
form.getRadioGroup('Group4').select('Choice1');

form.getCheckBox('Check Box3').check();
form.getCheckBox('Check Box4').uncheck();

form.getDropdown('Dropdown7').select('Infinity');

form.getOptionList('List Box6').select('Honda');
```

```
// Flatten the form's fields
form.flatten();

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Copy Pages

*This example produces **this PDF*** (when **this PDF** is used for the `firstDonorPdfBytes` variable and **this PDF** is used for the `secondDonorPdfBytes` variable).

Try the JSFiddle demo

```
import { PDFDocument } from 'pdf-lib'

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// These should be Uint8Arrays or ArrayBuffers
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const firstDonorPdfBytes = ...
const secondDonorPdfBytes = ...

// Load a PDFDocument from each of the existing PDFs
const firstDonorPdfDoc = await PDFDocument.load(firstDonorPdfBytes)
const secondDonorPdfDoc = await PDFDocument.load(secondDonorPdfBytes)

// Copy the 1st page from the first donor document, and
// the 743rd page from the second donor document
const [firstDonorPage] = await pdfDoc.copyPages(firstDonorPdfDoc, [0])
const [secondDonorPage] = await pdfDoc.copyPages(secondDonorPdfDoc, [74
```

```
// Add the first copied page
pdfDoc.addPage(firstDonorPage)

// Insert the second copied page to index 0, so it will be the
// first page in `pdfDoc`
pdfDoc.insertPage(0, secondDonorPage)

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Embed PNG and JPEG Images

*This example produces **this PDF** (when **this image** is used for the `jpgImageBytes` variable and **this image** is used for the `pngImageBytes` variable).*

Try the JSFiddle demo

```
import { PDFDocument } from 'pdf-lib'

// These should be Uint8Arrays or ArrayBuffers
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const jpgImageBytes = ...
const pngImageBytes = ...

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Embed the JPG image bytes and PNG image bytes
const jpgImage = await pdfDoc.embedJpg(jpgImageBytes)
const pngImage = await pdfDoc.embedPng(pngImageBytes)
```

```
// Get the width/height of the JPG image scaled down to 25% of its orig
const jpgDims = jpgImage.scale(0.25)

// Get the width/height of the PNG image scaled down to 50% of its orig
const pngDims = pngImage.scale(0.5)

// Add a blank page to the document
const page = pdfDoc.addPage()

// Draw the JPG image in the center of the page
page.drawImage(jpgImage, {
  x: page.getWidth() / 2 - jpgDims.width / 2,
  y: page.getHeight() / 2 - jpgDims.height / 2,
  width: jpgDims.width,
  height: jpgDims.height,
})

// Draw the PNG image near the lower right corner of the JPG image
page.drawImage(pngImage, {
  x: page.getWidth() / 2 - pngDims.width / 2 + 75,
  y: page.getHeight() / 2 - pngDims.height,
  width: pngDims.width,
  height: pngDims.height,
})

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Embed PDF Pages

This example produces *this PDF* (when *this PDF* is used for the `americanFlagPdfBytes` variable and *this PDF* is used for the `usConstitutionPdfBytes` variable).

Try the JSFiddle demo

```
import { PDFDocument } from 'pdf-lib'

// These should be Uint8Arrays or ArrayBuffers
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const americanFlagPdfBytes = ...
const usConstitutionPdfBytes = ...

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Embed the American flag PDF bytes
const [americanFlag] = await pdfDoc.embedPdf(americanFlagPdfBytes)

// Load the U.S. constitution PDF bytes
const usConstitutionPdf = await PDFDocument.load(usConstitutionPdfBytes)

// Embed the second page of the constitution and clip the preamble
const preamble = await pdfDoc.embedPage(usConstitutionPdf.getPages()[1]
  left: 55,
  bottom: 485,
  right: 300,
  top: 575,
}))

// Get the width/height of the American flag PDF scaled down to 30% of
// its original size
const americanFlagDims = americanFlag.scale(0.3)

// Get the width/height of the preamble clipping scaled up to 225% of
// its original size
```



```

const preambleDims = preamble.scale(2.25)

// Add a blank page to the document
const page = pdfDoc.addPage()

// Draw the American flag image in the center top of the page
page.drawPage(americanFlag, {
  ...americanFlagDims,
  x: page.getWidth() / 2 - americanFlagDims.width / 2,
  y: page.getHeight() - americanFlagDims.height - 150,
})

// Draw the preamble clipping in the center bottom of the page
page.drawPage(preamble, {
  ...preambleDims,
  x: page.getWidth() / 2 - preambleDims.width / 2,
  y: page.getHeight() / 2 - preambleDims.height / 2 - 50,
})

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>

```

Embed Font and Measure Text

pdf-lib relies on a sister module to support embedding custom fonts: [@pdf-lib/fontkit](#). You must add the `@pdf-lib/fontkit` module to your project and register it using `pdfDoc.registerFontkit(...)` before embedding custom fonts.

See below for detailed installation instructions on installing `@pdf-lib/fontkit` as a UMD or NPM module.

*This example produces **this PDF** (when **this font** is used for the `fontBytes` variable).*

Try the JSFiddle demo

```
import { PDFDocument, rgb } from 'pdf-lib'
import fontkit from '@pdf-lib/fontkit'

// This should be a Uint8Array or ArrayBuffer
// This data can be obtained in a number of different ways
// If you're running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const fontBytes = ...

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Register the `fontkit` instance
pdfDoc.registerFontkit(fontkit)

// Embed our custom font in the document
const customFont = await pdfDoc.embedFont(fontBytes)

// Add a blank page to the document
const page = pdfDoc.addPage()

// Create a string of text and measure its width and height in our custom font
const text = 'This is text in an embedded font!'
const textSize = 35
const textWidth = customFont.widthOfTextAtSize(text, textSize)
const textHeight = customFont.heightAtSize(textSize)

// Draw the string of text on the page
page.drawText(text, {
  x: 40,
  y: 450,
  size: textSize,
  font: customFont,
  color: rgb(0, 0.53, 0.71),
```

```
    })

    // Draw a box around the string of text
    page.drawRectangle({
      x: 40,
      y: 450,
      width: textWidth,
      height: textHeight,
      borderColor: rgb(1, 0, 0),
      borderWidth: 1.5,
    })
```

```
// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Add Attachments

*This example produces **this PDF*** (when **this image** is used for the `jpgAttachmentBytes` variable and **this PDF** is used for the `pdfAttachmentBytes` variable).

Try the JSFiddle demo

```
import { PDFDocument } from 'pdf-lib'

// These should be Uint8Arrays or ArrayBuffers
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const jpgAttachmentBytes = ...
const pdfAttachmentBytes = ...

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()
```

```
// Add the JPG attachment
await pdfDoc.attach(jpgAttachmentBytes, 'cat_riding_unicorn.jpg', {
  mimeType: 'image/jpeg',
  description: 'Cool cat riding a unicorn! 🦄🐱👁️',
  creationDate: new Date('2019/12/01'),
  modificationDate: new Date('2020/04/19'),
})

// Add the PDF attachment
await pdfDoc.attach(pdfAttachmentBytes, 'us_constitution.pdf', {
  mimeType: 'application/pdf',
  description: 'Constitution of the United States US 🇺🇸',
  creationDate: new Date('1787/09/17'),
  modificationDate: new Date('1992/05/07'),
})

// Add a page with some text
const page = pdfDoc.addPage();
page.drawText('This PDF has two attachments', { x: 135, y: 415 })

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Set Document Metadata

This example produces *this PDF*.

Try the JSFiddle demo

```
import { PDFDocument, StandardFonts } from 'pdf-lib'

// Create a new PDFDocument
```

```

const pdfDoc = await PDFDocument.create()

// Embed the Times Roman font
const timesRomanFont = await pdfDoc.embedFont(StandardFonts.TimesRoman)

// Add a page and draw some text on it
const page = pdfDoc.addPage([500, 600])
page.setFont(timesRomanFont)
page.drawText('The Life of an Egg', { x: 60, y: 500, size: 50 })
page.drawText('An Epic Tale of Woe', { x: 125, y: 460, size: 25 })

// Set all available metadata fields on the PDFDocument. Note that these
// are visible in the "Document Properties" section of most PDF readers
pdfDoc.setTitle('🥚 The Life of an Egg 🔍')
pdfDoc.setAuthor('Humpty Dumpty')
pdfDoc.setSubject('📖 An Epic Tale of Woe 📖')
pdfDoc.setKeywords(['eggs', 'wall', 'fall', 'king', 'horses', 'men'])
pdfDoc.setProducer('PDF App 9000 🤖')
pdfDoc.setCreator('pdf-lib (https://github.com/Hopding/pdf-lib)')
pdfDoc.setCreationDate(new Date('2018-06-24T01:58:37.228Z'))
pdfDoc.setModificationDate(new Date('2019-12-21T07:00:11.000Z'))

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>

```

Read Document Metadata

Try the JSFiddle demo

```

import { PDFDocument } from 'pdf-lib'

// This should be a Uint8Array or ArrayBuffer

```

```
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer
const existingPdfBytes = ...

// Load a PDFDocument without updating its existing metadata
const pdfDoc = await PDFDocument.load(existingPdfBytes, {
  updateMetadata: false
})

// Print all available metadata fields
console.log('Title:', pdfDoc.getTitle())
console.log('Author:', pdfDoc.getAuthor())
console.log('Subject:', pdfDoc.getSubject())
console.log('Creator:', pdfDoc.getCreator())
console.log('Keywords:', pdfDoc.getKeywords())
console.log('Producer:', pdfDoc.getProducer())
console.log('Creation Date:', pdfDoc.getCreationDate())
console.log('Modification Date:', pdfDoc.getModificationDate())
```

This script outputs the following (when *this PDF* is used for the *existingPdfBytes* variable):

```
Title: Microsoft Word - Basic Curriculum Vitae example.doc
Author: Administrator
Subject: undefined
Creator: PScript5.dll Version 5.2
Keywords: undefined
Producer: Acrobat Distiller 8.1.0 (Windows)
Creation Date: 2010-07-29T14:26:00.000Z
Modification Date: 2010-07-29T14:26:00.000Z
```

Set Viewer Preferences

```
import {
  PDFDocument,
  StandardFonts,
  NonFullScreenPageMode,
```

```
    ReadingDirection,
    PrintScaling,
    Duplex,
    PDFName,
} from 'pdf-lib'

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Embed the Times Roman font
const timesRomanFont = await pdfDoc.embedFont(StandardFonts.TimesRoman)

// Add a page and draw some text on it
const page = pdfDoc.addPage([500, 600])
page.setFont(timesRomanFont)
page.drawText('The Life of an Egg', { x: 60, y: 500, size: 50 })
page.drawText('An Epic Tale of Woe', { x: 125, y: 460, size: 25 })

// Set all available viewer preferences on the PDFDocument:
const viewerPrefs = pdfDoc.catalog.getOrCreateViewerPreferences()
viewerPrefs.setHideToolbar(true)
viewerPrefs.setHideMenubar(true)
viewerPrefs.setHideWindowUI(true)
viewerPrefs.setFitWindow(true)
viewerPrefs.setCenterWindow(true)
viewerPrefs.setDisplayDocTitle(true)

// Set the PageMode (otherwise setting NonFullScreenPageMode has no mea
pdfDoc.catalog.set(PDFName.of('PageMode'), PDFName.of('FullScreen'))

// Set what happens when fullScreen is closed
viewerPrefs.setNonFullScreenPageMode(NonFullScreenPageMode.UseOutlines)

viewerPrefs.setReadingDirection(ReadingDirection.L2R)
viewerPrefs.setPrintScaling(PrintScaling.None)
viewerPrefs.setDuplex(Duplex.DuplexFlipLongEdge)
```

```
viewerPrefs.setPickTrayByPDFSize(true)

// We can set the default print range to only the first page
viewerPrefs.setPrintPageRange({ start: 0, end: 0 })

// Or we can supply noncontiguous ranges (e.g. pages 1, 3, and 5-7)
viewerPrefs.setPrintPageRange([
  { start: 0, end: 0 },
  { start: 2, end: 2 },
  { start: 4, end: 6 },
])

viewerPrefs.setNumCopies(2)

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
//   • Downloaded from the browser
//   • Rendered in an <iframe>
```

Read Viewer Preferences

```
import { PDFDocument } from 'pdf-lib'

// This should be a Uint8Array or ArrayBuffer
// This data can be obtained in a number of different ways
// If your running in a Node environment, you could use fs.readFile()
// In the browser, you could make a fetch() call and use res.arrayBuffer()
const existingPdfBytes = ...

// Load a PDFDocument without updating its existing metadata
const pdfDoc = await PDFDocument.load(existingPdfBytes)
const viewerPrefs = pdfDoc.catalog.getOrCreateViewerPreferences()

// Print all available viewer preference fields
```



```
console.log('HideToolbar:', viewerPrefs.getHideToolbar())
console.log('HideMenubar:', viewerPrefs.getHideMenubar())
console.log('HideWindowUI:', viewerPrefs.getHideWindowUI())
console.log('FitWindow:', viewerPrefs.getFitWindow())
console.log('CenterWindow:', viewerPrefs.getCenterWindow())
console.log('DisplayDocTitle:', viewerPrefs.getDisplayDocTitle())
console.log('NonFullScreenPageMode:', viewerPrefs.getNonFullScreenPageM
console.log('ReadingDirection:', viewerPrefs.getReadingDirection())
console.log('PrintScaling:', viewerPrefs.getPrintScaling())
console.log('Duplex:', viewerPrefs.getDuplex())
console.log('PickTrayByPDFSize:', viewerPrefs.getPickTrayByPDFSize())
console.log('PrintPageRange:', viewerPrefs.getPrintPageRange())
console.log('NumCopies:', viewerPrefs.getNumCopies())
```

This script outputs the following (when *this PDF* is used for the *existingPdfBytes* variable):

```
HideToolbar: true
HideMenubar: true
HideWindowUI: false
FitWindow: true
CenterWindow: true
DisplayDocTitle: true
NonFullScreenPageMode: UseNone
ReadingDirection: R2L
PrintScaling: None
Duplex: DuplexFlipLongEdge
PickTrayByPDFSize: true
PrintPageRange: [ { start: 1, end: 1 }, { start: 3, end: 4 } ]
NumCopies: 2
```

Draw SVG Paths

This example produces *this PDF*.

Try the JSFiddle demo

```
import { PDFDocument, rgb } from 'pdf-lib'

// SVG path for a wavy line
const svgPath =
  'M 0,20 L 100,160 Q 130,200 150,120 C 190,-40 200,200 300,150 L 400,9

// Create a new PDFDocument
const pdfDoc = await PDFDocument.create()

// Add a blank page to the document
const page = pdfDoc.addPage()
page.moveTo(100, page.getHeight() - 5)

// Draw the SVG path as a black line
page.moveDown(25)
page.drawSvgPath(svgPath)

// Draw the SVG path as a thick green line
page.moveDown(200)
page.drawSvgPath(svgPath, { borderColor: rgb(0, 1, 0), borderWidth: 5 })

// Draw the SVG path and fill it with red
page.moveDown(200)
page.drawSvgPath(svgPath, { color: rgb(1, 0, 0) })

// Draw the SVG path at 50% of its original size
page.moveDown(200)
page.drawSvgPath(svgPath, { scale: 0.5 })

// Serialize the PDFDocument to bytes (a Uint8Array)
const pdfBytes = await pdfDoc.save()

// For example, `pdfBytes` can be:
//   • Written to a file in Node
```

```
// • Downloaded from the browser
// • Rendered in an <iframe>
```

Deno Usage

pdf-lib fully supports the exciting new **Deno** runtime! All of the **usage examples** work in Deno. The only thing you need to do is change the imports for pdf-lib and @pdf-lib/fontkit to use the **Skypack** CDN, because Deno requires all modules to be referenced via URLs.

See also **How to Create and Modify PDF Files in Deno With pdf-lib**

Creating a Document with Deno

Below is the **create document** example modified for Deno:

```
import {
  PDFDocument,
  StandardFonts,
  rgb,
} from 'https://cdn.skypack.dev/pdf-lib@^1.11.1?dts';

const pdfDoc = await PDFDocument.create();
const timesRomanFont = await pdfDoc.embedFont(StandardFonts.TimesRoman)

const page = pdfDoc.addPage();
const { width, height } = page.getSize();
const fontSize = 30;
page.drawText('Creating PDFs in JavaScript is awesome!', {
  x: 50,
  y: height - 4 * fontSize,
  size: fontSize,
  font: timesRomanFont,
  color: rgb(0, 0.53, 0.71),
});

const pdfBytes = await pdfDoc.save();
```

```
await Deno.writeFile('out.pdf', pdfBytes);
```

If you save this script as `create-document.ts`, you can execute it using Deno with the following command:

```
deno run --allow-write create-document.ts
```

The resulting `out.pdf` file will look like [this PDF](#).

Embedding a Font with Deno

Here's a slightly more complicated example demonstrating how to embed a font and measure text in Deno:

```
import {
  degrees,
  PDFDocument,
  rgb,
  StandardFonts,
} from 'https://cdn.skypack.dev/pdf-lib@^1.11.1?dts';
import fontkit from 'https://cdn.skypack.dev/@pdf-lib/fontkit@^1.0.0?dts';

const url = 'https://pdf-lib.js.org/assets/ubuntu/Ubuntu-R.ttf';
const fontBytes = await fetch(url).then((res) => res.arrayBuffer());

const pdfDoc = await PDFDocument.create();

pdfDoc.registerFontkit(fontkit);
const customFont = await pdfDoc.embedFont(fontBytes);

const page = pdfDoc.addPage();

const text = 'This is text in an embedded font!';
const textSize = 35;
const textWidth = customFont.widthOfTextAtSize(text, textSize);
const textHeight = customFont.heightAtSize(textSize);
```

```

page.drawText(text, {
  x: 40,
  y: 450,
  size: textSize,
  font: customFont,
  color: rgb(0, 0.53, 0.71),
});

page.drawRectangle({
  x: 40,
  y: 450,
  width: textWidth,
  height: textHeight,
  borderColor: rgb(1, 0, 0),
  borderWidth: 1.5,
});

const pdfBytes = await pdfDoc.save();

await Deno.writeFile('out.pdf', pdfBytes);

```

If you save this script as `custom-font.ts`, you can execute it with the following command:

```
deno run --allow-write --allow-net custom-font.ts
```

The resulting `out.pdf` file will look like [this PDF](#).

Complete Examples

The **usage examples** provide code that is brief and to the point, demonstrating the different features of `pdf-lib`. You can find complete working examples in the **apps/** directory. These apps are used to do manual testing of `pdf-lib` before every release (in addition to the **automated tests**).

There are currently four apps:

- **node** - contains **tests** for `pdf-lib` in Node environments. These tests are a handy reference when trying to save/load PDFs, fonts, or images with `pdf-lib` from the filesystem. They also allow you to

quickly open your PDFs in different viewers (Acrobat, Preview, Foxit, Chrome, Firefox, etc...) to ensure compatibility.

- **web** - contains **tests** for `pdf-lib` in browser environments. These tests are a handy reference when trying to save/load PDFs, fonts, or images with `pdf-lib` in a browser environment.
- **rn** - contains **tests** for `pdf-lib` in React Native environments. These tests are a handy reference when trying to save/load PDFs, fonts, or images with `pdf-lib` in a React Native environment.
- **deno** - contains **tests** for `pdf-lib` in Deno environments. These tests are a handy reference when trying to save/load PDFs, fonts, or images with `pdf-lib` from the filesystem.

Installation

NPM Module

To install the latest stable version:

```
# With npm
npm install --save pdf-lib
```

```
# With yarn
yarn add pdf-lib
```

This assumes you're using **npm** or **yarn** as your package manager.

UMD Module

You can also download `pdf-lib` as a UMD module from **unpkg** or **jsDelivr**. The UMD builds have been compiled to ES5, so they should work **in any modern browser**. UMD builds are useful if you aren't using a package manager or module bundler. For example, you can use them directly in the `<script>` tag of an HTML page.

The following builds are available:

- <https://unpkg.com/pdf-lib/dist/pdf-lib.js>
- <https://unpkg.com/pdf-lib/dist/pdf-lib.min.js>
- <https://cdn.jsdelivr.net/npm/pdf-lib/dist/pdf-lib.js>
- <https://cdn.jsdelivr.net/npm/pdf-lib/dist/pdf-lib.min.js>

NOTE: if you are using the CDN scripts in production, you should include a specific version number in the URL, for example:

- <https://unpkg.com/pdf-lib@1.4.0/dist/pdf-lib.min.js>
- <https://cdn.jsdelivr.net/npm/pdf-lib@1.4.0/dist/pdf-lib.min.js>

When using a UMD build, you will have access to a global `window.PDFLib` variable. This variable contains all of the classes and functions exported by `pdf-lib`. For example:

```
// NPM module
import { PDFDocument, rgb } from 'pdf-lib';

// UMD module
var PDFDocument = PDFLib.PDFDocument;
var rgb = PDFLib.rgb;
```

Fontkit Installation

`pdf-lib` relies upon a sister module to support embedding custom fonts: **@pdf-lib/fontkit**. You must add the `@pdf-lib/fontkit` module to your project and register it using `pdfDoc.registerFontkit(...)` before embedding custom fonts (see the **font embedding example**). This module is not included by default because not all users need it, and it increases bundle size.

Installing this module is easy. Just like `pdf-lib` itself, `@pdf-lib/fontkit` can be installed with `npm / yarn` or as a UMD module.

Fontkit NPM Module

```
# With npm
npm install --save @pdf-lib/fontkit

# With yarn
yarn add @pdf-lib/fontkit
```

To register the `fontkit` instance:

```
import { PDFDocument } from 'pdf-lib'
import fontkit from '@pdf-lib/fontkit'
```

```
const pdfDoc = await PDFDocument.create()
pdfDoc.registerFontkit(fontkit)
```

Fontkit UMD Module

The following builds are available:

- <https://unpkg.com/@pdf-lib/fontkit/dist/fontkit.umd.js>
- <https://unpkg.com/@pdf-lib/fontkit/dist/fontkit.umd.min.js>
- <https://cdn.jsdelivr.net/npm/@pdf-lib/fontkit/dist/fontkit.umd.js>
- <https://cdn.jsdelivr.net/npm/@pdf-lib/fontkit/dist/fontkit.umd.min.js>

NOTE: if you are using the CDN scripts in production, you should include a specific version number in the URL, for example:

- <https://unpkg.com/@pdf-lib/fontkit@0.0.4/dist/fontkit.umd.min.js>
- <https://cdn.jsdelivr.net/npm/@pdf-lib/fontkit@0.0.4/dist/fontkit.umd.min.js>

When using a UMD build, you will have access to a global `window.fontkit` variable. To register the fontkit instance:

```
var pdfDoc = await PDFLib.PDFDocument.create()
pdfDoc.registerFontkit(fontkit)
```

Documentation

API documentation is available on the project site at <https://pdf-lib.js.org/docs/api/>.

The repo for the project site (and generated documentation files) is located here:

<https://github.com/Hopding/pdf-lib-docs>.

Fonts and Unicode

When working with PDFs, you will frequently come across the terms "character encoding" and "font". If you have experience in web development, you may wonder why these are so prevalent. Aren't they just annoying details that you shouldn't need to worry about? Shouldn't PDF libraries and readers be able to handle all of this for you like web browsers can? Unfortunately, this is not the case. The nature

of the PDF file format makes it very difficult to avoid thinking about character encodings and fonts when working with PDFs.

`pdf-lib` does its best to simplify things for you. But it can't perform magic. This means you should be aware of the following:

- **There are 14 standard fonts** defined in the PDF specification. They are as follows: *Times Roman* (normal, bold, and italic), *Helvetica* (normal, bold, and italic), *Courier* (normal, bold, and italic), *ZapfDingbats* (normal), and *Symbol* (normal). These 14 fonts are guaranteed to be available in PDF readers. As such, you do not need to embed any font data if you wish to use one of these fonts. You can use a standard font like so:

```
import { PDFDocument, StandardFonts } from 'pdf-lib'
const pdfDoc = await PDFDocument.create()
const courierFont = await pdfDoc.embedFont(StandardFonts.Courier)
const page = pdfDoc.addPage()
page.drawText('Some boring latin text in the Courier font', {
  font: courierFont,
})
```

- **The standard fonts do not support all characters** available in Unicode. The Times Roman, Helvetica, and Courier fonts use WinAnsi encoding (aka **Windows-1252**). The WinAnsi character set only supports 218 characters in the Latin alphabet. For this reason, many users will find the standard fonts insufficient for their use case. This is unfortunate, but there's nothing that PDF libraries can do to change this. This is a result of the PDF specification and its age. Note that the **ZapfDingbats** and **Symbol** fonts use their own specialized encodings that support 203 and 194 characters, respectively. However, the characters they support are not useful for most use cases. See [here](#) for an example of all 14 standard fonts.
- **You can use characters outside the Latin alphabet** by embedding your own fonts. Embedding your own font requires to you load the font data (from a file or via a network request, for example) and pass it to the `embedFont` method. When you embed your own font, you can use any Unicode characters that it supports. This capability frees you from the limitations imposed by the standard fonts. Most PDF files use embedded fonts. You can embed and use a custom font like so ([see also](#)):

```
import { PDFDocument } from 'pdf-lib'
import fontkit from '@pdf-lib/fontkit'

const url = 'https://pdf-lib.js.org/assets/ubuntu/Ubuntu-R.ttf'
const fontBytes = await fetch(url).then((res) => res.arrayBuffer())

const pdfDoc = await PDFDocument.create()

pdfDoc.registerFontkit(fontkit)
const ubuntuFont = await pdfDoc.embedFont(fontBytes)
```

```
const page = pdfDoc.addPage()
page.drawText('Some fancy Unicode text in the ŪBùñłü font', {
  font: ubuntuFont,
})
```

Note that encoding errors will be thrown if you try to use a character with a font that does not support it. For example, `Ω` is not in the WinAnsi character set. So trying to draw it on a page with the standard Helvetica font will throw the following error:

```
Error: WinAnsi cannot encode "Ω" (0x03a9)
    at Encoding.encodeUnicodeCodePoint
```

Font Subsetting

Embedding a font in a PDF document will typically increase the file's size. You can reduce the amount a file's size is increased by subsetting the font so that only the necessary characters are embedded. You can subset a font by setting the **subset option** to `true`. For example:

```
const font = await pdfDoc.embedFont(fontBytes, { subset: true });
```

Note that subsetting does not work for all fonts. See <https://github.com/Hopding/pdf-lib/issues/207#issuecomment-537210471> for additional details.

Creating and Filling Forms

`pdf-lib` can create, fill, and read PDF form fields. The following field types are supported:

- Buttons
- Check Boxes
- Dropdowns
- Option Lists
- Radio Groups
- Text Fields

See the **form creation** and **form filling** usage examples for code samples. Tests 1, 14, 15, 16, and 17 in the **complete examples** contain working example code for form creation and filling in a variety of different JS environments.

IMPORTANT: The default font used to display text in buttons, dropdowns, option lists, and text fields is the standard Helvetica font. This font only supports characters in the latin alphabet (see **Fonts and Unicode** for details). This means that if any of these field types are created or modified to contain text outside the latin alphabet (as is often the case), you will need to embed and use a custom font to update the field appearances. Otherwise an error will be thrown (likely when you save the PDFDocument).

You can use an embedded font when filling form fields as follows:

```
import { PDFDocument } from 'pdf-lib';
import fontkit from '@pdf-lib/fontkit';

// Fetch the PDF with form fields
const formUrl = 'https://pdf-lib.js.org/assets/dod_character.pdf';
const formBytes = await fetch(formUrl).then((res) => res.arrayBuffer())

// Fetch the Ubuntu font
const fontUrl = 'https://pdf-lib.js.org/assets/ubuntu/Ubuntu-R.ttf';
const fontBytes = await fetch(fontUrl).then((res) => res.arrayBuffer())

// Load the PDF with form fields
const pdfDoc = await PDFDocument.load(formBytes);

// Embed the Ubuntu font
pdfDoc.registerFontkit(fontkit);
const ubuntuFont = await pdfDoc.embedFont(fontBytes);

// Get two text fields from the form
const form = pdfDoc.getForm();
const nameField = form.getTextField('CharacterName 2');
const ageField = form.getTextField('Age');

// Fill the text fields with some fancy Unicode characters (outside
// the WinAnsi latin character set)
nameField.setText('Măřîö');
ageField.setText('24 yêàřš');
```

```
// **Key Step:** Update the field appearances with the Ubuntu font
form.updateFieldAppearances(ubuntuFont);

// Save the PDF with filled form fields
const pdfBytes = await pdfDoc.save();
```

Handy Methods for Filling, Creating, and Reading Form Fields

Existing form fields can be accessed with the following methods of **PDFForm** :

- **PDFForm.getButton**
- **PDFForm.getCheckBox**
- **PDFForm.getDropdown**
- **PDFForm.getOptionList**
- **PDFForm.getRadioGroup**
- **PDFForm.getTextField**

New form fields can be created with the following methods of **PDFForm** :

- **PDFForm.createButton**
- **PDFForm.createCheckBox**
- **PDFForm.createDropdown**
- **PDFForm.createOptionList**
- **PDFForm.createRadioGroup**
- **PDFForm.createTextField**

Below are some of the most commonly used methods for reading and filling the aforementioned subclasses of **PDFField** :

- **PDFCheckBox.check**
- **PDFCheckBox.uncheck**
- **PDFCheckBox.isChecked**

-
- **PDFDropdown.select**
 - **PDFDropdown.clear**
 - **PDFDropdown.getSelected**
 - **PDFDropdown.getOptions**

- `PDFDropdown.addOptions`
-

- `PDFOptionList.select`
 - `PDFOptionList.clear`
 - `PDFOptionList.getSelected`
 - `PDFOptionList.getOptions`
 - `PDFOptionList.addOptions`
-

- `PDFRadioGroup.select`
 - `PDFRadioGroup.clear`
 - `PDFRadioGroup.getSelected`
 - `PDFRadioGroup.getOptions`
 - `PDFRadioGroup.addOptionToPage`
-

- `PDFTextField.setText`
- `PDFTextField.getText`
- `PDFTextField.setMaxLength`
- `PDFTextField.getMaxLength`
- `PDFTextField.removeMaxLength`

Limitations

- `pdf-lib` **can** extract the content of text fields (see `PDFTextField.getText`), but it **cannot** extract plain text on a page outside of a form field. This is a difficult feature to implement, but it is within the scope of this library and may be added to `pdf-lib` in the future. See [#93](#), [#137](#), [#177](#), [#329](#), and [#380](#).
- `pdf-lib` **can** remove and edit the content of text fields (see `PDFTextField.setText`), but it does **not** provide APIs for removing or editing text on a page outside of a form field. This is also a difficult feature to implement, but is within the scope of `pdf-lib` and may be added in the future. See [#93](#), [#137](#), [#177](#), [#329](#), and [#380](#).
- `pdf-lib` does **not** support the use of HTML or CSS when adding content to a PDF. Similarly, `pdf-lib` **cannot** embed HTML/CSS content into PDFs. As convenient as such a feature might be, it would be extremely difficult to implement and is far beyond the scope of this library. If this capability is something you need, consider using [Puppeteer](#).

Help and Discussion

Discussions is the best place to chat with us, ask questions, and learn more about pdf-lib!

See also [MAINTAINERSHIP.md#communication](#) and [MAINTAINERSHIP.md#discord](#).

Encryption Handling

pdf-lib does not currently support encrypted documents. You should not use pdf-lib with encrypted documents. However, this is a feature that could be added to pdf-lib. Please **create an issue** if you would find this feature helpful!

When an encrypted document is passed to `PDFDocument.load(...)`, an error will be thrown:

```
import { PDFDocument, EncryptedPDFError } from 'pdf-lib'

const encryptedPdfBytes = ...

// Assignment fails. Throws an `EncryptedPDFError`.
const pdfDoc = PDFDocument.load(encryptedPdfBytes)
```

This default behavior is usually what you want. It allows you to easily detect if a given document is encrypted, and it prevents you from trying to modify it. However, if you really want to load the document, you can use the `{ ignoreEncryption: true }` option:

```
import { PDFDocument } from 'pdf-lib'

const encryptedPdfBytes = ...

// Assignment succeeds. Does not throw an error.
const pdfDoc = PDFDocument.load(encryptedPdfBytes, { ignoreEncryption: true })
```

Note that **using this option does not decrypt the document**. This means that any modifications you attempt to make on the returned `PDFDocument` may fail, or have unexpected results.

You should not use this option. It only exists for backwards compatibility reasons.

Contributing

We welcome contributions from the open source community! If you are interested in contributing to `pdf-lib`, please take a look at the **CONTRIBUTING.md** file. It contains information to help you get `pdf-lib` setup and running on your machine. (We try to make this as simple and fast as possible! 🚀)

Maintainership

Check out **MAINTAINERSHIP.md** for details on how this repo is maintained and how we use **issues**, **PRs**, and **discussions**.

Tutorials and Cool Stuff

- **labelmake** - a library for declarative PDF generation created by @hand-dot
- **Möbius Printing helper** - a tool created by @shreevatsa
- **Extract PDF pages** - a tool created by @shreevatsa
- **Travel certificate generator** - a tool that creates travel certificates for French citizens under quarantine due to COVID-19
- **How to use pdf-lib in AWS Lambdas** - a tutorial written by Crespo Wang
- **Working With PDFs in Node.js Using pdf-lib** - a tutorial by Valeri Karpov
- **Electron app for resizing PDFs** - a tool created by @vegarringdal
- **PDF Shelter** - online PDF manipulation tools by Lucas Morais

Prior Art

- **pdfkit** is a PDF generation library for Node and the Browser. This library was immensely helpful as a reference and existence proof when creating `pdf-lib`. `pdfkit`'s code for **font embedding**, **PNG embedding**, and **JPG embedding** was especially useful.
- **pdf.js** is a PDF rendering library for the Browser. This library was helpful as a reference when writing `pdf-lib`'s parser. Some of the code for stream decoding was **ported directly to TypeScript** for use in `pdf-lib`.
- **pdfbox** is a PDF generation and modification library written in Java. This library was an invaluable reference when implementing form creation and filling APIs for `pdf-lib`.
- **jspdf** is a PDF generation library for the browser.
- **pdfmake** is a PDF generation library for the browser.
- **hummus** is a PDF generation and modification library for Node environments. `hummus` is a Node wrapper around a **C++ library**, so it doesn't work in many JavaScript environments - like the Browser or React Native.

- **react-native-pdf-lib** is a PDF generation and modification library for React Native environments. `react-native-pdf-lib` is a wrapper around **C++** and **Java** libraries.
- **pdfassembler** is a PDF generation and modification library for Node and the browser. It requires some knowledge about the logical structure of PDF documents to use.

Git History Rewrite

This repo used to contain a file called `pdf_specification.pdf` in the root directory. This was a copy of the **PDF 1.7 specification**, which is made freely available by Adobe. On 8/30/2021, we received a DMCA complaint requiring us to remove the file from this repo. Simply removing the file via a new commit to `master` was insufficient to satisfy the complaint. The file needed to be completely removed from the repo's git history. Unfortunately, the file was added over two years ago, this meant we had to rewrite the repo's git history and force push to `master` 😞.

Steps We Took

We removed the file and rewrote the repo's history using **BFG Repo-Cleaner** as outlined [here](#). For full transparency, here are the exact commands we ran:

```
$ git clone git@github.com:Hopding/pdf-lib.git
$ cd pdf-lib
$ rm pdf_specification.pdf
$ git commit -am 'Remove pdf_specification.pdf'
$ bfg --delete-files pdf_specification.pdf
$ git reflog expire --expire=now --all && git gc --prune=now --aggressive
$ git push --force
```

Why Should I Care?

If you're a user of `pdf-lib`, you shouldn't care! Just keep on using `pdf-lib` like normal 😊 ✨!

If you are a `pdf-lib` developer (meaning you've forked `pdf-lib` and/or have an open PR) then this does impact you. If you forked or cloned the repo prior to 8/30/2021 then your fork's git history is out of sync with this repo's `master` branch. Unfortunately, this will likely be a headache for you to deal with. Sorry! We didn't want to rewrite the history, but there really was no alternative.

It's important to note that `pdf-lib`'s *source code* has not changed at all. It's exactly the same as it was before the git history rewrite. The repo still has the exact same number of commits (and even the same

commit contents, except for the commit that added pdf_specification.pdf). What has changed are the SHAs of those commits.

The simplest way to deal with this fact is to:

- 1. Reclone pdf-lib
- 2. Manually copy any changes you've made from your old clone to the new one
- 3. Use your new clone going forward
- 4. Reopen your unmerged PRs using your new clone

See this [StackOverflow answer](#) for a great, in depth explanation of what a git history rewrite entails.

License

MIT

Keywords

pdf-lib pdf document create modify creation modification edit editing
typescript javascript library

Install

```
> npm i pdf-lib
```

Repository

 github.com/Hopding/pdf-lib

Homepage

 pdf-lib.js.org/


Weekly Downloads

541,951



Version

License

1.17.1	MIT
	Unpacked Size
	19.5 MB
Total Files	Issues
1647	175
Pull Requests	
29	
Last publish	
2 years ago	
Collaborators	
	

>Try on RunKit

🚩Report malware



Support

Help

Advisories

Status

Contact npm

Company

About

Blog

Press

Terms & Policies

Policies

Terms of Use

Code of Conduct

Privacy