

PdfPecker manual

Fast HTML and Markdown to PDF for .NET, with no dependencies.

For PdfPecker 1.24.0 · pdfpecker.net/docs

Contents

Getting started 9
- Install 10
- Usage 10
- System requirements 10
- Where to go next 11

HTML to PDF in C# 12
- Install the NuGet package 12
- Convert an HTML string to PDF 12
- Convert an HTML file to PDF 12
- Generate an invoice from a template 12
- Return a PDF from ASP.NET Core 13
- Page size, margins, headers and footers 14
- Fonts 14
- Images 14
- Common questions 14

Options 16
- Files, network and untrusted HTML 16
- Page size from CSS 16
- Document properties 17
- Viewer settings and page labels 17
- Fast Web View and photo metadata 17
- Reproducible output 18
- Errors, cancellation and output files 18
- Copying options 18
- More options 19

HTML and CSS 20
- HTML 20
- CSS 20
- CSS properties . . . . . 21 Changed in 1.24.0
- Units and colours . . . . . 26 Changed in 1.24.0
- Subgrid 27
- Multi-column layout 27
- Percentage heights 27
- Container queries 28
- Tables . . . . . 28 Changed in 1.24.0
- display: contents 29
- Text wrapping and justification 29
- Vertical text 29
- Float shapes 30
- Drop caps 30
- Masks and backdrop-filter 30
- 3D transforms 31
- Trimming text boxes 31

Emphasis marks . . . 31
Corner shapes . . . 31
Scoped styles . . . 32
Typed attr() . . . 32
Zoom . . . 32
Gap decorations . . . 32
Check boxes and radio buttons . . . 33
Scale content to fit a box . . . 33
Pagination . . . 33
Links and bookmarks . . . 34
SVG . . . 34
Images . . . 35

Paged media . . . 37
Page rules and margin boxes . . . 37 Changed in 1.24.0
Footnotes . . . 38
Running header and footer from HTML . . . 38
Shrink to fit . . . 39

Markdown . . . 41
What is supported . . . 41
Footnotes, definition lists and front matter . . . 41

Fluent C# API . . . 43
What it covers . . . 43 Changed in 1.24.0
Coming from QuestPDF . . . 45
Coming from MigraDoc . . . 45

Razor components . . . 47
A template . . . 47
What works . . . 48
Safety . . . 48
Trimming and Native AOT . . . 49

Command line . . . 50
Native executable . . . 50 New in 1.24.0
Commands . . . 50
Replacing wkhtmltopdf . . . 51
Signing . . . 51

Python . . . 53
Converting . . . 53
MCP server . . . 53
Agent skill . . . 54
What to know . . . 54

Rust . . . 55
Converting . . . 55
Deploying . . . 55
Speed . . . 56
What to know . . . 56

Images to PDF **57**
Page size 57
Output settings 57
Errors 57

Fonts **58**
@font-face 58
Variable fonts 58
OpenType features . . . . . 59 Changed in 1.24.0
Colour fonts 59
Font properties 60
Standard fonts 60
Fonts read when needed . . . . . 60 New in 1.24.0
Missing fonts and characters 61
How fonts are embedded . . . . . 61 Changed in 1.24.0

Right-to-left, CJK and hyphenation **62**
Right-to-left text: Arabic and Hebrew 62
Indic, Thai and Lao scripts 62
Chinese, Japanese and Korean line breaking 63
CJK punctuation trimming 63
Hyphenation . . . . . 64 Changed in 1.24.0

PDF/A, passwords and watermarks **66**
Watermarks and passwords . . . . . 66 Changed in 1.24.0
PDF/A 66
Passwords on existing PDFs 67

Accessibility **68**
Tagged PDF 68
PDF/UA-1 68
PDF/UA-2 68
What your HTML needs 69

Fillable forms **70**
Field names and behaviour 70
Filling and flattening existing PDFs 71

Digital signatures **72**
Signing into an existing field 72
Formats and compatibility 73
External signers: HSMs, smart cards and key vaults 73
Certification signatures 74
Verifying signatures . . . . . 74 Changed in 1.24.0
Long-term validation 75

E-invoices **77**
Invoice XML from invoice data 77
Ready-made invoice template 78
Factur-X and ZUGFeRD 79
Reading the invoice from a received PDF 79
PDF attachments in UBL e-invoices (Peppol, OIOUBL) 80

Barcodes **81**
QR codes 81
EPC payment QR codes 81
Swiss QR-bills 82
Data Matrix 82
Code 128 and EAN-13 82
Options 82
Reading barcodes . . . . . 83 New in 1.24.0

Attachments **84**
How files are stored 84
Adding to an existing PDF 84

Merging and splitting **85**
Merging 85
Merging PDFs as they arrive 85
Extracting, reordering and splitting pages 86

Stamping **88**
Placement 88
What is kept 88

Compacting **90**
What it does 90
Options 90
Signed and encrypted PDFs 91
Fast Web View 91

Removing active content **92**
What it removes . . . . . 92 Changed in 1.24.0
The output . . . . . 92 Changed in 1.24.0

Redaction . . . . . **94** New
Reviewing before redacting 94
What is removed 94

PDF to PDF/A **96**
What it fixes 96
What it cannot fix 97
Encrypted and signed PDFs 97

Text extraction **98**
Pages, lines and words 98
What is read 98
Reading order . . . . . 99 New in 1.24.0
Images 99
Document properties 100

PDF to image **101**
HTML to images 101
Output formats 101
TIFF 102

Options . . . 102
Pixels for OCR and image processing . . . 103 New in 1.24.0
What the renderer supports . . . 103

PDF to HTML . . . 104
The HTML . . . 104
Structured HTML . . . 104
Options . . . 105
What it is not . . . 106

PDF to Markdown . . . 107
What it finds . . . 107
Options . . . 107
What it is not . . . 108

From wkhtmltopdf . . . 109
Replace the wkhtmltopdf call . . . 109
Options side by side . . . 110
What changes in the output . . . 111
Speed and size . . . 111
Next steps . . . 111

From HtmlRenderer.PdfSharp . . . 112
Replace the HtmlRenderer call . . . 112
Options side by side . . . 113
What changes in the output . . . 113
Speed and memory . . . 113
When HtmlRenderer is still a fine choice . . . 114

From IronPDF . . . 115
Replace the IronPDF call . . . 115
Headers and footers in HTML . . . 115
Matching IronPDF's output exactly . . . 116
Options side by side . . . 116
Speed . . . 116
When IronPDF is the better choice . . . 117

From PuppeteerSharp or Playwright . . . 118
Replace the browser code . . . 118
Headers and footers . . . 119
Options side by side . . . 119
What changes . . . 120
Speed and memory . . . 120
When to keep Chrome . . . 120

Changelog . . . 121
1.25.0 . . . 121
1.24.0 . . . 121
1.23.0 . . . 124
1.22.0 . . . 126
1.21.0 . . . 127
1.20.0 . . . 129

1.19.0 . . . 130
1.18.0 . . . 131
1.17.0 . . . 131
1.16.0 . . . 132
1.15.0 . . . 133
1.14.2 . . . 134
1.14.1 . . . 134
1.14.0 . . . 134
1.13.0 . . . 134
1.12.0 . . . 135
1.11.0 . . . 135
1.10.0 . . . 136
1.9.0 . . . 136
1.8.0 . . . 136
1.7.0 . . . 137
1.6.0 . . . 137
1.5.1 . . . 138
1.4.0 . . . 138
1.3.0 . . . 138
1.2.0 . . . 138
1.1.1 . . . 138
1.1.0 . . . 138
1.0.0 . . . 138

Limitations . . . 140
By design . . . 140
Text and fonts . . . 140 Changed in 1.24.0
CSS . . . 141 Changed in 1.24.0
Backgrounds and transparency . . . 142 Changed in 1.24.0
Layout . . . 142 Changed in 1.24.0
Images and SVG . . . 143 Changed in 1.24.0
Reading and changing existing PDFs . . . 143 Changed in 1.24.0
Markdown and the PDF renderer . . . 144

CSS support . . . 145
Properties . . . 145
At-rules . . . 167
Selectors . . . 168
Units, colours and functions . . . 170

PDF feature checklist . . . 172
File format and structure . . . 172 Changed in 1.24.0
Creating content . . . 173
Text and fonts . . . 174
Graphics, images and colour . . . 174
Navigation and interactivity . . . 175
Annotations . . . 176
Actions . . . 177
Forms (AcroForm) . . . 177
Security and encryption . . . 178 Changed in 1.24.0
Digital signatures . . . 179

Tagging and accessibility 180
Print production 180
Standards 180
Internet standards (RFCs) 181
Metadata and attachments 181
Editing existing PDFs 182
Reading, rendering and conversion . . . . . 182 Changed in 1.24.0

Snapshot testing 184
Snapshots 184
What has to match 185
Text and pages 186
Limitations 186

Performance 187
Compared with other converters 187
Conversion speed 188

How it works 190

Getting started

New and changed in PdfPecker 1.24.0 (since 1.23.0):

- HTML and CSS: CSS properties Changed
- HTML and CSS: Units and colours Changed
- HTML and CSS: Tables Changed
- Paged media: Page rules and margin boxes Changed
- Fluent C# API: What it covers Changed
- Command line: Native executable New
- Command line: Local files and passwords New
- Fonts: OpenType features Changed
- Fonts: Fonts read when needed New
- Fonts: How fonts are embedded Changed
- Right-to-left, CJK and hyphenation: Hyphenation Changed
- PDF/A, passwords and watermarks: Watermarks and passwords Changed
- Digital signatures: Verifying signatures Changed
- Barcodes: Reading barcodes New
- Removing active content: What it removes Changed
- Removing active content: The output Changed
- Redaction New
- Text extraction: Reading order New
- PDF to image: Pixels for OCR and image processing New
- Limitations: Text and fonts Changed
- Limitations: CSS Changed
- Limitations: Backgrounds and transparency Changed
- Limitations: Layout Changed
- Limitations: Images and SVG Changed
- Limitations: Reading and changing existing PDFs Changed
- PDF feature checklist: File format and structure Changed
- PDF feature checklist: Security and encryption Changed
- PDF feature checklist: Reading, rendering and conversion Changed

Every release in the changelog

PdfPecker turns HTML and Markdown into PDF, and PDF pages into PNG, JPEG, WebP or TIFF images, in .NET. It has its own HTML parser, CSS engine, layout engine, PDF writer and PDF renderer, so it never starts a browser, ships no native binaries and depends on nothing but the .NET base class library. These pages describe the latest release; the version picker has the manual for earlier ones.

```
using PdfPecker;

byte[] pdf = HtmlToPdf.Convert("<h1>Hello</h1><p>World</p>");
File.WriteAllBytes("hello.pdf", pdf);
```

Install

```
dotnet add package PdfPecker
```

The package targets `net8.0`, `net9.0` and `net10.0`, so it works on .NET 8 and later.

Other packages

Four more packages are released with PdfPecker under the same version:

Package	What it is
`PdfPecker.Fluent`	A fluent C# API in the style of QuestPDF
`PdfPecker.Razor`	Razor components to PDF
`PdfPecker.Cli`	The `pdfpecker` command-line tool (`dotnet tool install -g PdfPecker.Cli`)
`PdfPecker.Testing`	Snapshot tests for PDFs

From Python, `pip install pdfpecker` installs the same library from PyPI, with an MCP server for AI agents; see Python.

Usage

```
// String to bytes
byte[] pdf = HtmlToPdf.Convert(html);

// String to a stream
using (var file = File.Create("out.pdf")) HtmlToPdf.Convert(html, file);

// ASP.NET Core
app.MapGet("/invoice", () => Results.File(HtmlToPdf.Convert(html), "application/pdf",
"invoice.pdf"));

// File to file; relative <img> and <link> paths resolve against the HTML file's folder
HtmlToPdf.ConvertFile("invoice.html", "invoice.pdf");

// Off the calling thread
byte[] pdf = await HtmlToPdf.ConvertAsync(html);
```

The conversion methods are static and thread-safe, so you can convert many documents in parallel. The few instance types are `FontCollection` and `PdfRenderer`, which parallel work can share, and `StreamingPdfMerger`, which builds one output and is used from one thread at a time. Page size, margins, headers, footers and the rest are set with `PdfOptions`.

System requirements

Required: the .NET 8, .NET 9 or .NET 10 runtime (or later). Nothing else: the library is pure managed code with no native binaries, browser or third-party packages, so it runs wherever .NET runs (Windows, Linux, macOS; x64 and ARM64). The rendered pages are the same on every version; only the compressed bytes can differ, because each .NET version ships its own zlib build.

Recommended:

- Register the fonts your templates use with `options.Fonts` (see Fonts); other fonts fall back to Helvetica, Times or Courier metrics.
- Memory and CPU are small. Measured on a 4-core 2.1 GHz Xeon cloud VM (Linux, .NET 10, Release), each document in a fresh process:

Document	Pages	Time, warm	Time, first in process	Peak process memory	Smallest heap that works
Payslip, invoice, voucher	1 to 3	1 to 3 ms	0.2 s	50 MB	under 16 MB
Same with PDF/A or embedded fonts	1 to 3	3 to 5 ms	0.2 s	56 MB	under 16 MB
Report with tables	30	30 to 40 ms	0.3 s	75 MB	under 16 MB
Report with tables	1,000	1.7 s	3 s	480 MB	245 MB

  The process itself starts at about 32 MB; a conversion uses about one core, so throughput grows with cores. PDF/A and font embedding add a few milliseconds and about 5 MB for the embedded font data. Memory grows roughly with page count, so plan about 0.5 MB per page for very long documents.

From a clone of the PdfPecker repository, `dotnet run --project bench/PdfPecker.Benchmarks -c Release -- --requirements [your.html ...]` measures this on your own machine and documents.

Where to go next

- Options: page size, margins, headers and footers, document properties and file access.
- HTML and CSS support and Paged media: what your templates can use.
- Fonts: register your own fonts and use any Unicode text.
- Fluent C# API: build PDFs from C# code instead of HTML, in the style of QuestPDF.
- Razor components and the command line: other ways in.
- Snapshot testing: check in your tests that PDFs come out as before.
- Limitations: what PdfPecker does not do (yet).

HTML to PDF in C#

This guide shows how to convert HTML to PDF in C# with PdfPecker, from a one-line conversion to an invoice built from a template and served from ASP.NET Core. PdfPecker is a pure C# library: it does not start Chrome or another browser, needs no native binaries such as wkhtmltopdf, and has no NuGet dependencies, so the same code runs on Windows, Linux, macOS, in Docker and in serverless functions.

Install the NuGet package

```
dotnet add package PdfPecker
```

PdfPecker works with .NET 8, .NET 9 and .NET 10 and is free under the MIT license.

Convert an HTML string to PDF

```
using PdfPecker;

byte[] pdf = HtmlToPdf.Convert("<h1>Hello</h1><p>World</p>");
File.WriteAllBytes("hello.pdf", pdf);
```

`HtmlToPdf.Convert` returns the PDF as a byte array. To write it straight to a file, a `MemoryStream` or an HTTP response, pass a stream instead:

```
using (var file = File.Create("hello.pdf")) HtmlToPdf.Convert(html, file);
```

Convert an HTML file to PDF

```
HtmlToPdf.ConvertFile("invoice.html", "invoice.pdf");
```

Relative image and stylesheet paths in the file, such as `<img src="logo.png">` or `<link rel="stylesheet" href="invoice.css">`, are read from the HTML file's folder. When you convert a string, set `BaseDirectory` to say where those files are:

```
var options = new PdfOptions { BaseDirectory = "/var/app/templates" };
byte[] pdf = HtmlToPdf.Convert(html, options);
```

Generate an invoice from a template

Build the HTML with whatever produces a string, such as a Razor view rendered to a string, a template engine, or plain string interpolation, and convert the result. Encode values that come from users or a

database, so that a customer name with a `<` in it cannot break the page.

```
using System.Net;
using PdfPecker;

record Line(string Description, int Quantity, decimal Price);

static byte[] Invoice(string customer, IEnumerable<Line> lines)
{
    var rows = string.Concat(lines.Select(l =>
$"<tr><td>{WebUtility.HtmlEncode(l.Description)}</td><td>{l.Quantity}</td><td>{l.Quantity *
l.Price:N2}</td></tr>"));
    var total = lines.Sum(l => l.Quantity * l.Price);

    var html = $$"""
        <style>
          body { font-family: Helvetica, sans-serif; font-size: 10pt }
          table { width: 100%; border-collapse: collapse }
          th, td { padding: 6px; border-bottom: 1px solid #ddd; text-align: left }
          td:last-child, th:last-child { text-align: right }
        </style>
        <h1>Invoice</h1>
        <p>To: {{WebUtility.HtmlEncode(customer)}}</p>
        <table>
          <thead><tr><th>Item</th><th>Qty</th><th>Amount</th></tr></thead>
          <tbody>{{rows}}</tbody>
        </table>
        <p style="text-align: right"><b>Total: {{total:N2}}</b></p>
        """;

    return HtmlToPdf.Convert(html, new PdfOptions { Title = "Invoice" });
}
```

Table headers repeat on every page when a table runs over a page break, and flexbox, grid and the rest of the CSS PdfPecker supports work in templates as they do in a browser.

Return a PDF from ASP.NET Core

```
app.MapGet("/invoice/{id}", (int id) =>
{
    byte[] pdf = Invoice("ACME Ltd", LoadLines(id));
    return Results.File(pdf, "application/pdf", $"invoice-{id}.pdf");
});
```

In an MVC controller, `return File(pdf, "application/pdf", "invoice.pdf");` does the same. All PdfPecker methods are static and thread-safe, so many requests can convert at once, and `HtmlToPdf.ConvertAsync` moves the work off the calling thread.

Page size, margins, headers and footers

```
var options = new PdfOptions
{
    PageSize = PageSize.Letter,                          // A4 is the default
    Margins = PageMargins.FromMillimeters(20, 15, 20, 15),
    HeaderText = "ACME Ltd",
    FooterText = "Page {page} of {pages}",
};
byte[] pdf = HtmlToPdf.Convert(html, options);
```

The page can also be set from CSS, which keeps it with the template:

```
@page {
  size: A4 landscape;
  margin: 20mm 15mm;
  @bottom-right { content: "Page " counter(page) " of " counter(pages) }
}
```

For a header or footer written in HTML, with a logo or a table, see Paged media, and for every setting see Options.

Fonts

Text without a registered font uses the standard PDF fonts Helvetica, Times and Courier. To use your own font, register the TrueType, OpenType, WOFF or WOFF2 file and refer to it by its family name in CSS; only the characters the document uses are embedded.

```
var options = new PdfOptions();
options.Fonts.AddFile("fonts/OpenSans-Regular.ttf");
byte[] pdf = HtmlToPdf.Convert("<p style=\"font-family: 'Open Sans'\">Hello</p>", options);
```

Fonts covers `@font-face`, bold and italic files, and what happens when a font is missing.

Images

`<img>` and CSS backgrounds take PNG, JPEG, GIF, WebP and SVG files, and `data:` URLs. Files are read only from inside `BaseDirectory`, and PdfPecker never downloads anything by itself: to load `http` or `https` images, hand them over with `ResourceResolver`.

```
var http = new HttpClient();
var options = new PdfOptions
{
    ResourceResolver = url => url.StartsWith("https://cdn.example.com/")
        ? http.GetByteArrayAsync(url).GetAwaiter().GetResult()
        : null,
};
```

Common questions

Does PdfPecker need Chrome, Puppeteer or wkhtmltopdf? No. It has its own HTML parser, CSS engine, layout engine and PDF writer, written in C#. There is nothing to install on the server besides the .NET runtime.

Does it run on Linux, in Docker and in Azure Functions or AWS Lambda? Yes. It is pure managed code, so it runs wherever .NET 8 or later runs, on x64 and ARM64.

Does it run JavaScript? No. PdfPecker converts the HTML and CSS you give it, which suits templates you control, such as invoices, reports, payslips and letters. Pages that build themselves with JavaScript need a browser-based converter; Compare lists them.

How fast is it? A one-page invoice takes about a millisecond once the process is warm. See Performance for the measurements.

Is it free for commercial use? Yes. PdfPecker is open source under the MIT license.

Can I also convert Markdown, or make PDF/A? Yes: `MarkdownToPdf.Convert` takes Markdown (see Markdown), and `PdfOptions.Conformance` makes PDF/A for archiving (see PDF/A, passwords and watermarks).

Options

`PdfOptions` controls the page, the document properties and how resources are loaded. Every conversion method takes one; leave it out for A4 pages with the defaults below.

```
var options = new PdfOptions
{
    PageSize = PageSize.Letter.Landscape,            // A3, A4 (default), A5, Letter, Legal or
new PageSize(w, h) in points
    Margins = PageMargins.FromMillimeters(20, 15, 20, 15),
    Scale = 1,                                       // like Chrome's print scale (see its
docs to match IronPDF's output)
    ShrinkToFit = true,                              // lay out too-wide content wider and
draw it smaller, as Chrome prints (default)
    TruncateBandsToWholePoints = false,              // round header/footer height attributes
down to whole points, as Chrome does
    BaseDirectory = "/var/app/templates",            // where relative image/stylesheet paths
are resolved
    ResourceResolver = url => myCache.Get(url),      // serve http(s) or embedded resources
yourself; null = default handling
    UserStyleSheet = "body { font-size: 10pt }",     // extra CSS applied after the
document's own
    FooterText = "Page {page} of {pages}",           // also HeaderText
    Title = "Invoice 42", Author = "Accounts",       // document properties (Title defaults
to <title>)
    Keywords = "invoice", Creator = "MyApp",         // also Subject, and Producer (default
"PdfPecker")
    CustomProperties = { ["RunId"] = "42" },         // your own properties, in the PDF's
document information
    Bookmarks = true,                                // PDF bookmarks from h1-h6 (default)
    FormFields = true,                               // inputs, selects and textareas become
fillable fields (default)
    Compress = true,                                 // Flate-compress page content (default)
    AllowFileAccess = false,                         // true: also read files outside
BaseDirectory (trusted HTML only)
    MaxPages = 1000,                                 // refuse longer documents with
PdfLimitException (default 100,000)
    Linearize = true,                                // Fast Web View: the first page shows
before the rest has loaded
    StripImageMetadata = true,                       // drop EXIF (GPS location too), XMP and
IPTC from embedded JPEGs
};
byte[] pdf = HtmlToPdf.Convert(html, options);
```

Files, network and untrusted HTML

Local images and stylesheets are read only from inside `BaseDirectory` (the input file's folder for `ConvertFile`): absolute paths, `file:` URLs, `..` and symbolic links that lead outside it are ignored unless `AllowFileAccess = true`, and converting a string without a `BaseDirectory` reads no files at all. For HTML from untrusted users, keep it that way, set a low `MaxPages`, and pass a cancellation token with a timeout (`HtmlToPdf.Convert(html, options, token)`): cancelling stops the conversion where it is.

Page size from CSS

A CSS `@page { size: A4 landscape; margin: 15mm }` rule overrides the page size and margins unless `IgnoreCssPageRules` is set. Conversion never makes network requests: `http(s)` images and stylesheets are only loaded through `ResourceResolver`. Only signing goes online, and only when asked: a timestamp from `TimestampUrl`, revocation data with `FetchRevocationData`, and online revocation checks when verifying.

Document properties

Custom properties are written to the document information dictionary, where Acrobat shows them under Document Properties > Custom and `pdfinfo -custom` lists them. For PDF/A they are also written to the XMP metadata (Acrobat's `pdfx` namespace, with the extension schema PDF/A requires), alongside Keywords, Creator and Producer.

Viewer settings and page labels

`Viewer` sets how a PDF viewer opens the document: the start page and zoom, the page layout (one page, a scrolling column, or two pages side by side like a book), the panel next to the pages (bookmarks, thumbnails or attachments) or full screen, the window (fit, centre, hide the toolbar, menu bar or controls, show the title instead of the file name) and the print dialog (100% instead of fit to paper, so forms and labels print at their real size, and duplex). `PageLabels` sets the page numbers a viewer shows, such as "i" to "iv" for the front matter and "1" onwards after it, or "A-1" for an appendix; they do not change `{page}` or CSS page counters. Every setting's default leaves the choice to the viewer, and viewers treat them as suggestions: browsers in particular ignore some of them.

```
var options = new PdfOptions
{
    Viewer = new PdfViewerSettings
    {
        PageMode = PdfPageMode.Bookmarks,             // also PagesOnly, Thumbnails,
Attachments, FullScreen
        PageLayout = PdfPageLayout.OneColumn,         // also SinglePage, TwoColumnLeft/Right,
TwoPageLeft/Right
        Zoom = PdfZoom.FitWidth,                      // also FitPage, ActualSize
        StartPage = 1,
        DisablePrintScaling = true,
        Duplex = PdfDuplex.FlipLongEdge,
    },
    PageLabels =
    {
        new PdfPageLabel(1, PdfPageNumberStyle.LowerRoman),   // pages 1-4: i, ii, iii, iv
        new PdfPageLabel(5),                                  // from page 5: 1, 2, 3, ...
        new PdfPageLabel(20, PdfPageNumberStyle.Decimal, prefix: "A-"),
    },
};
```

`PdfMergeOptions` has the same `Viewer` and `PageLabels` for a merged file. A tagged PDF/UA-2 file opens at the first element of its start page, as PDF/UA-2 requires.

Fast Web View and photo metadata

`Linearize = true` writes a linearized PDF (Fast Web View): the first page and a table of where every later page lies come first, so a browser or viewer loading the file over the web shows the first page before the rest has arrived. It costs a second pass over the file and adds a little to its size, so it pays off for large documents served over a network. It works with `Security`, `Conformance` and `Accessibility`, but not with `Signature`, since a signature is added as an incremental update, which ends the linearization. `PdfLinearizer` does the same for any existing PDF, see Compacting.

JPEG images are embedded byte for byte, so a photo in a template carries its EXIF data into the PDF: the camera, the time it was taken and often its GPS location. `StripImageMetadata = true` removes EXIF, XMP, IPTC and comments from each JPEG as it is embedded, and keeps the colour profile, so the images look the same. Other image formats are decoded on embedding and never keep their metadata.

Reproducible output

By default every PDF records the time it was made and gets a random file identifier, so two conversions of the same HTML differ. Set `CreationDate`, or the `SOURCE_DATE_EPOCH` environment variable (seconds since 1970, as used by reproducible builds), and the dates are fixed and the identifier is derived from a hash of the file, so the same input gives a byte-identical PDF. `PdfMergeOptions`, `PdfPagesOptions`, `PdfStampOptions` and `PdfFormFillOptions` have the same `CreationDate`. Signed and encrypted output still differs each time, since signing time and encryption keys are part of it.

```
var options = new PdfOptions { CreationDate = new DateTimeOffset(2026, 1, 1, 0, 0, 0,
TimeSpan.Zero) };
```

Errors, cancellation and output files

Errors the library raises itself derive from `PdfPeckerException`: `PdfLimitException` (a limit such as `MaxPages` reached), `PdfPasswordException` and `PdfSignatureException`. Unreadable input PDFs throw `InvalidDataException`, and bad arguments the usual `ArgumentException`s; each method's documentation lists what it throws. Every method that writes a file (`ConvertFile`, `StampFile`, `SignFile`, `MergeFiles`, ...) writes to a temporary file first and replaces the target only when it succeeds, so a failure leaves an existing file untouched. Methods that read or write PDFs take a `CancellationToken` and have `Stream` overloads, and `ConvertAsync` can write straight to a `Stream`.

Copying options

`Clone()` copies a `PdfOptions`, so shared settings can stay in one place while a single conversion changes one of them. The copy is shallow: `Fonts`, `Attachments`, `CustomProperties` and the other settings that hold objects refer to the same objects in both, so replace such a setting on the copy rather than changing what it holds.

```
var draft = options.Clone();
draft.HeaderText = "Draft";
byte[] draftLetter = HtmlToPdf.Convert(html, draft);
```

More options

The other options have pages of their own: `Fonts`, `EmbedStandardFonts`, `MissingFont` and `MissingCharacter`, `Language` and `Hyphenation`, `Watermark`, `Security` and `Conformance`, `Accessibility`, `EInvoice`, `Attachments`, `Signature` and `FormFields`.

HTML and CSS support

> **Changed in 1.24.0:** CSS properties, Units and colours, Tables.

PdfPecker lays out HTML and CSS the way Chrome prints it. Anything not listed here should be assumed missing; the known gaps are listed under Limitations, and CSS support marks every CSS property, selector and unit as supported or not.

HTML

The common document elements: headings, paragraphs, `div`/`section`-style blocks, `span`, `a` (external links and `#id` links inside the document), `b`/`strong`, `i`/`em`, `u`, `s`, `sub`, `sup`, `small`, `big`, `mark`, `code`, `pre`, `blockquote`, `hr`, `br`, `ul`/`ol`/`li` (nested, with `start`, `value`, `type` and `reversed`), `dl`, `table` (with `thead`, `tbody`, `tfoot`, `caption`, `colspan`, `rowspan`, `col width`), `img` (with `srcset` and `sizes`), `<picture>` and `<source>`, pictures in `<embed>` and `<object>` (see Images), inline `<svg>`, `<details>` and `<summary>` (with the disclosure triangle; a closed `details` shows only its summary), and `<progress>` and `<meter>` drawn as Chrome draws them (meter colours from `low`, `high` and `optimum`). Legacy attributes work as in browsers: `align`, `valign`, `bgcolor`, `width`, `height`, `border`, `cellpadding`, `cellspacing`, `<font>`, `<center>`, `img align=left|right` (a float, with `hspace` and `vspace`), `table bordercolor` and `rules`, and `align=center` on a `div` or cell, which centres a nested table as e-mail templates expect, `<br clear=all>` (and `left`, `right`, `both`), which starts the next line below the floats, and `<hr size>`, `color` and `noshade`; the default `<hr>` has Chrome's shaded grey edges. The `dir` attribute (`ltr`, `rtl` and `auto`), `<bdi>` and `<bdo>` set the text direction. The parser is forgiving about unclosed and misnested tags, the way browsers are (a stray `</p>` makes an empty paragraph, as in Chrome), and a document that leaves out `html`, `head` or `body` gets them, as in a browser: head content (`title`, `meta`, `link`, `style`) goes into the head and the first other tag or text starts the body, so `body { ... }`, `html { ... }` and `:root` rules apply to a fragment too. A body the document leaves out has no default margin, so fragment templates keep their layout; a `<body>` tag brings the usual 8px, as in Chrome. A page with a doctype such as `<!DOCTYPE html>` is in standards mode, as in a browser, so a length without a unit (`width: 200`) is ignored and an earlier declaration wins (a shorthand with such a part is dropped whole, as in Chrome); pages without a doctype read it as pixels. Without a doctype, line heights also follow Chrome's quirks mode: a line of small `<font>` text, or an image alone in a table cell, is only as tall as its content. The old `<body>` attributes `topmargin`, `leftmargin`, `marginheight` and `marginwidth` set the page margins, as in Chrome.

Markdown is converted to this HTML first; see Markdown.

CSS

`<style>` blocks, `<link rel="stylesheet">`, `@import` (with `layer()`, `supports()` and media queries; a relative URL is relative to the importing stylesheet; stylesheets in `data:` URLs are read from the URL itself), inline `style` attributes, `@media` (media queries level 4: `print`, `width`/`height`,

`aspect-ratio` and `orientation` of the page area, ranges such as `(400px <= width <= 800px)`, `and`, `or`, `not`; the other features answer as a printer does: `prefers-color-scheme: light`, `hover` and `pointer: none`, `scripting: none`, `overflow-block: paged`, and `resolution`, `color` and `color-gamut: srgb`), `@supports` (property tests, `selector()`, `and`/`or`/`not`), `@layer` cascade layers, CSS nesting (`&`), `@property` (initial values and inheritance of registered custom properties; a rule whose `syntax` or `initial-value` is invalid is ignored, as in Chrome), `@scope` (see Scoped styles), `@container` (see Container queries) and `@page` (see Paged media). Selectors: type, `.class`, `#id`, `*`, attribute selectors (`[a]`, `=`, `~=`, `|=`, `^=`, `$=`, `*=`, with `i`/`s` flags), descendant, `>`, `+`, `~`, `:first-child`, `:last-child`, `:only-child`, `:nth-child()` and `:nth-last-child()` (incl. `of S`), `:first-of-type`, `:last-of-type`, `:only-of-type`, `:nth-of-type()`, `:nth-last-of-type()`, `:not()`, `:is()`, `:where()` and `:has()` (all with lists of complex selectors, such as `:is(nav > a, .menu li)`, nested to any sensible depth; `:is()` and `:where()` skip entries they do not understand), `:root`, `:link`, `:empty`, `:lang()`, `:dir(ltr|rtl)` (from the `dir` attribute, `auto` by the first strong character; the `direction` property does not count, as in browsers), form states (`:checked`, `:disabled`, `:enabled`, `:required`, `:optional`, `:read-only`, `:placeholder-shown` and the like; `:valid` and `:invalid` check a field's `required`, `pattern`, `min`, `max`, `step` and the `email` and `url` types as Chrome does for a page nobody has typed into, `:in-range` and `:out-of-range` compare number, range and date inputs with their `min` and `max` as Chrome does, and `:open` matches `details` and `dialog` with the `open` attribute), `::before`, `::after`, `::marker`, `::first-letter` (drop caps, floated or with `initial-letter`), `::first-line` (with a background) and `::placeholder` (also `::-webkit-input-placeholder`) on text fields and textareas, with specificity and `!important` handled as in browsers. As when Chrome prints, interaction states such as `:hover`, `:focus` and `:target` never match, and neither does `:indeterminate`. The CSS-wide keywords `inherit`, `initial`, `unset`, `revert` and `revert-layer` work on every property, shorthands included (`border: inherit`), and `all` resets everything but `direction` and `unicode-bidi`. Escapes in selectors and attribute values (`.\31 0`, `[class=second\ two]`) resolve as CSS syntax says, and broken rules and declarations are skipped the way browsers skip them. The old `-webkit-` names that Chrome still accepts (`-webkit-transform`, `-webkit-border-radius`, `-webkit-flex` and the like) stand for their standard property, `display: -webkit-box` lays out as the old flexbox with `-webkit-box-orient`, `-pack`, `-align` and `-flex`, and `-webkit-linear-gradient()`, `-webkit-radial-gradient()` and `-webkit-gradient()` draw as gradients.

CSS properties Changed in 1.24.0

Area	Properties
Box model	`display` (block, inline, inline-block, flow-root, flex, inline-flex, grid, inline-grid, list-item, none, the two-value forms such as `block flex`, `-webkit-box`, CSS tables on any element, and `contents`), `margin`, `padding` (also the logical `margin-inline`/`-block` and `padding-inline`/`-block` properties and their `-start`/`-end` longhands, which follow `direction` and `writing-mode`; auto margins on an inline-block count as 0, as in Chrome), `border` (solid, dashed, dotted, double, groove, ridge, inset, outset, with dashes, dots, corners and 3D shades as

Area	Properties
Chrome draws them, and sides of different colours meeting in mitred corners; also the logical `border-inline`, `border-block`, their `-start` / `-end`, `-width`, `-style` and `-color` forms and the logical radii such as `border-start-end-radius`, placed by `direction` and `writing-mode`), `border-image` (images, SVG files and gradients, with `-slice` incl. `fill`, `-width`, `-outset` and `-repeat`: stretch, repeat, round, space), `border-radius` (incl. elliptical `/` radii and percentages; rounds backgrounds, borders, shadows and images), `corner-shape` (squircle, bevel, scoop, notch, square and `superellipse()` corners), `box-shadow` (outer and `inset` shadows; a blur is drawn as fading bands, truly transparent over whatever is behind it), `overflow` (`hidden`, `clip`, `auto` and `scroll` clip to the padding box, following its rounded corners, also in table cells; `overflow-x` / `overflow-y`; `overflow-clip-margin` moves the edge of `overflow: clip` to the content, padding or border box, grown or pulled in by a length), `outline` and `outline-offset` (all border styles, `outline-style: auto` as Chrome's focus ring; follows rounded corners, and a wrapped inline gets one outline round all its lines, as in Chrome), `width`, `height`, `min-` / `max-width`, `min-` / `max-height` (percentages as in browsers; see Percentage heights), the logical `inline-size`, `block-size` and their `min-` / `max-` forms, `min-content`, `max-content` and `fit-content` widths (also `-webkit-` spelled) that size a box from its content, also as a flex or grid item, `stretch` and `-webkit-fill-available`, which fill the containing block less the box's margins (also for heights and the minimums and maximums), `box-sizing`, `aspect-ratio` (incl. `auto <ratio>`; also in shrink-to-fit widths, with `min-` / `max-height`, and on tables), `object-fit` and `object-position` on images (SVG files too), `object-view-box` (`inset()`, `rect()` and `xywh()`), which crops and zooms an image to a part of it, `clip-path` (`inset()`, `rect()`, `xywh()`, `circle()`, `ellipse()`, `polygon()`, `path()`, reference boxes, and `url(#id)` of a `clipPath` in inline SVG), and the legacy `clip: rect()` on absolutely and fixed positioned boxes	

Area	Properties
Text	`color`, `font`, `font-family`, `font-size`, `font-weight`, `font-style`, `font-stretch` (also by its new name `font-width`), `font-variation-settings`, `font-optical-sizing` (see Variable fonts), `font-kerning`, `font-synthesis` and its longhands, `font-size-adjust`, `font-variant-position` and `font-variant-east-asian` (see Font properties), `line-height`, `text-align` (incl. justify, `start` and `end`), `text-justify` (see Text wrapping and justification), `direction`, `unicode-bidi`, `text-decoration`, `text-transform`, `text-indent` (lengths, percentages, `hanging` and `each-line`), `text-emphasis` and its longhands, `text-box`, `text-box-trim` and `text-box-edge`, `letter-spacing`, `word-spacing`, `white-space`, `overflow-wrap`/`word-wrap`, `word-break` (`break-all`, `keep-all`), `line-break` (`auto`, `strict`, `normal`, `loose`, `anywhere`; see CJK line breaking), `vertical-align` (keywords, lengths and percentages such as `-0.125em` for icons; `top` and `bottom` align to the line box and `text-top`, `text-bottom` and `middle` to the parent's font, images and inline-blocks too, also for smaller inline text such as a currency sign at the top of a price, and nested `sup` and `sub` are raised again from their parent, as in Chrome), `text-shadow` (several shadows, a true Gaussian blur, and underlines and line-throughs get the shadow too), `text-overflow` (`ellipsis`, `clip` or a string such as `" [more]"`, on a box that clips its overflow), `text-decoration-style` (`solid`, `double`, `dotted`, `dashed`, `wavy`, the last four drawn with Chrome's thickness, dash lengths and wave shape), `text-decoration-color`, `text-decoration-thickness`, `text-underline-offset`, `text-underline-position` (`under` below the descenders and `from-font`, as in Chrome), `text-decoration-skip-ink` (`auto`, the default, leaves gaps where descenders cross an underline, as in Chrome; `none` and `all`), `-webkit-text-stroke` (outlined text that stays selectable) with `paint-order`, `text-align-last`, `tab-size`, `font-variant: small-caps` (the font's own small capitals, or synthesized from capitals when it has none), `font-variant-numeric` and `font-feature-settings` (see OpenType features), `line-clamp` and `-webkit-line-clamp` (with an ellipsis, or the mark `block-ellipsis` sets, such as `"… read more"`, or `line-clamp: 3 "… more"`), `hyphens: manual`

Area	Properties
Backgrounds	`background-color`, `background-image` and `background` with `url()` images (PNG, JPEG, GIF, WebP, BMP, TIFF, and SVG files drawn as vectors; an SVG file without a `viewBox` stretches to `background-size`, as in Chrome) and `image-set()` (Chrome's print choice: the first option of 1x or more by resolution, drawn at its pixel size over that resolution), `linear-gradient()`, `radial-gradient()`, `conic-gradient()` (pie charts and colour wheels, drawn as smooth PDF shadings) and their `repeating-` forms (several layers allowed; colour interpolation methods such as `in oklch`, `in oklab` and `in hsl longer hue`, as Tailwind 4 writes them, also work); `background-repeat` (incl. `space`, `round`), `background-position` (incl. edge offsets like `right 10px bottom 5px`), `background-size` (`cover`, `contain`, lengths), `background-origin`, `background-clip` (incl. `text`, also as `-webkit-background-clip`), `background-blend-mode`, and `background-attachment: fixed`, which places an image against the paper, the same on every page, and shows it where the box is, as Chrome prints it (gradients move with their box). Inline elements such as `<span>`, `<mark>` and `<code>` get the same backgrounds, borders and outer and inset shadows as blocks, including images, gradients, `border-radius` and clipping to text; `box-decoration-break: slice`
Flexbox	`flex-direction`, `flex-wrap`, `flex-flow`, `flex`, `flex-grow` / `-shrink` / `-basis`, `order`, `justify-content`, `align-items`, `align-self`, `align-content`, `gap`, `margin: auto`. The first and last child's margins stay inside a flex item. A wrapping column flex box of fixed height starts a new column when one is full, `min-width: 0` lets a flex item shrink below its longest word, and an item taller than its fixed-height row (a Bootstrap progress bar) stays in place, as in Chrome. Images stretch like other flex items (`align-items: normal` or `stretch`), and an item stretched to its row is laid out at that height, so an absolutely positioned child with `bottom` sits at its bottom edge. Charts.css charts draw as in Chrome
Grid	the `grid` shorthand (template forms and `auto-flow`), `grid-auto-flow: dense` packing, `grid-template-columns` / `-rows` (lengths, `%`, `fr`, `auto`, `min-content`, `max-content`, `minmax()`, `fit-content()` capped at its length, `repeat()` incl. `auto-fill` / `auto-fit`), `grid-auto-rows` / `-columns`, `grid-auto-flow`, `grid-template-areas`, `grid-template`, named lines (`[name]`), `grid-column` / `-row` (line numbers, names and `span`), `grid-area` (incl. area names), `subgrid` in `grid-template-columns` / `-rows` (see Subgrid), `gap`, `justify-items` / `-self`, `align-items` / `-self`, `justify-content`, `align-content`, `place-*` (an image in a grid stretches with an explicit `stretch`)
Columns	`column-count`, `column-width`, `columns`, `column-gap`, `column-rule` (all border styles), `column-span: all`, `column-fill`, `break-before` / `-after: column`, `break-inside: avoid-column` (see Multi-column layout)

Area	Properties
Variables	custom properties (`--name: value`) and `var(--name, fallback)`, anywhere including shorthands; a declaration whose `var()` cannot be substituted unsets its property, as in Chrome. `if()` picks a value by `media()`, `supports()` or `style()` conditions, such as `width: if(media(width < 500px): 100%; else: 50%)`
Math functions	`calc()`, `min()`, `max()`, `clamp()`, `round()`, `mod()`, `rem()`, `abs()`, `sign()`, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `atan2()`, `pow()`, `sqrt()`, `hypot()`, `log()` and `exp()`, and `sibling-index()` and `sibling-count()` (an element's place among its siblings, for staggered colours and sizes), for lengths, numbers and angles, nested in any combination and mixing units (`calc(100% - 2em)`, `min(100%, 600px)`, `clamp(1rem, 2.5vw, 2rem)`), with `var()` inside them. Percentages resolve against the property's own reference (the containing block's width for `width` and margins, the parent's font size for `font-size`), so they work in widths, heights, margins, padding, flex bases, grid tracks, gaps, positions, translations, font sizes and borders
Positioning	`float` and `clear` (with text wrapping round float shapes; an empty clearfix `div` moves what follows below the floats and makes its container grow round them); `position` (`relative`, `absolute`, `fixed`; `sticky` is drawn as `relative`, as in Chrome's print output) with `top`, `right`, `bottom` and `left`, `inset`, and the logical `inset-inline` and `inset-block` with their `-start` / `-end` longhands (the inline side follows `dir` and `direction`); `z-index`. An absolutely positioned child of a flex or grid container takes no room in its rows or columns and is placed by its offsets, and a `position: relative` table cell is the containing block of its positioned content, so corner ribbons and badges on cards and cells land where Chrome puts them. An absolutely positioned box inside a `position: relative` inline element (a tooltip under a word, a badge on its corner) is placed against that inline element, and a `position: fixed` box is painted by its `z-index` among the page's content, both as in Chrome
Transparency	`opacity` (a block is painted as one PDF transparency group; an inline `<span>` fades its text and background), translucent colours everywhere (`rgba()`, `hsla()`, `#rrggbbaa`, `transparent`), gradients that fade to `transparent` (as a soft mask). Blurred box shadows, repeated gradient tiles with translucent stops and SVG gradients with `stop-opacity` are truly see-through, over images too. PDF/A-2 and 3 and PDF/UA allow all of this
Effects	`filter`: `grayscale()`, `sepia()`, `saturate()`, `hue-rotate()`, `invert()`, `opacity()`, `brightness()` and `contrast()` recolour the box's text, borders and backgrounds as vectors (images through their pixels), while `blur()` and `drop-shadow()` are applied to the rendered box at 144 dpi, and text under `blur()` stays selectable and searchable; `mix-blend-mode` (all separable and non-separable modes, as PDF blend modes) and `isolation`; masks and `backdrop-filter`

Area	Properties
Transforms	`transform` (`translate()`, `translateX()`/`Y()`, `scale()`, `scaleX()`/`Y()`, `rotate()`, `skew()`, `skewX()`/`Y()`, `matrix()`), `transform-origin`, and the individual `translate`, `rotate` and `scale` properties. Gradients, background tiles and SVG patterns inside a transformed box turn and scale with it. 3D transforms are drawn flat
Generated content	`::before`, `::after` and `::marker` with `content` (strings, `url()` images at their own size and gradients, as in Chrome, `attr()`, `counter()` (also `counter(list-item)`), `counters()`, `open-quote`/`close-quote` with the `quotes` property (`none`, `auto`, or string pairs; `auto` follows `lang` as in Chrome, with the CLDR quotes for German, French, Italian, Dutch, Swedish, Polish, Russian, Japanese and more), and for paged media `target-counter()`, `target-text()`, `leader()`, `string()` and `content()`), `counter-reset`, `counter-increment`, `counter-set`, `string-set`
SVG	`fill`, `stroke`, `stroke-width`, `fill-opacity`, `stroke-opacity`, `fill-rule`, `stroke-linecap`, `stroke-linejoin`, `stroke-miterlimit`, `stroke-dasharray`, `stroke-dashoffset`, `clip-path`, `clip-rule`, `mask`, `mask-type`, `filter`, `marker`, `marker-start`/`-mid`/`-end`, `flood-color`, `flood-opacity`, `color-interpolation-filters`, `text-anchor`, `dominant-baseline` (as CSS or presentation attributes)
Lists	`list-style-type` (every predefined style of CSS Counter Styles 3, such as `lower-greek`, `hebrew`, `japanese-formal`, `cjk-decimal`, `disclosure-open`, written as Chrome writes them, or a string such as `'– '`), `@counter-style` rules (every system and descriptor) and `symbols()`, also in `counter()` and `counters()`; `list-style-image` (raster images, SVG files and gradients, drawn as Chrome draws a gradient bullet; falling back to the type when the image does not load), `list-style-position` (`outside` and `inside`), `list-style`, `::marker` (content, colour, font, and its own `direction`; a larger marker font or a tall marker image makes its line taller, and `content: none` removes the marker, as in Chrome)
Tables	`border-collapse`, `border-spacing`, cell `vertical-align`, automatic column widths, `table-layout: fixed`, a table `height` shared out over the rows, `caption-side` (`top`, `bottom`), `empty-cells`, cell `min-width` and `max-width`, and backgrounds on `thead`, `tbody`, `tfoot`, `col` and `colgroup` (see Tables)
Paging	`page-break-before`/`after`/`inside`, `break-before`/`after`/`inside` (`always`, `page`, `avoid`), `orphans`, `widows`, `@page` size, margins, borders, padding and background (colours and images), `@page :first`/`:left`/`:right`, the sixteen page-margin boxes (`@top-center` and the like), named pages (`page`), `string-set`, footnotes (`float: footnote`, `@footnote`) (see Paged media), `bookmark-level`, `bookmark-label` and `bookmark-state` (see Links and bookmarks)

Units and colours Changed in 1.24.0

Units: `px`, `pt`, `pc`, `in`, `cm`, `mm`, `q`, `em`, `rem`, `%`, the font-relative `ex` (the font's x-height), `ch` (the advance of "0"), `cap` (cap height), `ic` (the advance of 水, or one em), `lh` (the line height) and their root versions `rex`, `rch`, `rcap`, `ric` and `rlh`, measured from the font, standard and embedded fonts alike; `vw`, `vh`, `vmin`, `vmax`, `vi`, `vb` and the small, large and dynamic sets (`svh`, `lvw`, `dvh`, `svmin` and the like), which all equal `vw` and `vh` on paper (the viewport is the page's content area, minus any running header and footer, in whole pixels as in Chrome); and the container query units `cqw`, `cqh`, `cqi`, `cqb`, `cqmin` and `cqmax`. Colours: names, `#rgb`, `#rrggbb`, `#rrggbbaa`, `rgb()`, `rgba()`, `hsl()`, `hsla()` (comma or space syntax), `hwb()`, `lab()`, `lch()`, `oklab()`, `oklch()`, `color()` (sRGB, Display P3 and the other predefined spaces, mapped to sRGB), `color-mix()`, `light-dark()` (the light value), `contrast-color()` (black or white, whichever contrasts more with the colour given), `currentColor`, and relative colours (`rgb(from var(--brand) r g b / 50%)`, `oklch(from teal calc(l + 0.1) c h)` and the like, in every colour function, with the channels usable in math functions).

Subgrid

`grid-template-columns: subgrid` and `grid-template-rows: subgrid` (CSS Grid Level 2) let a grid item that is itself a grid share its parent's tracks on that axis, so the rows of nested cards or the columns of a form line up across the whole grid. The subgrid's items take part in sizing the parent's tracks, gaps come from the parent unless the subgrid sets its own, and line names come from the parent and from the subgrid's own list (including `repeat()` and `repeat(auto-fill, …)`). Subgrids nest, and a subgrid that is not a grid item is laid out as `none`, as in browsers. Rows paginate as the parent grid's rows do.

```
.plans { display: grid; grid-template-columns: repeat(3, 1fr); gap: 16px }
.plan { display: grid; grid-row: span 3; grid-template-rows: subgrid }   /* title, features
and price line up */
```

Multi-column layout

`column-count`, `column-width` and `columns` lay content out the way Chrome does. Paragraphs, lists, tables and other boxes continue from one column into the next, every column on every page is filled, and the last row of columns is balanced (`column-fill: balance`) or filled in turn (`column-fill: auto`). `column-span: all`, `column-gap`, `column-rule`, `break-before`/`break-after: column`, `break-inside: avoid` and `avoid-column`, orphans and widows, repeated table headers and right-to-left columns work as in Chrome, which also ignores `break-before: page` inside columns.

Percentage heights

Percentage `height`, `min-height` and `max-height` resolve as in browsers when the containing block's height is definite: a length, a chain of percentages up to the page (`html, body { height: 100% }`), a stretched flex or grid item, a flex item grown in a container of definite height, an `aspect-ratio` height, or the padding box for absolutely positioned boxes. Children of a table cell resolve against it when the

cell has a fixed height or the table has a height. Against an `auto` height a percentage behaves as `auto`, as in browsers.

Container queries

`container-type` (`inline-size` or `size`), `container-name` and the `container` shorthand make an element a query container, and `@container` rules style its descendants by the container's size: `width`, `height`, `inline-size`, `block-size`, `aspect-ratio` and `orientation`, with `min-`/`max-` prefixes or range syntax, combined with `and`, `or` and `not`. A name picks a container (`@container sidebar (width < 300px)`), and `style(--compact: 1)` queries a custom property. The `cq` units are percentages of the nearest container's size. One component can then lay itself out for a narrow sidebar and a wide main column without knowing where it is placed.

```
.card-slot { container: card / inline-size }
.card { display: grid; gap: 8px }
@container card (width >= 400px) { .card { grid-template-columns: 120px 1fr } }
.card h3 { font-size: clamp(14px, 5cqi, 22px) }
```

A document that uses container queries is laid out again until the container sizes settle, which takes about two to four times as long; documents without them are laid out as before.

Tables Changed in 1.24.0

`display: table`, `inline-table`, `table-row`, `table-cell` and the other table values work on any element, with the anonymous table, row and cell boxes CSS adds around them, so a lone `display: table-cell`, a table `div` with only text and an `inline-table` lay out as in Chrome; `border-spacing` starts at 0 except on `<table>`. Column widths are shared out as Chrome does: columns under a cell that spans several are sized by their own content, a percentage width on a spanning cell is shared by the columns it spans, and percentages that add up to more than 100% are cut down from the last column that has one. A table is never narrower than its columns' narrowest content, so a long word or URL widens it instead of running into the next cell, and never narrower than its caption's longest word. Percentage columns that would squeeze the other columns below their narrowest content give way in proportion, so the table keeps its width, and an image with a percentage width in a cell shrinks with the column instead of widening the table. Percentage columns that add up to less than 100% of a table with a width share out the rest in proportion, and a table of `auto` width with percentage columns is as wide as the percentages need, as in Chrome. Text with `word-break: break-all` or `overflow-wrap: anywhere` lets its cell shrink, and a CSS table without rows keeps the width it is given.

With `border-collapse: collapse`, each shared border is drawn once and conflicts are resolved as in CSS 2.1 and Chrome: `hidden` wins, then the wider border, then the style (double, solid, dashed, dotted, ridge, outset, groove, inset), then cell before row, row group and table. Borders on `tr`, `thead`, `tbody` and `tfoot` take part, and cells sit inside the border that wins on each side. Backgrounds on `thead`, `tbody`, `tfoot`, `col` and `colgroup` are painted in the CSS 2.1 order, under the cells' own. `caption-side: bottom`, `empty-cells: hide` and cell `min-width` and `max-width` work too; a `max-width` caps its column even below the text's narrowest width, so a `nowrap` cell with `text-overflow: ellipsis` is cut at that width. A `position: relative` cell holds its absolutely positioned content,

which moves with `vertical-align` and is clipped with the cell, and outlines on cells of a collapsed table are drawn above the table. Row, row group and column backgrounds no longer cover the collapsed borders above them, and a collapsed table that breaks across pages is closed on every page: under the repeated footer, which gets the table's bottom border, or under the last row, as in Chrome.

Borders on `col` and `colgroup` take part in a collapsed table's conflicts, and `visibility: collapse` on a row, row group, column or column group removes it from the table, as in Chrome. `border-spacing` is rounded to whole pixels, as Chrome rounds it. When a table breaks across pages, its header and footer repeat only while they take up no more than a quarter of the page, a table whose header and first row do not fit on what is left of the page moves to the next one whole, and a cell that spans several rows breaks with them instead of running past the foot of the page.

display: contents

`display: contents` makes no box: the element's children, `::before` and `::after` included, are laid out as children of its parent, so they become flex items, grid cells, or rows and cells of the table around it, such as a `<ul>` whose items join a grid or a `div` that wraps table rows. The element's own border, padding, margin, background, transforms and position do not apply, but inherited properties such as colour and font still reach its children, and counters still count. On replaced elements and form controls (`img`, `svg`, `input`, `select`, `progress` and the like) it hides the element, as CSS Display asks, and on the root element it acts as `block`.

Text wrapping and justification

`text-wrap: balance` evens out the lines of a heading or short paragraph (up to six lines, as in Chrome), and `text-wrap: pretty` rewraps a paragraph's last four lines so it does not end on a single short word. `text-wrap-mode` and `text-wrap-style` are the longhands; `stable` lays out as `auto`. Neither applies beside floats, with `::first-line`, `word-break: break-all` or `line-break: anywhere`, or on pages of different widths.

```
h1, h2 { text-wrap: balance }
p { text-wrap: pretty }
```

With `text-align: justify`, `text-justify: auto` spreads the free space at spaces and, as Chrome does, between Chinese and Japanese characters and CJK punctuation (Korean Hangul only at spaces). `inter-word` uses only spaces, `inter-character` (or `distribute`) every character, as Firefox does, and `none` leaves lines unjustified. No-break spaces stretch like spaces. Right-to-left and bidirectional lines are justified at spaces only.

`white-space-collapse` (`collapse`, `preserve`, `preserve-breaks`, `break-spaces`) sets how spaces and newlines collapse on its own, as newer style sheets write it instead of `white-space`. `hyphenate-limit-lines: 2` lets at most two lines in a row end in a hyphen; the line that would be the third breaks at a space instead. Chrome ignores it, so the specification is the reference.

Vertical text

`writing-mode: vertical-rl`, `vertical-lr`, `sideways-rl` and `sideways-lr` on blocks, inline-blocks, flex items and table cells turn the box's text a quarter turn, as Chrome draws Latin text in those modes (in `vertical-lr` the lines run from left to right, as in Chrome; floats and boxes with a size inside vertical text get their size and side as Chrome gives them): column headers in narrow table cells, labels along the side of a card, and spine text. See Limitations for what is not done yet, such as upright CJK.

```
<th style="writing-mode: sideways-lr">Quarter 1</th>   <!-- reads from bottom to top -->
```

Float shapes

`shape-outside` makes the text beside a float wrap round a shape instead of the float's rectangle, as in Chrome: `circle()`, `ellipse()`, `inset()` (with `round` corners), `polygon()` and `path()`, optionally with a reference box (`margin-box`, the default, `border-box`, `padding-box`, `content-box`); a box keyword on its own follows the float's `border-radius`. Images (PNG, JPEG, GIF, WebP) and gradients work too: pixels more opaque than `shape-image-threshold` (0 to 1, default 0) count as the shape, so text follows the outline of a cut-out photo. `shape-margin` grows the shape outwards. Lines beside a shape step down a pixel at a time until they fit, as in Chrome.

```
.portrait { float: left; width: 160px; shape-outside: circle(50%); shape-margin: 12px; border-
radius: 50% }
.logo { float: right; shape-outside: url(logo.png); shape-image-threshold: 0.3 }
```

Drop caps

`initial-letter` on `::first-letter` makes a drop cap, as in Chrome. `initial-letter: 3` (or `3 drop`) is three lines tall and sits on the third line's baseline, `3 raise` stands on the first line, and `3 2` sinks two lines; sizes such as `2.5` work too. The letter's font size is chosen so its cap height spans the lines, the lines beside it flow round it like a float, and padding, borders, backgrounds and colours on the letter apply.

```
article > p:first-of-type::first-letter { initial-letter: 3; color: #8b1e3f; margin-right: 6px
}
```

Masks and backdrop-filter

`mask-image`, `mask-size`, `mask-position`, `mask-repeat`, `mask-origin`, `mask-clip`, `mask-mode`, `mask-composite` and the `mask` shorthand work on HTML elements, and so do the `-webkit-mask` spellings. Mask layers take the same images as backgrounds (gradients, PNG, JPEG, WebP, GIF, SVG files and `data:` URIs) and are sized and tiled the same way; `url(#id)` points at an SVG `<mask>` in the document. A mask is a PDF soft mask, so masked text stays text: selectable, searchable and tagged. If no mask layer loads, the element is drawn unmasked, as in Chrome.

`backdrop-filter` (and `-webkit-backdrop-filter`) takes every filter function. What is behind the box is rendered at 144 dpi, filtered and drawn under the box, clipped to its border box and rounded corners; the box's own content stays vector. Chrome's own print to PDF leaves out `backdrop-filter` and `mask-composite`; PdfPecker draws them as Chrome shows them on screen.

```
.fade { mask-image: linear-gradient(to bottom, black 60%, transparent) }
.glass { background: rgb(255 255 255 / 40%); backdrop-filter: blur(8px) saturate(1.4) }
```

3D transforms

`translate3d()`, `translateZ()`, `scale3d()`, `scaleZ()`, `rotateX()`, `rotateY()`, `rotate3d()`, `matrix3d()`, `perspective()`, the `perspective` and `perspective-origin` properties, and the `translate`, `scale` and `rotate` properties with a z value or an axis are accepted. A PDF page has only flat (affine) transforms, so a 3D transform is drawn flat: `rotateY(60deg)` makes a box half as wide, and with a perspective the box gets the scale and shear the projection has at its centre, so it stays a rectangle or parallelogram where Chrome draws a trapezoid. A box turned edge-on or away from the viewer is not drawn. `transform-style` and `backface-visibility` are accepted and, as in Chrome's print output, both faces of a flip card are drawn.

Trimming text boxes

`text-box: trim-both cap alphabetic` (or `text-box-trim` and `text-box-edge`) trims a block's first line to the font's cap height and its last line to the baseline, as Chrome does, so the text of a button, badge or tag sits optically centred in its padding. The text keeps its place on the baseline and reaches past the trimmed box. Trimming passes into a box's first and last child blocks unless their own border or padding comes between; inline-blocks, flex and grid items and table cells trim their own lines. The edges are `text` (the font's ascent and descent), `cap`, `ex` and `alphabetic`.

```
.badge { text-box: trim-both cap alphabetic; padding: 4px 8px; }
```

Emphasis marks

`text-emphasis` (and `-style`, `-color`, `-position`, also with `-webkit-`) puts a mark over or under each character, as East Asian text uses for emphasis: `dot`, `circle`, `double-circle`, `triangle`, `sesame`, filled or open, or a string. The marks are drawn in the text's font at half its size, characters such as spaces and most punctuation get none, and lines move apart to make room for them, all as Chrome draws them.

Corner shapes

`corner-shape` gives `border-radius` corners other shapes: `round` (the default), `squircle`, `bevel`, `scoop`, `notch`, `square` and `superellipse(<number>)`, also per corner, per side and with the logical longhands. The shape applies to backgrounds, borders of every style, box shadows, outlines, overflow clips, images and `clip-path` reference boxes, with the inner edge Chrome 153 draws.

```
.card { border-radius: 24px; corner-shape: squircle; }
.ticket { border-radius: 12px; corner-shape: scoop; }
```

Scoped styles

Rules in `@scope (.card) to (.card .content) { ... }` apply below each `.card` and above its limits, as in Chrome, so component styles do not leak into the content nested inside. `:scope` and `&` stand for the root, a selector without them matches below it, and between rules of equal specificity the one with the closest root wins. `@scope` without a start inside a `<style>` element is rooted at the style element's parent.

```
@scope (.card) to (.card-body) {
  :scope { border: 1px solid #ccc }
  h3 { margin: 0 }          /* the card's own heading, not those in its body */
}
```

Typed attr()

`attr()` with a type works in any property, as in Chrome 133 and later, so bar charts, colours and badges can come from data attributes alone:

```
<style>
  .bar { width: attr(data-share type(<percentage>), 0%); background: attr(data-color
type(<color>), gray) }
</style>
<div class="bar" data-share="64%" data-color="#2a9d8f"></div>
```

`type()` takes any `@property` syntax, and `raw-string`, `number` and units such as `attr(data-gap px)` work too. A failed substitution leaves the property unset, as in Chrome. A value from an attribute can never become a `url()`, `image-set()` or any other function that loads a resource, so data from users cannot make the PDF fetch a file.

Zoom

`zoom: 0.8` draws an element and its content at 80%, as Chrome computes it: every absolute length on it and its descendants (px, pt, mm, viewport units) is multiplied by the zoom, nested zooms multiply, percentages stay, and images show their own size zoomed. Templates that Chrome users shrink with `zoom` come out the same size.

Gap decorations

`column-rule` now also draws lines between the columns of grid and flex containers, and `row-rule` between their rows, as Chromium draws CSS Gap Decorations: table-like lines without borders on every cell.

```
.grid { display: grid; grid-template-columns: repeat(3, 1fr); gap: 8px;
        column-rule: 1px solid #ccc; row-rule: 1px dashed #ccc; rule-break: intersection }
```

Each rule property takes a list, with `repeat()`, for different lines in different gaps; `rule-break` (`none`, `normal`, `intersection`) sets where lines stop at spanning items and crossings, and `rule-inset` with `rule-overlap` moves their ends. Rules are cut where a container breaks across pages, and a gap that falls on a page break draws nothing.

Check boxes and radio buttons

`accent-color` paints checked check boxes and radio buttons as Chrome does, with a dark mark when white would not stand out from the accent. `appearance: none` takes the native look away: the control is drawn only by the page's CSS (`:checked`, `::before` and `::after` included), for custom check boxes and toggle switches, and it is not made a form field, because a PDF field cannot know what the page's `:checked` style looks like.

Scale content to fit a box

`-pdfpecker-scale-to-fit: shrink` on a box with a `height` or `max-height` draws its content smaller until it fits the box, for labels, name badges and certificates that have a fixed space for text of any length. The text reflows wider into fewer lines before it gets smaller, and the box's own border and background keep their size. Content that already fits is not changed, and table cells with a height scale their content too. A scaled box is drawn whole, so like a box with `break-inside: avoid` it moves to the next page rather than across a page break. Scaled boxes nest at most two levels deep.

```
<div style="width: 80mm; height: 25mm; -pdfpecker-scale-to-fit: shrink">Dr. Maximiliane
Alexandra von Rosenberg-Hohenstein</div>
```

Pagination

Lines and images are never cut in half at a page boundary; table header rows repeat at the top and `<tfoot>` rows at the bottom of every page a table spans, as in Chrome; backgrounds and borders continue across pages. A table row that does not fit at the bottom of a page breaks inside, cell by cell, as Chrome breaks it, with the header and footer repeated around it; it moves to the next page whole only when a cell would have nothing before the break (its first lines with `orphans`, an image or a drawing), when it has `break-inside: avoid`, or when it has cells spanning rows. `break-before` and `break-after` work on table rows and row groups, `break-inside: avoid` on a `tbody` keeps a group together, and `break-after: avoid` on a group's header row keeps it with the next row, as in Chrome. A flex row that crosses the end of a page breaks inside its items, as in Chrome, and moves to the next page whole only when one of its items would move whole anyway (an image, or a box with `break-inside: avoid`); `break-before` or `break-after: page` on a flex item starts a new page between flex lines. A forced break before a box's first child breaks before the box itself, as CSS Fragmentation says and Chrome does, so a section whose first heading has `break-before: page` starts on the new page with its border and background, and a table of contents points at it. A grid row breaks across the page

boundary where it starts, as in Chrome, and moves whole only when one of its items cannot break (a line of text, an image, or a box with `break-inside: avoid`). A table inside a flex or grid item breaks across pages like any other table, with its header repeated, and the line or row grows to match. A paragraph that splits leaves at least `orphans` lines (default 2) at the bottom of a page and carries at least `widows` lines (default 2) to the next. Headings keep with the content that follows them (`break-after: avoid` in the default styles), and `break-before`/`break-after: avoid` does the same for any block. `break-inside: avoid` keeps a block or a whole table on one page when it fits; a kept box that is taller than a page starts on the next page and breaks there, as in Chrome. As in Chrome, a margin that runs past the end of a page is dropped or cut at the break rather than carried to the next one (except after a forced break), so a tall bottom margin makes no blank page, and a box's top and bottom borders and padding stay on the same page as its first and last content. A box shadow paints on the pages its box reaches but never adds a page of its own.

`box-decoration-break: clone` closes each page's part of a block box that splits across pages, as Chrome prints it: every part gets the box's top and bottom border and padding, its own corners, shadow and background, so a framed section or card looks whole on each page. `continue: discard` keeps a box to the page it starts on (or to its `height` or `max-height`) and leaves out the content that does not fit. `-pdfpecker-repeat: each-page`, PdfPecker's own property, draws a box again at the top of each later page its parent continues on, such as a term's label beside a definition several pages long.

Links and bookmarks

`<a href>` becomes a clickable link, whether inline or a block (a card wrapped in a link), and stays clickable on both pages when a page break splits it. External URLs open in the viewer's browser and `#id` links jump to the element. Headings `h1`-`h6` become the PDF's bookmarks, nested by level; set `PdfOptions.Bookmarks = false` to leave them out. The CSS properties from Generated Content for Paged Media control them: `bookmark-level` makes any block a bookmark at that level, or leaves a heading out with `none`; `bookmark-label` sets its text from strings, `attr()`, counters and `content()`; and `bookmark-state: closed` shows it collapsed.

```
.chapter-title { bookmark-level: 1; bookmark-label: "Chapter " counter(chapter) ": " content()
}
.appendix h2 { bookmark-state: closed }
.no-bookmark { bookmark-level: none }
```

SVG

Inline `<svg>` and SVG files (in `<img>` and as `url()` backgrounds, including `data:image/svg+xml` URIs) with `path` (all commands, arcs included), `rect` (incl. rounded), `circle`, `ellipse`, `line`, `polyline`, `polygon`, `text` and `tspan` (`x`, `y`, `dx`, `dy`, `text-anchor`, `dominant-baseline`, `direction`, registered fonts and right-to-left text), `textPath` (text along a path, with `startOffset` and the path's `pathLength`, as in Chrome), `g`, `use` (also of a `symbol`, gradient or clip path defined in another inline SVG on the page, with the symbol's `viewBox` fitted into the `use` size), nested `svg`, `image` (raster images and SVG files, with `preserveAspectRatio`; inside an SVG file only `data:` URIs load, as in browsers), `viewBox`, `preserveAspectRatio` and `transform`, drawn as vector paths. Percentages in coordinates and sizes resolve against the viewport, as in Bootstrap's `width="100%"`

placeholder images. Icons drawn with `<use href="#id">` from a sprite of symbols that is itself hidden (`<svg class="d-none">`, as Bootstrap's icon sprites are) show as in Chrome. Strokes can be dashed with `stroke-dasharray` and `stroke-dashoffset`, scaled by `pathLength`. `opacity` on a `g`, `use` or nested `svg` fades the group as one, so overlapping children do not show through each other, and `text-decoration` on `text` and `tspan` draws underlines, overlines and line-throughs in the text's fill. `fill="context-fill"` and `stroke="context-stroke"` paint with the `use` element that draws the shape, or the shape a marker sits on, as in Chrome. A `foreignObject` lays out its HTML content, as wide and tall as the element, through the SVG's transforms and `viewBox`. `linearGradient` and `radialGradient` (both units, `gradientTransform`, `spreadMethod`, `href` inheritance, and `stop-opacity` that is truly transparent) become PDF shadings, and `pattern` fills and strokes (both units, `viewBox`, `patternTransform`, `href` inheritance) become PDF tiling patterns. `clipPath` clips the elements that reference it, and `mask` (luminance or alpha, both units) becomes a PDF soft mask. `marker` draws arrowheads and dots at the vertices of paths, lines, polylines and polygons (`orient` incl. `auto` and `auto-start-reverse`, `markerUnits`, `viewBox`, `refX`/`refY`, clipped to the marker unless `overflow="visible"`). `filter` is applied in pixels, at 144 dpi, and drawn as an image, with every filter primitive: `feGaussianBlur`, `feOffset`, `feDropShadow`, `feFlood`, `feMerge`, `feComposite` (all operators), `feBlend` (separable modes), `feColorMatrix`, `feComponentTransfer`, `feMorphology`, `feTurbulence`, `feImage`, `feTile`, `feConvolveMatrix`, `feDisplacementMap`, and `feDiffuseLighting` and `feSpecularLighting` with distant, point and spot lights and `lighting-color`, in linearRGB or sRGB. Primitive subregions (`x`, `y`, `width`, `height`) work as in Chrome. Text under a filter stays selectable and searchable: an invisible copy of it is kept on top of the image. As in browsers, an SVG file is styled only by its own `<style>` elements and attributes, not by the page's CSS, and loads nothing else.

Shapes drawn by a `use` inherit its `fill`, `stroke` and other paint, as in Chrome, so sprite icons take the colour of their `use` and MathJax formulas follow the text colour. The geometry properties `cx`, `cy`, `r`, `rx`, `ry`, `x`, `y`, `width`, `height` and `d: path(...)`, and `stop-color` and `stop-opacity` on gradient stops, come from style sheets too, winning over the attributes as in Chrome. `vector-effect: non-scaling-stroke` keeps a stroke at the width the page gives it however the drawing is scaled, so hairline gridlines survive a chart scaled to fit. CSS `transform` works on SVG elements, turning about `transform-origin` within `transform-box` (`view-box`, `fill-box`, `stroke-box`). The `viewBox` scale is written exactly, so formulas from MathJax, drawn at about 1,000 units per em, keep their last glyph.

Images

JPEG (embedded as-is, RGB, grayscale and CMYK), PNG (all colour types and bit depths, transparency, interlacing), GIF (transparency, interlacing), WebP (lossy, lossless and with alpha), BMP (1 to 32 bits, palettes, RLE, bit fields and alpha) and TIFF (uncompressed, LZW, Deflate, PackBits, CCITT G3 and G4, JPEG; strips or tiles, planar, alpha, 16-bit; bilevel images stay 1 bit per pixel), from files, `data:` URIs or your `ResourceResolver`. Animated GIF and WebP files show their first frame, as a PDF page is still. SVG files are drawn as vectors (see SVG above). Opaque PNGs are passed through without re-encoding. A JPEG photo stored on its side is turned upright by its EXIF orientation, as browsers do (`image-orientation: from-image`, the default; `none` draws it as stored). Images are placed on whole CSS pixels, like their borders, so a bordered image meets its border evenly on every side.

An `<img>` with `srcset` and `sizes`, and a `<picture>` with `<source>` elements, show the picture Chrome chooses when it prints: the first source whose `media` and `type` match (with screen media, as Chrome chooses as the page loads, and skipping types PdfPecker cannot decode, such as AVIF), and from the set the candidate nearest to 1x, with `w` descriptors resolved through `sizes` against the page area. A 2x picture is drawn at half its pixel size. An `<embed>` or `<object>` that points at a picture shows it as an `<img>` would, and an `<object>`'s fallback content is left out; one that is no picture or does not load stays empty or shows its fallback. An image wider than its box overflows it, as in Chrome; an image wider than the page makes the whole page shrink to fit, as Chrome prints it, and is scaled down on its own only past 1.5 times the page width or with `ShrinkToFit` off. A percentage `width`, `max-width` or padding on an image is a share of the containing block, inline images draw their border, padding, background and shadow, and vertical margins on inline images and SVG take room in the line, all as in Chrome.

`image-rendering: pixelated` or `crisp-edges` keeps an enlarged small image, such as a QR code or pixel art, sharp in every viewer: its pixels are written as blocks, where some viewers would smooth them. `-pdfpecker-image-quality: 80` writes images as JPEGs at that quality, and `-pdfpecker-image-max-resolution: 150dpi` downsamples images drawn denser than that, judged by the size each is drawn at; both are inherited, so a document sets them once on its root. A smaller version replaces an image only when it is smaller, and transparency stays lossless.

An `<img>` whose picture is missing (no `src`, a file or `ResourceResolver` that finds nothing, or data that cannot be decoded) prints its `alt` text instead, as Chrome does, in the text style around it plus the image's own border, padding, colour and font; this works for Markdown's `![alt](missing.png)` too. Without alt text, or with `alt=""`, a missing image keeps an empty box of the width and height given.

Paged media

> **Changed in 1.24.0:** Page rules and margin boxes.

Page rules and margin boxes Changed in 1.24.0

`@page` rules set the page size, margins and background, and their margin boxes (`@top-left`, `@top-center`, `@bottom-right` and the other thirteen) print headers and footers with `counter(page)` and `counter(pages)`, chosen per page with `:first`, `:left` and `:right`. Those three can also change the page's size, margins and background: `@page :first { margin-top: 70mm }` leaves room for a letterhead on page 1 only, and mirrored `:left` and `:right` margins give a bound booklet a wider inner margin, and each page is laid out at its own content width. The first page is a right page; in a right-to-left document (`dir="rtl"` on `<html>`) it is a left page, and PDF viewers are told the pages run right to left. A border and padding on the `@page` box are drawn inside the margins, with the content inside them, and the page's background images cover the whole sheet, as Chrome prints them (the images are positioned in the page box, as CSS Paged Media 3 says). On top of that:

- **Named pages**: `page: wide` puts an element on pages styled by `@page wide { ... }`, with their own size, orientation, margins, background and margin boxes. Content on another named page starts a new page, so a landscape table or a full-bleed cover can sit between portrait pages.
- **Running headers**: `string-set` copies text from the document (`content()`, `attr()`, strings and counters) into a named string, and `string(name)` shows it in a margin box: the chapter title in the page header, say. As in the spec, `string(name, first | start | last | first-except)` picks which value a page shows.
- **Running elements**: `position: running(logo)` takes an element out of the flow, and `content: element(logo)` in a margin box lays it out there on each page, with the styles it has in the document, as Prince, WeasyPrint and Paged.js do: a logo, a small table or a formatted title in the header. `element(name, first | start | last | first-except)` picks which one a page shows, and page counters in its `::before` and `::after` count the page that shows it.
- **Pages as tall as their content**: `@page { size: 80mm auto }` gives each page the width and as much height as its content needs, for receipts and labels on a roll. Content longer than 200 inches, the most PDF readers accept, continues on pages of that height.
- **Tables of contents**: `target-counter(attr(href), page)` shows the page number of the element a link points to, and `leader(dotted)` (or `solid`, `space`, or a string) fills the space before it. `target-text()` shows the linked element's text. `target-counters(attr(href), section, ".")` writes every level of a nested counter, for cross references such as "see 2.3.1". The document is laid out again with the numbers it finds, so they are always right.
- **Page count in the content**: `counter(pages)` works anywhere, not only in margin boxes, as in Prince: "This report has 12 pages" on its first page.
- **Sections**: `-pdfpecker-target-pages(attr(href), first | last | count | current)` gives the first or last page of the element a link points to, how many pages it covers, or the page

counted from its first (with an optional counter style, as `target-counter()` takes): "Chapter 3, pages 12–17" in a table of contents, or "page 2 of 6" within a section. In a sticky header or footer, `{section-page:id}` (the id percent-encoded) is the page within that section.

- **Every nth page and page groups**: `@page :nth(2n+1)` matches pages by their number, and `@page :nth(1 of chapter)` the first page of each run of `chapter` pages, as in WeasyPrint and CSS GCPM. They style the margin boxes and the page background, not the page size or margins.
- **Bleed and printer's marks**: `@page { bleed: 3mm; marks: crop cross }` makes the sheet larger by the bleed on every side, so backgrounds that reach the page edge run on past where it is cut, and draws crop marks and registration crosses in the bleed. The paper size becomes the PDF's trim box and the larger sheet its bleed box, as print shops expect and as WeasyPrint writes them. Marks without a `bleed` get one of 6pt, as the spec says.

```
@page { size: A4; margin: 20mm;
  @top-center { content: string(chapter) }
  @bottom-right { content: "Page " counter(page) " of " counter(pages) } }
@page cover { margin: 0; background: #1d3557; @top-center { content: none } @bottom-right {
content: none } }
@page wide { size: A4 landscape }
.cover { page: cover }
.wide-table { page: wide }
h2 { string-set: chapter content(); break-before: page }
nav a::after { content: leader(dotted) target-counter(attr(href), page) }
```

Chrome supports named pages and margin boxes, but not running strings or elements, `target-counter()`, `target-counters()`, `leader()`, the page count in the content or section page numbers; `-pdfpecker-target-pages()` and `size: <width> auto` are PdfPecker's own.

Footnotes

`float: footnote` moves an element to the footnote area at the bottom of the page its call is on, as in CSS Generated Content for Paged Media. The `footnote` counter numbers them; `::footnote-call` styles the number in the text (a superscript by default) and `::footnote-marker` the number before the note. `@page { counter-reset: footnote }` restarts the numbering on every page, and `@footnote` inside `@page` styles the area with margins, padding, borders, a background and `max-height`. `footnote-display: inline` runs short notes together on one line, and `footnote-policy: line` moves the line with the call to the next page when its note does not fit. A long note continues on the next page. Each call links to its note, and tagged PDFs mark the notes as `Note` (PDF/UA-1) or `FENote` (PDF/UA-2).

```
.note { float: footnote; font-size: 8pt }
.note::footnote-call { content: "[" counter(footnote) "]"; vertical-align: baseline }
@page { @footnote { border-top: 0.5pt solid #999; padding-top: 4pt; margin-top: 8pt } }
```

```
<p>Paid on the last banking day<span class="note">Or the day before a holiday.</span>.</p>
```

Running header and footer from HTML

An element with the `data-sticky-pdf-header` or `data-sticky-pdf-footer` attribute is taken out of the flow and drawn at the top or bottom of every page, and the page content flows in the space between. Inside it, `{page}` and `{total-pages}` (or `{pages}`) are replaced with the page number and page count. An optional `height` attribute (`height="17mm"`) reserves at least that much space.

Each of the tokens, `{section-page:id}` included, takes a counter style after a colon: `{page:upper-roman}` for a preface numbered i, ii, iii, `{page:lower-alpha}`, or a style from the document's own `@counter-style` rules, such as one that pads numbers to three digits. An unknown style name gives decimal numbers, as in CSS.

```
<header data-sticky-pdf-header height="17mm">ACME Ltd · Payslip</header>
<main>...</main>
<footer data-sticky-pdf-footer height="12mm">Page {page} of {total-pages}</footer>
```

An element with `data-sticky-pdf-first-header` or `data-sticky-pdf-first-footer` takes the place of the running header or footer on the first page only: a letterhead on page one and a slim header after it, or, left empty, no header or footer on page one at all. The content on page one uses the space its own bands leave.

```
<header data-sticky-pdf-first-header height="40mm"><img src="letterhead.png" alt="ACME
Ltd"></header>
<header data-sticky-pdf-header height="12mm">ACME Ltd · Offer 2041</header>
<footer data-sticky-pdf-first-footer></footer>
<footer data-sticky-pdf-footer>Page {page} of {total-pages}</footer>
```

A band with `data-sticky-pdf-page="name"` is the header or footer of the pages of that named page instead of the document's bands, and a first-page band with it shows on the first page of each run of that page. Pages without bands of their own keep the document's. One HTML document can so hold a cover, a portrait letter and a landscape appendix, each with its own size, margins, header and footer, while page numbers run on through the whole document.

```
<style>
  @page appendix { size: A4 landscape }
  .appendix { page: appendix }
</style>
<footer data-sticky-pdf-footer>Page {page} of {total-pages}</footer>
<footer data-sticky-pdf-footer data-sticky-pdf-page="appendix">Appendix · page {page}</footer>
```

Elements with `data-sticky-pdf-even-header` and `data-sticky-pdf-even-footer` take the place of the running bands on the even pages (2, 4, ...), for documents printed on both sides. On a named page the page's own even band wins, then its running band, then the document's even band.

A `position: fixed` box, such as a watermark or a page frame, paints over the running header and footer when its `z-index` is `auto` or 0 or more, as over a header Chrome prints in the flow; with a negative `z-index` it stays under them.

Shrink to fit

When content sticks out past the right edge of the page area, Chrome prints the page laid out as much wider as the content reaches and drawn smaller, and PdfPecker does the same. A box or image wider than the page, a long word or `white-space: nowrap` text, a Bootstrap `.row` without a `.container` (its negative margin), a relative offset, a `translateX()` or an absolutely positioned box can each widen the layout, by at most 1.5 times (drawn at 67%), on top of `Scale`. The paper keeps its size and the margins keep theirs. As in Chrome, overflow to the left, content inside an `overflow: hidden` or `clip` box (other than `html` and `body`) and `position: fixed` boxes do not count.

Set `ShrinkToFit = false` to lay the document out at the page width as before 1.18.0, with the overflow running into the margin; an image wider than the page is then scaled down on its own.

```
var pdf = HtmlToPdf.Convert(html, new PdfOptions { ShrinkToFit = false });
```

Markdown

Markdown goes to PDF through the same engine as HTML: it is turned into HTML with a GitHub-like stylesheet first, so everything on HTML and CSS support applies to it. You can also stop at the HTML.

```
// Markdown to PDF, styled like a GitHub README
byte[] pdf = MarkdownToPdf.Convert(File.ReadAllText("README.md"));

// File to file; relative image paths resolve against the Markdown file's folder
MarkdownToPdf.ConvertFile("README.md", "README.pdf", new PdfOptions { FooterText = "Page
{page} of {pages}" });

// Markdown to HTML: a body fragment, or a complete page with the stylesheet
string fragment = MarkdownToHtml.Convert("# Hello\n\n*World*");     // <h1
id="hello">Hello</h1>\n<p><em>World</em></p>
string page = MarkdownToHtml.ConvertDocument(markdown);
MarkdownToHtml.ConvertFile("README.md", "README.html");
```

`MarkdownOptions` switches the GitHub extensions off for strict CommonMark (`GitHubExtensions = false`), drops heading ids (`HeadingIds = false`), shows raw HTML as text (`AllowRawHtml = false`), replaces the stylesheet (`StyleSheet`, see `MarkdownToHtml.DefaultStyleSheet`) or sets the title (default: the front matter's `title`, else the first heading). `MarkdownToPdf` takes `PdfOptions` and `MarkdownOptions` side by side.

What is supported

CommonMark 0.31 (651 of the 652 spec examples pass; the one miss is an exotic HTML5 entity name) plus the GitHub Flavored Markdown extensions: tables with column alignment, `~~strikethrough~~`, task list items (`- [x]`) and bare `www.`/`https://`/e-mail autolinks, along with footnotes, definition lists and YAML front matter (see below). Headings get GitHub-style ids, so `[see below](#heading-text)` links work in the PDF too. The parser runs in linear time on hostile input (deep nesting, unclosed brackets, long delimiter runs). See Limitations for what is not supported.

Footnotes, definition lists and front matter

Footnotes use GitHub's syntax and are numbered in the order they are first referenced. `MarkdownToPdf` puts each note at the bottom of the page that references it, with the CSS `float: footnote` engine, so this also works with your own `StyleSheet`; `MarkdownToHtml` lists the notes at the end with links back to the text, as GitHub does.

```
PdfPecker has no dependencies.[^bcl]

[^bcl]: Only the .NET base class library.
```

A definition list is a term on its own line followed by one or more lines starting with `: `, as in PHP Markdown Extra and markdown-it. Front matter between `---` lines at the very start of the file is not

shown; its `title` becomes the document's title (the PDF title, and `<title>` in HTML) unless `MarkdownOptions.Title` is set.

```
---
title: Release notes
---

Subset
: Only the glyphs a document uses are embedded.
```

All three are GitHub extensions, so `GitHubExtensions = false` turns them off along with tables and strikethrough.

Fluent C# API

> **Changed in 1.24.0:** What it covers.

PdfPecker.Fluent builds PDFs from C# code instead of HTML, in the style of QuestPDF. You describe pages, headers, footers, columns, rows, tables, text, images and barcodes in code; PdfPecker.Fluent writes compact HTML and CSS for them and renders it with PdfPecker. It needs no browser, no native binaries and no package other than PdfPecker.

```
dotnet add package PdfPecker.Fluent
```

It is released together with PdfPecker, always with the same version, and depends on the PdfPecker release of that version. It runs on .NET 8, 9 and 10, and is trimmable and Native AOT compatible.

```
using PdfPecker.Fluent;

Document.Create(container =>
{
    container.Page(page =>
    {
        page.Size(PageSizes.A4);
        page.Margin(2, Unit.Centimetre);
        page.DefaultTextStyle(x => x.FontSize(11));

        page.Header().Text("Hello
PdfPecker").FontSize(20).Bold().FontColor(Colors.Blue.Darken2);

        page.Content().PaddingVertical(1, Unit.Centimetre).Column(column =>
        {
            column.Spacing(10);
            column.Item().Text("A paragraph of text.");
            column.Item().Image(File.ReadAllBytes("logo.png")).FitWidth();
        });

        page.Footer().AlignCenter().Text(x =>
        {
            x.Span("Page ");
            x.CurrentPageNumber();
            x.Span(" of ");
            x.TotalPages();
        });
    });
})
.GeneratePdf("hello.pdf");
```

What it covers Changed in 1.24.0

- **Layout**: `Column`, `Row`, `Table` (row and column spans, header and footer rows repeated on every page, cell backgrounds and borders that fill the row in any call order, as in QuestPDF), padding, borders, corner radius, background, sizes, alignment, page breaks, `Layers`, `Inlined`, `MultiColumn`, `EnsureSpace`, `Decoration`, lists, transforms and `ScaleToFit`, which draws

content smaller until it fits a box of fixed size, for badges and labels. `Repeat` draws content again on each later page its parent continues on, `StopPaging` keeps content to its first page, and `ExtendLastCellsToTableBottom` runs short columns to the end of a table. The page's `Header()`, `Content()` and `Footer()` align vertically as in QuestPDF, so `Content().AlignMiddle()` centres a cover title on the page, and border calls in a row merge into one border.

- **Text**: spans with font, size, weight, colour, decoration, spacing, superscript and subscript, OpenType features, justified text, `ClampLines`, headings that become bookmarks, links, and page numbers and the page count anywhere: in the header, the footer and the content. Section page numbers: `BeginPageNumberOfSection`, `EndPageNumberOfSection`, `TotalPagesWithinSection` and `PageNumberWithinSection`, in Roman numerals, letters or zero-padded with `Format(PageNumberFormat)`. `Overline`, `BreakAnywhere`, `Hyphenate`, per-span direction, `Leader` (dots or a line up to a price or page number) and `Markdown` content, written with PdfPecker's own Markdown parser and safe for text from users. `TableOfContents()` lists the headings with dot leaders and page numbers.
- **Pages**: size, margins, colour, and a header and footer on every page with page numbers. `ShowOnce` and `SkipOnce` in the header or footer give the first page a band of its own, such as a letterhead followed by a slim header. Several `Page` calls give page sets with their own size, margins, header and footer in one document, and page numbers, links, PDF/A, PDF/UA and signatures work across all of them. `EvenPageHeader` and `EvenPageFooter` for documents printed on both sides, `Background` and `Foreground` for letterheads, frames and watermarks over the whole page, and `ContinuousSize` for receipts as tall as their content. `Document.Merge` makes one PDF of several documents, each with its own page numbers or numbered through.
- **Form fields and footnotes**, which QuestPDF does not have: `TextField` (one line, multiline or password), `CheckBox`, `RadioButton`, `ComboBox` and `ListBox` become fillable PDF fields, `SignatureField` an empty field that `PdfSigner` signs later, and `text.Footnote("...")` numbers a note and sets it at the bottom of its page.
- **Images and graphics**: PNG, JPEG, WebP, BMP, TIFF, GIF and SVG images, shared `Image` and `SvgImage` objects with `WithCompressionQuality` and `WithRasterDpi`, linear and radial gradients, `Opacity`, lines, column, bar, line and pie charts with `Chart`, and QR, EPC payment QR, Swiss QR, Code 128, EAN-13, Data Matrix and GS1 Data Matrix barcodes. `SwissQrBill` lays out the whole Swiss QR-bill payment part with its receipt.
- **Accessibility**: every one of QuestPDF's `Semantic*` calls, formulas, indexes, tables of contents and list labels included, writes the HTML PdfPecker tags a PDF/UA document from. `Language("da")` sets the language of a part.
- **Existing PDFs**: `DocumentOperation`, with QuestPDF's names, merges, overlays and underlays pages, adds attachments, encrypts, decrypts and linearizes.
- **Everything else in PdfPecker** through `PdfOptions`: PDF/A, PDF/UA, signatures, passwords, watermarks, e-invoices, attachments and fonts.
- **Output**: `GeneratePdf` (bytes, file or stream), `GeneratePdfAsync`, `GenerateImages` and `GenerateHtml`, which shows the HTML PdfPecker renders.

Text, links, colours and names you pass are encoded, so caller data never turns into markup. `Html()` and `Css()` are the exception: they insert trusted HTML and CSS as they are, for anything the fluent

calls don't cover yet, so never pass user input to them.

Coming from QuestPDF

Most QuestPDF code ports by changing the `using` lines: `Document.Create`, `Page`, `Header`, `Content`, `Footer`, `Column`, `Row`, `Table`, `Text`, `Padding`, `Border`, `Background`, `Align*`, `Width`, `Height` and `GeneratePdf` have the same names. `ScaleToFit`, `ShowOnce`, `SkipOnce`, page numbers in the content and every section page number work too, and so do `Repeat`, `StopPaging`, `ContinuousSize`, `ExtendLastCellsToTableBottom`, `Document.Merge`, `DocumentOperation`, `Image.FromFile` and the image compression settings, page `Background` and `Foreground`, and every `Semantic*` call. QuestPDF formats page numbers with a function, which PdfPecker cannot run during layout, so `Format(PageNumberFormat)` gives Roman numerals, letters and zero padding instead. Not available: `Dynamic`, `Canvas` and the Companion previewer. QuestPDF compresses images at High quality and 288 dpi by default and PdfPecker.Fluent leaves them as they are unless asked. The package's README lists every difference.

Coming from MigraDoc

MigraDoc builds a document as an object model (`Document`, `Section`, `Paragraph`, `Table`) and renders it through PDFsharp. With 6.5.0 (October 2026), PDFsharp and MigraDoc moved to AGPL-3.0 or a paid licence; 6.2.4 and earlier were MIT. PdfPecker.Fluent is MIT and describes the same documents in code. The usual MigraDoc calls map like this:

- `document.AddSection()`, `section.PageSetup` become `container.Page(page => ...)`, `page.Size`, `page.Margin`
- `section.Headers.Primary`, `section.Footers.Primary` become `page.Header()`, `page.Footer()`
- `Headers.FirstPage` with `DifferentFirstPageHeaderFooter` becomes `ShowOnce()` and `SkipOnce()` in the header or footer
- `AddParagraph`, `AddFormattedText` become `Text("...")`, or `Text(t => t.Span("...").Bold())`
- `AddTable`, `AddColumn`, `AddRow` become `Table` with `ColumnsDefinition` and `Cell()`
- `MergeRight`, `MergeDown` become `ColumnSpan`, `RowSpan`
- `row.HeadingFormat = true` becomes `table.Header(...)`, repeated on every page
- `KeepTogether`, `KeepWith` become `PreventPageBreak()`, `EnsureSpace()`
- `AddPageBreak` becomes `PageBreak()`
- `AddPageField`, `AddNumPagesField` become `CurrentPageNumber()`, `TotalPages()`
- `AddSectionPagesField`, `AddPageRefField` become `TotalPagesWithinSection`, `BeginPageNumberOfSection`
- `AddBookmark`, `AddHyperlink` become `Section("name")`, `SectionLink`, `Hyperlink`
- `AddFootnote` becomes `text.Footnote("...")`
- `AddImage`, `AddTextFrame` become `Image`, `Layers` or `Unconstrained`
- `AddChart` becomes `Chart`, and tab stops with leaders become `Leader`

- `Headers.EvenPage` becomes `EvenPageHeader()`
- `ListInfo` becomes `List`
- Styles such as `Normal` and `Heading1` become `DefaultTextStyle` and `TextStyle` objects you reuse

Not available: RTF output and charts beyond column, bar, line and pie. What PdfPecker.Fluent adds over MigraDoc: complex scripts and right-to-left text, hyphenation, PDF/A and PDF/UA built from the layout, signatures, form fields, QR codes and output as images or HTML.

The fluent layer adds well under a millisecond to a typical document. On the same invoice, built call for call, it is about four times as fast as QuestPDF 2026.9.1, as measured in the PdfPecker.Benchmarks repository.

Razor components

PdfPecker.Razor turns Razor components (`.razor` files) into PDFs. Write a document template as a component, with typed parameters, `@foreach`, child components, layouts and dependency injection, and get a PDF from it in one call. The component is rendered to HTML with Microsoft's `HtmlRenderer`, the static renderer ASP.NET Core uses for server-side rendering, and PdfPecker converts the HTML. There is no browser, no ASP.NET Core host and no JavaScript, so it works the same in a console app, a background worker or a web app.

```
dotnet add package PdfPecker.Razor
```

The package is released with PdfPecker under the same version. It depends on PdfPecker and on `Microsoft.AspNetCore.Components.Web` of the app's .NET version (8, 9 or 10), which holds `HtmlRenderer`. The project with the `.razor` files uses the Razor SDK, `<Project Sdk="Microsoft.NET.Sdk.Razor">`, so they compile to C# classes.

A template

`Invoice.razor`:

```
@using System.Globalization

<!DOCTYPE html>
<html>
<head>
    <style>
        @@page { size: A4; margin: 20mm; }
        td.amount { text-align: right; }
    </style>
</head>
<body>
    <h1>Invoice @Data.Number</h1>
    <table>
        @foreach (var line in Data.Lines)
        {
            <tr><td>@line.Description</td><td class="amount">@line.Amount.ToString("N2",
CultureInfo.InvariantCulture)</td></tr>
        }
    </table>
</body>
</html>

@code {
    [Parameter, EditorRequired] public InvoiceModel Data { get; set; } = default!;
}
```

and the code that turns it into a PDF:

```
using PdfPecker;
using PdfPecker.Razor;

var parameters = new Dictionary<string, object?> { [nameof(Invoice.Data)] = invoice };
byte[] pdf = await RazorToPdf.ConvertAsync<Invoice>(parameters);

// Every PdfPecker option works: page size, fonts, PDF/A, signatures, e-invoices.
byte[] archived = await RazorToPdf.ConvertAsync<Invoice>(parameters, new PdfOptions {
Conformance = PdfConformance.PdfA2b });

// Straight to an ASP.NET Core response, with the request's services for components that
@inject them.
app.MapGet("/invoice/{id}", async (int id, HttpContext context, InvoiceStore store) =>
{
    var parameters = new Dictionary<string, object?> { ["Data"] = await store.LoadAsync(id) };
    context.Response.ContentType = "application/pdf";
    await RazorToPdf.ConvertAsync<Invoice>(context.Response.Body, parameters, services:
context.RequestServices,
        cancellationToken: context.RequestAborted);
});

// The HTML the component renders to, to open in a browser while working on the template.
string html = await RazorToPdf.RenderHtmlAsync<Invoice>(parameters);
```

A complete example, with a layout and a child component for the line table, is in samples/PdfPecker.Razor.Sample.

What works

- **Typed templates.** Parameters are C# properties, so a misspelt field or a wrong type is a compile error. A parameter name without a `[Parameter]` property fails with `InvalidOperationException`.
- **Child components and layouts.** Split a document into components and pass data down with parameters or `CascadingValue`. A layout is a component with `ChildContent`, or a `LayoutComponentBase` used through `LayoutView`.
- **Dependency injection.** `@inject` and constructor injection take services from the `IServiceProvider` given as `services`, such as `RequestServices`. Without it there are none, and a component that injects one fails with a message that names it.
- **Asynchronous data.** `OnInitializedAsync` and `OnParametersSetAsync` are awaited before the HTML is taken.
- **Everything PdfPecker does.** The HTML goes to `HtmlToPdf`, so CSS, paged media, fonts and every option apply. Relative image paths resolve against `PdfOptions.BaseDirectory`.
- **Same output.** `ConvertAsync` gives byte for byte the PDF that `HtmlToPdf.ConvertAsync` gives for the HTML of `RenderHtmlAsync`, so a fixed `CreationDate` makes the output reproducible.

Safety

Razor encodes every value it writes into the markup, so names, addresses and other caller data cannot become HTML. Only a `MarkupString` inserts HTML as it is; use it only for markup you trust.

Each call renders in an `HtmlRenderer` of its own, disposed when the call returns, so nothing is kept between conversions and parallel calls are safe. Components render once and statically, as in static server-side rendering: event handlers, timers and JavaScript interop do not run. The cancellation token stops the PDF conversion; pass it to the component as a parameter when the component does slow work of its own.

Trimming and Native AOT

PdfPecker.Razor is trimmable and AOT-compatible. Microsoft's `Microsoft.AspNetCore.Components` reports trim warnings from its own code, folded into one `IL2104` warning when you publish trimmed or with Native AOT; the sample's project file shows how to ignore only that assembly's warning. `@layout` and `LayoutView` set a layout's type by reflection, which the trim analyzer warns about in your project; a layout written as a component with `ChildContent` avoids it.

Rendering the sample's 20-line invoice component takes about 0.03 ms, against about 0.5 ms for converting its HTML. The package adds about 1.4 MB of Microsoft assemblies to an app without ASP.NET Core; an ASP.NET Core app already has them in its shared framework.

Command line

> **Changed in 1.24.0:** Native executable, Local files and passwords.

PdfPecker.Cli is the `pdfpecker` dotnet tool: HTML and Markdown to PDF, and the everyday jobs on existing PDFs, from a terminal, a script or a build pipeline. It is a thin layer over the library, so it needs no browser, no JavaScript engine and no native binaries, and it never fetches anything over the network.

```
dotnet tool install -g PdfPecker.Cli
pdfpecker convert invoice.html            # writes invoice.pdf
pdfpecker watch invoice.html              # converts again on every save
pdfpecker merge a.pdf b.pdf:1-3 -o all.pdf
```

`pdfpecker help` lists the commands and `pdfpecker help <command>` gives the options of one. The tool is released with every PdfPecker release under the same version, and carries the library inside it, so it has no dependencies.

Native executable New in 1.24.0

A .NET 10 SDK installs the tool as a Native AOT executable from a package for its platform (`PdfPecker.Cli.linux-x64`, `win-arm64`, `osx-arm64`, ...: Linux x64 and Arm64 with glibc or musl, macOS and Windows on x64 and Arm64). It starts a command in a few milliseconds, where the framework-dependent tool spends about a third of a second starting the runtime and compiling: the sample invoice converts in 17 ms instead of 355 ms. On other platforms a .NET 10 SDK installs `PdfPecker.Cli.any`, the framework-dependent tool, and .NET 8 and 9 SDKs install the framework-dependent tool as before. The commands and their output are the same in every case. On Windows the SDK starts the native tool through a `pdfpecker.cmd` file, which Command Prompt and PowerShell run as `pdfpecker`; Git Bash needs the full name, `pdfpecker.cmd`.

Commands

Command	What it does
`convert`	HTML or Markdown file (or `-` for standard input) to PDF
`watch`	`convert`, again each time the file, its folder or the style sheet changes
`merge`, `split`, `extract`, `remove`, `rotate`	Join, cut, keep, drop and turn pages
`stamp`	Lay a PDF or an HTML page over or under every page
`compact`, `linearize`, `sanitize`	Smaller files, Fast Web View, no scripts or launch actions
`sign`, `verify`, `encrypt`, `decrypt`	PAdES signatures, AES-256 passwords
`to-image`, `to-html`, `text`, `info`	PNG, JPEG, WebP or TIFF, one HTML file, plain text, a summary

`-` stands for standard input or output, so PDFs can be piped from one command to the next. Progress and errors go to standard error. The exit code is 0 on success, 1 when the work failed (a damaged file, a wrong password, an invalid signature) and 2 when the command line is wrong. A command that takes several inputs needs an output ending in `.pdf`, so a mistyped call cannot overwrite one of the source documents.

Local files and passwords New in 1.24.0

A document reads local files (images, style sheets) only from its own folder, unless `--enable-local-file-access` is given. HTML or Markdown piped in on standard input has no folder, so it reads local files only from a `--base-dir` it is given: `pdfpecker convert - --base-dir .` keeps relative images working. Every password option has an `-env` form (`--user-password-env`, `--owner-password-env`, `--password-env`, `--cert-password-env`, `--stamp-password-env`) that reads it from an environment variable, where other users of the machine cannot see it in the process list.

With several inputs, `convert` merges them into one PDF and then applies what belongs to the whole file: the title, author and creation date, PDF/A, Fast Web View and encryption. `[page]` and `[topage]` in the header and footer texts count the whole output, as in wkhtmltopdf, and `[sitepage]` and `[sitepages]` each input's own pages.

`watch` is meant for designing a template: keep the PDF open in a viewer that reloads it, and every save of the HTML or its style sheet shows up there.

Replacing wkhtmltopdf

A call without a command word is read like wkhtmltopdf's, with its 10 mm default margins, so scripts written for wkhtmltopdf keep working:

```
pdfpecker -s A4 -O Landscape -T 15mm --footer-center "Page [page] of [topage]" in.html out.pdf
```

The page size and orientation, margins, `--title`, `--zoom`, `--disable-smart-shrinking`, `--user-style-sheet`, `--no-outline`, `--no-pdf-compression`, `--enable-local-file-access`, `-q` and the `--header-left/-center/-right` and `--footer-left/-center/-right` texts (with `[page]`, `[topage]`, `[date]`, `[time]` and `[title]`) work as they do there; the header and footer texts become margin boxes. Options that exist only because wkhtmltopdf runs a browser (`--javascript-delay`, `--cookie`, `--header-html`, `--grayscale`, ...) are accepted and ignored with a warning, so a script keeps running. Web addresses are refused: save the page to a file first. The wkhtmltopdf page covers moving the HTML itself.

Signing

```
export PDFPECKER_CERT_PASSWORD=...
pdfpecker sign contract.pdf signed.pdf --cert me.pfx --reason "I approve" --page 1 --rect
36,36,200,60
pdfpecker verify signed.pdf --trusted-root ca.cer
```

Passwords given on the command line are visible to other users of the machine; prefer the `-env` options and `PDFPECKER_CERT_PASSWORD`. Digital signatures has what the signatures contain.

Python

pdfpecker on PyPI is PdfPecker for Python: HTML and Markdown to PDF, and PDFs to images, text and Markdown, from a `pip install`. It is the same library as the NuGet package, compiled ahead of time into a native library inside the wheel, so there is no .NET to install, no browser and no Python dependencies.

```
pip install pdfpecker
```

The package version is the PdfPecker version it holds, so everything this manual says about HTML, CSS and paged media applies to it. Wheels cover Linux (glibc 2.28 or later and musl, x86-64 and ARM64), macOS (Apple Silicon and Intel) and Windows (x64 and ARM64), on Python 3.9 and later.

Converting

```
import pdfpecker

pdf = pdfpecker.html_to_pdf("<h1>Invoice 42</h1><p>Total: <b>1,250.00 EUR</b></p>")
pdfpecker.html_file_to_pdf("templates/invoice.html", "invoice.pdf")   # images and CSS load
next to the file
pdfpecker.markdown_file_to_pdf("README.md", "readme.pdf")

pngs = pdfpecker.pdf_to_images(pdf, dpi=96)    # one PNG per page, or format="jpeg" or "webp"
text = pdfpecker.pdf_to_text(pdf)              # the text in reading order, pages separated by
\f
both = pdfpecker.merge_pdfs([pdf, "terms.pdf"])
```

The HTML and Markdown functions take keyword options that match PdfOptions: `page_size`, `landscape`, `margin`, `header_text`, `footer_text`, `fonts`, `title`, `conformance` (PDF/A), `accessibility` (tagged PDF and PDF/UA), `creation_date` for byte-identical output, and more. A document from a string loads no local files unless `base_dir` names the folder its images, style sheets and fonts are in. The other functions are `pdf_to_text_pages`, `pdf_to_markdown`, `pdf_to_pdfa`, `pdf_page_count` and `library_version`.

Every function takes bytes or a path for PDFs and raises a subclass of `PdfPeckerError` when the work fails. The calls are thread-safe and release the GIL while they work, so threads convert in parallel. The package README lists every function, option and error.

MCP server

The package includes a Model Context Protocol server, so chat apps and coding agents can make PDFs, look at their pages and read them. It runs over stdio and needs nothing but the package. In Claude Code:

```
claude mcp add pdfpecker -- uvx pdfpecker mcp
```

For Claude Desktop and other apps with an `mcpServers` setting:

```
{ "mcpServers": { "pdfpecker": { "command": "uvx", "args": ["pdfpecker", "mcp", "--root",
"/Users/me/Documents/pdfs"] } } }
```

The tools convert HTML and Markdown to PDF files, render pages to PNG for the model to look at, read text and Markdown, convert to PDF/A and merge files. They read and write only inside the `--root` folders, because the model picks the paths. The server is listed in the MCP Registry as `io.github.Moonwalker1307/pdfpecker`.

Agent skill

The PdfPecker Skills repository holds an Agent Skill that teaches an agent to write HTML with print CSS, convert it with pdfpecker, look at the pages and fix them, with an invoice and a report template. In Claude Code, add the marketplace and install the plugin:

```
/plugin marketplace add Moonwalker1307/PdfPecker.Skills
/plugin install pdfpecker@pdfpecker-skills
```

The repository's README covers Claude.ai, the Claude API and other agents that read skill folders.

What to know

- There is no JavaScript: charts must be SVG or images, not scripts that draw them.
- Nothing loads from the network: images, style sheets and fonts come from `base_dir` or `data:` URIs.
- The built-in fonts cover Latin, Greek and Cyrillic. For Chinese, Japanese, Arabic, Hebrew, Hindi or Thai, pass `fonts=[...]` with a font that covers the script, or `use_system_fonts=True`. See Fonts.
- The package is built from PdfPecker.Python after each PdfPecker release on NuGet, so a new version reaches PyPI shortly after.

Rust

pdfpecker on crates.io is PdfPecker for Rust: HTML and Markdown to PDF, and PDFs to images, text and Markdown, with a safe Rust API and no runtime dependencies. It is the same library as the NuGet package, compiled ahead of time into a native library, so there is no .NET to install and no browser.

```
cargo add pdfpecker
```

The crate version is the PdfPecker version it holds, so everything this manual says about HTML, CSS and paged media applies to it. It needs Rust 1.85 or later on Linux (glibc 2.28 or later and musl, x86-64 and ARM64), macOS (Apple Silicon and Intel) or Windows (x64 and ARM64).

Converting

```
use pdfpecker::{ImageOptions, PageSize, PdfOptions, TextOptions};

let html = "<h1>Invoice 42</h1><p>Total: <b>1,250.00 EUR</b></p>";
let pdf = pdfpecker::html_to_pdf(html, &PdfOptions::new())?;

// Images and CSS load next to the file.
let invoice = pdfpecker::html_file_to_pdf("templates/invoice.html", &PdfOptions::new())?;
let letter = PdfOptions::new().page_size(PageSize::Letter);
let readme = pdfpecker::markdown_file_to_pdf("README.md", &letter)?;

// One PNG per page, or JPEG or WebP; the text has pages separated by \f.
let pngs = pdfpecker::pdf_to_images(&pdf, &ImageOptions::new().dpi(96.0))?;
let text = pdfpecker::pdf_to_text(&pdf, &TextOptions::new())?;
let both = pdfpecker::merge_pdfs(&[&invoice, &readme], &Default::default())?;
```

`PdfOptions` has the settings of PdfOptions as chained methods: `page_size`, `landscape`, `margin`, `header_text`, `footer_text`, `font`, `title`, `conformance` (PDF/A), `accessibility` (tagged PDF and PDF/UA), `creation_date` for byte-identical output, and more. A document from a string loads no local files unless `base_dir` names the folder its images, style sheets and fonts are in. The other functions are `pdf_to_text_pages`, `pdf_to_markdown`, `pdf_to_pdfa`, `pdf_page_count`, `library_version` and `library_path`.

Every function returns `pdfpecker::Result`, whose error has an `ErrorKind` such as `Unreadable`, `Password`, `Limit` or `Timeout`. The functions are safe to call from several threads at once and convert in parallel; they block, so async code runs them with `tokio::task::spawn_blocking`. The crate documentation lists every function and option.

Deploying

The build script downloads the target's native library from the pdfpecker wheel on PyPI, about 9 MB once, and checks it against SHA-256 values the crate carries; `cargo run` and `cargo test` then find it on their own. To ship a program, copy `pdfpecker_native.so` (`.dylib` on macOS, `.dll` on Windows)

from `target/<profile>/build/pdfpecker-*/out/` next to the executable. Offline builds set `PDFPECKER_LIB_DIR` to a folder that already holds the library.

On musl (Alpine), the program must be linked dynamically to load the library: build with `RUSTFLAGS="-C target-feature=-crt-static"`.

Speed

From Rust on a 4-core Linux cloud VM, median time per document after warm-up, against IronPress, a Rust HTML to PDF crate, and Chrome driven by chromiumoxide. The Chrome score compares each engine's pages with what Chrome prints, over 54 test pages, where 1 is identical:

	Invoice, 1 page	Report, 29 pages	New process, first PDF	Peak memory, one PDF	Chrome score
pdfpecker 1.23.0	**0.72 ms**	**28 ms**	**22 ms**	**26 MB**	**0.95**
IronPress 1.7.0	41 ms	1,873 ms	185 ms	40 MB	0.47
Chrome 153 via chromiumoxide 0.9	18 ms	231 ms	115 ms	396 MB	1

See Compare for more on the comparison.

What to know

- There is no JavaScript: charts must be SVG or images, not scripts that draw them.
- Nothing loads from the network: images, style sheets and fonts come from `base_dir` or `data:` URIs.
- The built-in fonts cover Latin, Greek and Cyrillic. For other scripts, add a font that covers them with `font(...)`, or turn on `use_system_fonts`. See Fonts.
- The crate is built from PdfPecker.Rust after each PdfPecker release reaches PyPI.

Images to PDF

`ImageToPdf` puts images into a PDF, one per page, such as scanned receipts, photos or drawings for an archive. It reads PNG, JPEG, GIF, WebP, BMP and TIFF (the first frame of an animated GIF or a multi-page TIFF).

```
// Each page the size of its image
byte[] pdf = ImageToPdf.Convert(new[] { File.ReadAllBytes("scan-1.jpg"),
File.ReadAllBytes("scan-2.png") });

// Files in, file out
ImageToPdf.ConvertFiles(new[] { "receipt-1.jpg", "receipt-2.jpg" }, "receipts.pdf");

// A4 pages with margins, each image centred and scaled down to fit, as PDF/A for archiving
byte[] archive = ImageToPdf.Convert(scans,
    new PdfOptions { PageSize = PageSize.A4, Conformance = PdfConformance.PdfA2b, Title =
"Receipts 2026" },
    new ImageToPdfOptions { FitPageToImage = false });
```

Page size

By default every page is exactly the size of its image, with no margins. `ImageToPdfOptions.Dpi` (default 96, one image pixel per CSS pixel) sets the printed size: a scan 2,480 pixels wide at 300 dpi becomes an A4-wide page.

With `FitPageToImage = false` every page has the `PdfOptions` page size and margins. Each image is centred, and scaled down to fit when it is larger than the space inside the margins; smaller images keep their size.

Output settings

The `PdfOptions` work as for HTML to PDF: metadata, PDF/A, passwords and watermarks, and a fixed `CreationDate` for byte-identical output. JPEG images are embedded as they are, without re-encoding.

Errors

An empty list or more than `ImageToPdf.MaxImages` (10,000) images throws `ArgumentException`, as does a `Dpi` that is not a positive number. An image that is not in a supported format or is damaged throws `InvalidDataException` naming its position ("Image 2 ..."), and more images than `PdfOptions.MaxPages` throws `PdfLimitException`. An overload takes a cancellation token.

Fonts

> **Changed in 1.24.0:** OpenType features, Fonts read when needed, How fonts are embedded.

Register TrueType or OpenType fonts and use them by name in CSS. Only the glyphs a document uses are embedded, and text can use any Unicode character the fonts have (Greek, Cyrillic, Central European, CJK, ...):

```
var options = new PdfOptions();
options.Fonts.AddFile("fonts/OpenSans-Regular.ttf")
             .AddFile("fonts/OpenSans-Bold.ttf");             // family, weight and style are
read from the files
options.Fonts.Add(bytes, familyName: "sans-serif");           // or override a generic family
options.Fonts.AddDirectory(@"C:\Windows\Fonts");               // read once, loaded only when
used
byte[] pdf = HtmlToPdf.Convert("<p style=\"font-family: 'Open Sans'\">Łódź, Ελλάδα,
Москва</p>", options);
```

@font-face

Documents can bring their own fonts with `@font-face` (files, `data:` URIs or `ResourceResolver` URLs; `local()` refers to registered fonts):

```
@font-face { font-family: Brand; src: url(fonts/brand.woff2) format("woff2"); font-weight:
400; }
@font-face { font-family: Brand; src: url(fonts/brand-bold.ttf); font-weight: 700; unicode-
range: U+0000-024F; }
```

A `FontCollection` can be shared between options and parallel conversions. Formats: `.ttf`, `.otf`, `.ttc` collections, WOFF and WOFF2 (including WOFF2 collections; decoded with .NET's built-in Brotli). Fonts are matched like browsers do: by family, then style, then the nearest weight; a missing bold or italic face is synthesized. With the `font-stretch` descriptor (a keyword, a percentage or a range such as `75% 100%`), faces of one family are matched by width first, as in browsers, and a variable font's `wdth` axis is kept within the declared range. `font-display` is accepted and ignored, since nothing loads progressively. `size-adjust`, `ascent-override`, `descent-override` and `line-gap-override` work as in Chrome, so a fallback face sized to match a web font keeps the web font's size and line height: `size-adjust` scales the face's widths, kerning and metrics, and the overrides replace its line metrics. A character that the chosen font lacks is drawn with the next font in the `font-family` list that has it, then the standard font, then any other registered font.

Variable fonts

Variable fonts (one file with a weight, width, slant or other axis, with TrueType or CFF2 outlines) work with the usual CSS properties. PdfPecker builds a static instance of the font, with its kerning, for each combination a document uses and embeds a subset of it, so bold text uses the font's real bold rather than a synthesized one:

```
@font-face { font-family: Inter; src: url(fonts/InterVariable.ttf); }    /* covers all of the
font's weights */
h1 { font-family: Inter; font-weight: 750; }
.narrow { font-stretch: 75%; }                                           /* the wdth axis */
em { font-style: italic; }                                               /* the ital axis, or
slnt */
.custom { font-variation-settings: "wght" 650, "GRAD" 50; }              /* any axis,
overrides the rest */
```

A variable `@font-face` without `font-weight` covers the font's whole weight range, and so does a variable font added through `options.Fonts`. A face declared with a single weight (`font-weight: 400`) stays at that weight, and bold is synthesized as for any other font. The optical size axis (`opsz`) follows the font size unless `font-optical-sizing: none` is set. CFF2 fonts are embedded as ordinary CFF fonts, since few PDF readers can draw CFF2.

OpenType features Changed in 1.24.0

`font-variant-numeric` and `font-feature-settings` turn on a font's optional glyphs, as in a browser. Tabular figures line up amounts in a column:

```
td.amount { font-variant-numeric: tabular-nums; }          /* every digit the same width */
.oldstyle { font-variant-numeric: oldstyle-nums slashed-zero; }
.alt { font-feature-settings: "ss01", "salt" 2; }           /* stylistic set 1, the second
alternate */
```

Features that swap one glyph for another (`tnum`, `pnum`, `lnum`, `onum`, `zero`, `ss01` to `ss20`, `salt`, `case` and the like) are applied; the font must have them. Fractions with `font-variant-numeric: diagonal-fractions` or `font-feature-settings: "frac"` are formed as Chrome forms them: the font's contextual rules turn `1/2` and `11/16` into fractions and leave `a/b` as it is, and `tabular-nums` used with them does not widen the fraction's digits. Text copied from the PDF keeps the original characters.

Contextual alternates and ligatures (`calt`, `clig`, `rclt`) are on by default, as Chrome applies them, so fonts such as Inter and Fira Code draw their arrows and programming ligatures; `font-feature-settings: "calt" 0` or `font-variant-ligatures: no-contextual` turns them off.

Colour fonts

Text in a COLR colour font, version 0 or 1, such as Noto Color Emoji or Nabla, prints in colour the way Chrome prints it: layers, gradients (also sweep gradients), blend and composite modes, clip boxes and variable colour fonts at the font's instance, with the palette's colours and the text colour where the font asks for it. The text stays text, so it can be selected, searched and copied, and colour fonts pass every PDF/A and PDF/UA profile. Emoji sequences join as in Chrome: skin tones, flags, zero width joiner sequences and subdivision flags. When `unicode-range` splits a font into parts, as Google Fonts serves Noto Color Emoji, each emoji is drawn with the part that has all of it.

```
var options = new PdfOptions();
options.Fonts.AddFile("fonts/NotoColorEmoji-Regular.ttf");     // family "Noto Color Emoji"
var pdf = HtmlToPdf.Convert("<p style='font-family: sans-serif, \"Noto Color Emoji\"'>Ready
  </p>", options);
```

Font properties

- `font-kerning: none` turns kerning off (so does `font-feature-settings: "kern" 0`), for the standard fonts too.
- `font-synthesis: none` (or the longhands `font-synthesis-weight`, `-style` and `-small-caps`) stops the made-up bold and italic for an `@font-face` family that has only a regular face, as in Chrome.
- `font-size-adjust` (a number, `from-font`, or `ex-height`, `cap-height`, `ch-width`, `ic-width` or `ic-height` with a number) draws text in the size that gives the first font of the family that proportion, so a fallback font with small letters does not look smaller; `em` units and `line-height` numbers keep the computed size.
- `font-variant-position: sub | super` uses the font's own `subs` and `sups` glyphs; a font without them is unchanged, as in Chrome.
- `font-variant-east-asian` (`jis78` to `jis04`, `simplified`, `traditional`, `full-width`, `proportional-width`, `ruby`) picks the font's alternate CJK glyphs.
- `font-variant-caps: small-caps` and `all-small-caps` use the font's own `smcp` and `c2sc` glyphs when it has them, and otherwise scaled capitals, unless `font-synthesis` turns that off.

Standard fonts

Text without a registered font uses the standard PDF fonts (Helvetica, Times, Courier), which viewers supply themselves, so files stay small but are limited to Western European characters. Set `EmbedStandardFonts = true` to embed the bundled Liberation fonts in their place instead: they have the same widths, so the layout does not change, the text looks the same in every viewer, and any character they cover (Latin, Greek, Cyrillic, ...) works. PDF/A always does this. It adds a subset of each font used to the file (typically 10-25 KB per font).

Fonts read when needed New in 1.24.0

`AddDeferred` registers a face by its family, weight, style and the characters it has, and reads the file only when a document uses it: when CSS names the family, or when text has a character that the fonts before it lack and that its `unicode-range` includes. Large fonts such as Noto CJK then cost no time or memory in documents that do not need them.

```
options.Fonts.AddDeferred("Noto Sans CJK SC", () => File.ReadAllBytes("NotoSansCJK-
Regular.ttc"),
                          unicodeRange: "U+3000-9FFF, U+FF00-FFEF", faceIndex: 2);
```

The loader is called at most once per face, and faces whose loaders return the same array share one parsed font, so a collection registered face by face is read and embedded once. A loader that throws an `IOException` or returns data that is not a font leaves the face out. Fallback skips any family whose ranges cannot draw a character, so a document in Latin script never opens the CJK file.

Missing fonts and characters

A family that no font provides falls back silently, like in a browser. To notice, set `MissingFont`: it is called once per conversion for each family that had to be substituted, and throwing from it fails the conversion. To use the fonts installed on the machine for families you have not registered, set `UseSystemFonts = true` (off by default, so the output does not depend on the machine):

```
options.MissingFont = family => logger.LogWarning("Font {Family} is not available", family);
options.UseSystemFonts = true;   // looks in C:\Windows\Fonts, /Library/Fonts,
/usr/share/fonts, ...
```

A character that none of the document's fonts has falls back to fonts built into PdfPecker, as a browser falls back to the system's fonts: Central European, Greek and Cyrillic letters are drawn with the bundled Liberation font that matches the standard font (so they no longer turn into `?` when the standard fonts are not embedded), and arrows, check marks, stars, dingbats, math and technical symbols, enclosed numbers, box drawing and geometric shapes with PdfPecker Symbols, a 1,424-character subset of DejaVu Sans. Only the characters used are embedded, and a document that needs no fallback is written exactly as before.

A character that no font has, the built-in ones included, is drawn as an empty box (blank space in PDF/A). `MissingCharacter` is called once for each such character before anything is written, so a test can make sure a template's fonts cover its data:

```
options.MissingCharacter = ch => throw new InvalidOperationException($"No font has
U+{char.ConvertToUtf32(ch, 0):X4}");
```

How fonts are embedded Changed in 1.24.0

Registered and `@font-face` TrueType/OpenType fonts are embedded as subsets (Type0/CIDFontType2 for TrueType outlines, CIDFontType0 with an OpenType program for CFF outlines) with a ToUnicode map, so text can be searched and copied. A font's Unicode characters are read from any of its cmap tables (formats 0, 2, 4, 6, 10, 12 and 13), so a font whose only Unicode cmap is format 10 or 13, as last-resort fonts have, works too. Without them, the PDF standard fonts are used: `font-family` maps to Helvetica (sans-serif, Arial, Verdana, ...), Times (serif, Georgia, ...) or Courier (monospace, Consolas, ...), each in regular, bold, italic and bold italic. With `EmbedStandardFonts` or PDF/A, these are replaced by embedded subsets of the bundled, metric-compatible Liberation Sans, Serif and Mono.

Subsets of large CFF fonts such as Noto CJK are small: a few Chinese characters in Noto Sans SC embed about 133 KB (1.93 MB in 1.23.0). A font's Unicode variation sequences (cmap format 14) draw the variant glyphs that variation selectors ask for.

Right-to-left text, CJK and hyphenation

> **Changed in 1.24.0:** Hyphenation.

Right-to-left text: Arabic and Hebrew

Arabic and Hebrew are laid out right to left and shaped like browsers do, with PdfPecker's own implementation of the Unicode Bidirectional Algorithm (UAX #9, passing all of Unicode's conformance tests) and of OpenType shaping (GSUB and GPOS, following HarfBuzz). Arabic letters join, lam-alef and other ligatures form, and harakat and Hebrew points are placed on their letters by the font's mark positioning. Mixed text (an English product name or a number inside an Arabic sentence) is put in the right order, and brackets are mirrored. Register a font that has the script, then set the direction with the `dir` attribute or CSS `direction`:

```
var options = new PdfOptions();
options.Fonts.AddFile("fonts/NotoNaskhArabic-Regular.ttf");
byte[] pdf = HtmlToPdf.Convert("""
    <html dir="rtl" lang="ar"><body style="font-family: 'Noto Naskh Arabic'">
      <h1>فاتورة رقم ٢٠٢٦/١٥</h1>
      <p>المبلغ الإجمالي: 1,250.00 DKK</p>
    </body></html>
    """, options);
```

In a right-to-left block, text aligns right (`text-align: start`), and table columns, flex rows, grid columns and list markers run from the right. `dir="auto"`, `<bdi>` and `<bdo>` work as in browsers, and so do `unicode-bidi`, `text-align: start`/`end` and the logical margin and padding properties (`margin-inline-start`, `padding-inline-end`, ...). `text-overflow: ellipsis` and `line-clamp` cut right-to-left lines at their left end. SVG `<text>` is reordered and shaped too, with `direction` deciding which way a text chunk runs from its `x`. Copied or extracted text gives back the original characters in logical order. A font without OpenType Arabic features is shaped with the Unicode presentation forms instead.

Indic, Thai and Lao scripts

Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada and Malayalam are shaped as HarfBuzz shapes them, from a port of its Indic shaper: conjuncts and half forms join, reph moves above its cluster and pre-base vowel signs are drawn before their consonant. Thai and Lao vowels and tone marks stack correctly, and combining marks in Latin, Greek and Cyrillic text are placed on their letters, or replaced by the font's precomposed letter. Register a font that has the script, such as one of the Noto fonts:

```
var options = new PdfOptions();
options.Fonts.AddFile("fonts/NotoSansDevanagari-Regular.ttf");
byte[] pdf = HtmlToPdf.Convert("<p lang='hi' style=\"font-family: 'Noto Sans Devanagari'\">
    : ₹1,250</p>", options);
```

Khmer, Myanmar, Sinhala, Tibetan, Mongolian, Javanese, Adlam and Syriac are not shaped yet (see Limitations).

Chinese, Japanese and Korean line breaking

Chinese and Japanese are written without spaces, so a line may break between almost any two characters. PdfPecker finds those breaks with the Unicode Line Breaking Algorithm (UAX #14) and the Japanese kinsoku rules, as browsers do: a line never starts with closing punctuation such as , , or , or with an iteration mark such as , and never ends with an opening bracket or quotation mark. Nothing is drawn at these breaks, and Latin words, numbers and emoji sequences inside the text stay whole. Text in other scripts breaks at spaces and hyphens, as before. The standard and bundled fonts have no CJK characters, so register a font that has them:

```
var options = new PdfOptions();
options.Fonts.AddFile("fonts/NotoSansJP-Regular.ttf");
byte[] pdf = HtmlToPdf.Convert("""
    <html lang="ja"><body style="font-family: 'Noto Sans JP'; line-break: strict">
      <p>                                                  </p>
    </body></html>
    """, options);
```

CSS `line-break` sets how strict the rules are, as in Chrome:

- `auto` (the default) and `strict` keep small kana (, , ...) and the prolonged sound mark off the start of a line.
- `normal` lets small kana and start a line, and allows a break before the hyphens `-`, `–`, and .
- `loose` also allows a break before an iteration mark or centred punctuation such as and , and on both sides of an ellipsis.
- `anywhere` breaks between any two characters and ignores the punctuation rules, in every script.

Korean text breaks between its syllables by default, as in Chrome. `word-break: keep-all` breaks CJK text only at spaces and punctuation instead, which suits Korean, written with spaces between words. `word-break: break-all` and `white-space: nowrap` work as for any other text. A justified Chinese or Japanese line is spread between its characters and punctuation, as in Chrome, and `text-justify` chooses where the space goes (see Text wrapping and justification).

CJK punctuation trimming

Fullwidth CJK punctuation such as , , and is drawn with blank space on one side. `text-spacing-trim` trims that blank half where it would look too wide, as Chrome does. The default, `normal`, trims it where two punctuation marks meet (, ,), and a closing bracket or quote at the end of a line also loses its blank half when that lets it fit. `space-first` also trims the opening bracket that starts each wrapped line, `trim-start` the one that starts any line, and `space-all` turns trimming off and keeps the spacing of PdfPecker 1.16 and earlier.

Trimming needs a font with the OpenType `halt` feature, which Noto Sans CJK, Source Han and most Chinese and Japanese fonts have. The font decides whether dots, colons and quotes count as opening or closing (Simplified Chinese fonts draw ` ` on the left, Traditional Chinese ones centre them), and `font-feature-settings` that set punctuation widths themselves (`palt`, `halt`, `pwid` and the like) turn trimming off, as in Chrome.

Hyphenation Changed in 1.24.0

`hyphens: auto` hyphenates words with the TeX hyphenation patterns (Liang's algorithm) of the text's language: the nearest `lang` attribute, else `PdfOptions.Language`. As in Chrome, a word is hyphenated only where it does not fit on its line, and a hyphen (or the `hyphenate-character`) is shown at the break. Words shorter than five letters, words with digits or other non-letters inside, URLs and e-mail addresses, and capitalized English words are not hyphenated, and a word with `` breaks only there. French and Italian also hyphenate words with an elided article or pronoun, such as *l'ordinateur* in French or *l'informazione* in Italian, with a straight or typographic apostrophe.

`hyphenate-limit-chars` sets the shortest word that is hyphenated and the fewest letters before and after a hyphen: up to three values, each `auto` or a whole number, such as `hyphenate-limit-chars: 8 3 3`. One value sets only the word length; with two, the letters after a hyphen are the same as before it. It is inherited, so it can be set on the whole document or on a single element. `auto` keeps the defaults above: words of at least five letters, and the patterns' own minimums around a hyphen (two letters before and three after in English, as in Chrome). Larger values make hyphenation more selective; smaller values do not go below the patterns' own minimums, as in Chrome.

Patterns for these languages are built in and loaded on first use, for every tag of the language (`fr-CA`, `pt-BR`, `es-MX` and so on):

Language	Tags
English (US patterns)	`en`
Danish	`da`
German (reformed spelling)	`de`
Dutch	`nl`
Norwegian	`nb`, `nn`, `no`
Swedish	`sv`
Spanish	`es`
French	`fr`
Italian	`it`
Portuguese	`pt`
Polish	`pl`

Others can be supplied as TeX pattern files, such as those of the hyph-utf8 project:

```
var options = new PdfOptions
{
    // Your own patterns for a language (they take precedence over the built-in ones).
    Hyphenation = new Dictionary<string, HyphenationPatterns>
    {
        ["fi"] = new HyphenationPatterns(File.ReadAllText("hyph-fi.tex"), leftMin: 2,
rightMin: 2),
    },
};
byte[] pdf = HtmlToPdf.Convert("""
    <html lang="da"><body style="hyphens: auto; text-align: justify">
      <p>Arbejdsmarkedsbidraget betales af lønmodtageren.</p>
      <p lang="fi">Tavutus omilla tavutussäännöillä.</p>
    </body></html>
    """, options);
```

`HyphenationPatterns.Deferred` reads its patterns only when a document hyphenates text in their language, so many languages can be offered without holding their patterns in memory:

```
options.Hyphenation = new Dictionary<string, HyphenationPatterns>
{
    ["fi"] = HyphenationPatterns.Deferred(() => File.ReadAllText("hyph-fi.tex")),
};
```

The function is called at most once, by the first conversion that needs the patterns.

PDF/A, passwords and watermarks

> **Changed in 1.24.0:** Watermarks and passwords.

Watermarks and passwords Changed in 1.24.0

```
var options = new PdfOptions
{
    // A diagonal stamp on every page. Also: Image (PNG, JPEG, GIF, WebP, BMP, TIFF or SVG
bytes; SVG stays vector), FontSize, LetterSpacing, Angle, Opacity, Bold, Behind.
    Watermark = new PdfWatermark { Text = "WATERMARK", Color = "#c00000", Opacity = 0.25f },

    // AES-256 encryption (PDF 2.0 / Acrobat X and later). Without an OwnerPassword a random
one is used,
    // so the permissions cannot be lifted.
    Security = new PdfSecurity
    {
        UserPassword = "1234",                                  // needed to open the file;
null = opens freely
        OwnerPassword = "owner-secret",                         // full access
        Permissions = PdfPermissions.Print | PdfPermissions.Copy,
    },
};
```

PDF/A

For archiving, set `Conformance = PdfConformance.PdfA2b` (or another level below): the PDF gets XMP metadata, an sRGB output intent and a file ID, and CMYK JPEGs are converted to RGB. All fonts are embedded: text without a registered font uses the bundled Liberation fonts (see Fonts). PDF/A does not allow encryption, so combining `Conformance` with `Security` throws an `ArgumentException`.

```
var options = new PdfOptions { Conformance = PdfConformance.PdfA4 };
```

Level	Standard	File	Use it for
`PdfA2b`, `PdfA3b`	ISO 19005-2/-3, level B	PDF 1.7	the usual archive format; part 3 also allows attachments (e-invoices)
`PdfA2u`, `PdfA3u`	ISO 19005-2/-3, level U	PDF 1.7	as level B, plus a Unicode mapping for every character, so text can be searched and copied
`PdfA4`	ISO 19005-4:2020	PDF 2.0	the current part; identified in the XMP only (`pdfaid:rev` 2020, no conformance letter), no document information dictionary
`PdfA4f`	ISO 19005-4:2020, with files	PDF 2.0	PDF/A-4 with attachments: only with `EInvoice` (it throws without one)

Every font is embedded with a ToUnicode map, and a character no font has is left out as blank space rather than drawn as `.notdef` (in SVG text too), so levels U cost nothing over B. A PDF 2.0 file keeps the title, author, keywords and custom properties in its XMP metadata, since PDF 2.0 deprecates the

information dictionary. Every level, on its own and with PDF/UA, passes veraPDF 1.28 (flavours 2b, 2u, 3b, 3u, 4 and 4f) on documents with text, tables, lists, images, links, form fields, SVG and Markdown. AES-256 keys come from a password hash that is slow on purpose, which adds about 3 ms to a conversion.

Passwords are prepared with SASLprep before they are hashed, as PDF 2.0 requires and as pypdf and PDFBox do, so a password typed in another normalization form (é as one character or as e and an accent) opens the file as it does in other PDF 2.0 readers. A password with control, private-use or bidi control characters, or with characters Unicode 3.2 had not assigned (most emoji), cannot be prepared and is refused with an `ArgumentException`. Files encrypted elsewhere with such passwords still open.

To make a PDF you already have into PDF/A, whichever tool made it, see PDF to PDF/A.

Passwords on existing PDFs

To add, change or remove the password of a PDF you already have (from any producer, not only PdfPecker), use `PdfProtector`:

```
// Encrypt any PDF with AES-256
byte[] locked = PdfProtector.Encrypt(pdf, new PdfSecurity { UserPassword = "1234", Permissions
= PdfPermissions.Print });

// Change the passwords or permissions of an encrypted PDF (give its current user or owner
password)
byte[] relocked = PdfProtector.Encrypt(locked, new PdfSecurity { UserPassword = "5678" },
password: "1234");

// Remove the encryption, and with it the permission restrictions
byte[] open = PdfProtector.Decrypt(locked, "1234");

// Stream overloads, and a check
PdfProtector.Encrypt(inputStream, outputStream, security);
bool encrypted = PdfProtector.IsEncrypted(pdf);
```

Encrypted inputs can use RC4 (40 to 128 bit), AES-128 or AES-256; the output of `Encrypt` is always AES-256. Like other PDF tools, `PdfProtector` accepts the user password as well as the owner password (or none, for a file that only restricts permissions), so anyone who can open a document can also lift its permissions. A wrong password throws `PdfPasswordException`. The document is rewritten as a whole and keeps its pages, bookmarks, links, form fields and values, tags, metadata, page labels and attachments; earlier revisions from incremental updates are dropped. Signed PDFs are refused with an `ArgumentException` (rewriting them would break the signatures; sign after encrypting instead), as is encrypting a PDF/A file, or a PDF/UA file without `PdfPermissions.ExtractForAccessibility`. `Decrypt` returns a PDF that is not encrypted unchanged.

`Watermark` applies while converting HTML; to stamp a PDF you already have, see Stamping PDFs.

Accessibility: tagged PDF and PDF/UA

```
var options = new PdfOptions
{
    Accessibility = PdfAccessibility.PdfUA1,   // or PdfUA2, or Tagged: the same structure
without the PDF/UA claim
    Language = "da",                           // default: <html lang="..."> for PDF/UA
without either, "en"
    Conformance = PdfConformance.PdfA2b,       // optional: PDF/UA-1 combines with PDF/A-2 and
3, PDF/UA-2 with PDF/A-4
};
```

Tagged PDF

`Tagged` gives the PDF a structure tree built from the HTML, so screen readers, reflow and text extraction follow the document rather than the page layout: headings (`H1` to `H6`), paragraphs, lists with their labels, tables with header scopes and spans, figures with their alt text, links (tied to their clickable areas), block quotes, code and sections, all in document order, with `lang` attributes as the language of the element. Text outside a paragraph (straight in a `div`, say) gets one. Backgrounds, borders, running headers and footers, `HeaderText` / `FooterText`, watermarks, repeated table headers, fixed boxes and decorative images are artifacts that assistive technology skips.

PDF/UA-1

`PdfUA1` adds what ISO 14289-1 asks beyond the tags: every font embedded (the bundled Liberation fonts stand in for the standard ones), the title shown in the viewer's title bar and the PDF/UA identification in the XMP metadata. The output passes veraPDF's PDF/UA-1 checks, on its own and together with PDF/A-2b or PDF/A-3b (e-invoices included). It needs a title: `Title`, the `<title>` element or else the first heading; without one, conversion throws `ArgumentException`, as it does when `Security` leaves out `PdfPermissions.ExtractForAccessibility`.

PDF/UA-2

`PdfUA2` (ISO 14289-2:2024) is the PDF 2.0 successor: the file is written as PDF 2.0, its tags are in the PDF 2.0 structure namespace (block quotes, quotes and code, which PDF 2.0 dropped, keep their PDF 1.7 type and are role mapped to `Div` and `Span`), links and bookmarks go to structure destinations (the element they point at, not only its page), lists state their numbering and form fields describe themselves, and the XMP says `pdfuaid:part` 2 with `pdfuaid:rev` 2024. It has the same title and language requirements as `PdfUA1` and passes veraPDF's PDF/UA-2 checks, on its own and with `PdfA4`. PDF/UA-1 and PDF/A-2 and 3 are based on PDF 1.7, PDF/UA-2 and PDF/A-4 on PDF 2.0, so `PdfUA2` with `PdfA2b` (or any PDF/A-2 or 3 level) and `PdfUA1` with `PdfA4` throw `ArgumentException`.

What your HTML needs

What the tags can't fix is up to the HTML: give images an `alt` (an image without `alt`, `aria-label` or `title`, or an `<svg>` without a `<title>`, counts as decorative), use real headings, lists and `th` cells, and mark purely visual content `aria-hidden="true"`. Heading levels that skip one (`h1` then `h3`) are tagged one level below the previous heading, as PDF/UA does not allow gaps.

Some PDF/UA structure types have no HTML element, so they come from ARIA roles: `role="math"` makes a Formula, with its `aria-label` as the alternative text, `role="doc-index"` an Index, and `role="doc-toc"` on a `nav` a table of contents whose list items are TOC entries. In a list laid out by hand, such as a number in a flex row beside the text, `data-pdf-list-label` marks the item's label, and the rest of the item becomes its body.

Fillable forms

Form controls become fillable PDF form fields (AcroForm), drawn as Chrome prints them and filled with their current values. Adobe Reader, Preview, Chrome, Firefox and other viewers let the reader fill them in and save the result. The fields of an existing PDF can be read, filled and flattened too.

```
var html = """
    <p>Name <input name="name" value="Jane Doe" required></p>
    <p><label><input type="checkbox" name="newsletter" checked> Send me the
newsletter</label></p>
    <p><input type="radio" name="size" value="S"> S <input type="radio" name="size" value="M"
checked> M</p>
    <p><select name="country"><option value="dk">Denmark</option><option
value="se">Sweden</option></select></p>
    <p><textarea name="comments" rows="4" cols="40"></textarea></p>
    """;
byte[] pdf = HtmlToPdf.Convert(html);

// Or print the controls as they are, without form fields:
byte[] flat = HtmlToPdf.Convert(html, new PdfOptions { FormFields = false });
```

HTML	PDF field
`<input>` of type `text`, `email`, `tel`, `url`, `search`, `number`, `date` and the like	Text field (`maxlength` sets the maximum length)
`<input type="password">`	Password text field; the value is drawn as dots and not stored
`<textarea>`	Multi-line text field
`<input type="checkbox">`	Check box; its `value` is the export value (default `Yes`)
`<input type="radio">`	Radio buttons: those with the same `name` are one field, and each `value` is a choice
`<select>`	Drop-down list (combo box); with `multiple` or `size` > 1 a list box, multi-select with `multiple`
`<button>`, `<input type="submit">`, `reset`, `button`	Drawn only: a PDF has nothing to submit

Field names and behaviour

Fields are named after the `name` attribute (a period becomes `_`, since PDF uses it to nest fields; a repeated name gets `_2`, `_3`, ...; a control without a name gets `text1`, `checkbox1`, ...). `readonly` and `required` set the field's flags, `title`, `aria-label` or its `<label>` becomes its tooltip, and a `placeholder` is drawn in grey while the field is empty. Disabled controls (or controls in a disabled `<fieldset>`) are drawn but not fillable, and hidden inputs are left out. Each field has its own appearance, so the PDF looks right before anyone touches it and in viewers that never redraw fields; while a field is edited, the viewer draws its text in Helvetica, Courier or Times at the control's size and colour. Forms work with `Security` (the values are encrypted with the rest), `Conformance` (veraPDF passes the fields as PDF/A-2b and 3b), `Accessibility` (each widget sits in a `Form` element described

by its label, title or name, and veraPDF passes PDF/UA-1), `Signature` (the signature field joins the form) and `PdfMerger` (repeated names are renamed).

Filling and flattening existing PDFs

`PdfForm` reads, fills and flattens the form fields of any PDF, not only those PdfPecker made, such as a government form or a contract template from another tool:

```
// The fields, by fully qualified name ("address.city" for a nested one).
foreach (PdfFormField field in PdfForm.GetFields(pdf))
    Console.WriteLine($"{field.Name} ({field.Type}) = {field.Value}, choices: {string.Join(",
", field.Options)}");

var values = new Dictionary<string, string>
{
    ["name"] = "Jane Doe",
    ["address.city"] = "Copenhagen",
    ["newsletter"] = "true",       // a check box: its on state, or true/yes/on/1;
Off/false/no/0/"" clears it
    ["size"] = "M",                // a radio button group: the state name of one button
    ["country"] = "dk",            // a combo box or list box: a choice's value or its text
};
byte[] filled = PdfForm.Fill(pdf, values);

// Fill and flatten in one go, so the values can no longer be changed; or flatten a PDF as it
is.
byte[] final = PdfForm.Fill(pdf, values, new PdfFormFillOptions { Flatten = true });
byte[] flat = PdfForm.Flatten(filled);
PdfForm.FillFile("application.pdf", "application-filled.pdf", values);
```

`PdfFormField` gives each field's `Type` (`Text`, `CheckBox`, `RadioButton`, `ComboBox`, `ListBox`, `PushButton` or `Signature`), its `Value` (and `Values` for a list box with several choices), the `Options` it accepts, its flags (`ReadOnly`, `Required`, `Multiline`, `MaxLength`, ...), its tooltip and the page it is on. A name that is no field, a choice the field does not have, or text longer than `MaxLength` throws `ArgumentException`; set `IgnoreUnknownFields` to skip names the PDF does not have. Push buttons and signature fields cannot be filled (sign the latter with `PdfSigner`).

Every filled field gets a new appearance in the field's own font, size, colour and alignment, with automatic font size, multi-line text and comb fields laid out, and the box and background of its old appearance kept, so the values show in every viewer and in `PdfToImage`. When the field's font cannot show a value, a subset of `PdfFormFillOptions.Font` (by default the bundled Liberation Sans, which covers Latin, Greek and Cyrillic) is embedded for it. XFA form data, which would override the filled fields in some viewers, is removed.

A signed or encrypted PDF is filled with an incremental update: the original bytes stay as they were, so the signatures stay valid (when their permissions allow form filling) and the encryption is kept; pass the password in `PdfFormFillOptions.Password`. Other PDFs are written out in full, without the old values in earlier revisions. Flattening draws each field into its page and removes it, keeping links, comments and other annotations; it changes the pages, so a signed PDF is refused with `InvalidOperationException`.

Digital signatures

> **Changed in 1.24.0:** Verifying signatures.

Sign the PDF with a certificate (a `.pfx` file or one from the certificate store). Adobe Reader then shows who signed it and whether the file was changed afterwards. The key can also stay in an HSM, smart card or cloud key vault (external signers), and the signatures of any PDF can be verified and made verifiable for years.

```
var cert = X509CertificateLoader.LoadPkcs12FromFile("signer.pfx", "pfx-password");
var options = new PdfOptions
{
    Signature = new PdfSignatureOptions(cert)
    {
        Reason = "Approved",
        Location = "Copenhagen",
        // Optional: a visible box (points from the top-left corner; Page = -1 is the last
page).
        Appearance = new PdfSignatureAppearance { Page = -1, X = 340, Y = 720, Width = 220,
Height = 70 },
        // Optional: a trusted RFC 3161 timestamp (PAdES B-T). The only option that uses the
network.
        TimestampUrl = "http://timestamp.digicert.com",
    },
};

// Or sign any existing PDF; earlier signatures stay valid.
byte[] signed = PdfSigner.Sign(File.ReadAllBytes("in.pdf"), new PdfSignatureOptions(cert));

// A contract template with empty signature fields: sign into one by name, at its place on the
page.
byte[] countersigned = PdfSigner.Sign(contract, new PdfSignatureOptions(cert) { FieldName =
"Buyer" });
```

Signing into an existing field

When `FieldName` names an empty signature field the PDF already has (`"Parties.Buyer"` for a nested one), the signature goes into it and the box fills the field's rectangle; signing a field of another type or one that is already signed throws `PdfSignatureException`. Any other name adds a new field.

A template can make the empty field itself: an element with `data-pdf-signature="Buyer"` becomes an empty signature field over its border box, which `PdfSigner` signs later with `FieldName = "Buyer"`. The element is drawn as usual, such as a box with a line and a caption, and in a tagged PDF the field is tagged as PDF/UA asks.

```
<div data-pdf-signature="Buyer" style="width: 60mm; height: 20mm; border-bottom: 1px
solid">Buyer</div>
```

Formats and compatibility

Signatures are PAdES baseline (`ETSI.CAdES.detached`) by default, or classic `adbe.pkcs7.detached` with `Format = PdfSignatureFormat.Pkcs7Detached`; RSA and ECDSA keys with SHA-256/384/512. The signature is appended as an incremental update, and it works together with `Security` (encrypted files) and `Conformance` (PDF/A: the visible box is drawn as outlines, so it needs no unembedded font). The CMS structure is built with .NET's own `System.Formats.Asn1` and `System.Security.Cryptography`; no extra package is needed. Signing adds a few milliseconds (the RSA or ECDSA operation and re-reading the file); conversions without a signature are unaffected.

External signers: HSMs, smart cards and key vaults

When the private key cannot leave its hardware security module, smart card, PKCS#11 token or cloud key vault, pass a `PdfExternalSigner` with the certificate. PdfPecker builds the PDF and the CMS structure, and asks your callback only for the raw signature value. The certificate needs only its public part. The callback gets a `PdfSignatureRequest` with `Digest` for "sign hash" APIs (Azure Key Vault `SignAsync`, AWS KMS `Sign` with `MessageType.DIGEST`, `RSA.SignHash`) and `Data` for APIs that hash themselves; both give the same signature.

```
// The certificate without its private key; the key stays in Azure Key Vault.
var cert = X509CertificateLoader.LoadCertificateFromFile("signer.cer");
var keyVault = new CryptographyClient(keyId, new DefaultAzureCredential());

var signer = new PdfExternalSigner(PdfSignatureAlgorithm.RsaPkcs1, async (request, ct) =>
    (await keyVault.SignAsync(SignatureAlgorithm.RS256, request.Digest.ToArray(),
ct)).Signature);
byte[] signed = await PdfSigner.SignAsync(pdf, new PdfSignatureOptions(cert, signer) { Reason
= "Approved" });

// A synchronous callback, here with a key .NET can use directly (such as one from a PKCS#11
wrapper).
var local = new PdfExternalSigner(PdfSignatureAlgorithm.RsaPss,
    request => rsa.SignHash(request.Digest.ToArray(), request.HashAlgorithm,
RSASignaturePadding.Pss));
byte[] signed2 = PdfSigner.Sign(pdf, new PdfSignatureOptions(cert, local));
```

`PdfSignatureAlgorithm` is `RsaPkcs1` (RS256 and the like), `RsaPss` (PS256, with a salt as long as the hash) or `Ecdsa` (ES256, DER-encoded or raw r||s); the hash is `PdfSignatureOptions.HashAlgorithm`, SHA-256 by default. The returned signature is checked against the certificate before it goes into the PDF, so a wrong key or padding throws `PdfSignatureException` instead of producing a broken file. The callback is normally called once per signature (twice in the rare case the signature does not fit the space reserved for it). `PdfSigner.SignAsync` awaits an asynchronous callback and the timestamp server instead of blocking a thread; `PdfSigner.Sign` and signing during `HtmlToPdf.Convert` with `PdfOptions.Signature` wait for it. Everything else, visible boxes, timestamps and existing fields, works as with a certificate's own key.

Certification signatures

The first signature of a document can certify it: it states what may change afterwards, and Acrobat shows a certified document with a blue ribbon and flags changes the level does not allow. `PdfCertificationLevel.NoChanges` allows none, not even another signature; `FormFilling` allows filling in fields and signing; `FormFillingAndAnnotations` also allows adding, changing and deleting comments.

```
// The issuer certifies the contract; the parties may still fill in its fields and sign.
byte[] certified = PdfSigner.Sign(contract, new PdfSignatureOptions(issuerCert)
{
    Certification = PdfCertificationLevel.FormFilling,
});
byte[] signed = PdfSigner.Sign(certified, new PdfSignatureOptions(buyerCert) { FieldName =
"Buyer" });
```

Signing refuses what would break a certification: certifying a document that is already signed, any signature on a `NoChanges` document, and a new visible signature box on a certified one (an invisible signature or a field the document was certified with is fine). The verifier reports `CertificationLevel` and the `DisallowedChanges` made after a certification, and a certification with disallowed changes is not `Valid`. Changes that hide in the cross-reference table count too: an update that only points an object back at an older body still in the signed bytes (the "hide and replace" shadow attack), deleted objects, a table that had to be rebuilt (viewers follow the table and can show a different document), and an update that drops `/Perms` to turn the certification into an ordinary signature.

Verifying signatures Changed in 1.24.0

`PdfSignatureVerifier` checks the signatures and document timestamps of any PDF, oldest first, and reports what it finds rather than throwing for a broken signature:

```
foreach (PdfSignatureVerification s in PdfSignatureVerifier.Verify(pdf))
{
    Console.WriteLine($"{s.FieldName}: signed by {s.SignerName} at {s.TimestampTime ??
s.SigningTime}");
    Console.WriteLine($"  valid: {s.Valid}, intact: {s.IsIntact}, covers the whole file:
{s.CoversWholeDocument}");
    foreach (string problem in s.Problems) Console.WriteLine("  " + problem);
}

// Trust only your own CA instead of the operating system's roots, and check revocation
online.
var results = PdfSignatureVerifier.VerifyFile("contract.pdf", new
PdfSignatureVerificationOptions
{
    TrustedRoots = new X509Certificate2Collection(companyRootCert),
    RevocationMode = X509RevocationMode.Online,
});
```

`IsIntact` means the signed bytes are unchanged (`DigestMatches`), the byte range is well formed (`ByteRangeValid`), the signature value verifies with the signer's certificate (`SignatureValid`) and an embedded timestamp, if any, is sound (`TimestampValid`). `Valid` adds that the certificate chains to a

trusted root (`ChainTrusted`, with `ChainErrors` saying why not). Chains are built from the certificates in the signature, the PDF's Document Security Store and `ExtraCertificates`. Verification is offline by default: `RevocationMode` is `NoCheck` unless you set it, and `VerificationTime` checks the chain at another time.

Each signature is read from the revision of the file it signed: its certification level, reason, location, name and signing time come from there, so a later update cannot strip a certification, and a signature dictionary that a later update wrote again or replaced is listed in `DisallowedChanges`. The byte range is checked in encrypted files too.

A signature that hashes with SHA-1 (as its digest, in its signature algorithm, or in the old `adbe.pkcs7.sha1` format) is not `Valid`: SHA-1 collisions can be computed, so a document someone was led to sign can be swapped for another with the same hash. `UsesSha1` says so and `Problems` lists it, and a SHA-1 timestamp is not trusted. Set `AllowSha1` in `PdfSignatureVerificationOptions` to accept them, for an old archive say. Signing refuses SHA-1.

Trusted timestamps

The chain is checked at the time of the signature's timestamp only when `TimestampTrusted` is true: the timestamp authority's certificate chains to the same trusted roots as the signer's and carries the timestamping key usage. Anyone can make a token that states any time, so an untrusted timestamp no longer lets an expired signer certificate pass; the chain is then checked now, and the untrusted timestamp is listed in `Problems` even when the signature is valid without it. The signing time the signer claims is never used. A signer certificate whose key usage allows neither digital signatures nor non-repudiation is not trusted, and a document timestamp's certificate must carry the timestamping usage. Add your timestamp authority's root to `TrustedRoots` when you trust only your own CA.

A signature followed by later changes stays valid, because that is how a second signature or validation data is added: `UpdatesAfterSigning` counts the incremental updates after it, and `CoversWholeDocument` is false. What those updates change is not analysed (see Limitations), so check `CoversWholeDocument` on the last signature when nothing may follow it. PAdES, `adbe.pkcs7.detached`, `adbe.pkcs7.sha1` and document timestamps (`ETSI.RFC3161`, `IsDocumentTimestamp`) are read, from PDFs made by any tool.

Long-term validation

A signature can normally be checked only while its certificates are valid and their CA answers. `PdfSigner.AddValidationData` makes it checkable for years: it adds a Document Security Store with every certificate in the chains of the PDF's signatures and timestamps, and the OCSP responses and CRLs that show none was revoked (PAdES B-LT). With `TimestampUrl` a document timestamp follows, which seals the signatures and the validation data with a trusted time (PAdES B-LTA). Both are incremental updates, so the signatures stay valid.

```
byte[] signed = PdfSigner.Sign(pdf, new PdfSignatureOptions(cert) { TimestampUrl =
"http://timestamp.digicert.com" });

byte[] ltv = PdfSigner.AddValidationData(signed, new PdfLtvOptions
{
    FetchRevocationData = true,                        // ask each certificate's OCSP
responder, else its CRL
    TimestampUrl = "http://timestamp.digicert.com",    // PAdES B-LTA
});

// Offline: only the chains, completed with your own certificates and revocation data.
byte[] offline = PdfSigner.AddValidationData(signed, new PdfLtvOptions
{
    Certificates = [intermediateCert, rootCert],
    OcspResponses = [ocspResponseDer],
});
```

Without `FetchRevocationData` or `TimestampUrl` nothing goes over the network; OCSP and CRL requests use only the http and https addresses in the certificates, through `HttpClient` if you set one. Those addresses come from the PDF, which may be hostile. Without a `UrlFilter`, URLs that name `localhost` or an internal address are skipped, redirects included, and the default client checks the addresses a host name resolves to when it connects, so a certificate cannot make your server call its own network. With a `UrlFilter`, the filter alone decides: it is asked about every URL, redirect targets included, and returning false skips it, so internal OCSP or CRL servers can be allowed.

```
var options = new PdfLtvOptions
{
    FetchRevocationData = true,
    UrlFilter = url => url.Host.EndsWith(".digicert.com", StringComparison.OrdinalIgnoreCase),
};
```

Whatever the filter says, one call contacts at most 32 distinct URLs (64 requests), follows at most 3 redirects per CRL download (never from https to http), bounds slow responses and stops fetching after two minutes in total. An `HttpClient` whose handler follows redirects itself does so without asking the filter. To make the document timestamp itself long-term verifiable, call `AddValidationData` again later without a `TimestampUrl`. A PDF without signatures throws `PdfSignatureException`.

`Sign` and `AddValidationData` take a `CancellationToken`, and `AddValidationDataAsync` awaits the OCSP, CRL and timestamp requests instead of blocking a thread. `HtmlToPdf.ConvertAsync` signs asynchronously with the caller's token. Cancelling throws `OperationCanceledException`, and an unreadable PDF throws `InvalidDataException`.

E-invoices

Invoice XML from invoice data

`InvoiceData` describes an invoice by the European e-invoicing standard EN 16931: the seller and buyer, the lines, allowances and charges, payment and references. PdfPecker calculates the totals and the VAT breakdown from the lines and writes the invoice as XML, so you do not need a separate e-invoicing library:

```
var invoice = new InvoiceData
{
    Number = "INV-2026-0042",
    IssueDate = new DateOnly(2026, 10, 13),
    DueDate = new DateOnly(2026, 11, 12),
    BuyerReference = "04011000-12345-34",
    Seller = new InvoiceParty
    {
        Name = "Example Seller GmbH",
        VatId = "DE123456789",
        Address = new InvoiceAddress { Line1 = "Musterstraße 1", City = "Berlin", PostalCode =
"10115", CountryCode = "DE" },
        ElectronicAddress = "DE123456789", ElectronicAddressScheme = "9930",
    },
    Buyer = new InvoiceParty
    {
        Name = "Example Buyer AG",
        Address = new InvoiceAddress { Line1 = "Beispielweg 2", City = "München", PostalCode =
"80331", CountryCode = "DE" },
        ElectronicAddress = "04011000-12345-34", ElectronicAddressScheme = "0204",
    },
    Payment = new InvoicePayment { Iban = "DE02 1203 0000 0000 2020 51", Reference = "INV-
2026-0042" },
};
invoice.Lines.Add(new InvoiceLine { Name = "Consulting", Quantity = 10, UnitCode = "HUR",
NetPrice = 100, VatRate = 19 });

byte[] facturX   = invoice.ToCii();                              // Factur-X / ZUGFeRD EN 16931
byte[] xrechnung = invoice.ToCii(EInvoiceProfile.XRechnung);    // XRechnung 3.0 CII
byte[] peppol    = invoice.ToUbl();                              // Peppol BIS Billing 3.0 UBL
byte[] pdf = HtmlToPdf.Convert(invoiceHtml, new PdfOptions { EInvoice = invoice.ToEInvoice()
});  // Factur-X PDF
```

- `ToCii()` writes Factur-X / ZUGFeRD EN 16931 CII, and `ToCii(EInvoiceProfile.XRechnung)` XRechnung 3.0 CII.
- `ToUbl()` writes Peppol BIS Billing 3.0 UBL, an Invoice or, for `InvoiceType.CreditNote`, a CreditNote; `ToUbl(UblFormat.XRechnung)` writes XRechnung 3.0 UBL. Attach the PDF with `UblInvoice.AttachPdf` (see below). Peppol needs real participant IDs as electronic addresses, with an EAS scheme such as 0184, 0088 or 9930; an email address (EM) is not accepted.
- `ToEInvoice()` returns a `PdfEInvoice` for `PdfOptions.EInvoice`, so one conversion makes the Factur-X PDF.
- `CalculateTotals()` returns the `InvoiceTotals`: line total, allowances, charges, net total, the VAT breakdown, grand total and amount due. Money is rounded to two decimals, half away from

zero, as EN 16931 expects.

- `Validate()` (with an `EInvoiceProfile` or a `UblFormat`) lists the problems with their EN 16931 rule numbers, plus XRechnung's BR-DE and the Peppol rules for those formats, or an empty list. `ToCii`, `ToUbl` and `ToEInvoice` throw `ArgumentException` with every problem in the message. The checks cover the fields the model has; they do not replace the receiver's own validator.
- VAT categories are standard (S), zero rated (Z), exempt (E, which needs a reason in `VatExemptionReasons`), reverse charge (AE), intra-community supply (K), export (G), outside scope (O), and the Canary Islands (L) and Ceuta and Melilla (M) taxes. Reverse charge, intra-community supply, export and outside scope get the standard EU wording and VATEX code when no reason is given.
- The same invoice always gives the same bytes. Only the EN 16931 and XRechnung profiles are written; MINIMUM, BASIC WL, BASIC and EXTENDED throw.

The XML passes the Mustang validator (CII and UBL, with XRechnung) and the Peppol BIS 3.0 schematrons.

Ready-made invoice template

`InvoiceTemplate` turns an `InvoiceData` into a print-ready A4 invoice, with the Factur-X EN 16931 XML embedded by default (which makes the file PDF/A-3b):

```
byte[] pdf = InvoiceTemplate.ToPdf(invoice, new InvoiceTemplateOptions
{
    Logo = File.ReadAllBytes("logo.png"),
    AccentColor = "#0b6e4f",
    Labels = InvoiceLabels.German,
});

string html = InvoiceTemplate.ToHtml(invoice);   // the HTML, to adapt and convert yourself
```

Option	Default	What it does
`Logo`	null	Your logo as PNG, JPEG, GIF, WebP, BMP or SVG bytes, drawn at most 180 by 64 pixels.
`LogoUri`	null	The logo as a data URI, or a URL your `ResourceResolver` loads; used when `Logo` is null.
`AccentColor`	`#1f4e79`	The colour of the title, headings and table header, as CSS.
`Labels`	`InvoiceLabels.English`	The texts, number and date formats: start from `English`, `German` or `Danish` and change any text, the decimal and group separators, the date format and the unit names.
`Css`	null	Extra CSS after the template's own, to change fonts, spacing or anything else.
`PaymentQrCode`	true	Prints an EPC payment QR code (GiroCode) next to the payment details for euro invoices paid to an IBAN.
`EInvoice`	`EInvoiceProfile.En16931`	The e-invoice `ToPdf` embeds: `En16931`, `XRechnung`, or null for a plain PDF.
`PdfOptions`	null	The PDF settings to start from, such as fonts, metadata or a signature. They are copied, not changed; page size and margins come from the template's CSS.

Long invoices continue on more pages with the table header repeated. The labels need no `CultureInfo`, so the template also works in apps that run with invariant globalization.

Factur-X and ZUGFeRD

Pass the invoice's CII XML (from your invoicing system, or from `InvoiceData.ToCii()`) and PdfPecker attaches it to the PDF, making a Factur-X / ZUGFeRD e-invoice: the pages are the human-readable invoice and the XML is what accounting software reads.

```
var options = new PdfOptions
{
    EInvoice = new PdfEInvoice(File.ReadAllBytes("invoice.xml")),
};
File.WriteAllBytes("invoice.pdf", HtmlToPdf.Convert(invoiceHtml, options));
```

The file is PDF/A-3b, so `Conformance` is raised to `PdfA3b` when it is left at `None`; `PdfA3u` and `PdfA4f` (PDF/A-4 with attachments) work too, while the other levels and `Security` throw. Factur-X and ZUGFeRD specify PDF/A-3, so keep the default unless the receiver asks for PDF/A-4f. The XML is embedded unchanged as `factur-x.xml` (`xrechnung.xml` for XRechnung), listed as an associated file, and the XMP metadata gets the Factur-X extension schema with the document type, version and profile. The profile (MINIMUM, BASIC WL, BASIC, EN 16931, EXTENDED or XRECHNUNG) is read from the XML's guideline ID; pass it as the second constructor argument to set it yourself. The relationship to the pages defaults to `Data` for MINIMUM and BASIC WL and `Alternative` otherwise; set `Relationship` to change it. PdfPecker checks that the XML is a well-formed Cross Industry Invoice (UBL is not allowed in Factur-X) but does not validate XML you pass in yourself: make sure the visible pages show the same invoice. XML from `InvoiceData` is checked before it is written. The output passes the Mustang validator (PDF/A-3b via veraPDF, and the XML's schematron). Other files can be attached next to the invoice with `Attachments` (see Attachments).

Reading the invoice from a received PDF

`PdfEInvoice.Read` gets the XML back out of a Factur-X, ZUGFeRD 2.x or XRechnung PDF, from PdfPecker or any other producer, so the accounting side can read what the pages show:

```
PdfEInvoice? invoice = PdfEInvoice.Read(File.ReadAllBytes("received.pdf"));
if (invoice is not null)
{
    File.WriteAllBytes("invoice.xml", invoice.Xml);
    Console.WriteLine(invoice.Profile);   // e.g. En16931
}
```

The invoice is the attachment the XMP metadata names, else one called `factur-x.xml`, `zugferd-invoice.xml` or `xrechnung.xml`, else any other attached Cross Industry Invoice. The XML comes back unchanged with its `Profile` (from the XML's guideline, else the XMP conformance level) and `Relationship`; PdfPecker does not validate it. The result is null when the PDF carries no invoice. ZUGFeRD 1.0 uses an older schema and is not found, but PdfAttachments.List still returns its file. Pass `password` for an encrypted PDF.

PDF attachments in UBL e-invoices (Peppol, OIOUBL)

Peppol BIS Billing 3.0, OIOUBL, EHF, XRechnung UBL and other UBL e-invoices are XML only: the XML is the invoice, and the PDF travels inside it as an attachment. `UblInvoice.AttachPdf` adds one to the invoice or credit note your invoicing system made:

```
byte[] pdf = HtmlToPdf.Convert(invoiceHtml);
byte[] xml = UblInvoice.AttachPdf(File.ReadAllBytes("invoice.xml"), pdf, "INV-1001.pdf");
```

The PDF goes in as an `AdditionalDocumentReference` with `EmbeddedDocumentBinaryObject` (`mimeCode="application/pdf"` and the file name), after any attachments already there and where the UBL schema expects it. The rest of the XML, including its indentation, encoding and declaration, is kept as it was. The reference ID defaults to the file name; pass `id` and `description` to set them. For OIOUBL 2.1 (UBL 2.0) the description is written as the `DocumentType` OIOUBL requires, `PDF` by default. The output passes the UBL schemas, the Peppol BIS 3.0 and EN 16931 schematrons, and the OIOUBL 2.1 schematron. PdfPecker does not validate the rest of an invoice you pass in; to build the UBL itself, use `InvoiceData.ToUbl()` (see above).

Barcodes

> **Changed in 1.24.0:** Reading barcodes.

`Barcode` draws QR codes, EPC payment QR codes, Code 128 and EAN-13 barcodes as SVG markup for your HTML template. The SVG stays vector graphics in the PDF, so the code scans at any print size. Like the rest of PdfPecker it needs no other package.

```
// Inline SVG in the template
string qr = Barcode.QrCode("https://example.com/invoice/1042");
string html = $"<p>Scan to pay online:</p>{qr}";

// Or as an image, a CSS background or a Markdown image
string src = Barcode.ToDataUri(Barcode.Ean13("590123412345"));
string img = $"<img src=\"{src}\" style=\"width:40mm\">";

// Code 128 for order numbers, shipping labels and other ASCII text
string label = Barcode.Code128("ORDER-2026-1042", new BarcodeOptions { Height = 40, ShowText =
false });
```

QR codes

`Barcode.QrCode` picks the smallest QR code (versions 1 to 40) that holds the text at the chosen error correction. Digits only and upper-case alphanumeric text are packed densely, and anything else is stored as UTF-8, up to 2,953 bytes at the lowest level. `BarcodeOptions.ErrorCorrection` is `Low` (about 7 percent of the code may be damaged), `Medium` (15 percent, the default), `Quartile` (25 percent) or `High` (30 percent, for codes printed small or with a logo on top). Text that does not fit throws `ArgumentException`.

EPC payment QR codes

`Barcode.EpcQrCode` draws the European Payments Council's QR code for a euro credit transfer, also called GiroCode, which banking apps read to fill in the payment. Set `Name` and `Iban` on an `EpcPayment`; the rest is optional.

```
string pay = Barcode.EpcQrCode(new EpcPayment
{
    Name = "Example Ltd",
    Iban = "DE89 3704 0044 0532 0130 00",
    Amount = 129.95m,
    Text = "Invoice 1042",
});
```

The payment follows guideline EPC069-12 (version 002, UTF-8) and is checked before drawing: the IBAN's check digits, an 8 or 11 character `Bic`, an `Amount` from 0.01 to 999,999,999.99 with at most two decimals (or null to let the payer enter it), a four-letter `Purpose` code, and either a structured creditor `Reference` (ISO 11649, "RF...") or a `Text` of up to 140 characters, not both. `Information` is a

note shown to the payer. A field that is missing, too long or invalid throws `ArgumentException`. The code always uses error correction level M, as the guideline requires. `ToPayload()` returns the text the code holds.

Swiss QR-bills

`Barcode.SwissQrCode` draws the QR code of a Swiss QR-bill, with the Swiss cross in the middle, from a `SwissQrBill`: the CH or LI IBAN of the creditor, structured addresses, the amount and currency (CHF or EUR), and a QR reference with its check digit for a QR-IBAN, or an ISO 11649 creditor reference or none for an ordinary IBAN. Every field is checked against the standard (guidelines version 2.3), and a wrong one throws `ArgumentException`; `ToPayload()` gives the text inside the code.

```
string qr = Barcode.SwissQrCode(new SwissQrBill
{
    Account = "CH58 0079 1123 0008 8901 2",
    Creditor = new SwissQrAddress { Name = "Robert Schneider AG", Street = "Rue du Lac",
HouseNumber = "1268",
                                    PostalCode = "2501", Town = "Biel" },
    Amount = 1949.75m,
    Reference = "RF18539007547034",
});
```

PdfPecker.Fluent lays out the whole payment part with its receipt with `SwissQrBill`.

Data Matrix

`Barcode.DataMatrix` draws an ECC 200 Data Matrix code in the smallest of the 24 square sizes that holds the text, the usual code for small labels, where a QR code of the same data is larger. `Barcode.Gs1DataMatrix` takes a GS1 element string with the application identifiers in parentheses, as shipping and pharmaceutical labels use:

```
string label = Barcode.Gs1DataMatrix("(01)09501101530003(17)261231(10)AB-123");
```

Code 128 and EAN-13

`Barcode.Code128` takes 1 to 1,000 ASCII characters and packs runs of digits two to a symbol. `Barcode.Ean13` takes 12 digits and adds the check digit, or 13 digits with a correct check digit, and draws the retail barcode with its guard bars. Both print the text below the bars unless `ShowText` is false.

Options

`BarcodeOptions` sets:

- `ModuleSize`: the width of the narrowest bar or one QR square in CSS pixels, which sets the SVG's size. The default is 3 for QR codes and 1.5 for barcodes. CSS `width` on the SVG or image scales it too.

- `Height`: the bar height of Code 128 and EAN-13 in CSS pixels; the default is 60 modules.
- `QuietZone`: the blank margin scanners need around the code (4 modules for QR, 10 for Code 128, 11 and 7 for EAN-13). On by default; turn it off only when the layout leaves that space.
- `Color` and `Background`: CSS colours, `#000` on `#fff` by default. Set `Background` to null for none, but keep the code dark on light, since scanners need the contrast.

Reading barcodes New in 1.24.0

`Barcode.Read` reads the QR codes, Data Matrix codes (square, rectangular and the DMRE sizes), Code 128 and EAN-13 barcodes in a PNG, JPEG, GIF, WebP, BMP or TIFF image, and `PdfBarcodes.Find` those on the pages of a PDF, whether they are drawn as vectors, embedded as images or scanned. Codes may be turned, tilted, mirrored or light on dark.

```
foreach (BarcodeResult code in Barcode.ReadFile("delivery-note.png"))
    Console.WriteLine($"{code.Format}: {code.Text}");

// Every code on the pages of a PDF, with its page and position in points from the top left
foreach (BarcodeResult code in PdfBarcodes.Find(pdfBytes, new PdfBarcodeOptions { Formats =
BarcodeFormat.QrCode }))
    if (code.EpcPayment is { } payment)
        Console.WriteLine($"Page {code.PageNumber}: {payment.Amount} EUR to {payment.Iban}");
```

Each `BarcodeResult` has its `Format`, its `Text` and `Bytes`, its `Corners` and bounding box (in pixels for an image, in points on the page for a PDF), `IsGs1` for GS1 data, and the payment of an EPC or Swiss QR-bill code parsed back into `EpcPayment` or `SwissQrBill`. `BarcodeReaderOptions` and `PdfBarcodeOptions` choose the `Formats`, whether to `TryInverted` light-on-dark codes and how many codes to return; `PdfBarcodeOptions` also takes a `Password`, `Pages` and the `Dpi` pages are rendered at (300 by default). Images and PDFs are handled as untrusted input, with a bounded amount of work per image.

Attachments

Any file can travel inside the PDF, listed in the viewer's attachments panel: the source spreadsheet of a report, the data behind a chart, a signed contract.

```
var options = new PdfOptions { Conformance = PdfConformance.PdfA3b }; // optional
options.Attachments.Add(new PdfAttachment("figures.csv", csvBytes)
{
    Description = "The figures behind the charts",          // default: the file name
    Relationship = PdfAttachmentRelationship.Data,           // Source, Data, Alternative,
Supplement or Unspecified
    // MimeType = "text/csv",                                // default: guessed from the
extension
    // CreationDate = ..., ModDate = ...                     // default: ModDate is the time
of conversion
});
options.Attachments.Add(PdfAttachment.FromFile("contract.pdf"));
byte[] pdf = HtmlToPdf.Convert(html, options);

// Read them back, or attach files to a PDF made elsewhere.
foreach (PdfAttachment a in PdfAttachments.List(pdf)) Console.WriteLine($"{a.FileName}:
{a.Data.Length} bytes");
byte[]? csv = PdfAttachments.Extract(pdf, "figures.csv");
byte[] more = PdfAttachments.Add(File.ReadAllBytes("scan.pdf"), new PdfAttachment("notes.txt",
notes));
```

How files are stored

Each file is written with its MIME type, size, dates and MD5 checksum, in the document's embedded-files list sorted by name, and as an associated file (`/AF`) with its relationship to the pages, as PDF/A-3 requires. File names must be unique (`ArgumentException` otherwise; an attachment can't take the e-invoice's `factur-x.xml`). Attachments don't change `Conformance`: without one the file is a plain PDF, PDF/A-3b, PDF/A-3u and PDF/A-4f allow any attachment, and PDF/A-2b, PDF/A-2u and PDF/A-4, which allow only PDF/A files as attachments, throw `ArgumentException` rather than produce a file that fails validation. With `Security` the attachments are encrypted with the rest of the file, and for PDF/UA each one gets the description ISO 14289 asks for. The output passes veraPDF's PDF/A-3b, PDF/A-3u, PDF/A-4f, PDF/UA-1 and PDF/UA-2 checks. `PdfAttachments.Add` likewise refuses PDF/A-1, PDF/A-2 and PDF/A-4 files, and adds to PDF/A-3 and PDF/A-4f files.

Adding to an existing PDF

`PdfAttachments.Add` appends the files as an incremental update, so the original bytes, and any digital signatures over them, stay as they were; an encrypted PDF (pass its password) stays encrypted, a PDF/A-3 file gets the new associated files, and PDF/A-1 and PDF/A-2 files are refused. `List` and `Extract` read the embedded-files list and associated files of any PDF, decrypting with the password you pass; files attached to page annotations (the paper-clip icons) are not read.

Merging and splitting PDFs

Merging

`PdfMerger` combines PDFs into one, any PDFs, not only those PdfPecker made:

```
byte[] merged = PdfMerger.Merge(invoicePdf, termsPdf, attachmentPdf);

// With passwords for encrypted inputs (by position), and properties for the result.
byte[] merged2 = PdfMerger.Merge(new[] { a, b }, new PdfMergeOptions
{
    Passwords = new string?[] { null, "secret" },
    Title = "Payslips October",
    CustomProperties = { ["RunId"] = "42" },
});

PdfMerger.MergeFiles("all.pdf", new[] { "a.pdf", "b.pdf" });

// Streams in and out: inputs are read one at a time and the result is written as it is made.
PdfMerger.Merge(new[] { invoiceStream, termsStream }, outputStream);
await PdfMerger.MergeAsync(uploads.Select(f => f.OpenReadStream()), response.Body);
```

Tagged inputs stay tagged: their structure trees are joined under one root, so the merged file is still accessible, and when every input is PDF/UA-1 the result says so too (veraPDF passes it). The merged file is PDF 1.7, so PDF/UA-2 inputs stay tagged but the result claims no PDF/UA level. An input in another language than the first keeps its language on its own part of the tree. Untagged inputs merge as before.

With streams (and `MergeFiles`), only one input is held in memory at a time and the merged document goes to the output as each input is done, so memory follows the largest input rather than the whole batch: merging 100 files of 32 MB in total peaks at 14 MB of managed memory, against 105 MB with byte arrays. `MergeAsync` reads and writes asynchronously, as ASP.NET Core requires for request and response bodies. Input streams are read from their current position and are not disposed; the sequence is enumerated lazily, so an iterator can open each one only when its turn comes. `MergeFiles` writes to a temporary file and moves it into place, so the output can be one of the inputs and a failed merge leaves no partial file behind.

Merging PDFs as they arrive

For PDFs that arrive one at a time from elsewhere, such as downloads read a few at a time, use `StreamingPdfMerger`. It writes each PDF's pages when it is added and then lets the PDF go, so memory stays at about one input, and between inputs you decide what comes next: skip a missing document, or check that anything was added before answering. The output can be an HTTP response body or a file; it does not need to be seekable.

```
var merger = new StreamingPdfMerger(response.Body, new PdfMergeOptions { Title = "Payslips
October" });
foreach (var path in paths)
{
    var pdf = await storage.DownloadAsync(path, ct);
    if (pdf.Length == 0) continue;          // skip a missing document
    await merger.AddAsync(pdf, ct);
}
if (merger.InputCount == 0) return Results.NotFound();
await merger.CompleteAsync(ct);             // writes the page tree, bookmarks, form and
cross-reference table

// Or, from an IAsyncEnumerable<byte[]>:
int pages = await PdfMerger.MergeAsync(DownloadInOrder(paths), response.Body,
cancellationToken: ct);
```

The result is the same as `PdfMerger.Merge`'s, but the output is only a valid PDF once `Complete` has run. If adding a PDF throws (a damaged or encrypted input, `MaxPages` reached), what was written cannot be used and the merger refuses further calls. `Passwords` are matched to inputs in the order they are added.

The pages keep their content, fonts, images, links, annotations, layers and fillable form fields, and each input's bookmarks follow each other. A form field named like one in an earlier input is renamed `name_2`, `name_3` and so on, so the copies keep their own values. Document properties come from the first input unless `PdfMergeOptions` sets them. The result is never encrypted (pass it to `PdfProtector.Encrypt` to protect it) and has no signatures (merging would break them); PDF/A conformance is not carried over. `PdfMergeOptions.Viewer` and `PageLabels` set the viewer settings and page labels of the result, and `PdfLinearizer` makes it load fast on the web. Without `PageLabels`, the inputs' page labels are kept (pages of an input without them are numbered as in that input), and attachments too, with one named like an earlier input's listed as `name (2).ext`. Images, fonts and other resources that are byte-for-byte the same in several inputs, such as a logo on every payslip of a run, are written once.

Inputs are handled as untrusted: damaged files are repaired where possible, decoding is limited per input, and `MaxPages` (default 100,000) caps the result. Inputs without any pages throw `InvalidDataException`. Merging was tested on PDFs from Chrome (as IronPDF makes them), Ghostscript, LibreOffice, MuPDF, qpdf, ReportLab, pypdf incremental updates, scanners (JPEG, CCITT G4, CMYK, OCR text layers) and damaged files (wrong or missing cross-reference tables, truncated files, junk before the header); every merged page rendered pixel-identical to its original in Poppler. Merging four small documents takes about 1 ms, and 1,000 one-page Chrome payslips about 1 s.

Extracting, reordering and splitting pages

`PdfPages` works on the pages of any existing PDF. Pages are numbered from 1, and ranges are written like `"1-3,5,8-"` (pages 1 to 3, 5, and 8 to the end): `"3,1,2"` reorders, `"1,1"` repeats, `"5-3"` runs backwards and `"last"` is the last page. `PdfPageRange.Parse` reads a range once for reuse; a page past the end throws `ArgumentOutOfRangeException` ("Page 9 is out of range: the PDF has 5 pages.").

```
byte[] summary = PdfPages.Extract(report, "1-3,5");              // or new[] { 3, 1, 2 }
byte[] withoutCover = PdfPages.Remove(report, "1");
byte[] turned = PdfPages.Rotate(scan, 90, "2,4");                 // clockwise; null pages =
every page
byte[][] parts = PdfPages.Split(report, 10);                      // 10 pages each
byte[][] chapters = PdfPages.Split(report, "1-4", "5-9", "10-");
int pages = PdfPages.Count(report);

// Encrypted input, files, streams.
byte[] page = PdfPages.Extract(secret, "2", new PdfPagesOptions { Password = "pw", Title =
"Page 2" });
PdfPages.ExtractFile("report.pdf", "summary.pdf", "1-3");
PdfPages.SplitFile("report.pdf", 1, n => $"report-{n}.pdf");

// Selected pages of several PDFs merged into one (also StreamingPdfMerger.Add(pdf, pages)).
byte[] merged = PdfMerger.Merge(new[] { new PdfMergeInput(invoice, "1"), new
PdfMergeInput(terms, "2-") });
```

For the pages taken, the result keeps their content, links and other annotations, the form fields with a widget on them, the bookmarks that go to them, page labels, the document properties and language. Links and bookmarks to pages left out are dropped (a bookmark with a kept bookmark below it stays, without a destination); links between pages taken go to their new place. A tagged input stays tagged: structure elements are kept when some of their content is on a page taken, so a PDF/UA-1 input gives a PDF/UA-1 result (veraPDF passes extracted, removed and split parts). A page taken more than once is copied with its links, but its form fields stay on the first copy and the later copies are not tagged, so such a result no longer claims PDF/UA. `Split` parses the PDF once for all parts. As with merging, the result is not encrypted and has no signatures.

Stamping PDFs

`PdfStamper` puts a page of one PDF on top of the pages of another, such as a "PAID" or "COPY" stamp, or underneath them, such as company letterhead. Any PDF can be stamped, not only those PdfPecker made:

```
// A one-page stamp PDF, fitted and centred on every page, half transparent.
byte[] paid = PdfStamper.Stamp(invoicePdf, paidStampPdf, new PdfStampOptions { Opacity = 0.5f
});

// Letterhead under the content: the stamp's first page on page 1, its second page on all the
others.
byte[] letter = PdfStamper.Stamp(letterPdf, letterheadPdf, new PdfStampOptions { Underlay =
true, MatchPages = true });

// Actual size in the top right corner, 20 pt in from each edge, on the first page only.
byte[] copy = PdfStamper.Stamp(pdf, copyStampPdf, new PdfStampOptions
{
    Position = PdfStampPosition.TopRight, Scale = 1, OffsetX = -20, OffsetY = 20, Pages = i =>
i == 0,
});

// The stamp as HTML, rendered at each page's size with no margins.
byte[] stamped = PdfStamper.StampHtml(pdf,
    "<div style='position:absolute; right:30pt; top:30pt; border:3pt solid #c00; color:#c00;
font:bold 28pt sans-serif'>PAID</div>");

PdfStamper.StampFile("in.pdf", "stamp.pdf", "out.pdf");
PdfStamper.Stamp(inputStream, stampStream, outputStream);
```

Placement

`StampHtml` renders the HTML with the options you pass but ignores their footer, header, watermark, attachments, bookmarks and tagging, so the main conversion's `PdfOptions` can be reused without drawing a second footer. `StampPage` picks the stamp PDF's page (0-based), `Pages` the pages to stamp (by 0-based index), `Scale` the size (0, the default, fits the stamp page on the page keeping its aspect ratio; 1 is its own size), and `Position`, `OffsetX` (right) and `OffsetY` (down) where it goes. The stamp is placed as the page is displayed, so it is upright on rotated pages and sits in the crop box. The stamp page is copied in once as a form XObject, so its text stays text and its graphics stay sharp, however many pages show it. The page's own content is wrapped so a graphics state it leaves behind cannot shift or recolour an overlay. Underlays are hidden by pages that paint an opaque background, and annotations (links, form fields) are always drawn above the page content, overlays included.

What is kept

Everything else in the document is kept: annotations, form fields and their values, bookmarks, named destinations, attachments, metadata, page labels and the structure tree of a tagged PDF, in which the stamp is marked as an artifact (a PDF/UA-1 document stamped this way still passes veraPDF, as does a PDF/A-2b one when the stamp has embedded fonts). The stamp page's own annotations, form fields

and tags are not copied. The result is written out in full, not as an incremental update. A signed PDF is refused with `InvalidOperationException`, since stamping would break the signatures; set `InvalidateSignatures = true` to stamp it anyway, or stamp before signing. Encrypted inputs open with `Password` (and `StampPassword` for the stamp); the result is not encrypted unless `Security` is set. Both inputs are handled as untrusted, like merge inputs: damaged files are repaired where possible, decoding is limited, and an unreadable file or one without pages throws `InvalidDataException`.

Compacting PDFs

`PdfCompactor` makes an existing PDF smaller, any PDF, not only those PdfPecker made. By default nothing visible changes: the pages, bookmarks, links, forms, tags, attachments and PDF/A and PDF/UA conformance are kept, and every image keeps its exact pixels.

```
// Lossless: the same pages, in fewer bytes.
byte[] smaller = PdfCompactor.Compact(File.ReadAllBytes("merged.pdf"));

// Smaller still: images above 150 dpi downsampled, and photos recompressed as JPEG at quality
75.
byte[] smallest = PdfCompactor.Compact(pdf, new PdfCompactOptions { MaxImageDpi = 150,
ImageQuality = 75 });

// Streams in and out; nothing is written when the input cannot be compacted.
PdfCompactor.Compact(inputStream, outputStream);
```

What it does

The document is written out again with only the objects its pages and catalog still use, so earlier revisions kept by incremental updates and orphaned objects go. Identical objects are stored once, such as the same font or image embedded many times, which merging documents often leaves behind. Streams with lossless filters are recompressed at the strongest Flate level, with PNG predictors for images, whenever that is smaller; JPEG, JPEG 2000, JBIG2 and CCITT images keep their data. Objects are written in minimal syntax and packed into compressed object streams.

The result depends only on the input and the options, so the same PDF always gives the same bytes. When rewriting does not make the file smaller, the input comes back unchanged.

Options

Option	Default	What it does
`ImageQuality`	null	Re-encodes 8-bit greyscale and RGB images as JPEG at this quality (1 to 100) where that makes them smaller. 75 to 85 suits most documents. This is lossy, also for images that were not JPEGs, such as screenshots and logos.
`MaxImageDpi`	null	Downsamples 8-bit greyscale and RGB images drawn at a higher resolution than this, judged by the largest size each image is drawn at on any page. A downsampled JPEG is re-encoded as JPEG (at `ImageQuality`, or 85); other images stay lossless unless `ImageQuality` is set. 150 suits the screen, 300 print.
`RemoveMetadata`	false	Removes the document information (title, author, dates) and XMP metadata. They are kept when the PDF claims PDF/A or PDF/UA, which require them.
`RemoveApplicationData`	true	Removes page thumbnails, which viewers draw themselves, and the private data editing applications keep in the file (`/PieceInfo`, such as Illustrator's editable copy), which no viewer shows.

Option	Default	What it does
`UseObjectStreams`	true	Packs objects into compressed object streams with a cross-reference stream (PDF 1.5, which every current reader supports). Turn it off for readers from before 2003. PDF/A-1 files never use them, as PDF/A-1 does not allow them.

An overload takes a `CancellationToken`, which stops the work between objects.

Signed and encrypted PDFs

Digitally signed PDFs are refused with `ArgumentException`, because rewriting them breaks the signatures. Compact before signing instead. Encrypted PDFs are refused too, since a rewrite would drop their protection: decrypt with `PdfProtector.Decrypt`, compact, and encrypt the result again with `PdfProtector.Encrypt` (see PDF/A, passwords and watermarks). A file that is not a readable PDF throws `InvalidDataException`.

Fast Web View

`PdfLinearizer` linearizes any existing PDF, such as a merged one, so a viewer loading it over the web shows the first page before the rest has arrived. New documents can be linearized as they are made with `PdfOptions.Linearize`.

```
byte[] web = PdfLinearizer.Linearize(PdfMerger.Merge(invoicePdf, termsPdf));
PdfLinearizer.Linearize(inputStream, outputStream);
```

Pages, bookmarks, links, forms, tags, metadata, attachments and PDF/A and PDF/UA conformance are kept, and the same input always gives the same bytes. Signed PDFs are refused, because rewriting them breaks the signatures, and so are encrypted ones, unless you give the password and the security to encrypt the result with in the same pass: `PdfLinearizer.Linearize(pdf, new PdfSecurity { UserPassword = "open" }, password: "old")` opens an encrypted input and encrypts the linearized result, which encrypting afterwards cannot do, since it would end the linearization. When converting, set `Linearize` together with `Security`. A signature added later keeps the file valid but ends its linearization, since it is an incremental update. `PdfDocumentProperties.Read` says whether a file is linearized (see Text extraction).

Removing active content

> **Changed in 1.24.0:** What it removes, The output.

PdfPecker never runs the scripts in a PDF, but whoever opens the PDF next might. When you merge, stamp, compact or pass on PDFs that come from others, `PdfSanitizer` takes out what a viewer could run, and keeps what people read.

```
// Strip JavaScript and other active content from a PDF someone sent you.
byte[] safe = PdfSanitizer.RemoveActiveContent(File.ReadAllBytes("received.pdf"));

// Attachments too: a viewer may open them, and an attachment can be a program.
byte[] safer = PdfSanitizer.RemoveActiveContent(pdf, new PdfSanitizeOptions {
RemoveAttachments = true });

// Only ask.
bool risky = PdfSanitizer.HasActiveContent(pdf);
```

What it removes Changed in 1.24.0

- JavaScript: document, page, field and link scripts.
- Actions that launch programs, open other files (including links to a network share), submit or import form data, or play media, and the additional actions that run when a page or field is opened or changed.
- XFA forms, and media and 3D annotations.
- Links to anything but absolute `http`, `https`, `mailto` and `tel` addresses: `javascript:`, `file:` and `data:` addresses, network paths and `smb:` addresses (which can send the reader's Windows credentials to the named server), protocol handlers such as `search-ms:` and `ms-msdt:`, and relative addresses, which the document's base address could turn into any of these.
- With `RemoveAttachments`, the document's attachments, its associated files and file attachment annotations. This also removes the XML of an e-invoice (ZUGFeRD, Factur-X).

Pages, text, images, bookmarks, links within the document, web and e-mail links, and form fields with their values stay.

The output Changed in 1.24.0

The document is written out again with only what its catalog still uses, so earlier revisions and unused objects go too, and the same input always gives the same bytes. A PDF without active content comes back as a copy, unless its cross-reference table was damaged and had to be rebuilt: a viewer repairs such a file its own way and might find what PdfPecker did not, so it is written out again from what PdfPecker read and checked. Digital signatures no longer verify after sanitizing, since the signed bytes change.

Encrypted PDFs are refused with an `ArgumentException`: decrypt them with `PdfProtector.Decrypt`, sanitize, and encrypt the result again (see PDF/A, passwords and watermarks). `HasActiveContent` takes a password for encrypted PDFs. There is also an overload with a `CancellationToken`, and one that reads from a stream and writes to another. The input is treated as untrusted.

Redaction

> **New in 1.24.0.**

`PdfRedactor` removes what the pages of any PDF draw in given areas, from the file itself rather than only from sight, and then draws a box over each area. Text in an area leaves the content streams, so neither text extraction nor copying finds it, the pixels of images under an area are overwritten, and shapes, links and form fields in an area go.

```
// Every personal ID number, wherever it appears.
byte[] redacted = PdfRedactor.RedactText(File.ReadAllBytes("contract.pdf"), new
Regex(@"\b\d{6}-\d{4}\b"));

// A phrase, compared without regard to case.
byte[] noName = PdfRedactor.RedactText(pdf, "Example Customer Ltd.", comparison:
StringComparison.OrdinalIgnoreCase);

// An area of a page, in points from its top-left corner, with no box drawn over it.
byte[] cut = PdfRedactor.Redact(pdf, new[] { new PdfRedactionArea(pageNumber: 1, x: 72, y:
100, width: 200, height: 40) },
                                new PdfRedactOptions { FillColor = null });
```

Reviewing before redacting

`FindText` returns the areas `RedactText` would remove, one per word, so they can be checked or shown first. Each line is searched as its words separated by single spaces, so a phrase matches across words but not across lines, and a match inside a word (an account number in "No.12345") takes the whole word. `PdfRedactionArea.Of` makes an area from a word that text extraction found, in the same coordinates.

```
IReadOnlyList<PdfRedactionArea> found = PdfRedactor.FindText(pdf, new Regex(@"IBAN \w+"));
byte[] result = PdfRedactor.Redact(pdf, found.Where(a => a.PageNumber > 1));
```

What is removed

- A glyph goes when an area covers a fifth or more of its width and height. The text around it keeps its place.
- A shape goes when it lies wholly inside an area. A shape that only crosses one, such as a table border, stays and is covered by the box.
- An image partly under an area is written again with its pixels there painted over, and one wholly under an area is removed. The originals of redacted images leave the file unless a page still shows them elsewhere.
- Links, form fields and other annotations that touch an area go.
- Forms (XObjects) drawn on several pages are redacted in a copy, so the other pages do not change.

- Replacement and alternative texts (`/ActualText`, `/Alt`) that held removed text are removed, so they cannot name what was there, and redacted pages lose their thumbnail.

The document is written out again with only what its catalog still uses, so earlier revisions, where an incremental update could keep the old page, and unused objects are gone, and the same input always gives the same bytes. Digital signatures no longer verify.

Text the document keeps elsewhere, such as its title and metadata, bookmarks and attachments, is not looked at: remove active content and attachments separately. Encrypted PDFs are refused with an `ArgumentException`: decrypt them with `PdfProtector.Decrypt`, redact, and encrypt the result again. There are overloads that read from a stream and write to another, and every call takes a `CancellationToken`. The input is treated as untrusted.

PDF to PDF/A

`PdfToPdfA` turns a PDF you already have, made by any tool, into PDF/A-2 or PDF/A-3 for long-term archiving. It fixes what it can, keeps the pages as they are (text stays text, nothing is rasterized), and tells you what it changed and what it could not fix. To make new documents as PDF/A, set `Conformance` instead (see PDF/A, passwords and watermarks).

```
var result = PdfToPdfA.Convert(File.ReadAllBytes("received.pdf"));
File.WriteAllBytes("archived.pdf", result.Pdf);

// What still breaks the PDF/A rules, one sentence each; empty when IsConformant is true.
if (!result.IsConformant)
    foreach (var problem in result.Problems) Console.WriteLine(problem);

// What the conversion changed, one sentence each, such as "Embedded 7 fonts that were not
// embedded, with glyphs from PdfPecker's bundled fonts at the document's widths (...)."
foreach (var change in result.Changes) Console.WriteLine(change);

// Another level, from one stream to another, with a cancellation token.
var options = new PdfAConversionOptions { Conformance = PdfConformance.PdfA3u };
PdfToPdfA.Convert(input, output, options, cancellationToken);
```

Level	Use it for
`PdfA2b` (the default)	the usual archive format
`PdfA2u`	as 2b, plus the Unicode text of every character, so the text can be searched and copied
`PdfA3b`, `PdfA3u`	as 2b and 2u, keeping every attachment

What it fixes

- **Fonts.** Fonts that are not embedded get glyphs from PdfPecker's bundled Liberation and DejaVu fonts at the document's own character widths, so the text keeps its place. Embedded TrueType fonts whose glyphs, widths or character maps disagree with the PDF are rewritten so that viewers keep drawing the glyphs they drew, and CFF fonts stored as TrueType move to where PDF/A expects them. Text drawn with a font the page does not define gets Helvetica, which Chrome draws it with. Level U adds the Unicode text of every character it can work out.
- **Colour.** An sRGB output intent is added. CMYK and grey colours get ICC profiles that keep the colours viewers showed, and spot colours get the colorant entries PDF/A requires.
- **Metadata.** XMP metadata with the document's properties and the PDF/A identification.
- **Active content.** JavaScript and the other actions PDF/A forbids, XFA forms, and the annotation types it forbids, such as sound, movie and 3D, are removed.
- **Annotations and forms.** Every annotation is set to print and gets an appearance of its own: drawn for highlights, notes, shapes and ink, empty for the rest. A form that asks viewers to draw its fields gets them drawn by PdfPecker's form filler.
- **Attachments.** PDF/A-3 keeps every attachment and gives it a MIME type and its relationship to the document. PDF/A-2 allows only PDF/A files as attachments, so attached PDFs are converted

to PDF/A-2b too (two levels deep) and other files are removed.

- **The rest.** Transfer functions, alternate images, external streams, operators PDF does not define, numbers PDF/A does not allow, layer settings, page boxes and other details PDF/A is strict about.

A change that alters how the document looks or behaves, such as a removed script or attachment, says so in its sentence.

What it cannot fix

The conversion checks the file it wrote against the same rules, so `Problems` lists only what is still wrong. The PDF claims the PDF/A level asked for even then, so check `IsConformant` before you archive it as PDF/A. These are reported and not fixed:

- Text in a font that is not embedded and whose glyphs the bundled fonts do not have: Chinese, Japanese and Korean fonts, and Symbol or Dingbats fonts used for Latin letters.
- At level U, characters whose Unicode cannot be worked out, such as those of Type 1 subsets with glyph names like `g123`. Level B accepts them.
- A file without pages.

A validator such as veraPDF gives the final word. In PdfPecker's CI, veraPDF 1.28 checks the converted output of PDFs from Chrome, LibreOffice, Ghostscript, ReportLab, cairo, qpdf and PdfPecker itself at all four levels.

Encrypted and signed PDFs

PDF/A forbids encryption, so an encrypted PDF throws `ArgumentException`: decrypt it with `PdfProtector.Decrypt` first. A signed PDF throws too, since converting it would break its signatures; convert first, then sign the PDF/A file. A file that is not a readable PDF throws `InvalidDataException`.

The input is treated as untrusted: decoding is bounded by the file's size, and the overloads with a `CancellationToken` stop between objects. The same input always gives the same bytes. The conversion makes PDF/A-2 and PDF/A-3 at levels B and U; it does not make PDF/A-1, PDF/A-4 or level A.

Text extraction

> **Changed in 1.24.0:** Reading order.

`PdfText` extracts the text of any PDF, not only those PdfPecker made, in reading order: as one string, or page by page with every line and word and where it is on the page. It reads what the PDF draws, using PdfPecker's own PDF parser and font engines, so it needs no other package either.

```
// All the text, pages separated by a form feed (\f) as pdftotext does.
string text = PdfText.Extract(pdfBytes);
string report = PdfText.ExtractFile("report.pdf");

// Some pages of an encrypted PDF.
string summary = PdfText.Extract(pdfBytes, new PdfTextOptions { Password = "pw", Pages =
PdfPageRange.Parse("1-3") });

// Lines and words with their positions, such as for finding a total on an invoice.
foreach (PdfTextPage page in PdfText.ExtractPages(pdfBytes))
    foreach (PdfTextLine line in page.Lines)
        if (line.Text.StartsWith("Total"))
            Console.WriteLine($"Page {page.PageNumber}, {line.Y:F0} pt from the top:
{line.Words[^1].Text}");

PdfTextPage first = PdfText.ExtractPage(pdfBytes, 1);
```

`Extract`, `ExtractPages` and `ExtractPage` also take a `Stream`, and `ExtractPagesFromFile` reads the pages of a file.

Pages, lines and words

`PdfTextPage.Text` is the page's text, with lines separated by a line feed and paragraphs by an empty line. Its `Lines` go from top to bottom, and each `PdfTextLine` has its `Text`, its `Words` in reading order and a bounding box (`X`, `Y`, `Width`, `Height`). Each `PdfTextWord` has its `Text`, its box, its `FontSize` and its `FontName` (such as `Helvetica-Bold`). Positions are in points from the top-left corner of the page as it is displayed, after its rotation, with y growing downward, like the signature box and the stamp offsets. Boxes span the font's usual ascent and descent, so they are close to, not exactly, the inked area. Text that runs up or down the page gets its own lines, with their `Angle`.

What is read

The characters come from the fonts' ToUnicode maps, encodings and glyph names, for the standard fonts and for embedded TrueType, CFF, Type 1 and Type 3 fonts, and `/ActualText` in marked content stands for the glyphs it covers. Where a PDF draws no space between words but leaves a gap, the gap becomes a space; kerning and letter spacing do not. Ligatures such as `fi` come out as their letters unless `ExpandLigatures` is false. Right-to-left text such as Arabic and Hebrew comes out in logical order, and `PdfTextLine.RightToLeft` says when most of a line is. Invisible text, such as the recognised text layer of a scanned page, is included, and text hidden by optional content (a layer that is

off) is not. The values shown in form fields and the text of stamps are included unless `IncludeAnnotations` is false.

Pages are read in parallel (`MaxDegreeOfParallelism`, default one per processor), `MaxPages` (default 100,000) limits the work, and inputs are handled as untrusted, as with merging. A damaged file or page throws `InvalidDataException`, a wrong password `PdfPasswordException`.

Text drawn only as outlines or images has no characters to extract; a scanned page needs OCR first. See Limitations.

Reading order New in 1.24.0

A tagged PDF (PDF/UA, or any PDF with a structure tree) is read in the order of its tags. Other pages are read from their layout: columns, boxes and text set side by side come one after the other instead of across the page line by line, also when the columns are of unequal length, when a column starts higher than the first, and when a paragraph runs on from the foot of one column to the top of the next. Right-to-left columns are read from the right, pages turned with `/Rotate` as their text runs, Chinese and Japanese vertical writing down its columns, and the lines of a skewed scan's text layer in order. Table rows still make one line each, and a letter's date and signature stay in their place.

```
// The order of 1.23.0 and earlier: every line from top to bottom, across columns.
var options = new PdfTextOptions { Order = PdfTextOrder.TopToBottom };
string text = PdfText.Extract(pdfBytes, options);
```

`PdfTextOrder.Reading` is the default. The same layout analysis gives PDF to Markdown and the structured PDF to HTML their order, so the three read a page alike. Numbers in Hebrew and Arabic lines keep their separators and signs (1,234.56 rather than 56.234,1), and Thai, Indic and Arabic text in PDFs from PdfPecker extracts as the letters written.

Images

`PdfImages` extracts the images the pages of any PDF draw: photos, logos, scans and the pictures inside forms and annotations, each once, in the order they are first drawn, with every page and place it is drawn.

```
foreach (PdfExtractedImage image in PdfImages.Extract(pdfBytes))
{
    image.Save($"image-{image.Placements[0].PageNumber}{image.FileExtension}");
    Console.WriteLine($"{image.Width}x{image.Height}, drawn {image.Placements.Count} times,
camera: {image.Exif?.Model}");
}
```

A JPEG that a viewer shows as it is comes out as its original bytes, with no loss and with its EXIF data (`KeepJpeg = false` decodes it instead); every other image (Flate, JPEG 2000, CCITT, JBIG2 and the rest) is decoded with its mask as alpha and written as PNG. Each image has its `Exif` summary (camera, dates, orientation, GPS and the whole EXIF block) and the XMP metadata the PDF attaches to it. Placements are in points from the top-left corner, like the text's. Images in hidden layers, soft

masks, tiling patterns and Type 3 glyphs are not returned. `Pages`, `Password`, `IncludeAnnotations`, `MaxImages` and `MaxPages` work as for text.

Document properties

`PdfDocumentProperties.Read` reads what a PDF says about itself, from its document information or, where that is missing or older, its XMP metadata (so PDF/A-4 and PDF 2.0 files without document information read too):

```
var p = PdfDocumentProperties.ReadFile("received.pdf");
Console.WriteLine($"{p.Title} by {p.Author}, created {p.CreationDate}, PDF {p.Version},
{p.PageCount} pages");
Console.WriteLine($"PDF/A: {p.PdfAConformance}, tagged: {p.IsTagged}, linearized:
{p.IsLinearized}, encrypted: {p.IsEncrypted}");
```

It gives the title, author, subject, keywords, creator, producer, dates, custom properties and language, the PDF version, every page's displayed size and rotation, encryption and permissions, the PDF/A and PDF/UA conformance the metadata claims (a claim, not a check), whether the file is tagged or linearized, and the XMP packet. It also takes a `Stream` and a password.

PDF to image

> **Changed in 1.24.0:** Pixels for OCR and image processing.

`PdfToImage` and `PdfRenderer` render PDF pages to PNG, JPEG, WebP or TIFF (or raw RGBA pixels). This works for any PDF, not only PdfPecker's own output, and also uses no third-party packages: the PDF parser, font engines, JPEG and fax decoders, rasterizer and the PNG, JPEG, WebP and TIFF encoders are all part of the library.

```
// Every page as PNG bytes (rendered in parallel), or one page (0-based index)
byte[][] pages = PdfToImage.Convert(pdfBytes, new ImageOptions { Dpi = 150 });
byte[] first = PdfToImage.ConvertPage(pdfBytes, 0);

// report.pdf -> out/report-1.png, out/report-2.png, ...
PdfToImage.ConvertFile("report.pdf", "out");

// JPEG or lossless WebP: report.pdf -> out/report-1.jpg, ... (or .webp)
PdfToImage.ConvertFile("report.pdf", "out", new ImageOptions { Format = PdfImageFormat.Jpeg,
Quality = 90 });
byte[][] webps = PdfToImage.Convert(pdfBytes, new ImageOptions { Format = PdfImageFormat.WebP
});

// Open once, render many pages; RenderedImage also exposes the RGBA pixels
var renderer = PdfRenderer.Open(pdfBytes, password: null);
RenderedImage image = renderer.RenderPage(2, new ImageOptions { Width = 800 });
image.SavePng("page3.png");
File.WriteAllBytes("page3.jpg", image.ToJpeg(quality: 85));
File.WriteAllBytes("page3.webp", image.ToWebP());

// The whole PDF as one multi-page TIFF, here black and white for fax or archiving
byte[] tiff = PdfToImage.ConvertToTiff(pdfBytes, new ImageOptions { Dpi = 200, TiffColorMode =
TiffColorMode.BlackAndWhite });
PdfToImage.ConvertFileToTiff("report.pdf", "report.tif");
```

HTML to images

`HtmlToImage` converts HTML straight to one image per page, for previews, thumbnails and e-mails. It is the shortcut for `HtmlToPdf.Convert` followed by `PdfToImage.Convert`, takes the same `PdfOptions` and `ImageOptions`, and gives the same images. `Security` and `Signature` are ignored, since they change nothing on the page.

```
byte[][] pages = HtmlToImage.Convert(html, new PdfOptions { PageSize = PageSize.A4 }, new
ImageOptions { Dpi = 96 });
byte[][] jpegs = await HtmlToImage.ConvertAsync(html, imageOptions: new ImageOptions { Format
= PdfImageFormat.Jpeg });
```

Output formats

Output formats (`ImageOptions.Format`, default `Png`; `RenderedImage.Encode()`/`Save()` use it too):

- **PNG**: lossless, RGB for opaque pages and RGBA otherwise.
- **JPEG**: baseline JFIF with the standard (IJG) quantisation tables scaled by `Quality` (1 to 100, default 85) and Huffman tables optimised per image. Below quality 90 colour is subsampled 4:2:0; from 90 it keeps full resolution (4:4:4), which keeps coloured text sharp. JPEG has no transparency, so a `Transparent` render is flattened onto `BackgroundColor` (white by default). At quality 90 a 150 DPI text page decodes at about 45 dB PSNR.
- **WebP**: always lossless (VP8L), alpha included, decoding to exactly the PNG's pixels and usually 50% to 70% smaller. Pages with up to 256 colours use a palette (packed 2, 4 or 8 pixels per byte when small); others use subtract-green and per-tile prediction, then LZ77, a colour cache and canonical prefix codes. Here `Quality` is the compression effort (default 29): higher spends more time searching for a smaller file and never changes a pixel; 85, the default before 1.19.0, makes files about 1% smaller and takes nearly twice as long. Lossy WebP is not written. WebP is limited to 16,384 pixels per side (JPEG to 65,535); larger images throw `NotSupportedException`.
- **TIFF**: lossless, one page per file with `Format = PdfImageFormat.Tiff` (`.tif` from `ConvertFile`, or `RenderedImage.ToTiff()`), or every page in one multi-page file with `ConvertToTiff`, `ConvertToTiffAsync`, `ConvertFileToTiff` and `PdfRenderer.RenderAllToTiff`. See TIFF.

TIFF

`ImageOptions.TiffColorMode` picks `Color` (the default: RGB, or RGBA when the render is transparent), `Grayscale` or `BlackAndWhite`. `TiffCompression` defaults to CCITT Group 4 for black and white and LZW otherwise; `None`, `Lzw`, `Deflate`, `PackBits` and `CcittGroup4` (always black and white) can be set, and all of them are lossless. For black and white, `TiffThreshold` (1 to 255, default 128) is the brightness below which a pixel becomes black. Greyscale and black and white flatten transparency onto `BackgroundColor`, and `Dpi` is written into the file so it prints at the right size. Files are classic TIFF, so output over 4 GB throws `NotSupportedException` (BigTIFF is not written).

```
// Greyscale archive copy with Deflate
byte[] archive = PdfToImage.ConvertToTiff(pdfBytes, new ImageOptions
{
    Dpi = 300,
    TiffColorMode = TiffColorMode.Grayscale,
    TiffCompression = TiffCompression.Deflate,
});
```

Options

`ImageOptions` sets `Dpi` (default 150) or a target `Width` / `Height` in pixels, `BackgroundColor` or `Transparent`, `RenderAnnotations`, `Password`, `MaxDegreeOfParallelism`, `PngCompression`, `Format`, `Quality` (85 for JPEG and 29 for WebP until it is set; an `ArgumentOutOfRangeException` outside 1 to 100) and the TIFF options. A `PdfRenderer` is thread-safe and shares decoded fonts and images between pages. Encrypted files open with the user or owner password (`PdfPasswordException` otherwise); unreadable files throw `InvalidDataException`, as does a page that needs more than 20 million drawing operations (nested forms can multiply a few bytes into billions). Pass a cancellation token to bound the time a page may take.

Pixels for OCR and image processing New in 1.24.0

`PdfRenderer.RenderPage` can draw a page into a buffer of yours as grey, RGB or RGBA bytes, rows from the top without padding, instead of making a new `RenderedImage`. `GetPixelSize` gives the size to allocate, so one buffer can serve every page. Grey and RGB flatten a transparent page onto `BackgroundColor` over white, as JPEG output does.

```
var renderer = PdfRenderer.Open(scanBytes, password: null);
var options = new ImageOptions { Dpi = 300 };
byte[] buffer = [];
for (int i = 0; i < renderer.PageCount; i++)
{
    var (w, h) = renderer.GetPixelSize(i, options);
    if (buffer.Length < w * h) buffer = new byte[w * h];
    renderer.RenderPage(i, buffer, ImagePixelFormat.Gray, options);
    // ... hand w x h grey pixels to OCR or image analysis
}
```

`ImageToPixels.Convert` does the same for image files: it decodes a PNG, JPEG, GIF, WebP, BMP or TIFF (any page of a multi-page TIFF, with `GetFrameCount` and `ImageToPixelsOptions.Frame`) straight to `ImagePixels` in grey, RGB or RGBA, with the colour conversions, EXIF orientation and size limits of ImageToPdf, and without making a PDF of the image first. A 300 dpi A4 PNG decodes in about a third of the time that going through a PDF took, with a small fraction of the memory.

```
ImagePixels scan = ImageToPixels.ConvertFile("page.png", new ImageToPixelsOptions { Format =
ImagePixelFormat.Gray });
Console.WriteLine($"{scan.Width} x {scan.Height}, {scan.Pixels.Length} bytes");
```

What the renderer supports

- **Files**: cross-reference tables and streams, object streams, incremental updates, damaged files (rebuilt by scanning), RC4 and AES-128/256 encryption, all standard filters (Flate, LZW, ASCII85, ASCIIHex, RunLength, predictors).
- **Graphics**: paths, fills (non-zero and even-odd), strokes with joins, caps and dashes, clipping, anti-aliasing, all blend modes, constant alpha, soft masks, transparency groups, axial, radial, function and mesh shadings (types 1 to 7), tiling patterns, form XObjects, optional content (layers) and annotation appearances.
- **Text**: embedded TrueType, OpenType, CFF and Type 1 fonts, Type 3 fonts, CID fonts with CMaps, vertical text, all text rendering modes. The 14 standard fonts and non-embedded fonts are drawn with built-in Liberation and DejaVu-derived faces stretched to the document's widths.
- **Images**: every bit depth and colour space (Gray, RGB, CMYK, Lab, ICC-based, Indexed, Separation, DeviceN), baseline and progressive JPEG, JPEG 2000 (`JPXDecode`, including embedded alpha), JBIG2 (generic, text, symbol, refinement and halftone regions, with `JBIG2Globals`), CCITT fax, stencil masks, colour-key masks and soft masks.

For what the renderer does not support yet, see Limitations.

PDF to HTML

`PdfToHtml` turns any PDF, not only those PdfPecker made, into one HTML file that looks like the PDF. Each page is an image of the page, and the page's text is laid invisibly over it line by line, so it can be selected, copied, found with the browser's search (whole phrases too) and read by screen readers. The PDF's web, mail and phone links and its links between pages keep working. Use it to show statements, invoices or reports in a web page or an e-mail portal without a PDF viewer. The structured mode writes the text instead as headings, paragraphs, lists and tables that flow, for search indexes, language models and small screens.

```
// Any PDF as one self-contained HTML page
string html = PdfToHtml.Convert(File.ReadAllBytes("statement.pdf"));

// File to file, or straight to a stream a few pages at a time, so a long PDF is never held in
memory whole
PdfToHtml.ConvertFile("report.pdf", "report.html");
using (var file = File.Create("report.html")) PdfToHtml.Convert(pdfBytes, file);

// Some pages of an encrypted PDF, as sharper JPEG images
string first = PdfToHtml.Convert(pdfBytes, new PdfHtmlOptions
{
    Password = "pw",
    Pages = PdfPageRange.Parse("1-3"),
    Dpi = 200,
    ImageFormat = PdfImageFormat.Jpeg,
    Quality = 80,
});
```

`Convert` also takes a `Stream`, and every method takes a `CancellationToken`.

The HTML

The output is one self-contained HTML file with no scripts: the page images, the style sheet and the small web font of the text layer are all inside it, so it can be saved, mailed or served as it is. Pages scale to the width of the window, down to a phone screen, and print on paper of their own size. Every page is a `div` with the class `pdf-page` and the id `page-` followed by its page number, so `report.html#page-3` opens the third page. Each line of the text layer is one box with its words inside, so the browser's find matches phrases across words, not only single words.

The pages are drawn by PdfPecker's own PDF renderer and the text comes from text extraction, so everything those can read works here: any fonts, JPEG 2000, JBIG2 and CCITT images, encrypted PDFs, form fields and annotations. Pages are converted in parallel.

Structured HTML

`Mode = PdfHtmlMode.Structured` writes the pages' text as flowing headings, paragraphs, lists and real `<table>` elements in reading order, with nothing placed at a fixed position. A tagged PDF (PDF/UA, or any PDF with a structure tree) is read from its tags, which say exactly what each heading,

list item and table cell is. Other PDFs are read from their layout: tables come from the rules and borders the PDF draws and from text that lines up in columns, with header rows, spanned cells, wrapped cells and indented row labels.

```
// Plain structure: one readable style, no pictures
string text = PdfToHtml.Convert(pdfBytes, new PdfHtmlOptions { Mode = PdfHtmlMode.Structured
});

// The same structure with the PDF's fonts, colours, backgrounds, borders and pictures
string styled = PdfToHtml.Convert(pdfBytes,
    new PdfHtmlOptions { Mode = PdfHtmlMode.Structured, IncludeStyles = true, IncludeImages =
true });
```

`IncludeStyles` keeps how the page looks as CSS classes of each page: the fonts, sizes and colours of the text, the backgrounds, rounded corners and border lines of table cells and coloured boxes (dashed ones too), where text sits in its cell, and the widths of tables and columns. The text still flows. `IncludeImages` adds the pictures the pages draw, such as a logo, as JPEG or PNG images where they sit in the flow; a picture that fills most of a page (a scan under its text), image masks, turned pictures and pictures too small to see are left out. The image options of the table below do not apply to structured HTML.

The same structure can also be written as GitHub Flavored Markdown with PDF to Markdown.

Options

`PdfHtmlOptions`	**Default**	
`Mode`	`Visual`	`Visual` pages that look like the PDF, or `Structured` flowing text and tables (see Structured HTML)
`IncludeStyles`	false	Structured HTML only: keep the PDF's fonts, colours, fills, borders and column widths as CSS classes
`IncludeImages`	false	Structured HTML only: include the pages' pictures as images in the flow
`Password`	none	The user or owner password of an encrypted PDF
`Pages`	every page	The pages to convert, such as `PdfPageRange.Parse("1-3")`
`Dpi`	150	The resolution of the page images; higher is sharper on high-density screens and makes larger files
`ImageFormat`	`WebP`	`WebP` (lossless), `Png`, or `Jpeg` (smaller for photos and scans, but lossy)
`Quality`	85 for JPEG, 29 for WebP	The JPEG quality, or the WebP compression effort, from 1 to 100; WebP never loses a pixel, and 85 (the default before 1.19.0) makes it about 1% smaller and slower
`IncludeText`	true	Lay the text invisibly over the images
`IncludeLinks`	true	Turn web, mail, phone and page links into links; `javascript:` and file links are always left out

`PdfHtmlOptions`	**Default**	
`RenderAnnotations`	true	Draw form fields, stamps and comments, and include their text
`Title`	the PDF's title	The HTML document's title
`MaxDimension`	16,384	The most pixels on either side of a page image; larger pages get a lower resolution
`MaxPages`	10,000	More pages throw `PdfLimitException`
`MaxDegreeOfParallelism`	processor count	The most pages converted at the same time

What it is not

Visual HTML keeps the PDF's layout rather than reflowing it into paragraphs and tables, and the text is shown in the image rather than in the PDF's fonts, so it zooms as an image does. Structured HTML reflows, but an untagged PDF's structure is read from its layout, so unusual layouts, floats and charts can come out as plain paragraphs. Text that the PDF draws only as outlines or images, such as a scan, can be seen but not selected, since there is no OCR. See Limitations.

PDF to Markdown

`PdfToMarkdown` turns any PDF into GitHub Flavored Markdown: headings, paragraphs, bulleted and numbered lists, bold and italic text, links and pipe tables. It works for PDFs from any tool, not only those PdfPecker makes, and is meant for search indexes, language models and anything else that reads text.

```
// The whole document as one Markdown string.
string markdown = PdfToMarkdown.Convert(File.ReadAllBytes("statement.pdf"));

// Mark where each page starts, so a passage can be traced back to its page.
string withPages = PdfToMarkdown.Convert(pdf, new PdfMarkdownOptions { IncludePageComments =
true });

// File to file, or into a stream.
PdfToMarkdown.ConvertFile("report.pdf", "report.md");
```

What it finds

The structure is the same one the structured PDF to HTML finds. A tagged PDF, such as one written for PDF/UA, is read from its structure tree, and other PDFs from their layout: headings from larger or bolder text, lists from their bullets and numbers, and tables from the borders and rules the PDF draws and from text that lines up in columns.

- Several header rows of a table are joined column by column, and a spanned cell's text is in its first row and column.
- Line breaks the PDF made on purpose, such as in an address, stay as line breaks.
- Text that would read as Markdown, such as a leading `#` or `*`, is escaped, so the output reads back as the same text.

Options

`PdfMarkdownOptions`	**Default**	**What it does**
`Pages`	every page	The pages to convert, such as `PdfPageRange.Parse("1-3")`
`Password`	none	The user or owner password of an encrypted PDF
`IncludeLinks`	true	Text that links to a web or e-mail address becomes a Markdown link
`RenderAnnotations`	true	The text of annotations, such as form field values, is included
`IncludeImages`	false	Pictures, such as a logo, as `data:` images where they sit in the text
`IncludePageComments`	false	Each page starts with `<!-- page 3 -->`
`MaxPages`	10,000	More pages throw `PdfLimitException`
`MaxDegreeOfParallelism`	processor count	Pages converted at the same time

Pictures in table cells, pictures that fill most of a page and pictures too small to see are left out even with `IncludeImages`. Many Markdown viewers do not show `data:` images, and they make the text much longer, so they are off by default.

What it is not

It reads the text a PDF holds; a scanned page without a text layer gives no text, since PdfPecker does no OCR. The order and structure are a reading of the page, so an unusual layout can come out in another order than a person would read it.

A wkhtmltopdf alternative for .NET

wkhtmltopdf was archived in January 2023 and gets no more fixes, security fixes included. It renders with a Qt WebKit engine from 2016, so flexbox, grid and most CSS written since then don't work, and it has to be deployed as a native binary of about 90 MB for each operating system. .NET wrappers such as DinkToPdf, Rotativa, NReco.PdfGenerator and WkHtmlToPdf-DotNet carry the same engine with them.

PdfPecker is a replacement written in plain C#. It is one NuGet package with no native files, it supports modern CSS, and it converts in your own process instead of starting a program for every PDF. It is free under the MIT license.

```
dotnet add package PdfPecker
```

Replace the wkhtmltopdf call

From the command line or a Process

```
// Before: one wkhtmltopdf process per document
var process = Process.Start("wkhtmltopdf", "--page-size A4 -T 20mm -B 20mm invoice.html
invoice.pdf");
process.WaitForExit();
```

```
// After
using PdfPecker;

HtmlToPdf.ConvertFile("invoice.html", "invoice.pdf", new PdfOptions
{
    PageSize = PageSize.A4,
    Margins = PageMargins.FromMillimeters(20, 10, 20, 10),
});
```

Scripts and build pipelines that call wkhtmltopdf itself can call the `pdfpecker` tool instead, which reads wkhtmltopdf's options, header and footer texts included:

```
dotnet tool install -g PdfPecker.Cli
pdfpecker --page-size A4 -T 20mm -B 20mm invoice.html invoice.pdf
```

From DinkToPdf or WkHtmlToPdf-DotNet

```
// Before
var converter = new SynchronizedConverter(new PdfTools());
var document = new HtmlToPdfDocument
{
    GlobalSettings = { PaperSize = PaperKind.A4, Orientation = Orientation.Landscape },
    Objects = { new ObjectSettings { HtmlContent = html } },
};
byte[] pdf = converter.Convert(document);
```

```
// After
byte[] pdf = HtmlToPdf.Convert(html, new PdfOptions { PageSize = PageSize.A4.Landscape });
```

There is no converter object to keep alive and no native library to load: `HtmlToPdf` is static and thread-safe, so many requests can convert at the same time without the `SynchronizedConverter` lock.

From Rotativa in ASP.NET Core

Rotativa renders a Razor view and passes the HTML to wkhtmltopdf. Render the view to a string the way you already do for emails, or build the HTML from a template, and return the PDF:

```
app.MapGet("/invoice/{id}", (int id) =>
    Results.File(HtmlToPdf.Convert(RenderInvoice(id)), "application/pdf",
$"invoice-{id}.pdf"));
```

Options side by side

wkhtmltopdf	PdfPecker
`--page-size A4`, `-O Landscape`	`PageSize = PageSize.A4.Landscape`, or CSS `@page { size: A4 landscape }`
`-T`, `-R`, `-B`, `-L`	`Margins = PageMargins.FromMillimeters(top, right, bottom, left)`, or CSS `@page { margin: ... }`
`--footer-center "[page] of [topage]"`	`FooterText = "{page} of {pages}"`, or an `@page` margin box with `counter(page)` and `counter(pages)`
`--header-html header.html`	An element with `data-sticky-pdf-header` in the page (see Paged media)
`--title`	`Title`, or the document's `<title>`
`--outline`	On by default: bookmarks come from `h1` to `h6`
`toc`	Write the table of contents in your HTML with `target-counter()` and `leader()` for page numbers
`--print-media-type`	Always on: `@media print` rules apply
`--enable-local-file-access`	Files under `BaseDirectory` are read; `AllowFileAccess = true` allows others
`--zoom`, smart shrinking	`Scale`. PdfPecker does not shrink content to fit, so check page breaks after switching

wkhtmltopdf	PdfPecker
`--javascript-delay`, `--run-script`	Not supported: render the content into the HTML before converting
`--enable-forms`	On by default: inputs become fillable fields (`FormFields = false` turns it off)

What changes in the output

- Pages can count differently. wkhtmltopdf shrinks content to fit the page by default, so a report that was 24 pages can become 29 at its real size. Set `Scale` if you need the old size back, but the real size is usually what the CSS meant.
- Flexbox, grid, `calc()`, CSS variables and other modern CSS now work, so workarounds written for the old WebKit, such as tables used for layout or `-webkit-box`, can be removed or left as they are.
- JavaScript is not run. Charts or text that scripts add to the page have to be in the HTML already, for example as SVG made on the server.
- Fonts come from files you register, not from the server's installed fonts, so the output is the same on every machine. See Fonts.

Speed and size

On the same 4-core Linux VM, a one-page invoice took 282 ms with wkhtmltopdf and 0.92 ms with PdfPecker, and a 30-page report 600 ms and 33 ms. Deployment drops from about 90 MB of native files to one 6.3 MB DLL. The full comparison is on the Performance page.

Next steps

- HTML to PDF in C#: a walk-through with templates, ASP.NET Core and fonts.
- HTML and CSS and Limitations: what your templates can use.
- Compare: PdfPecker next to wkhtmltopdf and the other .NET converters, feature by feature.

An HtmlRenderer.PdfSharp alternative

HtmlRenderer.PdfSharp is a free, managed HTML to PDF converter that runs on top of PDFsharp. Two things make people look for a replacement:

- **The licence of the engine underneath.** HtmlRenderer.PdfSharp 1.6.1 itself is BSD-licensed and still installs PDFsharp 6.2.4, the last MIT version. PDFsharp 6.5.0 (October 2026) and the 7.0 previews are AGPL-3.0 or a paid commercial licence. Updating PDFsharp, by hand or with a tool such as Dependabot, adding another package that needs a newer PDFsharp, or a future HtmlRenderer release built on it, brings in the AGPL, which requires you to publish your own application's source code unless you buy a licence. Staying on MIT means pinning PDFsharp at 6.2.4 and doing without its future fixes.
- **The CSS.** HtmlRenderer supports HTML 4 and CSS 2, so flexbox, grid, CSS variables, `calc()` and `@page` rules are ignored.

PdfPecker is also fully managed and runs in your own process, with no dependencies at all, under the MIT licence. It supports modern CSS and has PDF/A, PDF/UA, signatures, forms and merging built in.

```
dotnet add package PdfPecker
```

Then remove the `HtmlRenderer.PdfSharp` package, and `PDFsharp` too if nothing else in your project uses it.

Replace the HtmlRenderer call

```
// Before
using PdfSharp;
using TheArtOfDev.HtmlRenderer.PdfSharp;

var document = PdfGenerator.GeneratePdf(html, PageSize.A4);
document.Save("invoice.pdf");
```

```
// After
using PdfPecker;

File.WriteAllBytes("invoice.pdf", HtmlToPdf.Convert(html));
```

A4 is the default. With margins and a page footer:

```
// Before
var config = new PdfGenerateConfig { PageSize = PageSize.A4, MarginTop = 40, MarginBottom = 40
};
var document = PdfGenerator.GeneratePdf(html, config);
```

```
// After
byte[] pdf = HtmlToPdf.Convert(html, new PdfOptions
{
    Margins = PageMargins.FromMillimeters(14, 15, 14, 15),
    FooterText = "Page {page} of {pages}",
});
```

HtmlRenderer has no page numbers of its own; PdfPecker has `FooterText`, `@page` margin boxes and running headers and footers written in HTML (see Paged media).

Options side by side

HtmlRenderer.PdfSharp	PdfPecker
`PageSize`, `PageOrientation` in `PdfGenerateConfig`	`PageSize = PageSize.A4.Landscape`, or CSS `@page { size: A4 landscape }`
`MarginTop`, `MarginBottom`, `MarginLeft`, `MarginRight` (points)	`Margins = new PageMargins(top, right, bottom, left)` in points, or `PageMargins.FromMillimeters`
`cssData` argument	`UserStyleSheet`, or `<style>` and `<link>` in the HTML
`imageLoad` and `stylesheetLoad` events	`ResourceResolver`, or files under `BaseDirectory`
`PdfDocument` for further editing	PDF bytes; merge, stamp, sign and protect them with PdfPecker's own APIs
Fonts from PDFsharp's `GlobalFontSettings.FontResolver`	`options.Fonts.AddFile(...)` or `@font-face` (see Fonts)

What changes in the output

- Modern CSS now works. Layouts built with tables and floats to suit CSS 2 keep working, and you can move to flexbox and grid when you want.
- Backgrounds and colours that HtmlRenderer dropped, such as coloured table headers, are drawn.
- Long tables break across pages with their header row repeated, and rows are never cut in half.
- Text is laid out by PdfPecker's own engine, so line breaks and page breaks can fall in slightly different places. Check your longest templates after switching.

Speed and memory

On the same 4-core Linux VM, a one-page invoice took 4.9 ms with HtmlRenderer.PdfSharp and 0.92 ms with PdfPecker, a 30-page report 360 ms and 33 ms, and peak memory was 255 MB and 92 MB. See Performance.

When HtmlRenderer is still a fine choice

For simple HTML that already looks right, there is no urgent reason to move today: version 1.6.1 runs on the MIT PDFsharp 6.2.4. Pin that version so an update can't change the licence. PDFsharp itself remains a solid API for drawing PDFs from code. Compare shows the two side by side with the other .NET converters.

An IronPDF alternative

IronPDF converts HTML with an embedded Chrome engine and is sold per developer. PdfPecker converts the same kind of templates with its own engine written in C#: no Chrome binaries to deploy, no licence key, and an MIT licence that is free for commercial use. For invoices, payslips, reports and other HTML you control, the output can match IronPDF's to a fraction of a millimetre.

```
dotnet add package PdfPecker
```

Replace the IronPDF call

```
// Before
var renderer = new ChromePdfRenderer();
renderer.RenderingOptions.PaperSize = PdfPaperSize.A4;
renderer.RenderingOptions.MarginTop = 20;
renderer.RenderingOptions.MarginBottom = 20;
renderer.RenderingOptions.TextFooter = new TextHeaderFooter { CenterText = "{page} of {total-
pages}" };
using var document = renderer.RenderHtmlAsPdf(html);
byte[] pdf = document.BinaryData;
```

```
// After
using PdfPecker;

byte[] pdf = HtmlToPdf.Convert(html, new PdfOptions
{
    PageSize = PageSize.A4,
    Margins = PageMargins.FromMillimeters(20, 15, 20, 15),
    FooterText = "{page} of {pages}",
});
```

`HtmlToPdf` is static and thread-safe, so there is no renderer to create or dispose, and conversions can run in parallel. To write the PDF to a file or a response stream, pass the stream: `HtmlToPdf.Convert(html, stream)`.

Headers and footers in HTML

IronPDF's `HtmlHeaderFooter` takes a piece of HTML with `{page}` and `{total-pages}`. In PdfPecker the header or footer is an element of the page itself, marked with `data-sticky-pdf-header` or `data-sticky-pdf-footer`, and the same placeholders work:

```
<header data-sticky-pdf-header height="17mm"><img src="logo.png" height="40"> ACME
Ltd</header>
<main>...</main>
<footer data-sticky-pdf-footer height="12mm">Page {page} of {total-pages}</footer>
```

The element is drawn on every page and the content flows between them. See Paged media for `@page` margin boxes and running headers.

Matching IronPDF's output exactly

Chrome lays a page out at a whole number of CSS pixels and stretches it to the paper, so IronPDF's PDFs are a little larger than the CSS says. If you are switching an existing system and want every box and glyph where IronPDF put it, for example to compare old and new PDFs, use the same page geometry:

```
var options = new PdfOptions
{
    PageSize = new PageSize(594.96f, 841.92f),   // A4 as Chrome writes it
    Scale = 793.28f / 793f,
    TruncateBandsToWholePoints = true,           // header and footer heights rounded as
Chrome does
};
```

One user ran this on the 26 templates of their production app: 24 came out identical to IronPDF's output and the other two differed by less than 0.25 mm. For a new system, the defaults are fine.

Options side by side

IronPDF `RenderingOptions`	**PdfPecker** `PdfOptions`
`PaperSize`, `PaperOrientation`	`PageSize = PageSize.A4.Landscape`, or CSS `@page { size: A4 landscape }`
`MarginTop`, `MarginBottom`, `MarginLeft`, `MarginRight` (mm)	`Margins = PageMargins.FromMillimeters(top, right, bottom, left)`
`TextHeader`, `TextFooter`	`HeaderText`, `FooterText` with `{page}` and `{pages}`
`HtmlHeader`, `HtmlFooter`	`data-sticky-pdf-header` and `data-sticky-pdf-footer` elements
`CssMediaType = Print`	Always print: `@media print` rules apply
`PrintHtmlBackgrounds`	Always on, as in Chrome's print output
`Zoom`	`Scale`
`CreatePdfFormsFromHtml`	On by default (`FormFields = false` turns it off)
`Title` and `MetaData`	`Title`, `Author`, `Subject`, `Keywords` and `CustomProperties`
`EnableJavaScript`, `WaitFor`	Not supported: render the content into the HTML before converting
`RenderHtmlAsPdf(html, baseUrl)`	`BaseDirectory` for files, `ResourceResolver` for `http` URLs

PDF/A, PDF/UA, passwords, watermarks, digital signatures, ZUGFeRD and Factur-X, merging, stamping and PDF to image are built in too; each has a page in these docs.

Speed

On the same 26 production templates, IronPDF 2026.7.2 took 13.6 s for all of them one at a time and PdfPecker 17 ms. With 22 in parallel, IronPDF made 9.2 documents per second and PdfPecker 1,143. Details are on the Performance page.

When IronPDF is the better choice

IronPDF prints exactly what Chrome prints, runs JavaScript, converts live websites and comes with paid support and a large PDF editing API. If your pages build themselves with scripts, or you need output identical to a browser for arbitrary HTML, keep a Chrome-based converter. Compare lists what each converter can do.

PDF without Chrome: an alternative to PuppeteerSharp and Playwright

PuppeteerSharp and Playwright make PDFs by driving a real Chromium browser. That gives exactly what Chrome prints, but it means downloading and patching a browser of about 600 MB, keeping it running, and paying for a browser tab for every document. PdfPecker converts HTML to PDF in your own process with its own engine written in C#, so there is no browser to install and a one-page document takes about a millisecond. It is free under the MIT license.

```
dotnet add package PdfPecker
```

Replace the browser code

From PuppeteerSharp

```
// Before
await new BrowserFetcher().DownloadAsync();
await using var browser = await Puppeteer.LaunchAsync(new LaunchOptions { Headless = true });
await using var page = await browser.NewPageAsync();
await page.SetContentAsync(html);
byte[] pdf = await page.PdfDataAsync(new PdfOptions
{
    Format = PaperFormat.A4,
    PrintBackground = true,
    MarginOptions = new MarginOptions { Top = "20mm", Bottom = "20mm" },
});
```

```
// After
using PdfPecker;

byte[] pdf = HtmlToPdf.Convert(html, new PdfOptions
{
    PageSize = PageSize.A4,
    Margins = PageMargins.FromMillimeters(20, 15, 20, 15),
});
```

Both libraries have a class called `PdfOptions`, so in a project that still uses PuppeteerSharp elsewhere, write `PdfPecker.PdfOptions`.

From Playwright

```
// Before
using var playwright = await Playwright.CreateAsync();
await using var browser = await playwright.Chromium.LaunchAsync();
var page = await browser.NewPageAsync();
await page.SetContentAsync(html);
byte[] pdf = await page.PdfAsync(new PagePdfOptions { Format = "A4", PrintBackground = true
});
```

```
// After
byte[] pdf = await HtmlToPdf.ConvertAsync(html);
```

A4 is the default page size. `HtmlToPdf` is static and thread-safe, so there is no browser to share between requests and nothing to dispose or restart.

Headers and footers

Chrome's `headerTemplate` and `footerTemplate` use `<span class="pageNumber">` and `<span class="totalPages">`, and need `displayHeaderFooter`. In PdfPecker, use `FooterText = "{page} of {pages}"` for a simple line, or put the header or footer in the page itself:

```
<footer data-sticky-pdf-footer height="12mm">Page {page} of {total-pages}</footer>
```

Unlike Chrome's templates, these elements use the page's own stylesheets, fonts and images. `@page` margin boxes with `counter(page)` work too. See Paged media.

Options side by side

Chrome `page.pdf()` option	PdfPecker
`format`, `landscape`, `width`, `height`	`PageSize = PageSize.A4.Landscape`, or `new PageSize(width, height)` in points
`margin`	`Margins = PageMargins.FromMillimeters(top, right, bottom, left)`
`preferCSSPageSize`	Always on: CSS `@page` sets the size unless `IgnoreCssPageRules = true`
`printBackground`	Always on
`scale`	`Scale`
`displayHeaderFooter`, `headerTemplate`, `footerTemplate`	`HeaderText`, `FooterText`, or `data-sticky-pdf-header` and `data-sticky-pdf-footer` elements
`pageRanges`	Convert, then pick pages with the page tools
`outline`	On by default: bookmarks come from `h1` to `h6`
`tagged`	`Accessibility`, up to PDF/UA-2 (see Accessibility)

Chrome `page.pdf()` option	PdfPecker
`emulateMedia({ media: "print" })`	Always print: `@media print` rules apply
`waitForSelector`, `waitForNetworkIdle`	Not needed: nothing loads later, and JavaScript is not run

What changes

- JavaScript is not run, so content that scripts build in the browser has to be in the HTML before conversion. Templates rendered on the server (Razor, Scriban, string building) work as they are.
- Nothing is downloaded. Images and stylesheets come from files under `BaseDirectory`, `data:` URLs, or your own `ResourceResolver` for `http` URLs, so a template can't make the server fetch other addresses.
- PdfPecker supports a large part of modern CSS, including flexbox, grid, variables and `@page` rules, but not all of what Chrome does. Limitations lists the gaps.

Speed and memory

On the same 4-core Linux VM, a one-page invoice took 106 ms with PuppeteerSharp, 131 ms with Playwright and 0.92 ms with PdfPecker. Peak memory was 813 MB and 1,027 MB for the browser-based converters and 92 MB for PdfPecker, and in parallel PdfPecker made 1,132 documents per second against 12 to 15. The full table is on the Performance page.

When to keep Chrome

PuppeteerSharp and Playwright are free and print exactly what Chrome shows, so they remain the right tool for pages that build themselves with JavaScript, for live websites and for screenshots. Compare shows what each converter can do.

Changelog

What changed in each PdfPecker release on NuGet, newest first. The manual for any of these versions can be downloaded as a PDF with the version picker next to Download PDF; the source of every change is in the PdfPecker repository.

1.25.0

Released 12 November 2026.

- Announce 1.25.0 as the last release for .NET 8 and 9 (#739)
- Bound the stops premultiplied gradient interpolation adds (#740)
- Solid text decorations as thick and placed as in Chrome (#738)
- font-variant-alternates and @font-feature-values (#736)
- text-autospace between ideographs and Latin letters or digits (#734)
- Interpolate translucent CSS gradient stops premultiplied (#737)
- font-palette and @font-palette-values for colour fonts (#735)
- Ruby annotations above their base text (#733)
- A flex row with an object-fit: cover image moves whole to the next page (#732)
- Images keep a given height when min-width or max-width limits their width (#731)
- Table rows with cells that span rows break across pages (#718)
- Count positioned boxes carried up through captures against the relayout budget (#729)
- Tables count against the relayout budget (#728)
- Target .NET 11 (#725)
- PDF/A-2a and PDF/A-3a (#715)
- Look at most 64 characters for a cluster one font should draw (#724)
- Charge the renderer's budget for the pixel rows that paths cross (#723)
- Upright CJK, text-combine-upright and upright images in vertical text (#719)
- Draw annotations that have no appearance stream (#722)
- Shape Khmer, Myanmar, Sinhala and Tibetan as HarfBuzz does (#721)
- PDF/X-4 output for print shops (#720)
- CMYK colours with device-cmyk() (#717)
- Percentage heights in flex items with a definite flexed or stretched height (#716)
- SVG filters count the pixels their content fills (#714)
- Skip substituting declarations that a later one of the property overrides (#713)

1.24.0

Released 10 November 2026.

- Look up named character references without building a dictionary (#711)
- Decode images to pixels and render into a caller's buffer (#709)
- Compare redaction boxes only with the areas near them (#710)
- A clip path with a point at infinity no longer fails text extraction (1c75deb)
- Faster text extraction, Markdown and structured HTML from PDFs (#706)
- Prepare AES-256 passwords with SASLprep (#692)
- Grid items spanning flexible columns share their width by the fr (#697)
- SVG symbols are clipped to the viewport their use gives them (#696)
- Transparency groups and CSS masks get a bounding box round what they paint (#707)
- Table border spacing in whole pixels, as Chrome keeps it (#702)
- A letter's date and signature are read in their place (#704)
- Backgrounds inside a nonzero clip of nested boxes are kept (#705)
- Tables with wider border spacing, and lists counted in letters (#703)
- Object streams are decoded on the file's budget, not the page's (#701)
- Table headers and footers across page breaks as Chrome places them (#700)
- Variation sequences from cmap format 14 (#698)
- Non-isolated transparency groups blend with what is below them (#699)
- Letter spacing skips hidden characters, as Chrome does (#695)
- Grid fr rows of indefinite height share the size of one fr (#694)
- SVG text inherits through use, text positions take units, CSS overflow auto is visible (#693)
- Check that a signature's gap is its own /Contents in encrypted files too (#691)
- Long runs of combining marks keep their order, as in HarfBuzz (#690)
- Redaction carries the text state into forms (#689)
- Thai, Indic and Arabic text on an SVG path copies as the letters written (#688)
- Watch renders again when an input's name starts with the output's (#687)
- Bound the total Unicode mappings of a CMap (#686)
- Thai, Indic and Arabic text copies and extracts as the letters written (#677)
- Unicode normalization from the library's own tables, the same with or without ICU (#675)
- Find tables whose cells draw their own borders (#674)
- Bound the work of tagged tables on a page (#673)
- Variable font instances and feature variants no longer run out over a process's life (#676)
- Repeated table header after border spacing that crosses a page break (#671)
- Numbers in right-to-left lines keep their separators and signs (#670)
- Intrinsic widths that keep min-width, and the automatic minimum of grid and flex items (#672)
- A column that starts higher than the first is read after it (#668)
- Lines of skewed scans stay in order in text extraction (#669)
- Deferred fonts and hyphenation patterns (#655)
- Transparency groups and soft masks cost the area they draw (#667)
- Smaller subsets of CFF fonts that keep data before their subroutines (#652)

- Fluent page slots align their content, border calls merge, band Markdown keeps braces (#665)
- Verify signatures by the revision they signed (#666)
- Redaction removes the originals of what it replaces (#664)
- Column flex box with a min-height grows instead of pushing centred content up (#663)
- revert and revert-layer on every property, and invalid var() makes the property unset (#662)
- Barcode reader for images and PDF pages (#661)
- SVG currentColor where a use draws, lengths with units and nested svg viewports (#660)
- Invisible characters are no longer drawn as ? or boxes (#659)
- Parse each colour space once, and pages that do not depend on render order (#658)
- Bound colour space, function, ToUnicode and inline image sizes taken from the file (#656)
- Read references to freed objects as null instead of rebuilding the xref table (#657)
- HTML character references as browsers decode them (#654)
- CLI keeps whole-file options and page numbers when it joins several inputs (#653)
- Type 3 text honours invisible and clipping render modes (#651)
- Link annotations for links inside transformed elements (#650)
- Faster table search for text reading order, Markdown and structured HTML (#648)
- Snapshot comparison finds the stream keyword only after the dictionary (#649)
- PDF redaction: remove what pages draw in given areas (#647)
- Install the pdfpecker tool as Native AOT on .NET 10 SDKs (#645)
- Lone surrogates in a ToUnicode map draw no glyph (#646)
- Faster JPEG decoding, CMYK images and upright image drawing (#642)
- Faster JPEG encoding of flat blocks, sparse blocks and plain rows (#639)
- Reuse page pixel buffers in multi-page conversions (#638)
- Sanitizer keeps only web, e-mail and phone links, and rewrites damaged files (#640)
- LTV fetches without a proxy, piped HTML without the current folder, and password -env options (#641)
- Faster TIFF encoding (#637)
- Faster WebP encoding with pooled buffers (#636)
- Faster PNG encoding (#635)
- Fewer allocations and no lock waits in text extraction (#631)
- Inset box shadows on inline boxes (#633)
- object-view-box (#632)
- contrast-color() (#630)
- if() with media(), supports() and style() (#629)
- sibling-index() and sibling-count() (#628)
- @page :nth() and page groups (#627)
- Bleed and crop marks on @page (#626)
- Columns inside columns split a clipped shape only where the clip shows it (#625)
- Stretched flex items laid out again only within the relayout budget (#624)

- Nested SVG foreignObject content laid out within the relayout budget (#623)
- Markdown and structured HTML read unequal columns as PdfText does (#622)
- Read vertical writing drawn letter by letter down its columns (#621)
- Arabic words stay whole when letters are joined by cursive attachment (#620)
- Right-to-left text columns read from the right (#619)
- Read pages turned with /Rotate as their text runs (#618)
- Columns of unequal length in text extraction reading order (#617)
- Filter budget counts the shapes and clips a blurred box fills (#616)
- Table rows and columns with visibility: collapse leave the table (#614)
- Spanning cells no longer run past the page end (#615)
- Collapsed borders from col and colgroup (#613)
- SVG foreign objects drawn again by use elements count against the drawing budget (#612)
- Contextual alternates by default, as Chrome applies them (#584)
- Reading order in text extraction (#609)

1.23.0

Released 4 November 2026.

- SignatureField in PdfPecker.Fluent (#608)
- Empty signature fields from HTML (#605)
- Character references that overflow decode as U+FFFD (#607)
- Convert existing PDFs to PDF/A (#604)
- Page number formats in Fluent (#606)
- Formula, index, table of contents and list label semantics in Fluent (#603)
- Counter styles for page numbers in running headers and footers (#602)
- Tag formulas, indexes, tables of contents and hand-made list labels (#601)
- Shared images and image compression in PdfPecker.Fluent (#599)
- DocumentOperation in PdfPecker.Fluent (#596)
- PdfPecker.Razor: Razor components to PDF (#593)
- Typed attr() in any property (#586)
- Fluent Data Matrix calls and the Swiss QR-bill payment part (#600)
- PdfPecker.Cli: the pdfpecker command-line tool (#594)
- PDF to Markdown (#590)
- Swiss QR-bill codes (#598)
- Page background and foreground in PdfPecker.Fluent (#597)
- Data Matrix and GS1 Data Matrix barcodes (#595)
- Image quality and maximum resolution properties for smaller PDFs (#592)
- Linearize existing PDFs together with encryption (#589)
- Fixed boxes paint over the running headers and footers (#591)

- PdfPecker.Testing: snapshot tests for PDFs (#587)
- QuestPDF text, direction and semantic calls in Fluent (#588)
- Gap decorations in grid and flex containers (#583)
- size-adjust and the metric overrides in @font-face (#582)
- target-counters() in generated content (#585)
- box-decoration-break: clone for block boxes split across pages (#580)
- vector-effect non-scaling-stroke and transform-box for SVG elements (#579)
- Selectors :in-range and :out-of-range (#581)
- SVG geometry and gradient stop properties from style sheets (#577)
- hyphenate-limit-lines (#578)
- white-space-collapse (#576)
- Leave out SVG foreign objects inside patterns, masks and filter images (#575)
- Remove active annotations and actions in one pass (#574)
- Find @scope limits in one walk (#573)
- Float bands look only at the floats that can reach them (#572)
- Repeat in PdfPecker.Fluent (#570)
- accent-color and appearance: none for check boxes and radio buttons (#571)
- ContinuousSize in PdfPecker.Fluent (#569)
- StopPaging in PdfPecker.Fluent (#567)
- image-rendering: pixelated and crisp-edges keep enlarged images sharp (#568)
- Markdown content in PdfPecker.Fluent (#557)
- -pdfpecker-repeat: each-page draws a box again on its parent's later pages (#564)
- Write the SVG viewBox scale with six decimals (#562)
- ExtendLastCellsToTableBottom in PdfPecker.Fluent (#565)
- Pages as tall as their content with @page size: 80mm auto (#561)
- Table of contents from headings in PdfPecker.Fluent (#556)
- continue: discard keeps a box to its first fragment (#560)
- Shapes drawn by an SVG use element inherit from the use element (#566)
- Gradients and opacity in PdfPecker.Fluent (#554)
- CSS @scope (#563)
- Document.Merge in PdfPecker.Fluent (#558)
- Running elements in page-margin boxes (#559)
- CSS zoom (#552)
- Section pages over several named pages (#555)
- Hyphenate on text styles and Language per container in PdfPecker.Fluent (#553)
- Hand on a line's footnotes in linear time (#550)
- EvenPageHeader and EvenPageFooter in PdfPecker.Fluent (#549)
- Charts in PdfPecker.Fluent (#548)
- Leaders in PdfPecker.Fluent text (#546)

- Even-page headers and footers (#547)

1.22.0

Released 1 November 2026.

- The page within a section needs no id (#544)
- A forced break before a box's first child breaks before the box (#543)
- SVG backgrounds without a viewBox scale to background-size (#542)
- Icons from a hidden SVG sprite drawn as in Chrome (#541)
- Images and inline-blocks aligned to text-top and text-bottom as in Chrome (#539)
- The stretch sizing keyword and -webkit-fill-available (#538)
- Position paginated grid rows in one pass over the items (#536)
- Split table rows add no breaks past the page limit (#535)
- Bound the renderer's cache of pattern tiles (#534)
- Move inline layout out of LayoutEngine into its own class (#533)
- Keep rasterizer edges within their cells and coordinates finite (#532)
- Share the PNG filters, one PDF dictionary copy and the renderer's matrix (#531)
- Record the bug-type exceptions the library swallows (#530)
- One page-tree reader for the renderer, writers and signers (#529)
- One conversion context for the limits of an HTML conversion (#528)
- One cmap reader for HTML to PDF and the renderer (#527)
- Public PdfOptions.Clone, which the fluent package now uses (#525)
- Bound PDF stream decoding when no render scope is open (#526)
- corner-shape (#523)
- overflow-clip-margin (#522)
- text-box, text-box-trim and text-box-edge, as Chrome trims lines (#521)
- clip-path and shape-outside rect() and xywh() (#520)
- Keep number widths in border-image in standards mode (#519)
- Flex rows break inside their items at a page's end (#518)
- Section page numbers in the fluent package (#517)
- PdfSanitizer removes JavaScript and other active content (#516)
- First and last page of a section, and page numbers within it (#515)
- Page numbers in the content in PdfPecker.Fluent (#508)
- Inflate into pooled segments instead of a growing buffer (#513)
- Long-term validation fetches only from public addresses by default (#511)
- SHA-1 signatures are not valid unless the caller allows them (#507)
- Rarely set style properties in shared groups (#514)
- Legacy body margin attributes, as in Chrome (#512)
- The page count anywhere in the content (#506)

- Table cell content fills the row in any call order (#505)
- Percentage heights in table cells that span rows (#504)
- Auto margins on a float count as zero (#509)
- Columns nested in table cells keep their whole content between balancing passes (#510)
- Rarely used element fields in a side object (#501)
- Catch shadow attacks and dropped /Perms after a certification (#503)
- Total budget for decoded object streams (#502)
- Square dots for thin dotted borders, as in Chrome (#496)
- Keep OpenType feature keys between lookups (#500)
- Quirks-mode line heights like Chrome (#495)
- Make the cluster test's closure only for clusters in font chains (#498)
- Margins of a lowered icon no longer deepen its line (#499)
- Percentage lengths in inline SVG (#493)
- Drop a whole shorthand when one part has a unitless length (#492)
- A heading stays put before a box kept whole that is taller than a page (#494)
- Background on ::first-line (#481)
- Boxes with footnote calls count toward the relayout budget (#490)
- text-emphasis: marks over or under each character, as Chrome draws them (#487)
- Grid items kept whole go to the next page without a trial layout (#491)
- Draw image-set() in backgrounds, list markers and generated content (#479)
- text-indent with hanging, each-line and percentages (#475)
- ::marker direction orders the marker's own text (#484)
- counter(list-item) in ::marker and ::before content (#480)
- Match form patterns without a timeout when they cannot backtrack (#489)
- A marker image taller than its line moves the line down (#488)
- JPEG scans stop where their data ends, and an oversized frame keeps the old size (#482)
- Take font-width as the name of font-stretch (#486)
- Match :valid, :invalid and :open (#485)
- ::marker { content: none } removes the marker (#477)
- Expand the grid shorthand (#474)
- Border, padding and background images on the @page box (#478)
- A larger ::marker font makes its line taller (#476)

1.21.0

Released 28 October 2026.

- Limit the colour paints a document draws (#472)
- Charge images in ::before and ::after content to the generated-content budget (#471)
- Table rows break inside at a page's end as in Chrome (#470)

- SVG context-fill and context-stroke (#449)
- Fixed background images placed on the paper (#467)
- Fixed boxes paint by their z-index among the flow (#459)
- Bottom margins truncated at page breaks (#469)
- Unitless lengths only in quirks mode, as in browsers (#465)
- A stray </p> makes an empty paragraph, as in browsers (#460)
- Border sides of different colours meet in mitred corners (#464)
- Empty CSS tables keep their size, and tables in CSS tables are content (#468)
- Escapes in selector names and attribute values (#466)
- CSS error recovery as in browsers (#463)
- Margins collapse as one set of positive and negative margins (#462)
- Stylesheets in data: URLs (#457)
- Outline of a wrapped inline as one shape around all its lines (#461)
- 3D border shades as Chrome 153 draws them (#458)
- Outline and dashed border corners drawn like Chrome (#452)
- Left-to-right line order in vertical-lr (#448)
- break-all and overflow-wrap: anywhere shrink table cells (#437)
- Stretched images in flex and grid layouts, as in Chrome (#456)
- Percentage table columns fill the table as in Chrome (#450)
- Smaller default size for bare monospace text, as in browsers (#454)
- Repeat table headers and footers around a row split across pages (#453)
- Badges in empty inline wrappers, struts of empty inlines and cloned edges at wraps (#455)
- Boxes kept whole but taller than a page start on the next page (#451)
- Forced page breaks between flex lines (#447)
- HTML inside SVG foreignObject (#446)
- Percentage columns fill a table with a width (#445)
- Old -webkit- syntax: prefixed properties, the legacy box and gradients (#444)
- Floats and sized boxes in vertical writing modes (#443)
- Pictures in embed and object elements (#442)
- Charts.css charts: @property syntax checks and column flex item sizes (#441)
- Forced page breaks between table rows (#440)
- Absolutely positioned boxes inside positioned inline elements (#439)
- COLR colour fonts, versions 0 and 1 (#438)
- Faster QR code mask choice (#435)
- Form fields and footnotes in PdfPecker.Fluent (#436)
- Fluent page sets as named pages of one document (#434)
- Headers and footers of their own for named pages (#433)
- ScaleToFit, ShowOnce and SkipOnce in PdfPecker.Fluent (#432)
- Images and gradients in ::before and ::after content (#431)

- Custom ellipsis for clamped text (block-ellipsis) (#428)
- Gradient list markers (#430)
- Bookmarks from CSS (bookmark-level, bookmark-label, bookmark-state) (#429)
- Legacy clip: rect() on absolutely positioned boxes (#427)
- Underlines from the font or below the descenders (text-underline-position) (#426)
- Photos turned upright by their EXIF orientation (image-orientation) (#425)
- Images on whole CSS pixels, like their borders (#424)
- srcset, sizes and the picture element (#422)
- Gradient colour interpolation methods (in oklch, hsl longer hue, ...) (#423)
- Styles and images in structured PDF to HTML (#421)
- Stretched grid and flex items show their full height without a background (#420)
- Keep table row groups and group header rows together across pages (#419)
- Columns from broken row lines and indented row labels in structured PDF to HTML (#413)
- Collapsed tables close on every page they break across (#417)
- Read tagged PDFs from their structure tree in structured PDF to HTML (#411)
- Structured PDF to HTML with flowing text and real tables (#410)
- Keep small lines beside a large heading apart in text extraction (#409)
- Row backgrounds no longer cover collapsed table borders (#412)
- Headers and footers number only their own boxes for layer order (#408)

1.20.0

Released 25 October 2026.

- Keep-with-next chains move to the next page only within the relayout budget (#405)
- Option to strip metadata from embedded JPEGs (#403)
- Extract images from existing PDFs (#402)
- Isolated soft mask for blurred box shadows (#401)
- Browser find matches phrases in the PdfToHtml text layer (#400)
- Lay out nested flex rows on the pages a constant number of times (#399)
- Cap the pixels of a filter region far longer than it is wide (#398)
- Layouts of one document share the base of the relayout budget (#397)
- Bound CID width and vertical metric ranges (#396)
- Linearized output for Fast Web View (#386)
- Bound the layout of running headers and footers with page numbers (#394)
- Count boxes beside floats towards the depth keep-with-next chains move to (#395)
- Charge the JPEG 2000 budget before building code-blocks (#393)
- Charge the SVG drawing budget before the depth limit (#392)
- Count background gradients in the work budget of filters (#391)
- Charge JBIG2 bitmap rows to the work budget (#389)

- Keep surrogate pairs whole when hyphenating a decomposed word (#387)
- Fonts of SVG text in deeply nested groups (#388)
- Viewer settings and page labels (#381)
- Lay out a running header without page numbers once (#385)
- Check link schemes before System.Uri parses them (#384)
- Certification (DocMDP) signatures (#383)
- Read document properties of existing PDFs (#382)
- Grid row probes put the anchors they move back (#378)
- Columns count as depth dependent only when they lay a box out again (#377)

1.19.0

Released 22 October 2026.

- PdfPecker.Fluent, a fluent C# API in the style of QuestPDF, is on nuget.org. It builds pages, headers, footers, columns, rows, tables, text, images and barcodes in code and renders them with PdfPecker. It is released with every PdfPecker release, with the same version.
- A header and footer of their own on the first page: `data-sticky-pdf-first-header` and `data-sticky-pdf-first-footer` replace the running bands on page one, or remove them when left empty.
- Scale content to fit a box: `-pdfpecker-scale-to-fit: shrink` draws a box's content smaller until it fits the box's height, for labels, badges and certificates.
- Faster repeated conversions: documents made from one template reuse their fonts and font subsets, parsed CSS and the PDF writer's buffers, so an invoice converts about 10% faster than with 1.18.0, and documents with web fonts gain the most. The output is byte-identical.
- PDF to image and PDF to HTML: lossless WebP encodes faster, and its default effort is now 29 instead of 85, which is about 44% faster for files about 1% larger; `Quality = 85` gives the old bytes. Masks finer than their image keep their resolution, the words of Type 3 fonts drawn at their own scale stay whole, and mesh-shaded fills need far less memory.
- Fixes, all closer to Chrome: positioned boxes paint in document order; an absolutely positioned child of a flex container is aligned by `justify-content` and `align-self`, and one with `grid-row` or `grid-column` is placed in its grid area; `vertical-align: top` and `bottom` move a span's images and inline-blocks with its text; a `<br>` directly in a flex or grid container breaks the line instead of splitting the text into two items; and a word that may break no longer leaves its last letter on a line of its own in a box as wide as its content.
- Hostile input: deeply nested columns, flex boxes, tables and backdrop filters, and long runs of combining marks, now convert within bounded time and memory.
- Pages with overlapping positioned boxes, positioned children of flex and grid boxes, `vertical-align: top` or `bottom` spans holding other boxes and `<br>` in flex containers look different from 1.18.0, and WebP images at the default quality have different bytes but the same pixels.

1.18.0

Released 21 October 2026.

- Shrink to fit: a page whose content sticks out past the right edge (a wide table or image, a long URL, `nowrap` text, a Bootstrap `.row` without a container) is laid out wider and drawn smaller, as Chrome prints it, up to 1.5 times. It is on by default; `ShrinkToFit = false` keeps the 1.17.0 layout. An image wider than the page now shrinks the page instead of being squeezed on its own.
- Fallback fonts: characters none of the document's fonts has are drawn with built-in fonts, as browsers use system fonts: Central European, Greek and Cyrillic letters with the bundled Liberation fonts (no more `?` with the standard fonts), and arrows, check marks, stars, dingbats, math and box drawing with PdfPecker Symbols, a DejaVu Sans subset. Documents that need no fallback come out byte-identical.
- Positioning: absolutely positioned children of flex and grid containers (corner ribbons, sale badges) take no room and are placed by their offsets, a `position: relative` table cell holds its positioned content, and outlines on cells of a collapsed table are drawn.
- Logical border properties (`border-inline`, `border-block-start` and the rest, and the logical radii), and logical margins, paddings and insets now follow `writing-mode` too.
- Fractions with `diagonal-fractions` or `font-feature-settings: "frac"` are formed as in Chrome.
- Fixes, all closer to Chrome: an empty clearfix `div` and `<br clear=all>` clear floats; `vertical-align: top`, `middle` and `text-top` work on smaller inline text; auto margins on inline-blocks count as 0; a flex or grid item keeps its first and last child's margins; percentage columns that crowd out other columns give way and a cell's `max-width` caps its column (with an ellipsis where asked); an image with a percentage width no longer widens a table; a box shadow never adds a blank page; `backdrop-filter` works at every box size; the default `<hr>` has Chrome's lighter shading, `<hr size>`, `color` and `noshade` work, and a margin on a vertical inline-block no longer counts twice.
- About 10% faster on invoices, feature tours and reports than 1.17.0, with byte-identical output.
- Pages with content wider than the page area, clearfix divs, floats, tables with percentage columns or capped cells, inline-blocks with auto margins and default `<hr>` rules look different from 1.17.0.

1.17.0

Released 21 October 2026.

- PDF to HTML: `PdfToHtml` turns any PDF into one self-contained HTML page, with the page images, an invisible text layer that can be selected, searched and read aloud, and working links.
- Float shapes (`shape-outside`, `shape-margin`, `shape-image-threshold`), drop caps with `initial-letter`, masks and `backdrop-filter` on HTML elements, `display: contents`, 3D transforms drawn flat, and vertical text with `writing-mode`.
- Text wrapping and justification: `text-wrap: balance` and `pretty`, and `text-justify`; justified Chinese and Japanese text is spread between characters, as in Chrome.

- CJK punctuation trimming with `text-spacing-trim`. CJK text with adjacent punctuation is narrower than before and can break into lines differently; `text-spacing-trim: space-all` keeps the old spacing.
- Text: underlines and overlines skip descenders (`text-decoration-skip-ink`), dashed, dotted, double and wavy lines are drawn as Chrome draws them, outlined text with `-webkit-text-stroke` and `paint-order`, nested superscripts, and font properties `font-kerning`, `font-synthesis`, `font-size-adjust`, `font-variant-position` and `font-variant-east-asian`, with the font's own small capitals and the `@font-face` `font-stretch` descriptor.
- Units measured from the font (`ex`, `ch`, `cap`, `ic`, `lh` and the root versions), the viewport units `vi`, `vb`, `svh`, `dvh` and the rest, `q`, and the logical `inset-inline`, `inset-block`, `inline-size` and `block-size`.
- `width: min-content`, `max-content` and `fit-content`, wrapping column flex boxes, `min-width: 0` on flex items, capped grid `fit-content()` tracks, and flex items taller than their row staying in place (Bootstrap progress bars).
- Tables: CSS tables on any element with anonymous table boxes, `inline-table`, row group and column backgrounds, `caption-side`, `empty-cells`, cell `min-width` and `max-width`, column widths under spanning cells and percentages over 100% shared out as in Chrome, long words widening a table instead of overflowing, and collapsed borders resolved as in Chrome, on rows and row groups too.
- HTML: `<details>`, `<progress>` and `<meter>`, `list-style-position: inside`, old attributes (`img align`, `hspace`, `table bordercolor` and `rules`, list `type`, `reversed`, `align=center`), and the `html`, `head` and `body` elements a fragment leaves out (an implied body has no default margin, so fragments keep their layout).
- Images: a missing image prints its alt text, inline images draw their border, padding, background and shadow, take vertical margins in the line, and resolve percentage widths against the containing block, as in Chrome.
- Selectors and the cascade: `:dir()`, `::placeholder`, the `quotes` property (also `auto` by language), the `all` property and CSS-wide keywords on shorthands; `vertical-align` lengths and percentages.
- SVG: `use` of a symbol from another inline SVG, symbol `viewBox` fitting, group `opacity`, and `text-decoration` on SVG text. Gradients and background tiles turn and scale with transformed boxes.
- Despite the new features, warm conversions are within a few percent of 1.16.0's speed. Bounded work for nested columns, tables, generated content and backdrop filters on hostile input.
- Tables with colspans, percentages, long words or collapsed borders, lines with superscripts, decorated text, inline images and justified CJK text look different from 1.16.0, closer to Chrome.

1.16.0

Released 20 October 2026.

- Borders on inline elements: badges, keyboard keys and dotted underlines drawn with `border` on a `<span>` are painted, with `box-decoration-break` deciding how a border split over several lines is drawn.

- SVG dashed strokes with `stroke-dasharray`, `stroke-dashoffset` and `pathLength`, and text along a path with `textPath`.
- Grid rows break across pages where they start instead of moving whole and leaving a gap, and a margin that runs past the end of a page is dropped, as in Chrome. A box's top and bottom borders stay on the page with its content, and a paragraph that ends just past the page keeps its widows.
- A `<fieldset>`'s legend sits on its top border, and groove and ridge borders have Chrome's two shades.
- Form controls are sized as in Chrome: text fields in tables, textarea rows, drop-downs, date fields and check boxes, and controls the page gives its own border have Chrome's plain square look.
- Text can break before and after inline-blocks, images and form controls, so a label no longer wraps a word down next to its input.
- Line heights are rounded to Chrome's layout units, and line breaking counts the kerning next to spaces, so lines break where Chrome breaks them more often.
- Text extraction reads the standard Symbol and ZapfDingbats fonts (αβγ, ✓✔) instead of returning their raw codes.
- Bounded layout time for deeply nested flex, grid, tables, columns and keep-with-next chains found by fuzzing.
- Line positions, page breaks, forms, fieldsets and SVG dashes look different from 1.15.0; most documents move by a pixel or less.

1.15.0

Released 19 October 2026.

- Invoice XML from invoice data: `InvoiceData` writes Factur-X / ZUGFeRD EN 16931 and XRechnung CII, and Peppol BIS 3.0 and XRechnung UBL, with the totals and VAT breakdown calculated and the EN 16931 rules checked.
- A ready-made invoice template: `InvoiceTemplate.ToPdf` renders an A4 invoice with your logo and colour in English, German or Danish, with an EPC payment QR code and the Factur-X XML embedded.
- Compacting: `PdfCompactor` makes any PDF smaller, losslessly by default, or with images downsampled and recompressed as JPEG.
- A cell's `rowspan` stops at the end of its table header, body or footer, as in HTML, instead of throwing `NullReferenceException`.
- Deeply nested `break-inside: avoid` boxes in columns are laid out in bounded time; the output is unchanged.
- `PdfSignatureVerifier` reports a signature whose signer key cannot be read instead of throwing `OverflowException` (Windows, .NET 8 and 9).
- No other output changes: PDFs from 1.14.2 templates are byte-identical.

1.14.2

Released 18 October 2026.

- A table whose first row does not fit on the rest of a page starts on the next page, instead of leaving its header row behind at the bottom of the page. Chrome does the same.
- In tables with automatic layout, columns with a `width` keep it next to columns of long text, as in Chrome. Tables without column widths are laid out as before.

1.14.1

Released 18 October 2026.

- Collapsed table borders stay visible over cell backgrounds: shaded header and footer rows keep their vertical lines, and striped rows keep the line above them, as in Chrome. Tables with `border-collapse` and cell backgrounds look different from 1.14.0.

1.14.0

Released 18 October 2026.

- Backgrounds on inline elements: `<span>`, `<mark>`, `<code>` and badges get images, gradients, `border-radius` and clipping to text, not just a colour, and `box-decoration-break` decides how a background split over several lines is drawn.
- Markdown footnotes, definition lists and front matter, with footnotes at the bottom of the page in a PDF.
- `hyphenate-limit-chars` for the shortest hyphenated word and the fewest letters around a hyphen.
- Rows of collapsed tables with lines only under or over each row now have Chrome's height, and balanced columns keep `break-inside: avoid` boxes whole without growing taller than needed.
- Bounded work for repeating gradients and deeply nested decorated inline elements on hostile input. Tables with collapsed row lines, rounded inline badges and some multi-column layouts look different from 1.13.0.

1.13.0

Released 18 October 2026.

- Barcodes: QR codes (versions 1 to 40, error correction L to H), Code 128 and EAN-13 as vector SVG for your templates, and `Barcode.ToDataUri` for an `<img>`, a CSS background or a Markdown image.
- EPC payment QR codes (GiroCode) for euro transfers, with the IBAN, BIC, amount and field lengths checked.
- Images to PDF: PNG, JPEG, GIF, WebP, BMP and TIFF images one per page, at their own size or fitted to a page size with margins.

- HTML to images in one call with `HtmlToImage`.
- No output changes: PDFs from 1.12.0 templates are byte-identical.

1.12.0

Released 18 October 2026.

- Indic, Thai and Lao shaping: Devanagari, Bengali, Gurmukhi, Gujarati, Oriya, Tamil, Telugu, Kannada and Malayalam, Thai and Lao are shaped as HarfBuzz shapes them, and combining marks in Latin, Greek and Cyrillic are placed on their letters.
- `background-clip: text` (also as `-webkit-background-clip`).
- Tables across pages: `<tfoot>` repeats at the bottom of every page, and tables inside flex and grid items break across pages instead of moving whole.
- Reproducible output: with `CreationDate` or `SOURCE_DATE_EPOCH` the same input gives a byte-identical PDF, also for merging, page tools, stamping and form filling.
- Cancellable signing, `AddValidationDataAsync`, and cancellation-token and `Stream` overloads across the API.
- `PdfPeckerException` as the base of the library's own exceptions, and file output that leaves an existing file untouched when a conversion fails.
- `StampHtml` ignores the footer, header, watermark and other document-level options.
- `text-transform: uppercase` turns ß into SS and handles Turkish dotted and dotless i; `hyphens: auto` hyphenates decomposed words; CSS integers are read correctly on servers with an Arabic or Persian culture.
- Bounded work on malformed JPEG 2000, JBIG2, e-invoices, attachments and CSS, and less memory for JPEG and WebP output. Tables with a `<tfoot>`, tables in flex and grid, and Indic and Thai text look different from 1.11.0 on purpose.

1.11.0

Released 18 October 2026.

- Subgrid (CSS Grid Level 2): `grid-template-columns` and `grid-template-rows: subgrid` line nested grids up with their parent's tracks.
- Multi-column layout splits paragraphs, lists and tables across columns and onto later pages, as Chrome does, instead of keeping each child whole in the first columns.
- Percentage heights resolve against definite containing blocks, as in browsers.
- Blurred box shadows, translucent gradient stops and SVG `stop-opacity` are truly transparent, blending with what is behind them instead of with white.
- Hyphenation in Swedish, Spanish, French, Italian, Portuguese and Polish; French and Italian also hyphenate words with elisions such as *l'ordinateur*.
- Correct word widths in paragraphs with an inline-block, and shaping of Arabic Extended-C characters outside the Basic Multilingual Plane.

- Bounded work on crafted fonts and malformed PDFs, and linear time for Markdown autolinks, nested inline-blocks and headings kept with nested lists, flex rows and tables. No public API changes.

1.10.0

Released 18 October 2026.

- PDF to TIFF: a whole PDF as one multi-page TIFF, or single pages, in colour, greyscale or black and white with CCITT Group 4, LZW, Deflate or PackBits compression.
- Read the invoice back from a received Factur-X, ZUGFeRD or XRechnung PDF with `PdfEInvoice.Read`.
- Trusted timestamps: a timestamp's time is used to check the signer's certificate only when its authority is trusted (`TimestampTrusted`), so an untrusted timestamp no longer makes an expired certificate pass.
- `PdfLtvOptions.UrlFilter` decides which OCSP and CRL addresses long-term validation may contact, and fetching is bounded in URLs, redirects and time.
- `::first-line` styles on small capitals and mixed-direction text no longer lose their style or throw, and Arabic text in PDF/A-2u, 3u and PDF/UA maps every glyph back to Unicode.
- Tighter limits for hostile input found by fuzzing (images, SVG filters, soft masks, fonts, signature stores), more renderer features (soft-mask Matte, CalRGB and CalGray), and symbol packages with SourceLink on NuGet.

1.9.0

Released 16 October 2026.

- Chinese, Japanese and Korean line breaking with kinsoku rules, CSS `line-break` and `word-break: keep-all`.
- External signers: sign with a key in an HSM, smart card or cloud key vault, and `PdfSigner.SignAsync`.
- Signature verification and long-term validation (PAdES B-LT and B-LTA).
- Text extraction from any PDF, with the position of every line and word.
- Fill and flatten the forms of existing PDFs.
- Container queries and the `cq` units.
- Footnotes with `float: footnote`, numbered calls and a styled footnote area.
- Each page laid out at its own width when `:first`, `:left` or `:right` change it, and right-to-left page order.
- Blurred text shadows with a true Gaussian blur, shadows on underlines, and selectable text under `filter: blur()`.
- `aspect-ratio` in shrink-to-fit widths, with `min-` / `max-height` and on tables; `object-fit` for SVG images; and complex selectors inside `:is()`, `:where()`, `:not()` and `:has()`.

1.8.0

Released 16 October 2026.

- Variable fonts: weight, width, italic and optical size axes, `font-variation-settings`, kerning and CFF2 outlines (see Fonts).
- OpenType figures and features: `font-variant-numeric` (tabular, old-style, fractions and more) and `font-feature-settings` (see Fonts).
- `border-image`, CSS `filter`, `clip-path` on HTML elements, blend modes, 3D border styles, inset box shadows, `::first-letter` and `::first-line`, `@counter-style` and relative colours.
- Every SVG filter primitive, with primitive subregions, and text inside filtered SVG stays selectable.
- `<image>` in SVG, SVG watermarks, and BMP and TIFF images.
- `@import`, and `:first`, `:left` and `:right` page margins and sizes.
- The manual moved from the README to pdfpecker.net/docs, and the NuGet package links to it and has an icon.

1.7.0

Released 15 October 2026.

- Stamping: put a page of one PDF over or under the pages of another.
- Attachments: attach any file to a PDF and read attachments back.
- Automatic hyphenation with `hyphens: auto`.
- Add, change or remove passwords on existing PDFs.
- Extract, reorder, remove, rotate and split pages.
- JPEG and WebP output from `PdfToImage`.
- PDF/A-2u, 3u, 4 and 4f, and PDF/UA-2.
- Right-to-left text in SVG, and right-to-left ellipsis.

1.6.0

Released 15 October 2026.

- Right-to-left text with Arabic and Hebrew shaping.
- Named pages, running headers and footers from HTML, table of contents page numbers with `target-counter()` and `leader()`, and `conic-gradient()` (see Paged media).
- Much more of CSS3: modern colour syntax, `calc()` and the other math functions, `@layer`, nesting, `@supports`, `@property`, media queries, `:has()` and margin boxes.
- Sign into an existing empty signature field, merging keeps tagging and PDF/UA, and `PdfOptions.MissingCharacter` reports characters no font has.

1.5.1

Released 14 October 2026.

- Targets .NET 8 and 9 as well as .NET 10.
- Complete third-party licence notices, used as the NuGet package licence. 1.5.0 was never published, because nuget.org did not accept its licence expression.

1.4.0

Released 14 October 2026.

- Fillable PDF forms from HTML inputs, check boxes, radio buttons, selects and text areas.

1.3.0

Released 14 October 2026.

- Tagged PDF and PDF/UA-1 output.
- SVG masks, patterns, markers and filters, clipping in table cells, and double rounded borders.

1.2.0

Released 14 October 2026.

- Attach PDFs to Peppol and OIOUBL (UBL) e-invoices.
- Faster HTML to PDF, with half the memory allocations and decoded images reused.

1.1.1

Released 13 October 2026.

- The README shows its logo on nuget.org and has a NuGet badge. No changes to the library.

1.1.0

Released 13 October 2026.

- Factur-X and ZUGFeRD e-invoices.
- SVG, GIF and WebP images.
- Streaming PDF merging: stream and async overloads with one input in memory at a time.
- `calc()`, `min()`, `max()` and `clamp()`, `:nth-of-type`, `:is()`, `:where()`, `object-fit`, `text-overflow`, `outline`, `text-shadow` and `aspect-ratio`.
- Header and footer heights are rounded to whole points, as Chrome does.

1.0.0

Released 12 October 2026.

- The first release: HTML and Markdown to PDF and PDF to PNG with its own HTML parser, CSS engine, layout engine, PDF writer and renderer, and no dependencies outside the .NET base class library.

Limitations

> **Changed in 1.24.0:** Text and fonts, CSS, Backgrounds and transparency, Layout, Images and SVG, Reading and changing existing PDFs.

PdfPecker is built for fast conversion of HTML templates you control, not for rendering arbitrary web pages. These are the known gaps; most are scope choices for speed and size, and they are the first candidates for future work. Anything not listed in What is supported should be assumed missing; CSS support marks every CSS property one by one. The PDF feature checklist marks every PDF feature as supported, partly supported or missing.

By design

- No JavaScript. Scripts are never run, so content must be in the HTML before conversion (render it server-side first).
- No network access: `http:` / `https:` stylesheets and images are not downloaded. Supply them through your `ResourceResolver`, as files under `BaseDirectory`, or as `data:` URIs.
- `<canvas>`, `<iframe>`, `<video>` and `<audio>` are not drawn, `<object>` and `<embed>` show only pictures, and file, range, colour and image inputs are not drawn. Form fields have no JavaScript (no calculations, formats or submit actions), and a field inside a transformed box, a running header or footer, or a `position: fixed` box is drawn but not fillable.

Text and fonts Changed in 1.24.0

- Without registered fonts or `EmbedStandardFonts`, text uses the standard PDF fonts, so characters are limited to the Windows-1252 set (Western European languages) plus what the built-in fallback fonts add: Central European, Greek and Cyrillic letters and common symbols. They have no CJK or other scripts, and no emoji: register fonts for those.
- Complex-script shaping covers Arabic and Hebrew (and other scripts written with Arabic letters, such as Persian and Urdu), nine Indic scripts, Thai and Lao. Khmer, Myanmar, Sinhala, Tibetan, Mongolian, Javanese, Adlam and Syriac are not shaped, so their vowel signs and conjuncts come out wrong. Thai and Lao text breaks only at spaces, since there is no dictionary for word breaks.
- Vertical text is drawn as turned horizontal text: Chinese and Japanese characters are turned rather than upright (`text-orientation` and `text-combine-upright` are not supported), tables and images inside vertical text are turned with it (Chrome keeps images upright and lays tables out vertically), and a vertical `html` or `body` is laid out horizontally. `vi` and `vb` stay the page width and height in vertical boxes (Chrome swaps them), and `initial-letter` makes no drop cap in a vertical box.
- CJK punctuation trimming does not use the vertical `vhal` feature, and the `text-spacing` shorthand is not read. `hanging-punctuation` is not supported.

- In Latin and other simple scripts, shaping is limited to kerning, standard ligatures and features that swap one glyph for another (see OpenType features), contextual alternates, fractions and ordinals, and `small-caps` is synthesized from scaled capitals when the font has no small capitals of its own. `font-variant-alternates` and `font-palette` are not supported, so colour fonts use their default palette.
- Solid underlines, overlines and line-throughs are a little thinner than Chrome's (about 1.5 px for 20 px text where Chrome draws 2 px), and a solid overline sits about a pixel lower; dashed, dotted, double and wavy lines match Chrome. `text-decoration` on SVG text is always a solid line in the text's fill, and is not drawn on `textPath` text.
- Automatic hyphenation (`hyphens: auto`) has built-in patterns for English, Danish, German, Dutch, Norwegian, Swedish, Spanish, French, Italian, Portuguese and Polish; other languages need their patterns in `PdfOptions.Hyphenation`.
- SVG `<text>` uses only the first value of position lists; in a text chunk with right-to-left text, `dx` on its `tspan`s is ignored.

CSS Changed in 1.24.0

- Container queries: no `scroll-state()` queries, `style()` works only on custom properties, and style containment (counter scoping) and the `contain` property are not implemented. A document with container queries takes about two to four times as long to lay out, and queries that keep changing the layout stop after six passes. In running headers and footers, `@page` margin boxes and SVG images, the `cq` units fall back to viewport units.
- Footnotes: `@footnote` inside `@page :first`, `:left` or `:right` is ignored, `footnote-policy: block` acts like `line`, and `footnote-display: compact` is always inline. A note split across pages does not carry its background or border onto the next page, and margins between block notes do not collapse.
- When `@page :first`, `:left` or `:right` give pages different widths, floats, inline-blocks, flex, grid and multi-column containers, and boxes with transforms or filters keep the width of the page they start on (Chrome re-sizes floats, inline-blocks and multi-column per page). A named page's `:first` is the document's first page, as in Chrome, not the first page of the named run.
- Selectors: a rule with an unsupported selector is skipped. A complex selector may have at most 64 compound parts and 16 levels of nesting, and `:has()` cannot contain `:has()`.
- Blurred text shadows are drawn as images at 144 dpi (down to 72 dpi for wide blurs, with the blur capped at about 200px), so shadowing long body text adds to the file size; the shadow of an underline is painted under the glyphs rather than between them and their own underline. `filter: blur()` and `drop-shadow()` draw the box as an image; its text stays selectable, and the colour filters keep everything as vectors.
- `text-overflow` applies only at the end of the line, as in the spec and Chrome.
- Background images on the `@page` box are positioned in the page box, as CSS Paged Media 3 says; Chrome sizes that area like the root element, so `right`, `bottom` and percentage positions can land elsewhere than in Chrome.
- `inline-size`, `block-size` and their `min-` and `max-` forms always mean width and height, also in vertical writing modes; the logical margins, paddings, insets and borders do follow `writing-`

`mode`.

- Percentage heights still count as `auto` in a flexed item of a container that has only a `min-height`, against a table row's height on its own, and in footnotes and running headers and footers. When pages differ in size, a box resolves its percentage height against the page it starts on and keeps that height if it breaks onto a page of another height (Chrome recomputes it there).
- `background-attachment: fixed` places images against the paper, as Chrome prints them, but gradients still move with their box.
- Not supported: `contain`, `content-visibility`, `cross-fade()`, `element()` outside page-margin boxes, `paint()`, `env()`, `anchor()`, `::selection` and `:nth-col()`. In `@page` margins and media queries, `ex`, `ch` and the other font units use the initial 16 px font.
- 3D transforms are drawn flat, since a PDF page has only flat transforms: a box with a perspective stays a rectangle or parallelogram where Chrome draws a trapezoid, only the parent's `perspective` counts, and nested `preserve-3d` scenes, such as a cube, do not come out the way a screen shows them.
- `backdrop-filter` draws what is behind the box as a 144 dpi image, up to 64 per document. A mask with more than 16 layers is ignored, and `mask-clip: no-clip`, `stroke-box` and `view-box` act as `border-box`.
- Float shapes: an SVG image in `shape-outside` falls back to the margin box, and image and path shapes are measured at 1 px per CSS pixel, up to 4 million pixels and a `shape-margin` of 1,024 px (beyond that the float keeps its rectangle).
- `text-wrap: balance` and `pretty` do not apply beside floats, with `::first-line`, or on pages of different widths.
- `display: contents`: a link to the `id` of such an element finds no place on the page when its children are blocks, and in a tagged PDF the wrapper's own tag (a list or link around blocks) is left out; its children are tagged.
- Gradient stops that fade to transparent are mixed in unpremultiplied colours, so the middle of such a gradient is a little greyer than in Chrome.
- A document without a doctype reads lengths without a unit as pixels and takes Chrome's quirks-mode line heights, but is otherwise laid out in standards mode; Chrome's quirks mode also, among other things, ignores the first paragraph's top margin in the body.

Backgrounds and transparency Changed in 1.24.0

- Inline boxes nested more than 16 deep are not decorated, and one layout paints at most 50,000 inline box fragments with a shared budget for their layers, shadows and outlines; past it, fragments show only their background colour.
- A translucent box shadow without blur is blended onto white rather than truly see-through. Blurs are drawn as bands, so very wide blurs may show faint steps when zoomed in.

Layout Changed in 1.24.0

- Subgrid: percentage gaps, margins and padding on a subgrid resolve against the parent grid's width rather than the subgrid's area; subgrid items are not baseline-aligned with the parent's

items; and only the horizontal `writing-mode` is supported.

- Multi-column: `column-span: all` works only on direct children of the multi-column box, and multi-column boxes nested more than two levels deep keep their children whole instead of splitting them. Footnotes inside columns are placed at their call and do not reserve space at the bottom of the column, orphans and widows are not applied to lines beside floats, and content that overflows a fixed-height multi-column box with `overflow: hidden` is clipped from view but stays in the PDF's text.
- Overflow: a positioned child with a `z-index` inside a clipping box stays stacked within that box.
- A table row with a cell that spans rows moves to the next page whole rather than breaking inside, as other rows do.

Images and SVG Changed in 1.24.0

- No AVIF images: decoding them needs an AV1 video decoder, which is not planned. Convert them to WebP, PNG or JPEG.
- SVG filters are applied in pixels at 144 dpi, so filtered shapes become an image (text keeps an invisible, selectable copy).
- A missing image with alt text shows the text without Chrome's small broken-image icon, and a missing image without alt text or a size has no box rather than Chrome's 16 px icon box.
- A `foreignObject` inside an SVG group with a clip, mask or filter is drawn over the drawing without those effects, and one inside a pattern, mask or filter image is left out.
- With `vector-effect: non-scaling-stroke`, markers still scale with the drawing, where Chrome keeps them at their size.

Reading and changing existing PDFs Changed in 1.24.0

- Text extraction finds the reading order of an untagged page from its layout, so unusual layouts, floats and charts can be read in another order than a person would. Text drawn only as outlines or images is not extracted; there is no OCR. Indic vowel signs drawn before their consonant come out in that visual order unless the PDF carries `/ActualText`.
- PDF to HTML in its visual mode keeps the PDF's layout: the page is an image and the text an invisible layer over it, so text zooms as an image does, and what text extraction cannot read (outlines, scans) cannot be selected either. The structured mode reads an untagged PDF's structure from its layout, so unusual layouts, floats and charts can come out as plain paragraphs, and text of cards side by side in one row can run together.
- Form filling does not shape text, so values in Arabic, Hebrew, Devanagari and other scripts that need shaping are stored but not drawn correctly. XFA forms are not filled: their XFA data is removed and the AcroForm fields are filled instead. Field JavaScript (calculations, formats) is not run.
- Signature verification checks the changes made after a certification signature against its level, but not field locks (FieldMDP), so after an approval signature it counts the updates but cannot tell an allowed form fill from a changed page. Revocation is checked only when `RevocationMode` asks the platform to, not from the OCSP responses and CRLs stored in the PDF.

Markdown and the PDF renderer

- Markdown: no math. Definition list terms are one line each (no several terms for one definition), and `~` does not start a definition. A footnote's text cannot hold another page-bottom footnote: a reference inside a footnote links to its note, and a note that only other notes reference is listed after the text.
- `PdfRenderer` (PDF to image): no predefined CJK CMaps or non-embedded CJK fonts, and ICC profiles are not applied (ICC-based colours use their device equivalents). WebP output is lossless only (no lossy VP8), and JPEG output is baseline only (no progressive or arithmetic coding).

CSS support

> **Changed in 1.24.0:** .

Every standard CSS property, at-rule, selector and value type, and whether PdfPecker draws it. Of the 602 properties below, PdfPecker supports 409, supports 13 in part and leaves out 69. Another 111 are for scrolling, animation and interaction, which have no effect on a printed page, so PdfPecker ignores them on purpose. HTML and CSS support explains how the supported features behave, and Limitations describes the gaps.

A declaration PdfPecker does not support is dropped like any unknown property: the rest of the rule still applies and the conversion does not fail. The property list is the standards-track properties in MDN's browser compatibility data, grouped by the specification that defines them, plus the paged media properties PdfPecker reads. Property names link to MDN.

Properties

Box model and sizing

Property	PdfPecker	Notes
`aspect-ratio`	✓ Yes	
`block-ellipsis`	✓ Yes	auto or a string of your own ends a block cut short by line-clamp.
`block-size`	✓ Yes	
`box-sizing`	✓ Yes	
`column-width`	✓ Yes	
`contain`	✕ No	Layout is the same without it.
`contain-intrinsic-block-size`	✕ No	Only matters with content-visibility.
`contain-intrinsic-height`	✕ No	Only matters with content-visibility.
`contain-intrinsic-inline-size`	✕ No	Only matters with content-visibility.
`contain-intrinsic-size`	✕ No	Only matters with content-visibility.
`contain-intrinsic-width`	✕ No	Only matters with content-visibility.
`content-visibility`	✕ No	Content is drawn even when set to hidden.
`continue`	✓ Yes	auto and discard, which keeps a box to its first fragment.
`display`	✓ Yes	block, inline, inline-block, flow-root, flex, grid, list-item, contents, none, every table value, two-value forms and -webkit-box.

Property	PdfPecker	Notes
`frame-sizing`	✕ No	Iframes are not loaded.
`height`	✓ Yes	
`inline-size`	✓ Yes	
`line-clamp`	✓ Yes	Also -webkit-line-clamp with display: -webkit-box.
`margin`	✓ Yes	
`margin-block`	✓ Yes	
`margin-block-end`	✓ Yes	
`margin-block-start`	✓ Yes	
`margin-bottom`	✓ Yes	
`margin-inline`	✓ Yes	
`margin-inline-end`	✓ Yes	
`margin-inline-start`	✓ Yes	
`margin-left`	✓ Yes	
`margin-right`	✓ Yes	
`margin-top`	✓ Yes	
`margin-trim`	✕ No	
`max-block-size`	✓ Yes	
`max-height`	✓ Yes	
`max-inline-size`	✓ Yes	
`max-lines`	✕ No	Use the line-clamp shorthand.
`max-size`	✕ No	Newer shorthand; use max-width and max-height.
`max-width`	✓ Yes	
`min-block-size`	✓ Yes	
`min-height`	✓ Yes	
`min-inline-size`	✓ Yes	
`min-size`	✕ No	Newer shorthand; use min-width and min-height.
`min-width`	✓ Yes	
`order`	✓ Yes	
`overflow`	✓ Yes	auto and scroll clip like hidden, since paper cannot scroll.
`overflow-block`	✕ No	Only the media feature of that name is answered.
`overflow-clip-margin`	✓ Yes	padding-box, border-box or content-box, grown or pulled in by a length, for overflow: clip.
`overflow-inline`	✕ No	Only the media feature of that name is answered.

Property	PdfPecker	Notes
`overflow-x`	✓ Yes	As overflow.
`overflow-y`	✓ Yes	As overflow.
`padding`	✓ Yes	
`padding-block`	✓ Yes	
`padding-block-end`	✓ Yes	
`padding-block-start`	✓ Yes	
`padding-bottom`	✓ Yes	
`padding-inline`	✓ Yes	
`padding-inline-end`	✓ Yes	
`padding-inline-start`	✓ Yes	
`padding-left`	✓ Yes	
`padding-right`	✓ Yes	
`padding-top`	✓ Yes	
`reading-flow`	– No effect	Only changes keyboard focus order.
`reading-order`	– No effect	Only changes keyboard focus order.
`scroll-axis-lock`	– No effect	
`scroll-behavior`	– No effect	
`scroll-marker-group`	– No effect	
`scroll-target-group`	– No effect	
`scrollbar-gutter`	– No effect	
`size`	✓ Yes	The @page descriptor: A4, Letter, landscape or any length.
`text-overflow`	✓ Yes	ellipsis, clip and strings.
`visibility`	✓ Yes	collapse on table rows and columns removes them as in Chrome.
`width`	✓ Yes	Also min-content, max-content, fit-content, stretch and -webkit-fill-available.

Borders, outlines and shadows

Property	PdfPecker	Notes
`border`	✓ Yes	
`border-block`	✓ Yes	
`border-block-color`	✓ Yes	
`border-block-end`	✓ Yes	
`border-block-end-color`	✓ Yes	
`border-block-end-style`	✓ Yes	

Property	PdfPecker	Notes
`border-block-end-width`	✓ Yes	
`border-block-start`	✓ Yes	
`border-block-start-color`	✓ Yes	
`border-block-start-style`	✓ Yes	
`border-block-start-width`	✓ Yes	
`border-block-style`	✓ Yes	
`border-block-width`	✓ Yes	
`border-bottom`	✓ Yes	
`border-bottom-color`	✓ Yes	
`border-bottom-left-radius`	✓ Yes	
`border-bottom-right-radius`	✓ Yes	
`border-bottom-style`	✓ Yes	
`border-bottom-width`	✓ Yes	
`border-color`	✓ Yes	
`border-end-end-radius`	✓ Yes	
`border-end-start-radius`	✓ Yes	
`border-image`	✓ Yes	
`border-image-outset`	✓ Yes	
`border-image-repeat`	✓ Yes	
`border-image-slice`	✓ Yes	
`border-image-source`	✓ Yes	
`border-image-width`	✓ Yes	
`border-inline`	✓ Yes	
`border-inline-color`	✓ Yes	
`border-inline-end`	✓ Yes	
`border-inline-end-color`	✓ Yes	
`border-inline-end-style`	✓ Yes	
`border-inline-end-width`	✓ Yes	
`border-inline-start`	✓ Yes	
`border-inline-start-color`	✓ Yes	
`border-inline-start-style`	✓ Yes	
`border-inline-start-width`	✓ Yes	
`border-inline-style`	✓ Yes	

Property	PdfPecker	Notes
`border-inline-width`	✓ Yes	
`border-left`	✓ Yes	
`border-left-color`	✓ Yes	
`border-left-style`	✓ Yes	
`border-left-width`	✓ Yes	
`border-radius`	✓ Yes	Elliptical radii and percentages.
`border-right`	✓ Yes	
`border-right-color`	✓ Yes	
`border-right-style`	✓ Yes	
`border-right-width`	✓ Yes	
`border-shape`	✕ No	
`border-start-end-radius`	✓ Yes	
`border-start-start-radius`	✓ Yes	
`border-style`	✓ Yes	
`border-top`	✓ Yes	
`border-top-color`	✓ Yes	
`border-top-left-radius`	✓ Yes	
`border-top-right-radius`	✓ Yes	
`border-top-style`	✓ Yes	
`border-top-width`	✓ Yes	
`border-width`	✓ Yes	
`box-shadow`	✓ Yes	Inset shadows on inline elements are not drawn.
`corner-block-end-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-block-start-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-bottom-left-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-bottom-right-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-bottom-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-end-end-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-end-start-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-inline-end-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-inline-start-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-left-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-right-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.

Property	PdfPecker	Notes
`corner-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-start-end-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-start-start-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-top-left-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-top-right-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`corner-top-shape`	✓ Yes	Shapes the border-radius corners, as Chrome draws them.
`outline`	✓ Yes	
`outline-color`	✓ Yes	
`outline-offset`	✓ Yes	
`outline-style`	✓ Yes	
`outline-width`	✓ Yes	

Backgrounds and images

Property	PdfPecker	Notes
`background`	✓ Yes	Several layers, gradients and images.
`background-attachment`	✓ Yes	fixed images are placed against the paper on every page, as in Chrome; fixed gradients move with their box.
`background-clip`	✓ Yes	Also text and -webkit-background-clip.
`background-color`	✓ Yes	
`background-image`	✓ Yes	
`background-origin`	✓ Yes	
`background-position`	✓ Yes	
`background-position-x`	✓ Yes	
`background-position-y`	✓ Yes	
`background-repeat`	✓ Yes	
`background-repeat-x`	✕ No	Use background-repeat.
`background-repeat-y`	✕ No	Use background-repeat.
`background-size`	✓ Yes	
`image-orientation`	✓ Yes	from-image (the EXIF orientation) and none.
`image-rendering`	✓ Yes	pixelated and crisp-edges write enlarged images with their pixels repeated, so every viewer shows them sharp.

Property	PdfPecker	Notes
`object-fit`	✓ Yes	
`object-position`	✓ Yes	
`object-view-box`	✓ Yes	inset(), rect() and xywh(), without rounded corners.

Positioning, floats and shapes

Property	PdfPecker	Notes
`anchor-name`	✕ No	Anchor positioning is not supported.
`anchor-scope`	✕ No	Anchor positioning is not supported.
`bottom`	✓ Yes	
`clear`	✓ Yes	
`float`	✓ Yes	Also inline-start, inline-end and footnote.
`inset`	✓ Yes	
`inset-block`	✓ Yes	
`inset-block-end`	✓ Yes	
`inset-block-start`	✓ Yes	
`inset-inline`	✓ Yes	
`inset-inline-end`	✓ Yes	
`inset-inline-start`	✓ Yes	
`left`	✓ Yes	
`overlay`	– No effect	Only for dialogs and popovers in the top layer.
`position`	✓ Yes	fixed repeats on every page; sticky is drawn as relative, as in Chrome print.
`position-anchor`	✕ No	Anchor positioning is not supported.
`position-area`	✕ No	Anchor positioning is not supported.
`position-try`	✕ No	Anchor positioning is not supported.
`position-try-fallbacks`	✕ No	Anchor positioning is not supported.
`position-try-order`	✕ No	Anchor positioning is not supported.
`position-visibility`	✕ No	Anchor positioning is not supported.
`right`	✓ Yes	
`shape-image-threshold`	✓ Yes	
`shape-margin`	✓ Yes	
`shape-outside`	✓ Yes	Basic shapes, boxes and images; SVG images fall back to the margin box.
`top`	✓ Yes	
`z-index`	✓ Yes	

Flexbox, grid and alignment

Property	PdfPecker	Notes
`align-content`	✓ Yes	
`align-items`	✓ Yes	
`align-self`	✓ Yes	
`fit-tolerance`	✕ No	
`flex`	✓ Yes	
`flex-basis`	✓ Yes	
`flex-direction`	✓ Yes	
`flex-flow`	✓ Yes	
`flex-grow`	✓ Yes	
`flex-line-count`	✕ No	
`flex-shrink`	✓ Yes	
`flex-wrap`	✓ Yes	
`grid`	✓ Yes	Template forms and auto-flow forms.
`grid-area`	✓ Yes	
`grid-auto-columns`	✓ Yes	
`grid-auto-flow`	✓ Yes	Including dense.
`grid-auto-rows`	✓ Yes	
`grid-column`	✓ Yes	
`grid-column-end`	✓ Yes	
`grid-column-start`	✓ Yes	
`grid-row`	✓ Yes	
`grid-row-end`	✓ Yes	
`grid-row-start`	✓ Yes	
`grid-template`	✓ Yes	
`grid-template-areas`	✓ Yes	
`grid-template-columns`	✓ Yes	repeat(), minmax(), fit-content(), named lines and subgrid (horizontal writing modes).
`grid-template-rows`	✓ Yes	As grid-template-columns.
`justify-content`	✓ Yes	
`justify-items`	✓ Yes	
`justify-self`	✓ Yes	
`place-content`	✓ Yes	

Property	PdfPecker	Notes
`place-items`	✓ Yes	
`place-self`	✓ Yes	

Multi-column and gap rules

Property	PdfPecker	Notes
`column-count`	✓ Yes	
`column-fill`	✓ Yes	
`column-gap`	✓ Yes	
`column-height`	✕ No	
`column-rule`	✓ Yes	Between the columns of multi-column, grid and flex containers.
`column-rule-break`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-color`	✓ Yes	
`column-rule-inset`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-cap`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-cap-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-cap-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-junction`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-junction-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-junction-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-inset-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-style`	✓ Yes	
`column-rule-visibility-items`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`column-rule-width`	✓ Yes	
`column-span`	✓ Yes	all works on direct children of the multi-column box.
`column-wrap`	✕ No	
`columns`	✓ Yes	

Property	PdfPecker	Notes
`gap`	✓ Yes	
`row-gap`	✓ Yes	
`row-rule`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-break`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-color`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-cap`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-cap-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-cap-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-junction`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-junction-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-junction-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-inset-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-style`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-visibility-items`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`row-rule-width`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-break`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-color`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-inset`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.

Property	PdfPecker	Notes
`rule-inset-cap`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-inset-end`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-inset-junction`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-inset-start`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-overlap`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-style`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-visibility-items`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.
`rule-width`	✓ Yes	Gap decorations in grid, flex and multi-column containers, as Chromium draws them.

Fonts

Property	PdfPecker	Notes
`font`	✓ Yes	
`font-family`	✓ Yes	Bundled fonts, @font-face and fonts you register.
`font-feature-settings`	✓ Yes	Contextual lookups too, through the shaper, for the features you turn on.
`font-kerning`	✓ Yes	
`font-language-override`	✕ No	
`font-optical-sizing`	✓ Yes	
`font-palette`	✕ No	
`font-size`	✓ Yes	
`font-size-adjust`	✓ Yes	
`font-stretch`	✓ Yes	
`font-style`	✓ Yes	
`font-synthesis`	✓ Yes	
`font-synthesis-position`	✓ Yes	
`font-synthesis-small-caps`	✓ Yes	
`font-synthesis-style`	✓ Yes	
`font-synthesis-weight`	✓ Yes	
`font-variant`	½ Partly	Small caps are synthesized when the font lacks them.

Property	PdfPecker	Notes
`font-variant-alternates`	✕ No	
`font-variant-caps`	✓ Yes	Small caps are synthesized when the font has no smcp feature, as in Chrome.
`font-variant-east-asian`	✓ Yes	
`font-variant-emoji`	✕ No	
`font-variant-ligatures`	✓ Yes	Contextual alternates (calt, clig, rclt) are on by default, as in Chrome.
`font-variant-numeric`	✓ Yes	
`font-variant-position`	✓ Yes	
`font-variation-settings`	✓ Yes	
`font-weight`	✓ Yes	
`font-width`	✓ Yes	The new name of font-stretch.

Text, lines and writing modes

Property	PdfPecker	Notes
`alignment-baseline`	✓ Yes	In SVG text.
`baseline-shift`	✕ No	
`baseline-source`	✕ No	
`direction`	✓ Yes	
`dominant-baseline`	✓ Yes	In SVG text.
`glyph-orientation-vertical`	✕ No	
`hanging-punctuation`	✕ No	
`hyphenate-character`	✓ Yes	
`hyphenate-limit-chars`	✓ Yes	
`hyphenate-limit-lines`	✓ Yes	
`hyphens`	✓ Yes	Built-in dictionaries for many languages, chosen by lang.
`initial-letter`	✓ Yes	On ::first-letter, as in Chrome; not in vertical writing modes.
`letter-spacing`	✓ Yes	
`line-break`	✓ Yes	
`line-height`	✓ Yes	
`math-depth`	✕ No	MathML is not laid out.
`math-shift`	✕ No	MathML is not laid out.
`math-style`	✕ No	MathML is not laid out.
`overflow-wrap`	✓ Yes	

Property	PdfPecker	Notes
`ruby-align`	✕ No	Ruby annotations are not laid out.
`ruby-overhang`	✕ No	Ruby annotations are not laid out.
`ruby-position`	✕ No	Ruby annotations are not laid out.
`tab-size`	✓ Yes	
`text-align`	✓ Yes	Right-to-left and bidi lines justify at spaces only.
`text-align-last`	✓ Yes	
`text-autospace`	✕ No	
`text-box`	✓ Yes	Trims block lines as Chrome does; inline boxes are not trimmed, as in Chrome.
`text-box-edge`	✓ Yes	
`text-box-trim`	✓ Yes	
`text-combine-upright`	✕ No	
`text-fit`	✕ No	
`text-indent`	✓ Yes	
`text-justify`	✓ Yes	inter-character spreads CJK text.
`text-orientation`	✕ No	No upright CJK in vertical text.
`text-spacing-trim`	½ Partly	normal, space-all, space-first and trim-start; needs a font with the halt feature.
`text-transform`	✓ Yes	
`text-wrap`	✓ Yes	balance (up to six lines), pretty and stable.
`text-wrap-mode`	✓ Yes	
`text-wrap-style`	✓ Yes	As text-wrap.
`unicode-bidi`	✓ Yes	
`vertical-align`	✓ Yes	Keywords, lengths and percentages.
`-webkit-text-fill-color`	✓ Yes	
`-webkit-text-stroke`	✓ Yes	Text stays selectable text.
`-webkit-text-stroke-color`	✓ Yes	
`-webkit-text-stroke-width`	✓ Yes	
`white-space`	✓ Yes	
`white-space-collapse`	✓ Yes	break-spaces acts as preserve.
`white-space-trim`	✕ No	
`word-break`	✓ Yes	
`word-spacing`	✓ Yes	

Property	PdfPecker	Notes
`wrap-inside`	✕ No	
`writing-mode`	½ Partly	Vertical modes are drawn as turned horizontal text, as Chrome does for Latin; CJK is not set upright.

Text decoration

Property	PdfPecker	Notes
`text-decoration`	✓ Yes	
`text-decoration-color`	✓ Yes	
`text-decoration-inset`	✕ No	
`text-decoration-line`	✓ Yes	
`text-decoration-skip`	✕ No	Use text-decoration-skip-ink.
`text-decoration-skip-ink`	✓ Yes	
`text-decoration-skip-spaces`	✕ No	
`text-decoration-style`	✓ Yes	Wavy and dashed lines differ a little from Chrome.
`text-decoration-thickness`	✓ Yes	
`text-emphasis`	✓ Yes	Marks drawn as Chrome draws them.
`text-emphasis-color`	✓ Yes	
`text-emphasis-position`	✓ Yes	
`text-emphasis-style`	✓ Yes	
`text-shadow`	✓ Yes	
`text-underline-offset`	✓ Yes	
`text-underline-position`	✓ Yes	under and from-font, as in Chrome.

Colour, effects and masks

Property	PdfPecker	Notes
`backdrop-filter`	✓ Yes	The backdrop is a 144 dpi raster; up to 64 per document. Also -webkit-backdrop-filter.
`background-blend-mode`	✓ Yes	
`clip`	✓ Yes	rect() on absolutely and fixed positioned boxes. Deprecated; clip-path is the modern form.
`clip-path`	✓ Yes	Basic shapes (also rect() and xywh()), reference boxes and url(#id).
`clip-rule`	✓ Yes	
`color`	✓ Yes	Every colour syntax; wide-gamut colours are mapped to sRGB.
`color-adjust`	– No effect	Backgrounds always print, as with exact.

Property	PdfPecker	Notes
`color-interpolation-filters`	✓ Yes	
`color-scheme`	– No effect	Pages are always light, as in print.
`dynamic-range-limit`	– No effect	
`filter`	✓ Yes	blur() and drop-shadow() are rasterized at 144 dpi; the rest stay vector.
`flood-color`	✓ Yes	
`flood-opacity`	✓ Yes	
`forced-color-adjust`	– No effect	
`isolation`	✓ Yes	
`lighting-color`	✓ Yes	
`mask`	✓ Yes	Soft masks on HTML and SVG; text stays text. Also -webkit-mask.
`mask-border`	✕ No	
`mask-border-outset`	✕ No	
`mask-border-repeat`	✕ No	
`mask-border-slice`	✕ No	
`mask-border-source`	✕ No	
`mask-border-width`	✕ No	
`mask-clip`	✓ Yes	
`mask-composite`	✓ Yes	
`mask-image`	✓ Yes	
`mask-mode`	✓ Yes	
`mask-origin`	✓ Yes	
`mask-position`	✓ Yes	
`mask-repeat`	✓ Yes	
`mask-size`	✓ Yes	
`mask-type`	✓ Yes	
`mix-blend-mode`	✓ Yes	
`opacity`	✓ Yes	
`print-color-adjust`	– No effect	Backgrounds always print, as with exact.

Transforms and motion paths

Property	PdfPecker	Notes
`backface-visibility`	½ Partly	Drawn as Chrome print does.

Property	PdfPecker	Notes
`offset`	✕ No	Motion paths are not supported.
`offset-anchor`	✕ No	
`offset-distance`	✕ No	
`offset-path`	✕ No	
`offset-position`	✕ No	
`offset-rotate`	✕ No	
`perspective`	½ Partly	Flattened; perspective boxes stay parallelograms.
`perspective-origin`	½ Partly	Flattened with perspective.
`rotate`	✓ Yes	3D axes are flattened.
`scale`	✓ Yes	
`transform`	½ Partly	2D is exact; 3D functions are flattened because PDF is affine.
`transform-box`	✓ Yes	On SVG elements and HTML boxes.
`transform-origin`	✓ Yes	
`transform-style`	½ Partly	preserve-3d is flattened.
`translate`	✓ Yes	

Lists, counters and generated content

Property	PdfPecker	Notes
`content`	✓ Yes	Strings, attr(), counters, quotes, string(), target-counter(), target-text(), leader(), and url() and gradients as inline images.
`counter-increment`	✓ Yes	
`counter-reset`	✓ Yes	
`counter-set`	✓ Yes	
`list-style`	✓ Yes	
`list-style-image`	✓ Yes	Images, image-set() and gradients.
`list-style-position`	✓ Yes	
`list-style-type`	✓ Yes	Every Counter Styles 3 style and @counter-style.
`quotes`	✓ Yes	Including auto by lang.

Tables

Property	PdfPecker	Notes
`border-collapse`	✓ Yes	Collapsed-border conflicts can differ from Chrome in rare cases.

Property	PdfPecker	Notes
`border-spacing`	✓ Yes	
`caption-side`	✓ Yes	
`empty-cells`	✓ Yes	
`table-layout`	✓ Yes	

Paged media and breaks

Property	PdfPecker	Notes
`bookmark-label`	✓ Yes	content(), strings and attr().
`bookmark-level`	✓ Yes	Makes any block a bookmark, or leaves a heading out with none.
`bookmark-state`	✓ Yes	open or closed.
`box-decoration-break`	✓ Yes	clone and slice, on inline boxes and on blocks split across pages, as Chrome prints them.
`break-after`	✓ Yes	
`break-before`	✓ Yes	
`break-inside`	✓ Yes	
`footnote-display`	½ Partly	compact is drawn inline.
`footnote-policy`	½ Partly	block acts like line.
`orphans`	✓ Yes	
`page`	✓ Yes	Named pages with their own @page size and margins.
`page-break-after`	✓ Yes	
`page-break-before`	✓ Yes	
`page-break-inside`	✓ Yes	
`-pdfpecker-image-max-resolution`	✓ Yes	PdfPecker's own property: downsamples images drawn denser than a resolution.
`-pdfpecker-image-quality`	✓ Yes	PdfPecker's own property: writes images as JPEGs at a quality, when that makes them smaller.
`-pdfpecker-repeat`	✓ Yes	PdfPecker's own property: each-page draws a box again on its parent's later pages.
`-pdfpecker-scale-to-fit`	✓ Yes	PdfPecker's own property: shrink fits text of any length into a fixed-size box.
`position: running()`	✓ Yes	Running elements, placed with element() in the page-margin boxes.
`string-set`	✓ Yes	Running headers with string().
`widows`	✓ Yes	

SVG properties

Property	PdfPecker	Notes
`color-interpolation`	✕ No	
`cx`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`cy`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`d`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`fill`	✓ Yes	
`fill-opacity`	✓ Yes	
`fill-rule`	✓ Yes	
`marker`	✓ Yes	
`marker-end`	✓ Yes	
`marker-mid`	✓ Yes	
`marker-start`	✓ Yes	
`paint-order`	✓ Yes	On SVG and on HTML text with -webkit-text-stroke.
`path-length`	✕ No	
`pointer-events`	– No effect	
`r`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`rx`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`ry`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`shape-rendering`	– No effect	
`stop-color`	✓ Yes	From style sheets too, as in Chrome.
`stop-opacity`	✓ Yes	From style sheets too, as in Chrome.
`stroke`	✓ Yes	
`stroke-color`	✕ No	Use stroke.
`stroke-dasharray`	✓ Yes	
`stroke-dashoffset`	✓ Yes	
`stroke-linecap`	✓ Yes	
`stroke-linejoin`	✓ Yes	
`stroke-miterlimit`	✓ Yes	
`stroke-opacity`	✓ Yes	
`stroke-width`	✓ Yes	
`text-anchor`	✓ Yes	In SVG text.
`text-rendering`	– No effect	Kerning and ligatures follow font-kerning and font-variant.
`vector-effect`	✓ Yes	non-scaling-stroke; markers still scale.

Property	PdfPecker	Notes
`x`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.
`y`	✓ Yes	From style sheets too, winning over the attribute, as in Chrome.

Cascade, variables and containers

Property	PdfPecker	Notes
`all`	✓ Yes	
`container`	½ Partly	Size queries and style(--custom); no scroll-state().
`container-name`	✓ Yes	
`container-type`	✓ Yes	
`--*`	✓ Yes	Custom properties (--*) and var() anywhere, @property too.
`interpolate-size`	– No effect	Only affects animations.

Prefixed and legacy names

Property	PdfPecker	Notes
`grid-column-gap`	✓ Yes	Read as column-gap.
`grid-gap`	✓ Yes	Read as gap.
`grid-row-gap`	✓ Yes	Read as row-gap.
`-moz-tab-size`	✓ Yes	Read as tab-size.
`-webkit-backdrop-filter`	✓ Yes	Read as backdrop-filter.
`-webkit-background-clip`	✓ Yes	Read as background-clip.
`-webkit-box, -webkit-box-orient, -webkit-box-pack, -webkit-box-align, -webkit-box-flex`	✓ Yes	Read as the old flexbox, as in Chrome.
`-webkit-box-decoration-break`	✓ Yes	Read as box-decoration-break.
`-webkit-hyphenate-character`	✓ Yes	Read as hyphenate-character.
`-webkit-hyphens`	✓ Yes	Read as hyphens.
`-webkit-line-clamp`	✓ Yes	Read as line-clamp.
`-webkit-mask`	✓ Yes	Read as mask, and every -webkit-mask-* longhand.
`-webkit-transform, -webkit-border-radius, -webkit-box-shadow and the other -webkit- names Chrome accepts`	✓ Yes	Read as their standard property.
`-webkit-writing-mode`	✓ Yes	Read as writing-mode.
`word-wrap`	✓ Yes	Read as overflow-wrap.

Interaction, animation and scrolling

Property	PdfPecker	Notes
`accent-color`	½ Partly	Check boxes and radio buttons; progress bars and range sliders keep their default colours.
`animation`	– No effect	
`animation-composition`	– No effect	
`animation-delay`	– No effect	
`animation-direction`	– No effect	
`animation-duration`	– No effect	
`animation-fill-mode`	– No effect	
`animation-iteration-count`	– No effect	
`animation-name`	– No effect	
`animation-play-state`	– No effect	
`animation-range`	– No effect	
`animation-range-end`	– No effect	
`animation-range-start`	– No effect	
`animation-timeline`	– No effect	
`animation-timing-function`	– No effect	
`animation-trigger`	– No effect	
`appearance`	½ Partly	none turns check boxes and radio buttons into plain boxes you style; other controls keep their look.
`caret-animation`	– No effect	
`caret-color`	– No effect	
`caret-shape`	– No effect	
`cursor`	– No effect	
`field-sizing`	✕ No	Form fields keep their default size.
`ime-mode`	– No effect	
`interactivity`	– No effect	
`interest-delay`	– No effect	
`interest-delay-end`	– No effect	
`interest-delay-start`	– No effect	
`link-parameters`	✕ No	
`overflow-anchor`	– No effect	
`overscroll-behavior`	– No effect	

Property	PdfPecker	Notes
`overscroll-behavior-block`	– No effect	
`overscroll-behavior-inline`	– No effect	
`overscroll-behavior-x`	– No effect	
`overscroll-behavior-y`	– No effect	
`resize`	– No effect	
`scroll-initial-target`	– No effect	
`scroll-margin`	– No effect	
`scroll-margin-block`	– No effect	
`scroll-margin-block-end`	– No effect	
`scroll-margin-block-start`	– No effect	
`scroll-margin-bottom`	– No effect	
`scroll-margin-inline`	– No effect	
`scroll-margin-inline-end`	– No effect	
`scroll-margin-inline-start`	– No effect	
`scroll-margin-left`	– No effect	
`scroll-margin-right`	– No effect	
`scroll-margin-top`	– No effect	
`scroll-padding`	– No effect	
`scroll-padding-block`	– No effect	
`scroll-padding-block-end`	– No effect	
`scroll-padding-block-start`	– No effect	
`scroll-padding-bottom`	– No effect	
`scroll-padding-inline`	– No effect	
`scroll-padding-inline-end`	– No effect	
`scroll-padding-inline-start`	– No effect	
`scroll-padding-left`	– No effect	
`scroll-padding-right`	– No effect	
`scroll-padding-top`	– No effect	
`scroll-snap-align`	– No effect	
`scroll-snap-stop`	– No effect	
`scroll-snap-type`	– No effect	
`scroll-timeline`	– No effect	
`scroll-timeline-axis`	– No effect	

Property	PdfPecker	Notes
`scroll-timeline-name`	– No effect	
`scrollbar-color`	– No effect	
`scrollbar-width`	– No effect	
`speak`	– No effect	
`speak-as`	– No effect	
`text-size-adjust`	– No effect	Only for phone browsers.
`timeline-scope`	– No effect	
`timeline-trigger`	– No effect	
`timeline-trigger-activation-range`	– No effect	
`timeline-trigger-activation-range-end`	– No effect	
`timeline-trigger-activation-range-start`	– No effect	
`timeline-trigger-active-range`	– No effect	
`timeline-trigger-active-range-end`	– No effect	
`timeline-trigger-active-range-start`	– No effect	
`timeline-trigger-name`	– No effect	
`timeline-trigger-source`	– No effect	
`touch-action`	– No effect	
`transition`	– No effect	
`transition-behavior`	– No effect	
`transition-delay`	– No effect	
`transition-duration`	– No effect	
`transition-property`	– No effect	
`transition-timing-function`	– No effect	
`trigger-scope`	– No effect	
`user-select`	– No effect	
`view-timeline`	– No effect	
`view-timeline-axis`	– No effect	
`view-timeline-inset`	– No effect	
`view-timeline-name`	– No effect	
`view-transition-class`	– No effect	

Property	PdfPecker	Notes
`view-transition-group`	– No effect	
`view-transition-name`	– No effect	
`view-transition-scope`	– No effect	
`will-change`	– No effect	
`window-drag`	– No effect	
`zoom`	✓ Yes	Scales a box and its content as Chrome does.

At-rules

At-rule	PdfPecker	Notes
`@charset`	✓ Yes	
`@import`	✓ Yes	With layer(), supports() and media queries. Remote files load only through your ResourceResolver.
`@media`	✓ Yes	print and all match, screen does not. Page width, height, aspect-ratio, orientation, ranges, and/or/not, plus resolution, color, prefers-color-scheme (light), hover and pointer (none), overflow-block (paged) and more.
`@supports`	✓ Yes	Property tests, selector(), font-format() and font-tech().
`@layer`	✓ Yes	Including the !important reversal.
`@container`	½ Partly	Size queries and style(--custom); no scroll-state().
`@property`	✓ Yes	syntax, inherits and initial-value.
`@font-face`	✓ Yes	src (url(), local(), format()), font-weight and font-stretch ranges, font-style, unicode-range. font-display is accepted. size-adjust, ascent-override, descent-override and line-gap-override as in Chrome.
`@counter-style`	✓ Yes	Every system and descriptor.
`@page`	✓ Yes	size, margins, borders, padding, backgrounds and background images, :first, :left, :right, :blank, named pages and all 16 margin boxes. bleed and marks (crop, cross) with trim and bleed boxes, and :nth() with page groups. page-orientation is ignored.
`@footnote`	✓ Yes	The footnote area inside @page.
`@namespace`	½ Partly	Accepted and ignored; HTML needs no namespaces.
`CSS nesting (&)`	✓ Yes	
`@scope`	✓ Yes	Scope roots, limits and proximity, as in Chrome.

At-rule	PdfPecker	Notes
`@font-feature-values`	✕ No	
`@font-palette-values`	✕ No	
`@custom-media`	✕ No	
`@function`	✕ No	
`@position-try`	✕ No	Anchor positioning is not supported.
`@keyframes`	– No effect	No animations on paper.
`@starting-style`	– No effect	Only used for transitions.
`@view-transition`	– No effect	
`@document`	✕ No	Obsolete Firefox rule.

Selectors

A rule whose selector PdfPecker cannot parse is skipped as a whole, as browsers do. Interaction states are parsed and never match, as in Chrome's print output.

Selector	PdfPecker	Notes
Type, class, id and universal (*)	✓ Yes	
Attribute: [a], =, ~=, |=, ^=, $=, *= and the i and s flags	✓ Yes	
Combinators: descendant, >, + and ~	✓ Yes	
Nesting selector &	✓ Yes	
`:root`	✓ Yes	
`:empty`	✓ Yes	
`:first-child`, `:last-child`, `:only-child`	✓ Yes	
`:nth-child()`, `:nth-last-child()`	✓ Yes	Including An+B of S.
`:first-of-type`, `:last-of-type`, `:only-of-type`, `:nth-of-type()`, `:nth-last-of-type()`	✓ Yes	
`:not()`, `:is()`, `:where()`	✓ Yes	Complex selector lists; :is() and :where() are forgiving. Also :matches() and :-webkit-any().

Selector	PdfPecker	Notes
`:has()`	✓ Yes	Relative selectors; :has() cannot nest inside :has(), as the spec says.
`:lang()`	✓ Yes	
`:dir()`	✓ Yes	From the dir attribute.
`:link`, `:any-link`	✓ Yes	
`:scope`	✓ Yes	Matches the root.
`:defined`	✓ Yes	
`:checked`, `:default`	✓ Yes	
`:disabled`, `:enabled`	✓ Yes	
`:required`, `:optional`	✓ Yes	
`:read-only`, `:read-write`	✓ Yes	
`:placeholder-shown`	✓ Yes	
`::before`, `::after`	✓ Yes	
`::marker`	✓ Yes	
`::first-letter`	✓ Yes	With initial-letter for drop caps.
`::first-line`	✓ Yes	
`::placeholder`	✓ Yes	On text fields and textareas.
`::footnote-call`, `::footnote-marker`	✓ Yes	From CSS Generated Content for Paged Media.
`@page` `:first`, `:left`, `:right`, `:blank`	✓ Yes	
`:valid`, `:invalid`	✓ Yes	As Chrome matches them before anyone types: required, pattern, min, max, step, email and url.
`:in-range`, `:out-of-range`	✓ Yes	As Chrome matches a page nobody has typed into.
`:open`	✓ Yes	details and dialog with the open attribute.
`:heading`, `:heading()`	✕ No	
`:state()`, `:host`, `:host()`, `:host-context()`, `::part()`, `::slotted()`	✕ No	No shadow DOM: scripts do not run.
`::details-content`	✕ No	
`::checkmark`, `::picker()`, `::picker-icon`	✕ No	
Column combinator || and :nth-col()	✕ No	
Namespace selector ns|E	✕ No	
`:hover`, `:active`, `:focus`, `:focus-visible`, `:focus-within`, `:visited`, `:target`	– No effect	Parsed, never match, as in Chrome print.

Selector	PdfPecker	Notes
`:indeterminate`, `:autofill`, `:user-valid`, `:user-invalid`, `:fullscreen`, `:modal`, `:popover-open`, `:playing`, `:paused`	– No effect	Parsed, never match.
`::selection`, `::highlight()`, `::spelling-error`, `::grammar-error`, `::target-text`, `::search-text`	– No effect	Nothing is selected on paper.
`::backdrop`, `::file-selector-button`, `::cue`, `::view-transition-*`, `::scroll-marker`, `::scroll-button()`	– No effect	

Units, colours and functions

Value	PdfPecker	Notes
px, pt, pc, in, cm, mm, Q, %	✓ Yes	
em, rem	✓ Yes	
ex, ch, cap, ic, lh and rex, rch, rcap, ric, rlh	✓ Yes	Measured from the font.
vw, vh, vmin, vmax, vi, vb and the sv*, lv*, dv* sets	✓ Yes	The viewport is the page content area.
cqw, cqh, cqi, cqb, cqmin, cqmax	✓ Yes	
fr	✓ Yes	In grid tracks.
deg, rad, grad, turn	✓ Yes	
dpi, dpcm, dppx	✓ Yes	In media queries.
s, ms, Hz, kHz	– No effect	Only used by animations and speech.
Named colours, hex (3, 4, 6, 8 digits), currentColor, transparent	✓ Yes	
rgb(), rgba(), hsl(), hsla(), hwb()	✓ Yes	
lab(), lch(), oklab(), oklch(), color()	✓ Yes	Wide-gamut colours are mapped to sRGB.
color-mix(), relative colours (from)	✓ Yes	
light-dark()	✓ Yes	Takes the light colour.
contrast-color()	✓ Yes	Black or white, whichever contrasts more.
Device CMYK and spot colours (device-cmyk())	✕ No	Colours are written as RGB.
calc(), min(), max(), clamp()	✓ Yes	
round(), mod(), rem(), abs(), sign()	✓ Yes	
sin(), cos(), tan(), asin(), acos(), atan(), atan2()	✓ Yes	
pow(), sqrt(), hypot(), log(), exp()	✓ Yes	
sibling-index(), sibling-count()	✓ Yes	
if() with media(), supports() and style()	✓ Yes	
calc-size(), progress(), random()	✕ No	

Value	PdfPecker	Notes
var() and custom properties	✓ Yes	Anywhere, in shorthands and inside math.
env()	✕ No	
attr()	✓ Yes	Typed attr() in any property, as in Chrome 133; never in url() or other resource functions.
url()	✓ Yes	Data URIs, local files and, through your ResourceResolver, remote files. Stylesheets in data: URLs too.
linear-gradient(), radial-gradient(), conic-gradient() and repeating forms	✓ Yes	Drawn as PDF shadings, with colour interpolation methods such as in oklch. Also the old -webkit-gradient() syntax.
image-set()	✓ Yes	In backgrounds, list markers and generated content, chosen as Chrome chooses when it prints.
cross-fade(), element(), paint()	✕ No	element() as an image; element() of a running element in page-margin boxes is supported.
counter(), counters(), symbols()	✓ Yes	
string(), content(), target-counter(), target-text(), leader()	✓ Yes	Paged media functions.
target-counters()	✓ Yes	Every level of a nested counter at the linked element.
anchor(), anchor-size()	✕ No	
Basic shapes: inset(), circle(), ellipse(), polygon(), path()	✓ Yes	In clip-path and shape-outside.
xywh(), rect(), superellipse()	✓ Yes	rect() and xywh() in clip-path and shape-outside; superellipse() in corner-shape.
shape(), ray()	✕ No	
repeat(), minmax(), fit-content()	✓ Yes	In grid tracks.
Transform functions: translate, scale, rotate, skew, matrix	✓ Yes	
3D transform functions and perspective()	½ Partly	Flattened, because PDF is affine.
Filter functions: blur, brightness, contrast, drop-shadow, grayscale, hue-rotate, invert, opacity, saturate, sepia, url()	✓ Yes	blur and drop-shadow are rasterized at 144 dpi.
inherit, initial, unset, revert, revert-layer	✓ Yes	Also on shorthands.
!important	✓ Yes	

PDF feature checklist

> **Changed in 1.24.0:** File format and structure, Security and encryption, Reading, rendering and conversion.

What the PDF format and other PDF libraries offer, and whether PdfPecker does it. Of the 234 features below, PdfPecker supports 127, supports 38 in part and leaves out 69. Some of those are left out on purpose, such as JavaScript in PDFs and the old RC4 encryption. The list follows the PDF standard, ISO 32000-2 (PDF 2.0, which includes PDF 1.7), and the ISO profiles and RFCs built on it. The limitations page has the details of what is missing in HTML and CSS.

File format and structure Changed in 1.24.0

The PDF file itself: versions, cross-reference tables, pages and their boxes.

Feature	PdfPecker	Notes
Write PDF 1.4 and 1.7	✓ Yes	1.4 for plain output, 1.7 when tagged, encrypted or PDF/A.
Write PDF 2.0	✓ Yes	Used for PDF/A-4 and PDF/UA-2 output.
Compressed object streams and xref streams	✓ Yes	`PdfCompactor` packs objects into them; `HtmlToPdf` writes a classic xref table.
Incremental updates	✓ Yes	Signing and long-term validation append to the file, so earlier signatures stay valid.
Linearization (Fast Web View)	✓ Yes	`PdfOptions.Linearize` for new documents, `PdfLinearizer` for existing ones; encrypted, PDF/A and PDF/UA output too.
Open damaged files	✓ Yes	Rebuilds a broken cross-reference table from the object headers.
Page sizes, orientation and `@page`	✓ Yes	A4, Letter, Legal, A3, A5, any custom size, landscape, and per-page sizes from CSS `@page`.
Page rotation	✓ Yes	`PdfPages.Rotate` on existing files; `/Rotate` is honoured when rendering.
Page boxes (Crop, Bleed, Trim, Art)	½ Partly	CropBox is read and kept. CSS `@page { bleed; marks: crop cross }` writes the bleed and trim boxes with crop and cross marks; art boxes are not written.
Very large documents	✓ Yes	Streaming merge and `MaxPages` limits; the readers have resource limits for hostile input.
All object types (names, strings, arrays, dictionaries, streams)	✓ Yes	
Text strings in PDFDocEncoding and UTF-16	✓ Yes	Bookmarks, properties, field values and tags use UTF-16 when the text needs it.
Hybrid-reference files (/XRefStm)	½ Partly	Read, not written. Read; not written, since every current reader understands plain xref streams.
Name trees and number trees	✓ Yes	Embedded files, page labels and the structure parent tree.

Feature	PdfPecker	Notes
Page tree with inherited attributes	✓ Yes	
Developer extensions (/Extensions)	✓ Yes	Adobe extension level 8 is declared for AES-256 in PDF 1.7 files.
Document parts (/DPartRoot)	✕ No	PDF 2.0 page grouping, used by PDF/VT.
External file streams (/F on a stream)	✕ No	Stream data kept in another file. Left out on purpose: conversion never reads files by itself.
User units (/UserUnit)	½ Partly	Read, not written. Pages larger than 200 inches render at their real size.
Thumbnail images (/Thumb)	✕ No	Viewers make their own; `PdfCompactor` drops stored ones.
Page-piece dictionaries (/PieceInfo)	✕ No	Application data; `PdfCompactor` removes it.
FlateDecode with PNG and TIFF predictors	✓ Yes	Everything PdfPecker writes is Flate-compressed unless it is a JPEG.
Crypt filters	✓ Yes	AES-256 standard crypt filter; Identity and named filters read.

Creating content

Ways to produce a new PDF.

Feature	PdfPecker	Notes
HTML and CSS to PDF	✓ Yes	Own parser, CSS engine and layout: flexbox, grid, tables, floats, multi-column, SVG and more.
Markdown to PDF	✓ Yes	CommonMark 0.31 and GitHub Flavored Markdown, with footnotes.
Images to PDF	✓ Yes	PNG, JPEG, GIF, WebP, BMP and TIFF, one per page.
Fluent C# API	✓ Yes	PdfPecker.Fluent, in the style of QuestPDF.
Headers, footers and page numbers	✓ Yes	Running bands, `@page` margin boxes, `{page}` and `{pages}` tokens, and a different or no header and footer on the first page.
Table of contents with page numbers	✓ Yes	`target-counter()` and `leader()` from CSS Paged Media.
Footnotes	✓ Yes	CSS `float: footnote`, and Markdown footnotes at the page bottom.
Scale content to fit a box	✓ Yes	`-pdfpecker-scale-to-fit: shrink` for labels, badges and certificates.
Barcodes and QR codes	✓ Yes	QR, EPC payment QR, Swiss QR-bill, Code 128, EAN-13, Data Matrix and GS1 Data Matrix, drawn as vectors.
Content rendered by JavaScript	✕ No	By design: no scripts run. Render the HTML on the server first.
Low-level drawing API	✕ No	There is no canvas-style API to draw paths and text at coordinates; content comes from HTML, Markdown, images or the fluent API.

Text and fonts

How text is stored, so it looks right and stays searchable.

Feature	PdfPecker	Notes
Standard 14 fonts	✓ Yes	Used by default; `EmbedStandardFonts` embeds metric-compatible fonts instead.
TrueType and OpenType (CFF) embedding	✓ Yes	Subset to the glyphs used.
WOFF and WOFF2 web fonts	✓ Yes	Decoded by the library's own WOFF2 decoder.
Variable fonts	✓ Yes	Instanced at the weight and width the CSS asks for.
Copyable, searchable Unicode text	✓ Yes	`ToUnicode` maps on every embedded font.
Kerning, ligatures and OpenType features	✓ Yes	Numerals, fractions, small capitals, super and subscripts.
Complex scripts	½ Partly	Arabic, Hebrew, nine Indic scripts, Thai and Lao are shaped. Khmer, Myanmar, Sinhala, Tibetan and others are not.
Vertical text	½ Partly	Drawn as turned horizontal text; CJK characters are not set upright.
Fallback fonts	✓ Yes	Bundled fonts for Central European, Greek, Cyrillic and symbols when your fonts lack a character.
Colour fonts and emoji	½ Partly	COLR colour fonts, versions 0 and 1 (such as Noto Color Emoji), print in colour as text; SVG and bitmap colour glyphs are not drawn.
Outlined and stroked text	✓ Yes	`-webkit-text-stroke` and `paint-order`, written as PDF text render modes.
Text state (spacing, scaling, rise, leading)	✓ Yes	
Embedded Type 1 fonts	½ Partly	Read, not written.
Type 3 fonts	✓ Yes	Written for colour glyphs.
Composite (Type 0) fonts with CIDFonts	✓ Yes	Identity-H encoding.
Embedded CMaps and /UseCMap	✓ Yes	

Graphics, images and colour

What can be drawn on a page.

Feature	PdfPecker	Notes
Vector paths and inline SVG	✓ Yes	SVG becomes PDF vector paths, not pictures.
Gradients	✓ Yes	Linear, radial and conic gradients as PDF shadings.

Feature	PdfPecker	Notes
Transparency, soft masks and blend modes	✓ Yes	Opacity, `mask-image`, `mix-blend-mode`, `isolation`.
Clipping	✓ Yes	`overflow`, `border-radius`, `clip-path`.
Tiling patterns	✓ Yes	Repeated backgrounds and SVG patterns.
Images: JPEG, PNG, GIF, WebP, BMP, TIFF	✓ Yes	JPEG is passed through unchanged; PNG alpha becomes a soft mask.
AVIF images	✕ No	Needs an AV1 decoder, which is not planned.
sRGB output intent	✓ Yes	Written for PDF/A.
CMYK	½ Partly	CMYK JPEGs are embedded (converted to RGB for PDF/A). There is no CSS `device-cmyk()` colour, so text and shapes are always RGB.
Spot colours (Separation, DeviceN)	✕ No	Not written. Read and rendered in existing PDFs.
Overprint control	✕ No	Not written, and not simulated when rendering.
Filters and blur	½ Partly	CSS filter, `backdrop-filter` and blurred shadows; blurs are drawn as 144 dpi images, colour filters stay vectors.
Paths, fills and strokes	✓ Yes	Line widths, caps, joins, miter limits and dash patterns.
Graphics state (/ExtGState)	✓ Yes	Opacity, blend mode, soft mask and line settings.
Form XObjects	✓ Yes	Shared content, stamps, watermarks and appearances.
Reference XObjects (page from another file)	✕ No	
PostScript XObjects	✕ No	Deprecated in the spec; skipped when rendering, as viewers do.
Rendering intents	✕ No	Screen rendering ignores them, as most viewers do.
Halftones, transfer functions, black generation, undercolour removal	✕ No	Device-dependent printer settings; ignored when rendering. `PdfToPdfA` removes the ones PDF/A forbids.
Marked content (BMC, BDC, EMC)	✓ Yes	Written for tags, artifacts and /ActualText; read for reading order and extraction.
DeviceRGB and DeviceGray	✓ Yes	
Indexed colour	½ Partly	Read, not written. Palette PNGs are stored as RGB.
Functions: sampled, exponential, stitching, PostScript calculator	½ Partly	Exponential and stitching functions are written for gradients; all four types are read.
Knockout groups	✕ No	Rare outside print production.
Image soft masks with /Matte (pre-multiplied colour)	✕ No	
Inline images (BI ... EI)	½ Partly	Read, not written.
Stencil and colour-key masks	½ Partly	Read, not written.
Alternate images	✕ No	

Navigation and interactivity

What a reader can click or open in a viewer.

Feature	PdfPecker	Notes
Bookmarks (outlines)	✓ Yes	Built from h1 to h6 or from CSS `bookmark-level`, `bookmark-label` and `bookmark-state`; kept and nested when merging.
Internal links	✓ Yes	Links to #ids jump to the right page and position.
External web links	✓ Yes	URI link annotations.
Named destinations	½ Partly	Read when rendering and converting to HTML. New files use direct page destinations instead of a /Dests name tree.
Page labels (i, ii, iii, A-1)	✓ Yes	`PdfOptions.PageLabels` for new documents and `PdfMergeOptions.PageLabels` for merges; kept when merging otherwise.
Viewer preferences	✓ Yes	`PdfOptions.Viewer`: page mode, page layout, start page and zoom, window settings, print scaling and duplex.
Annotations	½ Partly	Links and form widgets are written. Notes, highlights, stamps, free text, ink and shapes cannot be created. Existing ones are rendered and their text extracted.
Layers (optional content)	½ Partly	Not created. Existing layers are kept on merge and their visibility is honoured when rendering.
JavaScript in the PDF	✕ No	By design: no document or field scripts are written or run; `PdfSanitizer` removes them from PDFs you receive.
Page transitions and presentation mode	✕ No	
Multimedia, 3D and rich media	✕ No	Sound, video, 3D models and RichMedia annotations.
PDF portfolios (collections)	✕ No	
Article threads	✕ No	
Bookmark colour and bold or italic style	✕ No	
Navigation nodes (presentation steps)	✕ No	
Measurement and geospatial features	✕ No	
Document requirements (/Requirements)	✕ No	

Annotations

Which kinds of annotation PdfPecker writes, and which it shows when rendering. Existing annotations are drawn from their stored appearance and kept when merging.

Feature	PdfPecker	Notes
Text (sticky note) and Popup	½ Partly	Shown when the PDF stores its appearance; not created. The note icon renders; pop-ups do not open, as on paper.
FreeText	½ Partly	Shown when the PDF stores its appearance; not created.
Line, Square, Circle, Polygon and PolyLine	½ Partly	Shown when the PDF stores its appearance; not created.
Highlight, Underline, Squiggly and StrikeOut	½ Partly	Shown when the PDF stores its appearance; not created.
Caret, Rubber stamp and Ink	½ Partly	Shown when the PDF stores its appearance; not created.
Redact (marked for redaction)	✕ No	`PdfRedactor` removes content directly instead of marking it.
Watermark annotations	½ Partly	Shown when the PDF stores its appearance; not created. Watermarks are written as page content.
Printer's mark and trap network	½ Partly	Shown when the PDF stores its appearance; not created. Crop and cross marks from `@page { marks }` are page content.
Projection	✕ No	
Annotations without an appearance stream	✕ No	PdfPecker draws only stored appearances; it does not build one for a note or highlight that has none.
Hidden, NoView and Print flags	✓ Yes	

Actions

What a link, bookmark or field can do when clicked.

Feature	PdfPecker	Notes
GoTo (jump inside the document)	✓ Yes	
GoToR and GoToE (other or embedded PDF)	✕ No	
GoToDp (document part, PDF 2.0)	✕ No	
Named actions (next page, print)	✕ No	
Launch	✕ No	By design; `PdfSanitizer` removes it.
Hide, SetOCGState, Rendition, Sound, Movie, Trans, GoTo3DView	✕ No	Media ones are removed by `PdfSanitizer`.
Additional actions (/AA) on pages, fields and the document	✕ No	

Forms (AcroForm)

Fillable fields.

Feature	PdfPecker	Notes
Create fields from HTML	✓ Yes	Text, multi-line, password, checkbox, radio group, drop-down, list box and button.
Read fields of any PDF	✓ Yes	Name, type, value, options, flags and page.
Fill fields	✓ Yes	With new appearance streams, optionally with your own font.
Flatten fields	✓ Yes	Turn the fields into ordinary page content.
Field calculations, formats and validation	✕ No	These need JavaScript.
Submit and reset actions	✕ No	
XFA forms	✕ No	XFA data is removed when filling; the AcroForm fields are filled instead.
Fill in complex scripts	½ Partly	Arabic, Hebrew and Indic values are stored but not shaped in the appearance.
Comb, file-select and rich-text fields	½ Partly	Not created. Read and filled as ordinary text fields, without comb spacing or rich text.
FDF and XFDF import and export	✕ No	

Security and encryption Changed in 1.24.0

Passwords, permissions and protection.

Feature	PdfPecker	Notes
AES-256 encryption	✓ Yes	Standard security handler revision 6, on new or existing PDFs.
User and owner passwords	✓ Yes	
Permissions	✓ Yes	Print, high-quality print, copy, modify, annotate, fill forms, assemble, accessibility.
Open encrypted PDFs	✓ Yes	RC4 40 to 128 bit, AES-128 and AES-256 (revisions 2 to 6).
Remove a password	✓ Yes	`PdfProtector.Decrypt`.
Write RC4 or AES-128	✕ No	Only the current AES-256 is written, on purpose.
Certificate (public-key) encryption	✕ No	The Adobe.PubSec handler is not supported.
Redaction	✓ Yes	`PdfRedactor` removes text, image pixels, shapes, links and form fields under areas or matching text from the file, and draws a box over them.
Remove JavaScript and active content	✓ Yes	`PdfSanitizer` takes out scripts, launch, submit and media actions, XFA, media and 3D annotations, and optionally attachments.
Sanitising	½ Partly	`PdfSanitizer` removes active content and `PdfCompactor` metadata and application data; hidden layers and hidden text stay.
AES-256 revision 5 (Adobe extension level 3)	½ Partly	Read, not written.
Unencrypted metadata (/EncryptMetadata false)	½ Partly	Read, not written.

Feature	PdfPecker	Notes
Unencrypted wrapper documents (PDF 2.0)	✕ No	
AES-GCM encryption (ISO/TS 32003)	✕ No	
Integrity protection (ISO/TS 32004)	✕ No	

Digital signatures

Signing and checking signatures.

Feature	PdfPecker	Notes
PAdES and PKCS#7 detached signatures	✓ Yes	`ETSI.CAdES.detached` and `adbe.pkcs7.detached`.
RSA, RSA-PSS and ECDSA keys	✓ Yes	SHA-256 and other hash algorithms.
Visible signature box	✓ Yes	Page, position, size, text and border.
Empty signature fields	✓ Yes	`data-pdf-signature` in the HTML makes a field that `PdfSigner` signs later.
External signers	✓ Yes	HSMs, smart cards, PKCS#11 tokens and cloud signing services, also async.
RFC 3161 timestamps	✓ Yes	From the timestamp server you name.
Document timestamps	✓ Yes	
Long-term validation	✓ Yes	DSS and VRI with OCSP and CRL data, PAdES B-LT and B-LTA.
Several signatures	✓ Yes	Each signature is an incremental update, so earlier ones stay valid.
Verify signatures	✓ Yes	Byte range, digest, signature, timestamp and certificate chain; SHA-1 signatures are not valid unless allowed.
Certification signatures (DocMDP)	✓ Yes	Levels 1 to 3; verification checks later changes against the level, shadow attacks and a dropped `/Perms` included.
Change analysis	½ Partly	Changes after a certification signature are judged against its level; after approval signatures, updates are only counted.
Revocation checks during verification	½ Partly	Uses the platform's online check; the OCSP and CRL data stored in the PDF is not used.
adbe.pkcs7.sha1	½ Partly	Not written. Verified only with `AllowSha1`.
adbe.x509.rsa_sha1	✕ No	An obsolete PKCS#1 format.
Field locking (FieldMDP, /Lock)	✕ No	
Usage rights (UR3)	✕ No	Adobe Reader extensions.
Seed values (/SV)	✕ No	
EdDSA and SHA-3 (ISO 32001, ISO 32002)	✕ No	

Tagging and accessibility

The structure that screen readers and reflow use.

Feature	PdfPecker	Notes
PDF 2.0 structure namespaces	½ Partly	Written for PDF/UA-2.
Alternate text (/Alt)	✓ Yes	From `alt` and form tooltips.
Replacement text (/ActualText)	✓ Yes	For ligatures and glyphs that copy as other characters.
Abbreviation expansions (/E)	✕ No	
Table and list attributes	✓ Yes	Headers, scope, row and column spans, list numbering.
Layout attributes (placement, spacing, colour)	✕ No	
Artifacts (headers, footers, decoration)	✓ Yes	Running bands and decoration are marked; extraction can leave them out.
Pronunciation hints	✕ No	
Tagging an untagged existing PDF	✕ No	

Print production

Settings for print shops and prepress.

Feature	PdfPecker	Notes
Printer's marks	½ Partly	Crop and cross marks from `@page { marks }`, as page content.
Separation dictionaries	✕ No	
Trapping	✕ No	
OPI (Open Prepress Interface)	✕ No	`PdfToPdfA` removes it.

Standards

ISO and industry profiles.

Feature	PdfPecker	Notes
PDF/A-1	✕ No	Only PDF/A-2 and later.
PDF/A-2b, 2u, 3b, 3u	✓ Yes	Checked with veraPDF in CI.
PDF/A-4 and PDF/A-4f	✓ Yes	
PDF/A level a (2a, 3a)	½ Partly	Tagging is there (PDF/UA), but the level a conformance itself is not declared.
PDF/A-4e (engineering)	✕ No	
Tagged PDF	✓ Yes	Structure tree built from the HTML.

Feature	PdfPecker	Notes
PDF/UA-1 and PDF/UA-2	✓ Yes	Accessible PDFs, also combined with PDF/A.
Factur-X and ZUGFeRD	✓ Yes	All profiles from Minimum to Extended and XRechnung; embedded and read back.
XRechnung, Peppol BIS 3.0 and OIOUBL XML	✓ Yes	Built from invoice data, totals checked against EN 16931.
PDF/X (print production)	✕ No	No PDF/X-1a, X-3 or X-4.
PDF/E and PDF/VT	✕ No	
Well-Tagged PDF (WTPDF)	½ Partly	PDF/UA-2 output follows the same rules; WTPDF itself is not declared.
PDF/R raster (ISO 23504)	✕ No	
PAdES with a signature policy or commitment type	✕ No	

Internet standards (RFCs)

The RFCs that PDF signatures and links depend on.

Feature	PdfPecker	Notes
RFC 8118: the application/pdf media type	✓ Yes	Files follow it; embedded files carry their own MIME type.
RFC 5652: Cryptographic Message Syntax	✓ Yes	Own CMS writer and reader.
RFC 5035: signing-certificate-v2 attribute	✓ Yes	Required by PAdES.
RFC 5280: X.509 certificates and CRLs	✓ Yes	Chains are checked with the platform's certificate store.
RFC 6960: OCSP	½ Partly	Fetched for long-term validation; stored responses are not used when verifying.
RFC 8017 and RFC 4055: RSA PKCS#1 and RSA-PSS	✓ Yes	
RFC 5758: ECDSA in CMS	✓ Yes	
RFC 8032 and RFC 8419: EdDSA in CMS	✕ No	
RFC 3986: URIs in links	✓ Yes	

Metadata and attachments

Information about the document and files inside it.

Feature	PdfPecker	Notes
Document properties	✓ Yes	Title, author, subject, keywords, creator, producer, dates.
Custom properties	✓ Yes	
XMP metadata	✓ Yes	

Feature	PdfPecker	Notes
Document language	✓ Yes	
Reproducible output	✓ Yes	A fixed `CreationDate` or `SOURCE_DATE_EPOCH` gives byte-identical files.
Embedded files	✓ Yes	Add, list and extract, with MIME type, description and PDF/A-3 relationship.
File attachment annotations	✕ No	Files can be attached to the document, not pinned to a spot on a page.
Read document properties	✓ Yes	`PdfDocumentProperties`: properties, XMP, version, page sizes, encryption, PDF/A and PDF/UA claims.
Strip photo metadata	✓ Yes	`PdfOptions.StripImageMetadata` removes EXIF (GPS too), XMP and IPTC from embedded JPEGs.
Associated files (/AF)	✓ Yes	Document-level, with /AFRelationship, for PDF/A-3 and e-invoices.
Web Capture	✕ No	

Editing existing PDFs

Changing files you already have.

Feature	PdfPecker	Notes
Merge	✓ Yes	Also streamed, with bookmarks, page labels and forms kept.
Split	✓ Yes	
Extract, remove and reorder pages	✓ Yes	
Rotate pages	✓ Yes	
Stamp, overlay and underlay	✓ Yes	Another PDF or HTML on every page, or selected pages.
Watermarks	✓ Yes	Text, image and SVG.
Compress and optimise	✓ Yes	Lossless recompression, duplicate fonts and images stored once, optional image downsampling.
Add or remove a password	✓ Yes	
Edit existing text and content	✕ No	No find-and-replace or content editing.
Insert blank pages, crop or resize pages	✕ No	
N-up and imposition	✕ No	
Convert an existing PDF to PDF/A	✓ Yes	`PdfToPdfA`: PDF/A-2b, 2u, 3b or 3u from any PDF, with what it changed and what it could not fix.

Reading, rendering and conversion Changed in 1.24.0

Getting things out of existing PDFs.

Feature	PdfPecker	Notes
Render pages to PNG, JPEG, WebP and TIFF	✓ Yes	Multi-page TIFF in colour, greyscale or CCITT Group 4.
Read barcodes	✓ Yes	`PdfBarcodes.Find` reads QR, Data Matrix, Code 128 and EAN-13 codes on pages, and `Barcode.Read` in images, with EPC and Swiss QR-bill payments parsed.
All PDF image filters	✓ Yes	Flate, LZW, RunLength, ASCII85, ASCIIHex, DCT, JPEG 2000, JBIG2 and CCITT.
All font types	✓ Yes	Type 1, CFF, TrueType, Type 3 and CID fonts.
Fonts that are not embedded	½ Partly	Standard fonts are substituted; CJK fonts and predefined CJK CMaps are not.
All seven shading types	✓ Yes	Function, axial, radial and the four mesh types.
Colour spaces	✓ Yes	Device, Cal, Lab, ICC-based, Indexed, Separation, DeviceN and Pattern.
ICC colour management	½ Partly	ICC-based colours use their device equivalents; profiles are not applied.
Transparency groups, blend modes, soft masks	✓ Yes	
Annotation appearances	✓ Yes	Can be switched off.
Text extraction with positions	✓ Yes	Pages, lines and words with coordinates, font and size.
Reading order	✓ Yes	Text extraction, PDF to Markdown and structured PDF to HTML follow a tagged PDF's structure tree, and read columns, boxes, right-to-left columns and vertical CJK text in order from the layout.
OCR	✕ No	Text in scans and outlines cannot be extracted.
PDF to HTML	✓ Yes	Page images with a selectable, searchable text layer and working links.
Reflowing PDF to HTML	✓ Yes	`PdfHtmlMode.Structured`: headings, paragraphs, lists and tables in reading order, from a tagged PDF's tags or else from the layout, optionally with styles and pictures.
PDF to Markdown	✓ Yes	`PdfToMarkdown`: GitHub Flavored Markdown with headings, lists, links and tables.
Extract images	✓ Yes	`PdfImages`: original JPEGs or PNG, with where each is drawn and its EXIF.
PDF to Word, Excel or PowerPoint	✕ No	

Snapshot testing

PdfPecker.Testing checks in your unit tests that a PDF still comes out as it did. `PdfAssert.MatchesSnapshot` compares the PDF your code makes with a stored one, leaving out what changes on every run (creation dates, file IDs, the compressed bytes of streams), and when they differ it writes page images with the changed pixels marked. It also checks for text and page counts.

```
dotnet add package PdfPecker.Testing
```

It works with xUnit, NUnit, MSTest or any other test framework, because a failed check throws `PdfAssertException`, which every framework reports as a failed test. Its only dependency is PdfPecker, released with it under the same version, and the PDFs under test can come from PdfPecker, PdfPecker.Fluent or anything else.

```
using PdfPecker.Testing;

[Fact]
public void Invoice_looks_as_before()
{
    byte[] pdf = InvoiceService.Render(SampleData.Invoice);

    PdfAssert.MatchesSnapshot(pdf, "invoice");          // Snapshots/invoice.pdf next to this
test file
    PdfAssert.ContainsText(pdf, "Total due: 1,500.00");
    PdfAssert.ContainsText(pdf, "Payment terms", pageNumber: 2);
    PdfAssert.DoesNotContainText(pdf, "{{");             // no template placeholder left
    PdfAssert.HasPageCount(pdf, 2);
}
```

Snapshots

The first run has no snapshot, so it fails and writes the PDF as `Snapshots/invoice.received.pdf`. Look at it, and rename it to `invoice.pdf` to make it the snapshot. Commit the snapshots with the tests.

When a later PDF differs, the check fails with the first difference it found, such as the line of a content stream that changed, and writes next to the snapshot:

- `invoice.received.pdf`, the new PDF;
- for each page that looks different, `invoice.page-1.expected.png`, `invoice.page-1.actual.png` and `invoice.page-1.diff.png`: the new page faded to grey with the changed pixels in magenta.

To accept a change, replace the snapshot with the received file, or run the tests with `PDFPECKER_UPDATE_SNAPSHOTS=1`, which writes every missing or changed snapshot and passes. A check that passes deletes the files an earlier failure left. Add the failure files to `.gitignore`:

```
*.received.pdf
*.page-*.expected.png
*.page-*.actual.png
*.page-*.diff.png
```

What has to match

By default the files' content is compared: every object in the same order, with streams inflated, and without dates, file IDs and XMP document IDs. PdfPecker writes the same objects for the same input, so this catches every change, including those that don't show on the page, such as a link target, a bookmark, the title or the text that copy and paste and screen readers get. A PDF made again from the same input matches without a fixed `CreationDate`.

`PdfSnapshotComparison.Appearance` compares the rendered pages instead, for PDFs from other producers or when only the look matters:

```
var options = new PdfSnapshotOptions
{
    Comparison = PdfSnapshotComparison.Appearance,
    ColorTolerance = 2,         // a channel may differ by up to 2 of 255
    MaxDifferentPixels = 50,    // per page
};
PdfAssert.MatchesSnapshot(pdf, "invoice", options);
```

`PdfSnapshotOptions`	**Default**	**What it does**
`Directory`	`Snapshots` next to the test's source file	Where the snapshots and failure files are
`Comparison`	`Content`	`Content` or `Appearance`, as above
`Dpi`	72	Resolution of the compared and written page images
`ColorTolerance`	0	How far a pixel's channel may differ before it counts as different
`MaxDifferentPixels`	0	How many different pixels a page may have in `Appearance` comparisons
`UpdateSnapshots`	false	Write the PDF as the snapshot instead of failing
`WriteDiffImages`	true	Write the page images on a failure
`Password`	none	Password of encrypted PDFs

The default folder comes from the source file path the compiler records for the calling test. A build with `ContinuousIntegrationBuild` or `PathMap` records other paths, and the check then asks for `Directory`, such as `Path.Combine(AppContext.BaseDirectory, "Snapshots")` with the snapshots copied to the output.

Text and pages

`ContainsText`, `DoesNotContainText` and `HasPageCount` read the PDF with PdfPecker's text extraction. Line breaks and other runs of white space count as one space, so text that wraps differently still matches.

Limitations

- The content comparison reads objects in file order, so a PDF whose producer orders or numbers its objects differently on every run needs `Appearance`.
- Signed and encrypted PDFs change on every run by design; compare them with `Appearance`, with `Password` for encrypted ones.
- `Appearance` compares PdfPecker's rendering of the pages, not a particular viewer's.

Performance

PdfPecker lays out documents in your own process, without a browser, so a typical one-page document takes about a millisecond. For memory and CPU per document, see System requirements.

Compared with other converters

The same three documents converted to A4 PDF by PdfPecker and six .NET alternatives, on one 4-core cloud VM (Intel Xeon @ 2.1 GHz, 260 MB L3, 16 GB, Linux, .NET 10.0.13). Times are medians after a 3-second warm-up, averaged over two runs; "first PDF" includes starting the browser where there is one. Parallel runs one conversion per core at once on the 5-page document. Peak memory is the proportional set size (PSS) of the whole process tree, so a browser's helper processes are included once and shared pages are not double-counted; .NET processes run with a 16 MB GC gen0 budget, as .NET would otherwise size it from the CPU cache.

Converter	Engine	To deploy	First PDF	Invoice, 1 page	Feature tour, 5 pages	Report, 30 pages	Parallel	Peak memory
PdfPecker	own, managed	6.3 MB DLL	0.40 s	**0.92 ms**	**2.9 ms**	**33 ms**	**1,132 docs/s**	**92 MB**
HtmlRenderer.PdfSharp 1.6.1	own, managed (CSS 2)	5.0 MB of DLLs	0.54 s	4.9 ms	16 ms	360 ms	114 docs/s	255 MB
iText pdfHTML 6.3.4	own, managed	11 MB of DLLs	1.18 s	24 ms	132 ms	2.2 s	15 docs/s	707 MB
wkhtmltopdf 0.12.6	Qt WebKit, native	about 90 MB with Qt	0.33 s	282 ms	309 ms	600 ms	12 docs/s	165 MB
PuppeteerSharp 25.13	Chromium 141	about 600 MB browser	1.15 s	106 ms	142 ms	496 ms	15 docs/s	813 MB
Microsoft.Playwright 1.63	Chromium 141	browser + 135 MB driver	0.98 s	131 ms	167 ms	517 ms	12 docs/s	1,027 MB
Syncfusion HTML to PDF 35.1.40	Chrome (Blink), bundled	about 545 MB with Chrome	2.46 s	183 ms	236 ms	968 ms	3 docs/s	2,239 MB

How to read it:

- PdfPecker is 50 to 200 times faster than the Chromium-based converters on the 1 and 5-page documents and 15 to 30 times faster on the 30-page report, and uses an eighth or less of their memory. Per-document overhead (a browser tab, a process start) dominates for the others on small documents.

- Output is not identical. Chromium is the reference for modern CSS and runs JavaScript, which PdfPecker does not. HtmlRenderer supports only HTML 4 and CSS 2 (it drops the invoice's table header colours), and wkhtmltopdf uses a 2016 WebKit, shrinks content to fit by default (24 pages for the report instead of 29) and was archived in 2023.
- iText pdfHTML lays out the documents itself, like PdfPecker, but its time grows faster than the page count: 132 ms for 5 pages and 2.2 s for the 30-page report, which is one long table.
- Syncfusion ran as a trial without a licence key, which adds a line of text to each page. Its `AdditionalDelay` (2 seconds of waiting for scripts by default) was set to 0, as the other Chromium converters don't wait either. With its default settings it scales content down, so the report took 13 pages instead of 29.
- PDF sizes for the 1 / 5 / 30-page documents: PdfPecker 12 / 28 / 126 KB, Chromium 72 / 139 / 171 KB, wkhtmltopdf 30 / 59 / 191 KB, HtmlRenderer 35 / 65 / 273 KB, iText 26 / 45 / 104 KB, Syncfusion 42 / 89 / 132 KB.
- Each converter was used the way its documentation suggests for many documents: PdfPecker and HtmlRenderer in process, wkhtmltopdf as one process per document, and PuppeteerSharp and Playwright with one shared browser and a new tab per document (Playwright in one shared browser context, tagged PDF output off for both). iText reuses one font provider per thread, as its knowledge base advises. Syncfusion keeps one converter per concurrent conversion with `ReuseBrowserProcess` on, as its support forum advises, so the parallel run starts four browsers.

IronPDF was not measured here. One user compared PdfPecker with **IronPDF 2026.7.2** (Chrome-based) on the 26 templates of their production app, on a 22-core development machine after warm-up: IronPDF took 13.6 s for all 26 documents one at a time and PdfPecker 17 ms (about 800 times faster), and with 22 in parallel IronPDF made 9.2 documents/s and PdfPecker 1,143 (about 125 times). 24 of the 26 documents came out identical; the other two differ by less than 0.25 mm.

To reproduce the table, clone the PdfPecker.Benchmarks repository (needs wkhtmltopdf on `PATH` and a Chromium at `CHROMIUM_PATH`) and run:

```
dotnet run --project src/PdfPecker.Comparison -c Release
```

It measures PdfPecker from nuget.org; `-p:PdfPeckerVersion=` picks another release. The comparison lives in its own repository, so its third-party packages never reach the library or its tests. Your numbers will differ with hardware and documents.

Conversion speed

Measured with `bench/PdfPecker.Benchmarks` (in the PdfPecker repository) on a 4-core cloud VM (.NET 10, Release, after JIT warm-up):

Document	Pages	Median	Throughput
Invoice with a PNG logo and a styled table	1	0.55 ms	about 1,800 docs/s on one core
Feature tour (text, lists, tables, JPEG and PNG)	5	2.5 ms	about 1,700 docs/s on 4 cores
Dense report with 1,000 table rows	30	19.5 ms	about 1,500 pages/s on one core

Document	Pages	Median	Throughput
Markdown guide (`guide.md`, headings, lists, code, a table) to PDF	2	0.9 ms	about 1,000 docs/s on one core

Parsing Markdown to HTML runs at roughly 20 to 45 MB/s, so it is a small part of a Markdown to PDF conversion.

The first conversion in a process also pays for JIT compilation; later ones run at the speeds above. About 80% of the first PDF's time is compiling PdfPecker's code, so an app that starts often (a command-line tool, a serverless function, a container that scales to zero) can publish with ReadyToRun to have it compiled ahead of time:

```
dotnet publish -c Release -r linux-x64 -p:PublishReadyToRun=true
```

In a measured app this took the first PDF from about 345 ms to about 130 ms, while the published PdfPecker.dll grows from about 5 MB to about 10 MB, since it holds native code for one platform next to the IL. Later conversions run at the same speed either way. Native AOT, which PdfPecker supports too, removes the JIT entirely.

Rendering PDF pages to images, same machine, one core per page (`PdfRenderer`, first render of each page of a freshly opened document, so fonts and images are decoded too):

Document	Render at 96 DPI	Render at 150 DPI	PNG encoding at 150 DPI
Invoice (1 page)	13 ms	19 ms	24 ms
Feature tour (5 pages)	7 ms per page	12 ms per page	26 ms per page
Large report (30 pages)	5 ms per page	10 ms per page	29 ms per page

Rendering the same page again (fonts and glyphs cached) takes about 5 ms at 150 DPI. Multi-page conversions render pages in parallel. Encoding one 150 DPI A4 page (1240 x 1754) takes about 15 to 60 ms as PNG, 20 to 70 ms as JPEG and 40 to 85 ms as WebP at the default quality, depending on content.

How it works

The source paths below are in the PdfPecker repository under `src/PdfPecker/`.

Markdown is first turned into HTML: `Markdown/BlockParser` finds the block structure line by line, following the CommonMark reference algorithm, `Markdown/InlineParser` handles emphasis (delimiter stack) and links (bracket stack), and `Markdown/HtmlRenderer` writes the HTML, which then takes the same path as any other document:

1. `Html/HtmlParser` builds a DOM in a single pass, applying the browser rules for implied end tags.
2. `Css/StyleResolver` cascades a user-agent stylesheet, presentational attributes, document stylesheets and inline styles. Rules are indexed by id, class and tag, and elements with identical inputs share one computed style. For a tagged PDF, `Pdf/StructureTree` then maps the elements to structure types and gives every text node, image and list marker a slot in that tree.
3. `Layout/LayoutEngine` lays out blocks, inline text (line breaking, justification), lists and tables (automatic column widths), paginating as it goes, and produces a flat list of draw operations. Inline content with right-to-left text gets its embedding levels from `Text/Bidi` (UAX #9), and each line is put in visual order after it is broken. `Fonts/ComplexShaper` shapes Arabic and Hebrew runs with the font's GSUB and GPOS tables (`Fonts/OpenTypeLayout`).
4. `Pdf/PdfDocumentWriter` writes PDF 1.4: content streams, fonts, image XObjects, link annotations, the bookmark outline and the cross-reference table. A tagged PDF's content is wrapped in marked-content sequences for its slots (or as artifacts), and the structure tree and its parent tree are written last. `Fonts/FontSubsetter` and `Fonts/CffSubsetter` cut embedded fonts down to the glyphs used (with their composite components and the CFF subroutines they call).

The renderer (`Rendering/`) works the other way round: `Parsing/PdfFile` reads the object graph, `PageRenderer` interprets content streams, `Fonts/` turns glyphs into outlines (cached as anti-aliased bitmaps), `Images/` decodes images, and `Graphics/Rasterizer` fills paths with exact area coverage before `PngEncoder`, `JpegEncoder` or `WebPEncoder` writes the result.

The third-party fonts, font data and code built into the library are listed, with their licenses, in THIRD-PARTY-NOTICES.md.