

PdfPecker manual

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

For PdfPecker 1.16.0 · pdfpecker.net/docs

Contents

Getting started 6
- Install 6
- Usage 6
- System requirements 7
- Where to go next 7

HTML to PDF in C# 8
- Install the NuGet package 8
- Convert an HTML string to PDF 8
- Convert an HTML file to PDF 8
- Generate an invoice from a template 8
- Return a PDF from ASP.NET Core 9
- Page size, margins, headers and footers 10
- Fonts 10
- Images 10
- Common questions 10

Options 12
- Files, network and untrusted HTML 12
- Page size from CSS 12
- Document properties 13
- Reproducible output 13
- Errors, cancellation and output files 13
- More options 13

HTML and CSS 14
- HTML 14
- CSS 14
- CSS properties . . . . . 15 Changed in 1.16.0
- Units and colours 16
- Subgrid 17
- Multi-column layout 17
- Percentage heights 17
- Container queries 17
- Pagination . . . . . 18 Changed in 1.16.0
- Links and bookmarks 18
- SVG . . . . . 18 Changed in 1.16.0
- Images 19

Paged media 20
- Page rules and margin boxes 20
- Footnotes 20
- Running header and footer from HTML 21

Markdown 22
- What is supported 22
- Footnotes, definition lists and front matter 22

Images to PDF 24
Page size 24
Output settings 24
Errors 24

Fonts 25
@font-face 25
Variable fonts 25
OpenType features 26
Standard fonts 26
Missing fonts and characters 26
How fonts are embedded 27

Right-to-left, CJK and hyphenation 28
Right-to-left text: Arabic and Hebrew 28
Indic, Thai and Lao scripts 28
Chinese, Japanese and Korean line breaking 29
Hyphenation 29

PDF/A, passwords and watermarks 31
Watermarks and passwords 31
PDF/A 31
Passwords on existing PDFs 32

Accessibility 33
Tagged PDF 33
PDF/UA-1 33
PDF/UA-2 33
What your HTML needs 34

Fillable forms 35
Field names and behaviour 35
Filling and flattening existing PDFs 36

Digital signatures 37
Signing into an existing field 37
Formats and compatibility 37
External signers: HSMs, smart cards and key vaults 38
Verifying signatures 38
Long-term validation 39

E-invoices 41
Invoice XML from invoice data 41
Ready-made invoice template 42
Factur-X and ZUGFeRD 43
Reading the invoice from a received PDF 43
PDF attachments in UBL e-invoices (Peppol, OIOUBL) 44

Barcodes 45
QR codes 45
EPC payment QR codes 45
Code 128 and EAN-13 46

Options . . . 46

Attachments . . . 47
How files are stored . . . 47
Adding to an existing PDF . . . 47

Merging and splitting . . . 48
Merging . . . 48
Merging PDFs as they arrive . . . 48
Extracting, reordering and splitting pages . . . 49

Stamping . . . 51
Placement . . . 51
What is kept . . . 51

Compacting . . . 53
What it does . . . 53
Options . . . 53
Signed and encrypted PDFs . . . 54

Text extraction . . . 55
Pages, lines and words . . . 55
What is read . . . 55

PDF to image . . . 57
HTML to images . . . 57
Output formats . . . 57
TIFF . . . 58
Options . . . 58
What the renderer supports . . . 59

From wkhtmltopdf . . . 60
Replace the wkhtmltopdf call . . . 60
Options side by side . . . 61
What changes in the output . . . 61
Speed and size . . . 62
Next steps . . . 62

From HtmlRenderer.PdfSharp . . . 63
Replace the HtmlRenderer call . . . 63
Options side by side . . . 64
What changes in the output . . . 64
Speed and memory . . . 64
When HtmlRenderer is still a fine choice . . . 65

From IronPDF . . . 66
Replace the IronPDF call . . . 66
Headers and footers in HTML . . . 66
Matching IronPDF's output exactly . . . 67
Options side by side . . . 67
Speed . . . 67
When IronPDF is the better choice . . . 68

From PuppeteerSharp or Playwright **69**
Replace the browser code 69
Headers and footers 70
Options side by side 70
What changes 71
Speed and memory 71
When to keep Chrome 71

Changelog **72**
1.16.0 72
1.15.0 72
1.14.2 73
1.14.1 73
1.14.0 73
1.13.0 74
1.12.0 74
1.11.0 75
1.10.0 75
1.9.0 75
1.8.0 76
1.7.0 76
1.6.0 77
1.5.1 77
1.4.0 77
1.3.0 77
1.2.0 77
1.1.1 77
1.1.0 78
1.0.0 78

Limitations **79**
By design 79
Text and fonts . . . . . 79 Changed in 1.16.0
CSS 80
Backgrounds and transparency . . . . . 80 Changed in 1.16.0
Layout 81
Images and SVG 81
Reading and changing existing PDFs 81
Markdown and the PDF renderer 81

Performance **83**
Compared with other converters 83
Conversion speed 84

How it works **86**

Getting started

> **New and changed in PdfPecker 1.16.0** (since 1.15.0):
>
> - HTML and CSS: CSS properties Changed
> - HTML and CSS: Pagination Changed
> - HTML and CSS: SVG Changed
> - Limitations: Text and fonts Changed
> - Limitations: Backgrounds and transparency 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 PdfPecker 1.13.

```
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.

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.
- 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)
    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)
};
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.

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`.

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 1.16.0:** CSS properties, Pagination, SVG.

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.

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` and `value`), `dl`, `table` (with `thead`, `tbody`, `tfoot`, `caption`, `colspan`, `rowspan`, `col width`), `img`, inline `<svg>`, and legacy attributes such as `align`, `valign`, `bgcolor`, `width`, `height`, `border`, `cellpadding`, `cellspacing` and `<font>`. 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.

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), inline `style` attributes, `@media` (media queries level 4: `print`, `width`/`height` and `orientation` of the page area, ranges such as `(400px <= width <= 800px)`, `and`, `or`, `not`), `@supports` (property tests, `selector()`, `and`/`or`/`not`), `@layer` cascade layers, CSS nesting (`&`), `@property` (initial values and inheritance of registered custom properties), `@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()`, form states (`:checked`, `:disabled`, `:enabled`, `:required`, `:optional`, `:read-only`, `:placeholder-shown` and the like), `::before`, `::after`, `::marker`, `::first-letter` (drop caps, floated or not) and `::first-line`, with specificity and `!important` handled as in browsers. As when Chrome prints, interaction states such as `:hover` and `:focus` never match.

CSS properties Changed in 1.16.0

Area	Properties
Box model	`display` (block, inline, inline-block, flex, inline-flex, grid, inline-grid, list-item, table parts, none), `margin`, `padding` (also the logical `margin-inline` / `-block` and `padding-inline` / `-block` properties and their `-start` / `-end` longhands), `border` (solid, dashed, dotted, double, groove, ridge, inset, outset), `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), `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`), `outline` and `outline-offset` (all border styles; follows rounded corners), `width`, `height`, `min-` / `max-width`, `min-` / `max-height` (percentages as in browsers; see Percentage heights), `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), `clip-path` (`inset()`, `circle()`, `ellipse()`, `polygon()`, `path()`, reference boxes, and `url(#id)` of a `clipPath` in inline SVG)
Text	`color`, `font`, `font-family`, `font-size`, `font-weight`, `font-style`, `font-stretch`, `font-variation-settings`, `font-optical-sizing` (see Variable fonts), `line-height`, `text-align` (incl. justify, `start` and `end`), `direction`, `unicode-bidi`, `text-decoration`, `text-transform`, `text-indent`, `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`, `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`), `text-decoration-color`, `text-decoration-thickness`, `text-underline-offset`, `text-align-last`, `tab-size`, `font-variant: small-caps` (synthesized from capitals when the font has none), `font-variant-numeric` and `font-feature-settings` (see OpenType features), `line-clamp` and `-webkit-line-clamp` (with an ellipsis), `hyphens: manual`
Backgrounds	`background-color`, `background-image` and `background` with `url()` images (PNG, JPEG, GIF, WebP, BMP, TIFF, and SVG files drawn as vectors), `linear-gradient()`, `radial-gradient()`, `conic-gradient()` (pie charts and colour wheels, drawn as smooth PDF shadings) and their `repeating-` forms (several layers allowed); `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`. Inline elements such as `<span>`, `<mark>` and `<code>` get the same backgrounds and borders 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`
Grid	`grid-auto-flow: dense` packing, `grid-template-columns` / `-rows` (lengths, `%`, `fr`, `auto`, `min-content`, `max-content`, `minmax()`, `fit-content()`, `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-*`
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)
Variables	custom properties (`--name: value`) and `var(--name, fallback)`, anywhere including shorthands

Area	Properties
Math functions	`calc()`, `min()`, `max()`, `clamp()`, `round()`, `mod()`, `rem()`, `abs()`, `sign()`, `sin()`, `cos()`, `tan()`, `asin()`, `acos()`, `atan()`, `atan2()`, `pow()`, `sqrt()`, `hypot()`, `log()` and `exp()` 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`; `position` (`relative`, `absolute`, `fixed`; `sticky` is drawn as `relative`, as in Chrome's print output) with `top`, `right`, `bottom` and `left`; `z-index`
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`
Transforms	`transform` (`translate()`, `translateX()`/`Y()`, `scale()`, `scaleX()`/`Y()`, `rotate()`, `skew()`, `skewX()`/`Y()`, `matrix()`), `transform-origin`, and the individual `translate`, `rotate` and `scale` properties
Generated content	`::before`, `::after` and `::marker` with `content` (strings, `attr()`, `counter()`, `counters()`, `open-quote`/`close-quote`, 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 and SVG files, falling back to the type when the image does not load), `list-style`, `::marker` (content, colour, font)
Tables	`border-collapse` (each shared border drawn once, the wider one winning), `border-spacing`, cell `vertical-align`, automatic column widths (a column width that mixes a percentage with a length, such as `calc(50% - 10px)`, counts as `auto`, as in Chrome), `table-layout: fixed`, a table `height` shared out over the rows
Paging	`page-break-before`/`after`/`inside`, `break-before`/`after`/`inside` (`always`, `page`, `avoid`), `orphans`, `widows`, `@page` size, margins and background, `@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)

Units and colours

Units: `px`, `pt`, `em`, `rem`, `%`, `mm`, `cm`, `in`, `pc`, `ex`, `ch`, `vw`, `vh`, `vmin`, `vmax` (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), `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.

Pagination Changed in 1.16.0

Lines, images and table rows 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. Flex lines move to the next page as a whole when they don't fit. 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. As in Chrome, a margin that runs past the end of a page is dropped rather than carried to the next one (except after a forced break), and a box's top and bottom borders and padding stay on the same page as its first and last content.

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.

SVG Changed in 1.16.0

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`, 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. Strokes can be dashed with `stroke-`

`dasharray` and `stroke-dashoffset`, scaled by `pathLength`. `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.

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. Images wider than the page are scaled down to fit.

Paged media

Page rules and margin boxes

`@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. 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.
- **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. The document is laid out again with the numbers it finds, so they are always right.

```
@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, `target-counter()` or `leader()`.

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.

```
<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>
```

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.

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

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. 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

`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. Features that depend on the surrounding characters, such as `diagonal-fractions` and `ordinal` in most fonts, are not. Text copied from the PDF keeps the original characters.

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).

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, ...
```

Likewise, a character that no font has 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

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. 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.

Right-to-left text, CJK and 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 ١٥/٢٦٠٢</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 line is stretched only at its spaces, so a Chinese or Japanese line without spaces is not justified (see Limitations).

Hyphenation

`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);
```

PDF/A, passwords and watermarks

Watermarks and passwords

```
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 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.

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

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.

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.

Verifying signatures

`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.

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, so set `UrlFilter` to keep requests away from internal hosts: it is asked about every URL, redirect targets included, and returning false skips it.

```
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.

Option	Default	What it does
`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

`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.

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.

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. 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`.

Text extraction

`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`.

Columns side by side are read across, line by line, not one column after the other, and table cells on one row make one line. Text drawn only as outlines or images has no characters to extract; a scanned page needs OCR first. See Limitations.

PDF to image

`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: higher spends more time searching for a smaller file and never changes a pixel. 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` (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.

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.

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),
});
```

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
`--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 310 ms with wkhtmltopdf and 0.86 ms with PdfPecker, and a 30-page report 780 ms and 45 ms. Deployment drops from about 90 MB of native files to one 4 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 6.3 ms with HtmlRenderer.PdfSharp and 0.86 ms with PdfPecker, a 30-page report 639 ms and 45 ms, and peak memory was 236 MB and 90 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 153 ms with PuppeteerSharp, 194 ms with Playwright and 0.86 ms with PdfPecker. Peak memory was 829 MB and 967 MB for the browser-based converters and 90 MB for PdfPecker, and in parallel PdfPecker made 818 documents per second against 8 to 11. 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.

1.16.0

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 1.16.0:** Text and fonts, Backgrounds and transparency.

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.

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>`, `<audio>`, `<object>` and `<embed>` are not drawn, nor are file, range, colour and image inputs. 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 1.16.0

- Without registered fonts or `EmbedStandardFonts`, text uses the standard PDF fonts, so characters are limited to the Windows-1252 set (Western European languages); other characters are replaced with `?`. The bundled Liberation fonts add Central European, Greek and Cyrillic, but not CJK or other scripts: 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. There is no vertical text (`writing-mode`).
- Chinese, Japanese and Korean text breaks into lines, but `text-align: justify` stretches only spaces, so CJK lines without spaces are not justified (`text-justify` is not supported), and punctuation is not compressed at the start or end of a line.
- 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 and other features that depend on neighbouring characters are not applied, and `small-caps` is synthesized from scaled capitals.
- 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

- 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.
- 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 or a rowspanning cell, 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` scrolls with the box, since paper has no scrolling viewport.

Backgrounds and transparency Changed in 1.16.0

- Inset box shadows of inline elements (`<span>`) are not drawn. `box-decoration-break` applies to inline elements, not to blocks split across pages or columns.
- 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

- 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.

Images and SVG

- 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).

Reading and changing existing PDFs

- Text extraction reads columns that sit side by side across, line by line, rather than one column after the other, and it does not use the structure tree of a tagged PDF to find the reading order. 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`.
- 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 counts the updates made after a signature but does not analyse them against its permissions (no DocMDP or FieldMDP change analysis), so it 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).

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 @ 2.8 GHz, 33 MB L3, 16 GB, Linux, .NET 10.0.12). 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	4.6 MB DLL	0.40 s	**0.86 ms**	**3.5 ms**	**45 ms**	**818 docs/s**	**90 MB**
HtmlRenderer.PdfSharp 1.6.1	own, managed (CSS 2)	2.4 MB of DLLs	0.58 s	6.3 ms	21 ms	639 ms	72 docs/s	236 MB
iText pdfHTML 6.3.4	own, managed	11 MB of DLLs	1.44 s	34 ms	209 ms	3.4 s	9 docs/s	624 MB
wkhtmltopdf 0.12.6	Qt WebKit, native	about 90 MB with Qt	0.36 s	310 ms	356 ms	780 ms	11 docs/s	164 MB
PuppeteerSharp 25.13	Chromium 141	about 600 MB browser	1.43 s	153 ms	217 ms	792 ms	11 docs/s	829 MB
Microsoft.Playwright 1.63	Chromium 141	browser + 135 MB driver	1.36 s	194 ms	247 ms	726 ms	8 docs/s	967 MB
Syncfusion HTML to PDF 35.1.40	Chrome (Blink), bundled	about 545 MB with Chrome	2.83 s	264 ms	320 ms	1.1 s	2 docs/s	2,193 MB

How to read it:

- PdfPecker is 60 to 220 times faster than the Chromium-based converters on the 1 and 5-page documents and about 17 times faster on the 30-page report, and uses a tenth 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: 209 ms for 5 pages and 3.4 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 / 27 / 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 from a clone of the PdfPecker repository (needs wkhtmltopdf on `PATH` and a Chromium at `CHROMIUM_PATH`):

```
dotnet run --project bench/PdfPecker.Comparison -c Release
```

The comparison project is not part of `PdfPecker.sln`, 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
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 (about 150 ms); later ones run at the speeds above.

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.