



PdfPecker manual

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

For PdfPecker 1.7.0 · pdfpecker.net/docs

Contents

Getting started	4
Install	4
Usage	4
System requirements	4
Where to go next	5
Options	6
Files, network and untrusted HTML	6
Page size from CSS	6
Document properties	7
More options	7
HTML and CSS	8
HTML	8
CSS	8
CSS properties	8
Units and colours	10
Pagination	10
Links and bookmarks	10
SVG	10
Images	11
Paged media	12
Page rules and margin boxes	12
Running header and footer from HTML	12
Markdown	13
What is supported	13
Fonts	14
@font-face	14
Standard fonts	14
Missing fonts and characters	14
How fonts are embedded	15
Right-to-left and hyphenation	16
Right-to-left text: Arabic and Hebrew	16
Hyphenation	16
PDF/A, passwords and watermarks	18
Watermarks and passwords	18
PDF/A	18
Passwords on existing PDFs	19
Accessibility	20
Tagged PDF	20
PDF/UA-1	20

PDF/UA-2	20
What your HTML needs	21
Fillable forms	22
Field names and behaviour	22
Digital signatures	24
Signing into an existing field	24
Formats and compatibility	24
E-invoices	25
Factur-X and ZUGFeRD	25
PDF attachments in UBL e-invoices (Peppol, OIOUBL)	25
Attachments	26
How files are stored	26
Adding to an existing PDF	26
Merging and splitting	27
Merging	27
Merging PDFs as they arrive	27
Extracting, reordering and splitting pages	28
Stamping	30
Placement	30
What is kept	30
PDF to image	32
Output formats	32
Options	33
What the renderer supports	33
Limitations	34
By design	34
Text and fonts	34
CSS	34
Backgrounds and transparency	35
Layout	35
Images and SVG	35
Markdown and the PDF renderer	36
Performance	37
Compared with other converters	37
Conversion speed	38
How it works	40

Getting started

PdfPecker turns HTML and Markdown into PDF, and PDF pages into PNG, JPEG or WebP 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 1.7.

```
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);
```

All methods are static and thread-safe, so you can convert many documents in parallel. 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).

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/stylesheets 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. PdfPecker never makes network requests on its own: `http(s)` images and stylesheets are only loaded through `ResourceResolver`.

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.

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

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">`, 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) 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 selector lists; `: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` and `::marker`, with specificity and `!important` handled as in browsers.

CSS properties

Area	Properties
Box model	<code>display</code> (<code>block</code> , <code>inline</code> , <code>inline-block</code> , <code>flex</code> , <code>inline-flex</code> , <code>grid</code> , <code>inline-grid</code> , <code>list-item</code> , <code>table parts</code> , <code>none</code>), <code>margin</code> , <code>padding</code> (also the logical <code>margin-inline/-block</code> and <code>padding-inline/-block</code> properties and their <code>-start/-end</code> longhands), <code>border</code> (<code>solid</code> , <code>dashed</code> , <code>dotted</code> , <code>double</code>), <code>border-radius</code> (incl. elliptical / radii and percentages; rounds backgrounds, borders, shadows and images), <code>box-shadow</code> (outer shadows; a blur is drawn as fading bands), <code>overflow</code> (<code>hidden</code> , <code>clip</code> , <code>auto</code> and <code>scroll</code> clip to the padding box, following its rounded corners, also in table cells; <code>overflow-x/overflow-y</code>), <code>outline</code> and <code>outline-offset</code> (all border styles; follows rounded corners), <code>width</code> , <code>height</code> , <code>min-/max-width</code> , <code>min-height</code> , <code>box-sizing</code> , <code>aspect-ratio</code> (incl. <code>auto <ratio></code>), <code>object-fit</code> and <code>object-position</code> on images

Area	Properties
Text	<code>color</code> , <code>font</code> , <code>font-family</code> , <code>font-size</code> , <code>font-weight</code> , <code>font-style</code> , <code>line-height</code> , <code>text-align</code> (incl. <code>justify</code> , <code>start</code> and <code>end</code>), <code>direction</code> , <code>unicode-bidi</code> , <code>text-decoration</code> , <code>text-transform</code> , <code>text-indent</code> , <code>letter-spacing</code> , <code>word-spacing</code> , <code>white-space</code> , <code>overflow-wrap/word-wrap</code> , <code>word-break</code> (<code>break-all</code>), <code>vertical-align</code> , <code>text-shadow</code> (several shadows; a blur is drawn as faint copies of the text), <code>text-overflow: ellipsis</code> (on a box that clips its overflow), <code>text-decoration-style</code> (<code>solid</code> , <code>double</code> , <code>dotted</code> , <code>dashed</code> , <code>wavy</code>), <code>text-decoration-color</code> , <code>text-decoration-thickness</code> , <code>text-underline-offset</code> , <code>text-align-last</code> , <code>tab-size</code> , <code>font-variant: small-caps</code> (synthesized from capitals when the font has none), <code>line-clamp</code> and <code>-webkit-line-clamp</code> (with an ellipsis), <code>hyphens: manual</code>
Backgrounds	<code>background-color</code> , <code>background-image</code> and <code>background</code> with <code>url()</code> images (PNG, JPEG, GIF, WebP, and SVG files drawn as vectors), <code>linear-gradient()</code> , <code>radial-gradient()</code> , <code>conic-gradient()</code> (pie charts and colour wheels, drawn as smooth PDF shadings) and their <code>repeating-</code> forms (several layers allowed); <code>background-repeat</code> (incl. <code>space</code> , <code>round</code>), <code>background-position</code> (incl. edge offsets like <code>right 10px bottom 5px</code>), <code>background-size</code> (<code>cover</code> , <code>contain</code> , <code>lengths</code>), <code>background-origin</code> , <code>background-clip</code> . Repeated images are written once as PDF tiling patterns. <code>url()</code> in a linked stylesheet is relative to the stylesheet and follows the same file access rules as <code></code> . As in Chrome's print output, the background of <code><html></code> , or else of <code><body></code> , fills every page area (images and gradients too, positioned against the root box and continuing from page to page); <code>@page { background: <colour> }</code> also colours the page margins. <code>opacity</code> on SVG
Flexbox	<code>flex-direction</code> , <code>flex-wrap</code> , <code>flex-flow</code> , <code>flex</code> , <code>flex-grow/-shrink/-basis</code> , <code>order</code> , <code>justify-content</code> , <code>align-items</code> , <code>align-self</code> , <code>align-content</code> , <code>gap</code> , <code>margin: auto</code>
Grid	<code>grid-auto-flow: dense</code> packing, <code>grid-template-columns/-rows</code> (<code>lengths</code> , <code>%</code> , <code>fr</code> , <code>auto</code> , <code>min-content</code> , <code>max-content</code> , <code>minmax()</code> , <code>fit-content()</code> , <code>repeat()</code> incl. <code>auto-fill/auto-fit</code>), <code>grid-auto-rows/-columns</code> , <code>grid-template-areas</code> , <code>grid-template</code> , named lines (<code>[name]</code>), <code>grid-column/-row</code> (line numbers, names and <code>span</code>), <code>grid-area</code> (incl. area names), <code>gap</code> , <code>justify-items/-self</code> , <code>align-items/-self</code> , <code>justify-content</code> , <code>align-content</code> , <code>place-*</code>
Columns	<code>column-count</code> , <code>column-width</code> , <code>columns</code> , <code>column-gap</code> , <code>column-rule</code> (all border styles), <code>column-span: all</code>
Variables	custom properties (<code>--name: value</code>) and <code>var(--name, fallback)</code> , anywhere including shorthands
Math functions	<code>calc()</code> , <code>min()</code> , <code>max()</code> , <code>clamp()</code> , <code>round()</code> , <code>mod()</code> , <code>rem()</code> , <code>abs()</code> , <code>sign()</code> , <code>sin()</code> , <code>cos()</code> , <code>tan()</code> , <code>asin()</code> , <code>acos()</code> , <code>atan()</code> , <code>atan2()</code> , <code>pow()</code> , <code>sqrt()</code> , <code>hypot()</code> , <code>log()</code> and <code>exp()</code> for lengths, numbers and angles, nested in any combination and mixing units (<code>calc(100% - 2em)</code> , <code>min(100%, 600px)</code> , <code>clamp(1rem, 2.5vw, 2rem)</code>), with <code>var()</code> inside them. Percentages resolve against the property's own reference (the containing block's width for <code>width</code> and margins, the parent's font size for <code>font-size</code>), so they work in widths, heights, margins, padding, flex bases, grid tracks, gaps, positions, translations, font sizes and borders
Positioning	<code>float</code> and <code>clear</code> ; <code>position</code> (<code>relative</code> , <code>absolute</code> , <code>fixed</code> ; <code>sticky</code> is drawn as <code>relative</code>) with <code>top</code> , <code>right</code> , <code>bottom</code> and <code>left</code> ; <code>z-index</code>
Transparency	<code>opacity</code> (a block is painted as one PDF transparency group; an inline <code></code> fades its text and background), translucent colours everywhere (<code>rgba()</code> , <code>hsla()</code> , <code>#rrggbbaa</code> , <code>transparent</code>), gradients that fade to <code>transparent</code> (as a soft mask). PDF/A-2 and 3 allow all of this
Transforms	<code>transform</code> (<code>translate()</code> , <code>translateX()/Y()</code> , <code>scale()</code> , <code>scaleX()/Y()</code> , <code>rotate()</code> , <code>skew()</code> , <code>skewX()/Y()</code> , <code>matrix()</code>), <code>transform-origin</code> , and the individual <code>translate</code> , <code>rotate</code> and <code>scale</code> properties
Generated content	<code>::before</code> , <code>::after</code> and <code>::marker</code> with <code>content</code> (strings, <code>attr()</code> , <code>counter()</code> , <code>counters()</code> , <code>open-quote/close-quote</code> , and for paged media <code>target-counter()</code> , <code>target-text()</code> , <code>leader()</code> , <code>string()</code> and <code>content()</code>), <code>counter-reset</code> , <code>counter-increment</code> , <code>counter-set</code> , <code>string-set</code>

Area	Properties
SVG	<code>fill</code> , <code>stroke</code> , <code>stroke-width</code> , <code>fill-opacity</code> , <code>stroke-opacity</code> , <code>fill-rule</code> , <code>stroke-linecap</code> , <code>stroke-linejoin</code> , <code>stroke-miterlimit</code> , <code>clip-path</code> , <code>clip-rule</code> , <code>mask</code> , <code>mask-type</code> , <code>filter</code> , <code>marker</code> , <code>marker-start</code> / <code>-mid</code> / <code>-end</code> , <code>flood-color</code> , <code>flood-opacity</code> , <code>color-interpolation-filters</code> , <code>text-anchor</code> , <code>dominant-baseline</code> (as CSS or presentation attributes)
Lists	<code>list-style-type</code> (<code>disc</code> , <code>circle</code> , <code>square</code> , <code>decimal</code> , <code>decimal-leading-zero</code> , <code>alpha</code> , <code>roman</code> , <code>none</code> , or a string such as <code>'—'</code>), <code>list-style</code> , <code>::marker</code> (content, colour, font)
Tables	<code>border-collapse</code> (each shared border drawn once, the wider one winning), <code>border-spacing</code> , <code>cell vertical-align</code> , automatic column widths, <code>table-layout: fixed</code> , a table <code>height</code> shared out over the rows
Paging	<code>page-break-before</code> / <code>after</code> / <code>inside</code> , <code>break-before</code> / <code>after</code> / <code>inside</code> (<code>always</code> , <code>page</code> , <code>avoid</code>), <code>orphans</code> , <code>widows</code> , <code>@page</code> size, margins and background, <code>@page :first</code> / <code>:left</code> / <code>:right</code> , the sixteen page-margin boxes (<code>@top-center</code> and the like), named pages (<code>page</code>), <code>string-set</code> (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). 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) and `currentColor`.

Pagination

Lines, images and table rows are never cut in half at a page boundary; table header rows repeat on every page a table spans; backgrounds and borders continue across pages. Flex lines and grid rows move to the next page as a whole when they don't fit. 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.

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

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), `g`, `use`, nested `svg`, `viewBox`,

`preserveAspectRatio` and `transform`, drawn as vector paths. `linearGradient` and `radialGradient` (both units, `gradientTransform`, `spreadMethod`, `href` inheritance) 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: `feGaussianBlur`, `feOffset`, `feDropShadow`, `feFlood`, `feMerge`, `feComposite` (all operators), `feBlend` (separable modes), `feColorMatrix`, `feComponentTransfer` and `feMorphology`, in linearRGB or sRGB. 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) and WebP (lossy, lossless and with alpha), from files, `data:` URIs or your `ResourceResolver`. Animated GIF and WebP files show their first frame. 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`. 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()`.

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 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. 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.

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,
Mockba</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.

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 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>٢٠٢٦/١٥ فورة رقم</h1>
        <p>1,250.00 المبلغ الإجمالي: 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.

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. Patterns for English (US patterns for every `en` tag), Danish, German, Dutch and Norwegian (`nb`, `nn`, `no`) are built in and loaded on first use; 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>
    {
        ["sv"] = new HyphenationPatterns(File.ReadAllText("hyph-sv.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="sv">Avstavning med egna mönster.</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 or WebP bytes), 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 <code>EInvoice</code> (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.

```
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).

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.

```
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.

E-invoices

Factur-X and ZUGFeRD

Pass the invoice's CII XML (from your invoicing system) 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 its content or build it from invoice data: make sure the visible pages show the same invoice. 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](#)).

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 build or validate the rest of the invoice.

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 IEnumerable<byte[]>:
int pages = await PdfMerger.MergeAsync(DownloadInOrder(paths), response.Body,
cancellationTokens: 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

`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`.

PDF to image

`PdfToImage` and `PdfRenderer` render PDF pages to PNG, JPEG or WebP (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 and WebP 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());
```

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`.

Options

`ImageOptions` sets `Dpi` (default 150) or a target `Width/Height` in pixels, `BackgroundColor` or `Transparent`, `RenderAnnotations`, `Password`, `MaxDegreeOfParallelism`, `PngCompression`, `Format` and `Quality` (an `ArgumentOutOfRangeException` outside 1 to 100). 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](#).

Limitations

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

- 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). Devanagari and other Indic scripts, Thai, Khmer, Myanmar and Syriac are not shaped, so their vowel signs and conjuncts come out wrong, and Thai text breaks only at spaces. There is no vertical text (`writing-mode`).
- In Latin and other simple scripts, shaping is limited to pair kerning and standard ligatures; no contextual alternates or other OpenType features (`small-caps` is synthesized from scaled capitals).
- Variable fonts use their default instance (`font-variation-settings` and `font-stretch` are ignored).
- Automatic hyphenation (`hyphens: auto`) has built-in patterns for English, Danish, German, Dutch and Norwegian only; other languages need their patterns in `PdfOptions.Hyphenation`. `hyphenate-limit-chars` is not supported.
- SVG `<text>` uses only the first value of position lists and has no `textPath`; in a text chunk with right-to-left text, `dx` on its `tspan`s is ignored.

CSS

- A table column width that mixes a percentage with other units (`calc(50% - 10px)`) is sized as `auto`.

- `@import` is ignored (use `<link rel="stylesheet">`), and rules inside `@container` blocks are skipped. `@page` rules for `:first`, `:left` and `:right` pages can style margin boxes but not change the page size or margins. A named page's `:first` is the document's first page, as in Chrome, not the first page of the named run.
- Selectors: no `::first-letter` / `::first-line`. A rule with an unsupported selector is skipped. `:hover`, `:focus` and similar interaction states never match. Selectors inside `:is()`, `:where()` and `:not()` may have at most eight compound parts (one when nested in another of them).
- Properties not supported: `filter`, `mix-blend-mode`, `clip-path` on HTML elements (only in SVG), `border-image`, `list-style-image`, `font-variant-numeric` and inset `box-shadow`. No relative colours (`rgb(from ...)`).
- `text-shadow` blur is approximated with faint copies of the text rather than a true Gaussian blur, and decorations (underlines) get no shadow. `text-overflow` only supports `ellipsis` at the end of the line (`clip` and custom strings do nothing). `aspect-ratio` does not take part in min-/max-content sizing (shrink-to-fit widths) and is ignored on tables. `object-fit` applies to raster images; an SVG file in `<img>` keeps its own `preserveAspectRatio` fit.
- Borders: `groove`, `ridge`, `inset` and `outset` draw as solid.
- `position: sticky` is drawn as `relative`; `background-attachment: fixed` scrolls with the box.
- Lists: `list-style-type` covers disc, circle, square, decimal, decimal-leading-zero, alpha, roman and strings only (no Greek, CJK or `@counter-style`).

Backgrounds and transparency

- No `background-clip: text`; backgrounds of inline elements (`<span>`) show only their colour, not images or gradients.
- Blurred box shadows and repeating gradients with translucent stops are blended onto the background colour rather than truly see-through; SVG gradient stops with transparency are blended onto white.

Layout

- Grid: no subgrid.
- Multi-column: children are kept whole and balanced between columns; a single block is never split across columns.
- Overflow: a positioned child with a `z-index` inside a clipping box stays stacked within that box.

Images and SVG

- Images: no AVIF, BMP or TIFF. Animated GIF and WebP files show only their first frame.
- SVG (inline and files): no embedded `<image>`. Filters have no `feTurbulence`, `feImage`, `feTile`, `feConvolveMatrix`, `feDisplacementMap` or lighting (these pass their input through unchanged) and ignore primitive subregions; filtered content becomes an image (text in it is no longer selectable). Watermark images must be raster images (PNG, JPEG, GIF or WebP).

Markdown and the PDF renderer

- Markdown: no footnotes, definition lists, math or front matter.
- 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, 16 GB, Linux, .NET 10.0.12). Times are medians after a 3-second warm-up; "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.

Converter	Engine	To deploy	First PDF	Invoice, 1 page	Feature tour, 5 pages	Report, 30 pages	Parallel	Peak memory
PdfPecker	own, managed	3.1 MB DLL	0.27 s	0.86 ms	2.7 ms	39 ms	962 docs/s	83 MB
HtmlRenderer.PdfSharp 1.6.1	own, managed (CSS 2)	2.4 MB of DLLs	0.47 s	5.0 ms	16 ms	473 ms	88 docs/s	255 MB
iText pdfHTML 6.3.4 *	own, managed	11 MB of DLLs	1.68 s	41 ms	242 ms	4.2 s	8 docs/s	614 MB
wkhtmltopdf 0.12.6	Qt WebKit, native	about 90 MB with Qt	0.31 s	289 ms	333 ms	693 ms	12 docs/s	164 MB
PuppeteerSharp 25.13	Chromium 141	about 600 MB browser	0.91 s	128 ms	166 ms	539 ms	12 docs/s	849 MB
Microsoft.Playwright 1.63	Chromium 141	browser + 135 MB driver	1.08 s	142 ms	191 ms	522 ms	11 docs/s	986 MB
Syncfusion HTML to PDF 35.1.40 *	Chrome (Blink), bundled	about 545 MB with Chrome	2.52 s	263 ms	314 ms	1.7 s	1 doc/s	2,195 MB

* Measured later (October 2026) on the same VM type with the same program. In that run the other converters were up to 70% slower than in the table above and PdfPecker slightly faster, so these two rows are approximate and, if anything, a little slower than a like-for-like run would show.

How to read it:

- PdfPecker is 60 to 150 times faster than the Chromium-based converters on the 1 and 5-page documents and about 14 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: 242 ms for 5 pages and 4.2 s for the 30-page report, which is one long table.
- Syncfusion ran as a trial without a licence key, which adds a line of text to each page. Its `AdditionalDelay` (2 seconds of waiting for scripts by default) was set to 0, as the other Chromium converters don't wait either. With its default settings it scales content down, so the report took 13 pages instead of 29.
- PDF sizes for the 1 / 5 / 30-page documents: PdfPecker 12 / 26 / 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

Document	Pages	Median	Throughput
Dense report with 1,000 table rows	30	19.5 ms	about 1,500 pages/s on one core
Markdown guide (<code>guide.md</code> , 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](#).