WEBLITZ

The complete documentation

Every page of https://weblitzpdf.com's documentation in one PDF, printed by Weblitz on 2026-11-29: to read offline or to give to an AI. The pages online are the latest: https://weblitzpdf.com/docs.

Contents

1	Getting started	https://weblitzpdf.com/docs
2	Every option, rendered	https://weblitzpdf.com/docs/options
3	API reference	https://weblitzpdf.com/docs/api
4	Node.js	https://weblitzpdf.com/docs/node
5	Python	https://weblitzpdf.com/docs/python
6	Java	https://weblitzpdf.com/docs/java
7	HTTP server and Docker	https://weblitzpdf.com/docs/server
8	Pictures of a page	https://weblitzpdf.com/docs/images
9	Printing like Chrome	https://weblitzpdf.com/docs/guides/printing
10	Fonts	https://weblitzpdf.com/docs/guides/fonts
11	Pages, headers and footers	https://weblitzpdf.com/docs/guides/pages
12	Scripts and the virtual clock	https://weblitzpdf.com/docs/guides/scripts
13	Network and security	https://weblitzpdf.com/docs/guides/network-security
14	Many documents	https://weblitzpdf.com/docs/guides/performance
15	Scrolling and printing a part	https://weblitzpdf.com/docs/guides/scrolling
16	Consent boxes and advertisements	https://weblitzpdf.com/docs/guides/consent-and-ads

17 Excel, Word and PowerPoint https://weblitzpdf.com/docs/guides/office

Getting started

Weblitz prints HTML, CSS and the page's scripts to PDF from a .NET library, as Chrome prints them. Every picture of a page in this documentation is a page of a PDF Weblitz printed from the code shown beside it, made on 2026-11-23 on macOS; each PDF is a download.

Install

Weblitz is the NuGet package `WeblitzPdf`, for .NET 8, 9, 10 and 11 on Windows, Linux and macOS. Version 2.0 is a release candidate: the latest is `2.0.0-rc.2`, a prerelease, so the command asks for prereleases.

```
dotnet add package WeblitzPdf --prerelease
```

Or in the project file:

```
<PackageReference Include="WeblitzPdf" Version="2.0.0-rc.2" />
```

Everything is in the package: the layout engine, the fonts it needs to print as Chrome does, and the script engine (V8, built in). Nothing to download at run time, no browser to start.

From Node.js, Python, Java or any language

The same engine, at the same version, in a package of each language and in a Docker image: each carries the engine and the runtime it needs, so no .NET is installed. Their calls run in worker processes the package starts and keeps warm.

.NET `dotnet add package WeblitzPdf --prerelease` NuGet 2.0.0-rc.2

.NET 8, 9, 10 and 11 on Windows, Linux and macOS, x64 and ARM64

Node.js `npm install weblitzpdf` npm 2.0.0-rc.2

Node.js 22 or later, on Linux with glibc (x64, Arm64), macOS (Arm64, x64) and Windows (x64, Arm64)

Python `pip install --pre weblitz` PyPI 2.0.0rc2

Python 3.9 or later, on Linux with glibc (x64, Arm64), macOS (Arm64, x64) and Windows (x64, Arm64)

Java `com.weblitzpdf:weblitzpdf:2.0.0-rc.2` Maven 2.0.0-rc.2

Java 17 or later, on Linux with glibc (x64, Arm64), macOS (Arm64, x64) and Windows (x64, Arm64)

HTTP

`docker run -p 8080:8080 ghcr.io/antoinegou/weblitz-server:2.0.0-rc.2`

2.0.0-rc.2

Docker, linux/amd64 and linux/arm64

The Node.js, Python and Java packages' options are Puppeteer's and Playwright's `page.pdf()` options under their own names; the server takes Puppeteer's. Every package is version 2.0.0-rc.2, which Python spells 2.0.0rc2. Pictures of a page, from each: pictures.

Your licence key

Weblitz is covered by the SSRS2 licence: one key, the one that starts with `SSRS2-`, for SSRS2's reports and for Weblitz. Activate it once when your application starts, or set the environment variable `SSRS2_LICENSE`, which is read at the first render.

```
using WeblitzPdf;

License.Activate("SSRS2-...");          // throws when the key is not valid
// or: License.TryActivate(key, out var error)
Console.WriteLine(License.LicensedTo);  // the company the key names
```

Without a key, Weblitz prints every page with a trial watermark, and does everything else the same: try it all before you buy. License in the reference lists what the key says (its company, expiry, the versions it covers).

Hello, Weblitz

Printed without a licence key: every page carries the trial watermark.

SSRS2 TRIAL — ssrs2.net

Printed without a key. The PDF

Your first PDFs

Four ways in: HTML text, a file with its folder, a web address, a stream. Each call is static and thread-safe, and returns an HtmlPdfResult: the PDF, its page count, and notes on what could not be done.

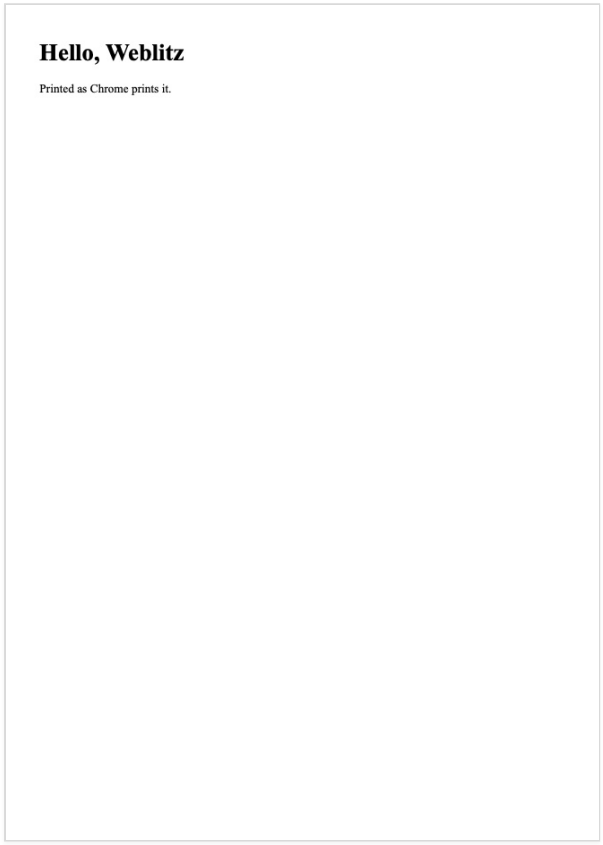
Your first PDF: HTML text

The smallest program: HTML in, PDF bytes out.

Reference: `HtmlToPdf.Render`

HTML text

```
var result = HtmlToPdf.Render("<h1>Hello, Weblitz</h1><p>Printed as Chrome prints it.
</p>");
File.WriteAllBytes("hello.pdf", result.Pdf!);
```



Pages 1
File 37,271 bytes (36.4 KB) · the PDF

A file and its folder

RenderFile reads the file's stylesheet, pictures, fonts and scripts from its own folder, and nothing outside it.

Reference: `HtmlToPdf.RenderFile`

▶ The page printed: `letter/index.html`

A folder: HTML, CSS, a logo

```
var result = HtmlToPdf.RenderFile(path);
```



Pages 1
File 50,768 bytes (49.6 KB) · the PDF

A web address

RenderUrl fetches the page and what it loads. With the network left off, the page's own host is allowed (here a server the generator starts on this machine, as `assets`).

Reference: `HtmlToPdf.RenderUrl`, `HtmlPdfOptions.Network`

▶ The page printed: `letter/index.html`

RenderUrl

```
var result = HtmlToPdf.RenderUrl(new Uri(assets, "letter/index.html"));
```



Pages 1
File 50,768 bytes (49.6 KB) · the PDF

Into a stream

Render(html, stream) writes the PDF to a stream (the result's Pdf is then null): a file, a response, a blob.

Reference: `HtmlToPdf.Render`

To a file stream

```
using var file = File.Create(output);
var result = HtmlToPdf.Render("<h1>Straight into a stream</h1>", file);
```

Pages	1
File	18,330 bytes (17.9 KB) · the PDF
Pdf	null: written to the stream

An ASP.NET Core endpoint

An endpoint returning a PDF made from your own HTML (a Razor view rendered to a string, a template, a page of your site). Render runs on the calling thread; RenderAsync runs it on the thread pool.

```
using WeblitzPdf;

app.MapGet("/invoices/{id}.pdf", async (int id, InvoiceViews views) =>
{
    var html = await views.Html(id);
    var result = await HtmlToPdf.RenderAsync(html, new HtmlPdfOptions {
Title = $"Invoice {id}" });
    return Results.File(result.Pdf!, "application/pdf", $"invoice-
{id}.pdf");
});
```

`InvoiceViews` stands for your own code that makes the invoice's HTML.

Next

Every option, rendered

Paper, margins, scale, backgrounds, pages, scripts, the network, compression, passwords, limits, notes, the browser a page sees.

API reference

Every public type and member, from the engine's own documentation.

Printing like Chrome

What happens between your HTML and the PDF.

Fonts

The machine's fonts, the bundled faces, Linux containers.

Every option, rendered

For each option that changes what comes out: a small page, the C# that sets the option, and the pages Weblitz printed, beside the default where that shows the difference. Made on 2026-11-23 on macOS; the pages are drawn from the PDFs, which are downloads. What the engine noted is shown as it came back, including what it does not do yet.

Paper size and orientation
Margins
Scale
Header and footer templates
Backgrounds
Choosing pages
The page's own @page size
Scripts and the virtual clock
The network
Pages behind a login
PDF compression
Passwords and permissions
Title, author and PDF/A
Limits
Notes: what could not be done
The browser the page sees
Locale, time zone and clock
Installed fonts
Lists that load as you scroll
Scrolling to a section first
Printing a part of the page
Chrome's experimental features
Printing documentation without its menus
Without the consent boxes
Without advertisements and trackers

Paper size and orientation

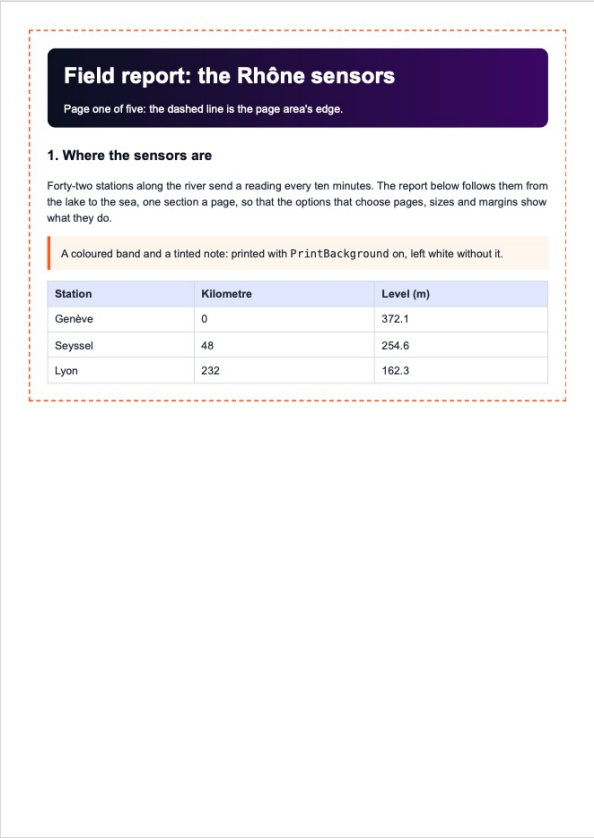
The page area changes with the paper: the dashed frame is the area's edge.

Reference: `HtmlPdfOptions.Format`, `HtmlPdfOptions.Landscape`, `HtmlPdfOptions.Width`, `HtmlPdfOptions.Height`

▶ The page printed: `article.html`

Default: A4, portrait

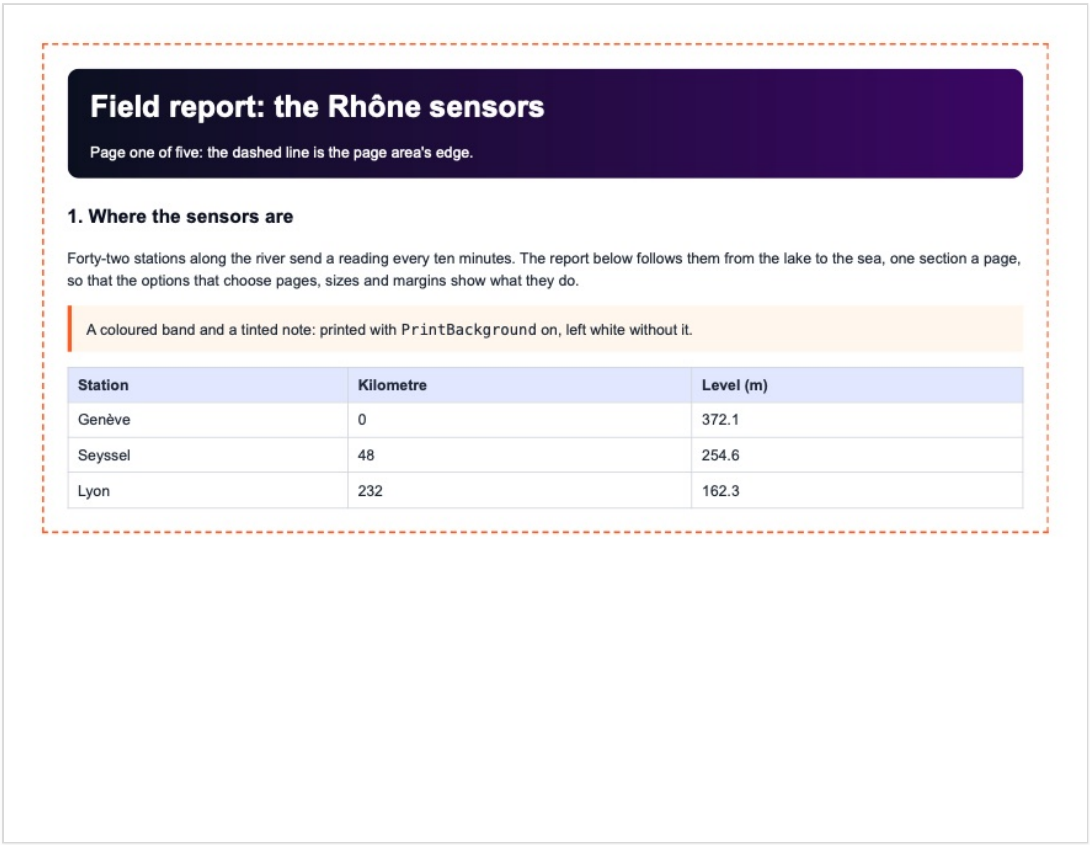
```
var result = HtmlToPdf.Render(html);
```



Pages 5
File 95,383 bytes (93.1 KB) · the PDF

Letter, landscape

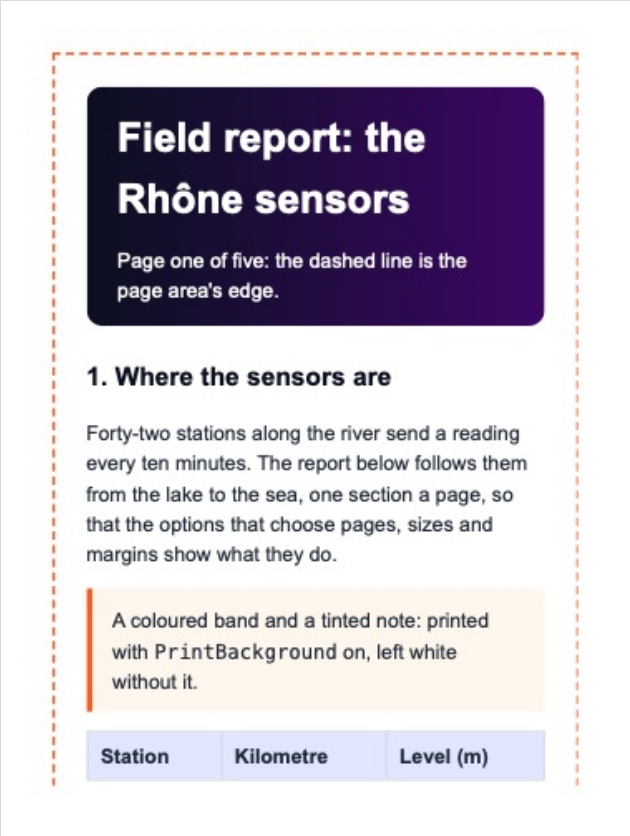
```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Format = "Letter", Landscape = true });
```



Pages 5
File 99,598 bytes (97.3 KB) · the PDF

Width and height: 120 × 160 mm

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Width = "120mm", Height =
"160mm" });
```



Pages 6
File 87,157 bytes (85.1 KB) · the PDF

Margins

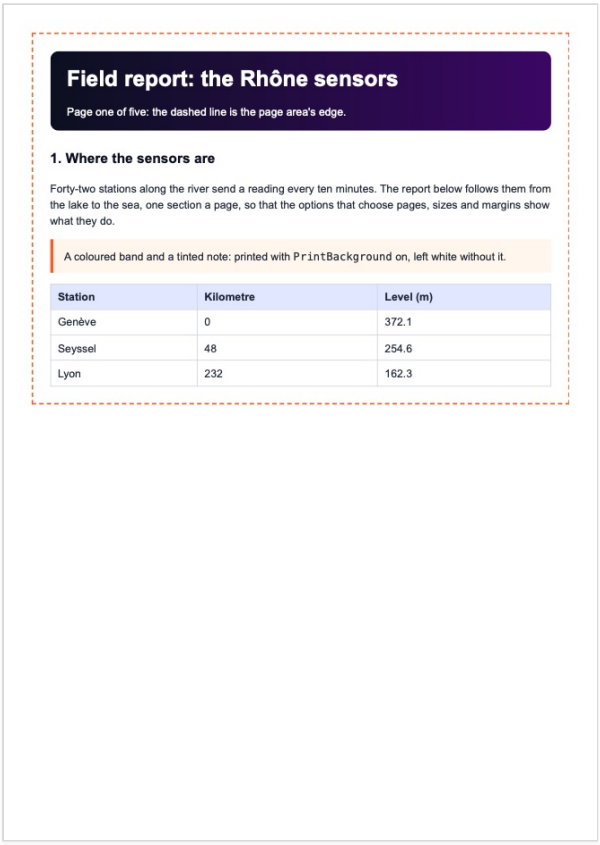
Without margins in the options or the page's @page rule, Chrome's default margins apply.

Reference: `HtmlPdfOptions.Margin`, `HtmlMargins`

▶ The page printed: `article.html`

Default

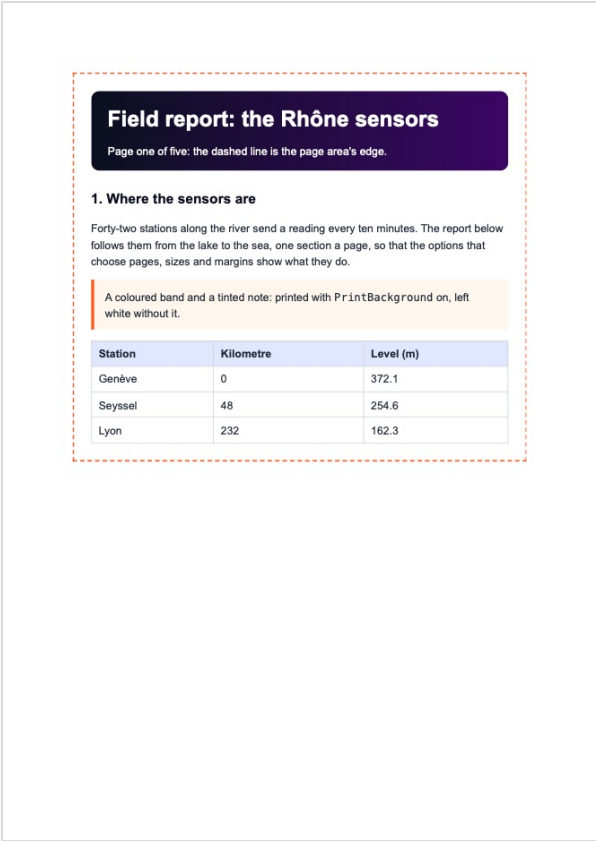
```
var result = HtmlToPdf.Render(html);
```



Pages 5
File 95,383 bytes (93.1 KB) · the PDF

25 mm all round

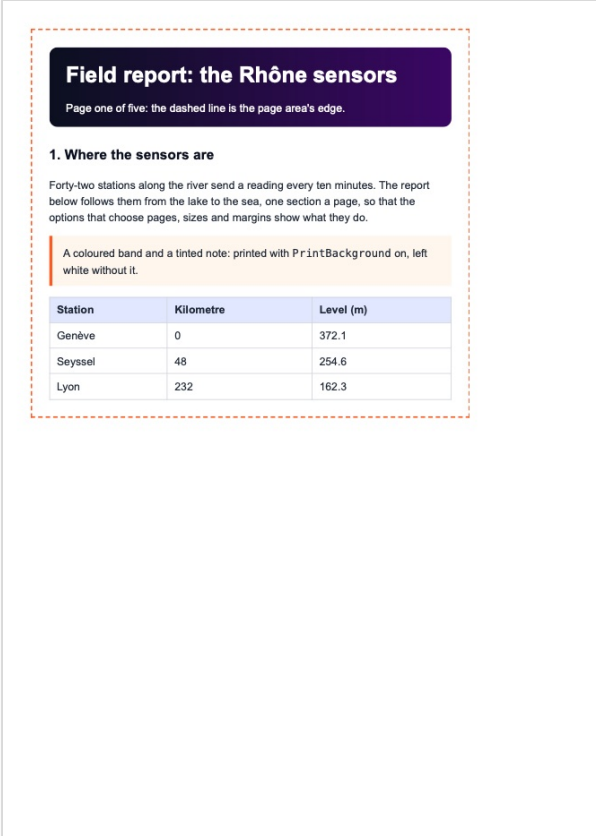
```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Margin =
HtmlMargins.All("25mm") });
```



Pages 5
File 91,048 bytes (88.9 KB) · the PDF

Top, right, bottom, left

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Margin = new
HtmlMargins("10mm", "45mm", "10mm", "10mm") });
```



Pages 5
File 91,538 bytes (89.4 KB) · the PDF

Scale

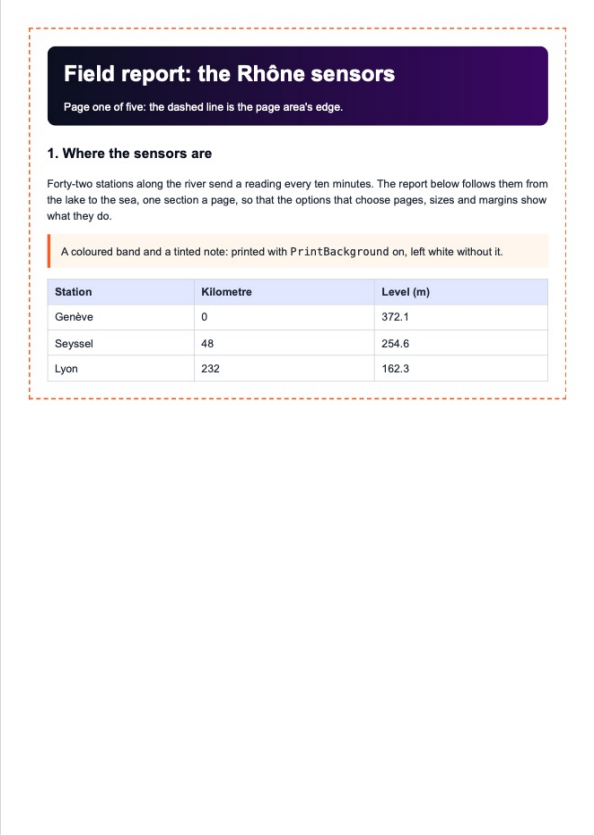
The page is laid out as if the paper were larger or smaller, then printed on it: 0.1 to 2.

Reference: `HtmlPdfOptions.Scale`

▶ The page printed: `article.html`

1 (default)

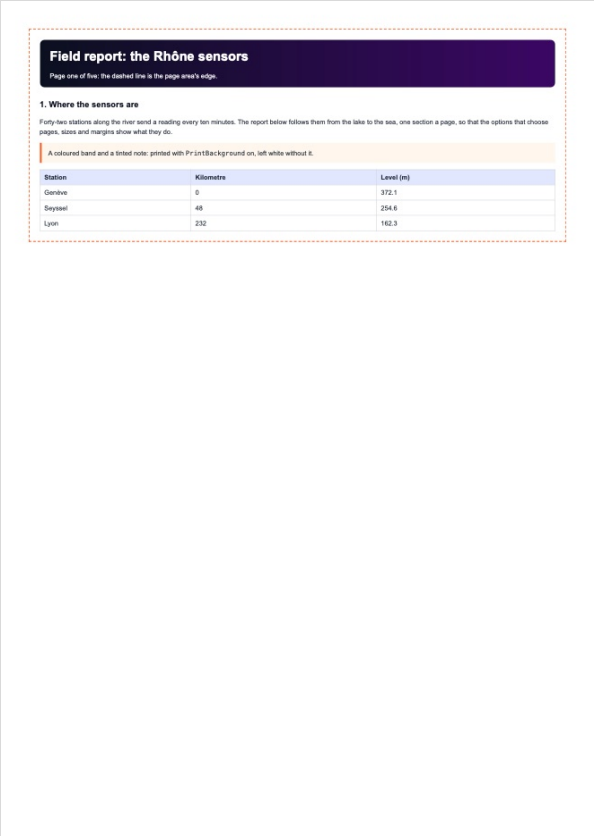
```
var result = HtmlToPdf.Render(html);
```



Pages 5
File 95,383 bytes (93.1 KB) · the PDF

0.6

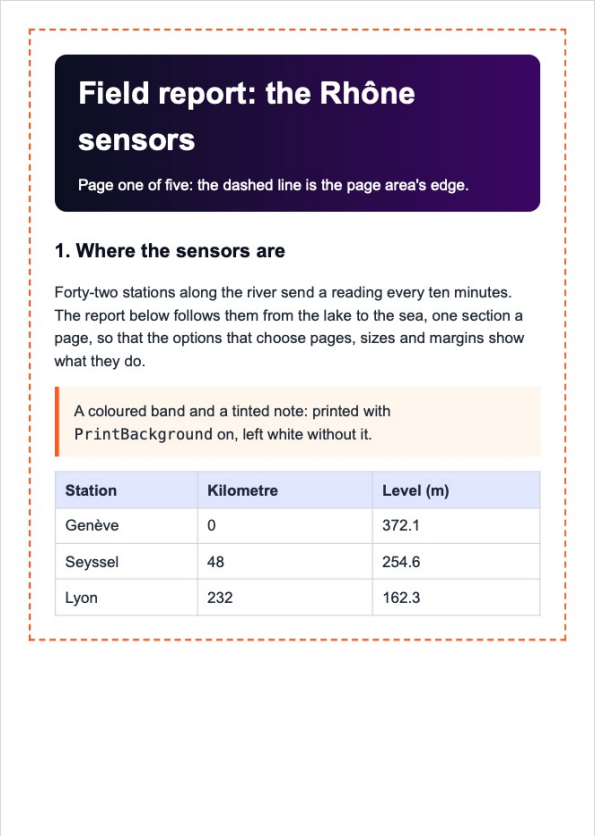
```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Scale = 0.6 });
```



Pages 5
File 102,606 bytes (100.2 KB) · the PDF

1.4

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Scale = 1.4 });
```



Field report: the Rhône sensors

Page one of five: the dashed line is the page area's edge.

1. Where the sensors are

Forty-two stations along the river send a reading every ten minutes. The report below follows them from the lake to the sea, one section a page, so that the options that choose pages, sizes and margins show what they do.

A coloured band and a tinted note: printed with `PrintBackground` on, left white without it.

Station	Kilometre	Level (m)
Genève	0	372.1
Seyssel	48	254.6
Lyon	232	162.3

Pages 5

File 92,984 bytes (90.8 KB) · the PDF

Header and footer templates

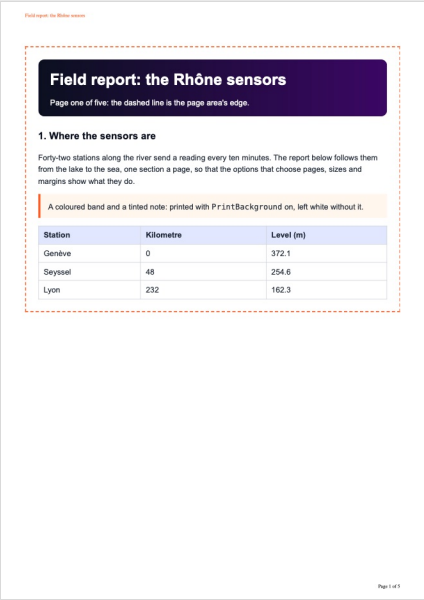
The templates are HTML printed in the top and bottom margins of every page, as Chrome prints them; elements with the classes pageNumber, totalPages, title, url and date are filled in. A template that sets no font size gets Chrome's tiny default: set one. Give the margins room for them.

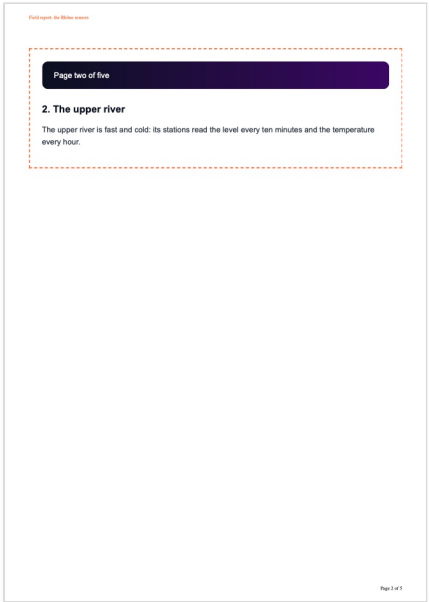
Reference: HtmlPdfOptions.DisplayHeaderFooter, HtmlPdfOptions.HeaderTemplate, HtmlPdfOptions.FooterTemplate

▶ The page printed: `article.html`

Header and footer

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    DisplayHeaderFooter = true,
    Margin = new HtmlMargins("22mm", "12mm", "20mm", "12mm"),
    HeaderTemplate = "<div style='font-size:9px; width:100%; padding:0 12mm;
color:#ff5a1f'><span class='title'></span></div>",
    FooterTemplate = "<div style='font-size:9px; width:100%; padding:0 12mm; text-
align:right'>Page <span class='pageNumber'></span> of <span class='totalPages'></span>
</div>",
});
```





Pages 5
File 108,127 bytes (105.6 KB) · the PDF

Backgrounds

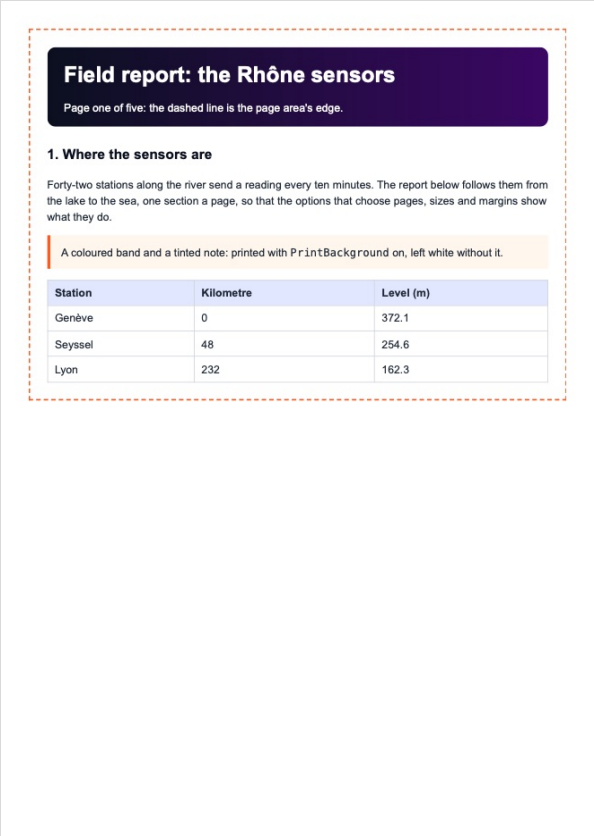
Background colours, gradients and pictures are printed by default; off, they are left out, as Chrome's "Background graphics" box unticked.

Reference: `HtmlPdfOptions.PrintBackground`

▶ The page printed: `article.html`

true (default)

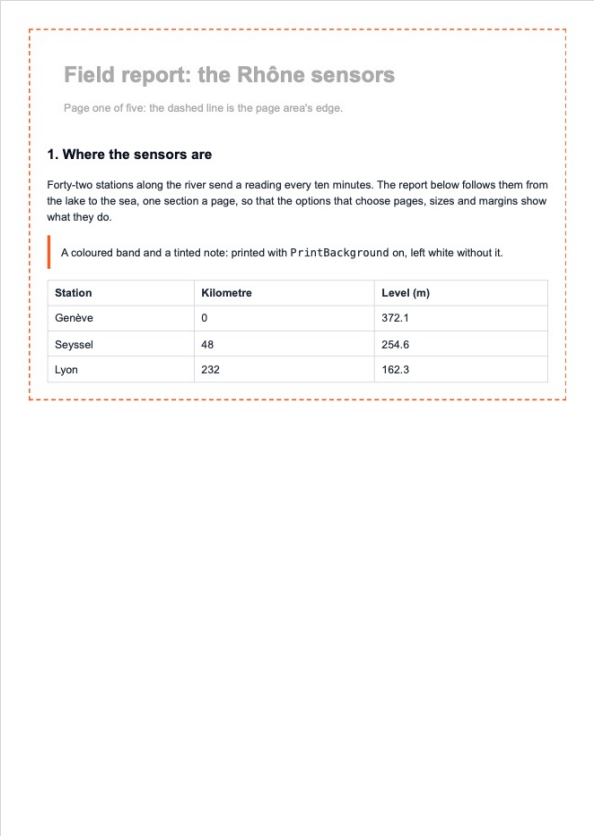
```
var result = HtmlToPdf.Render(html);
```



Pages 5
File 95,383 bytes (93.1 KB) · the PDF

false

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { PrintBackground = false });
```



Pages 5
File 87,547 bytes (85.5 KB) · the PDF

Choosing pages

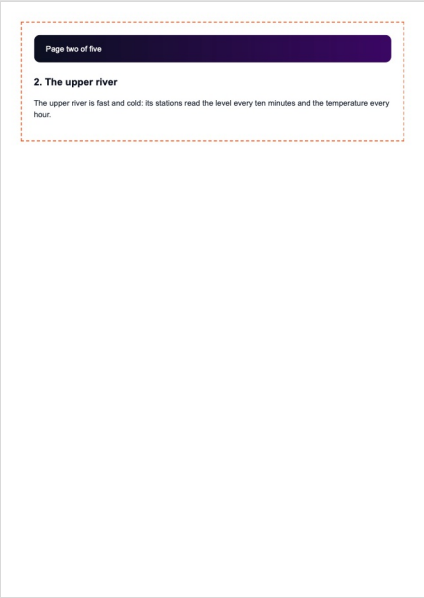
The article has five pages; the PDF holds the ones asked for.

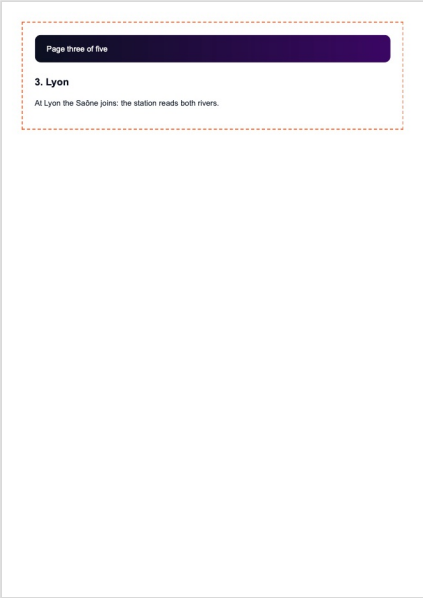
Reference: `HtmlPdfOptions.PageRanges`

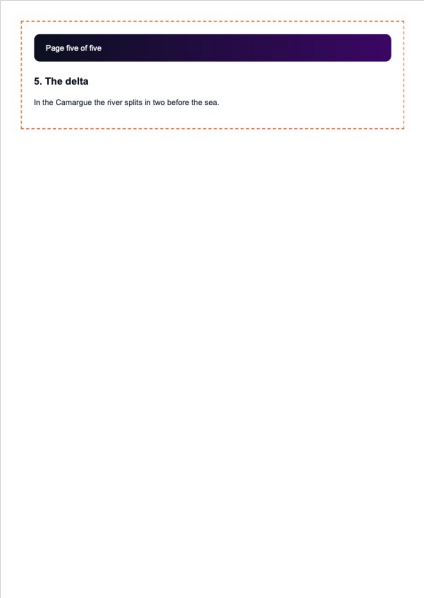
▶ The page printed: `article.html`

"2-3, 5"

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { PageRanges = "2-3, 5" });
```







Pages	3
File	59,266 bytes (57.9 KB) · the PDF

The page's own @page size

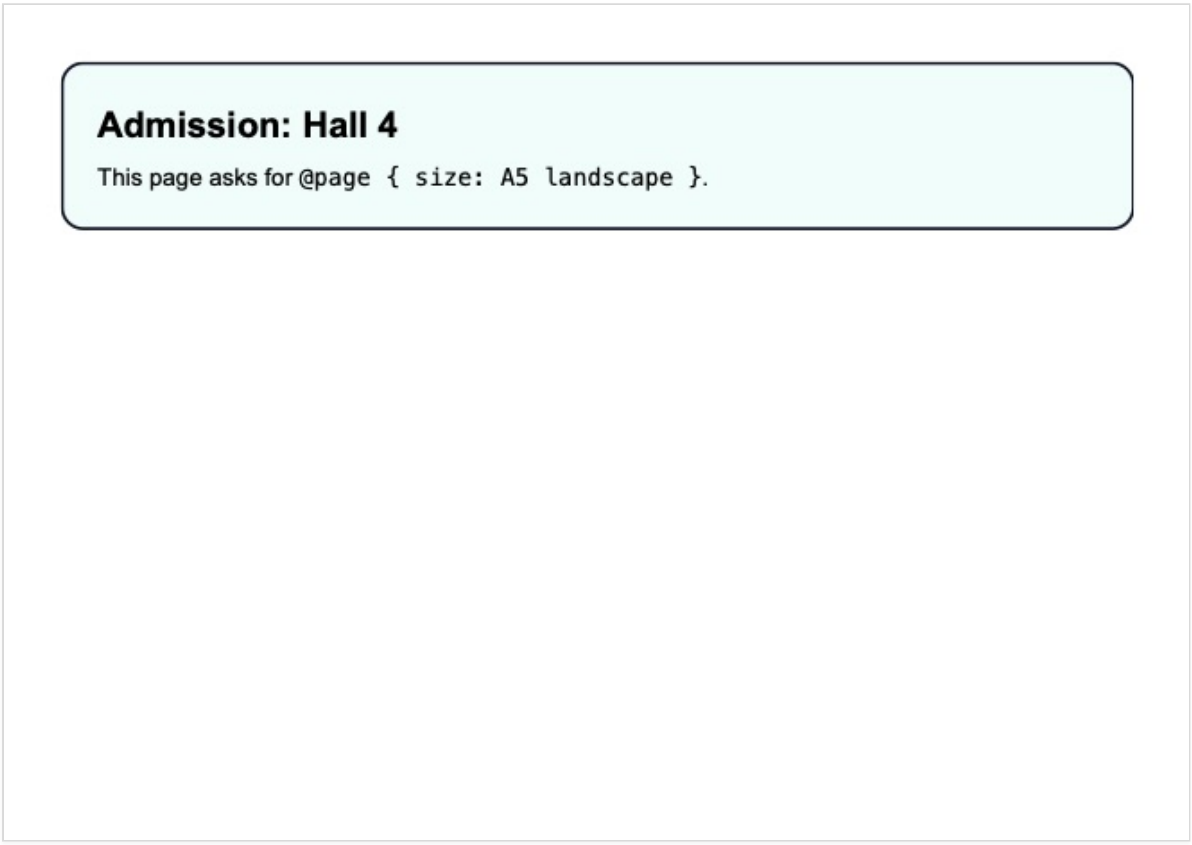
A page that asks for its paper size gets it by default; with PreferCssPageSize false, the options' Format wins.

Reference: `HtmlPdfOptions.PreferCssPageSize`

▶ The page printed: `css-page-size.html`

true (default): A5 landscape, as the page asks

```
var result = HtmlToPdf.Render(html);
```



Pages 1
File 39,665 bytes (38.7 KB) · the PDF

false: the options' A4

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { PreferCssPageSize = false,
Format = "A4" });
```



Pages 1
File 39,680 bytes (38.8 KB) · the PDF

Scripts and the virtual clock

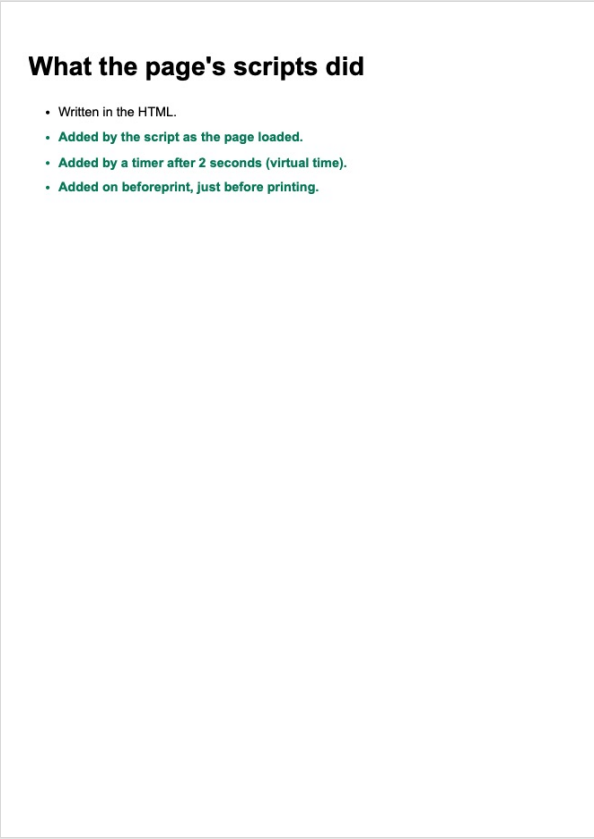
Scripts run once before printing; timers run in virtual time for WaitFor (5 seconds by default), so a page's 2-second timer fires and its 8-second one only with a longer wait. beforeprint runs just before the PDF is made.

Reference: `HtmlPdfOptions.Scripts`, `HtmlScripts`

▶ The page printed: `scripts.html`

Default: scripts on, 5 s of virtual time

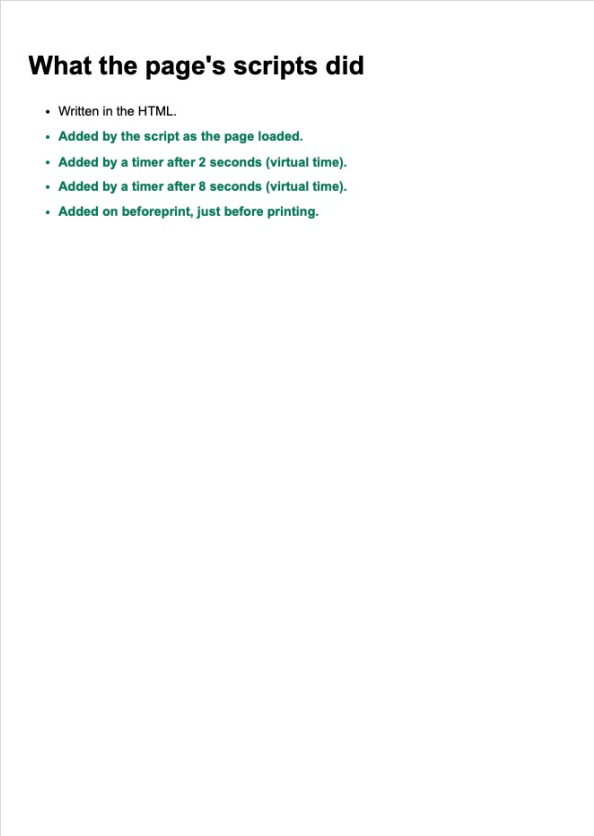
```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	36,218 bytes (35.4 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (2017 ms of virtual time, 5 tasks; 0 canvases drawn).

WaitFor 10 s

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Scripts = HtmlScripts.Default
with { WaitFor = TimeSpan.FromSeconds(10) } });
```



Pages	1
File	37,010 bytes (36.1 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (8017 ms of virtual time, 8 tasks; 0 canvases drawn).

Scripts off

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Scripts = HtmlScripts.Off });
```



Pages 1
File 30,676 bytes (30.0 KB) · the PDF

The network

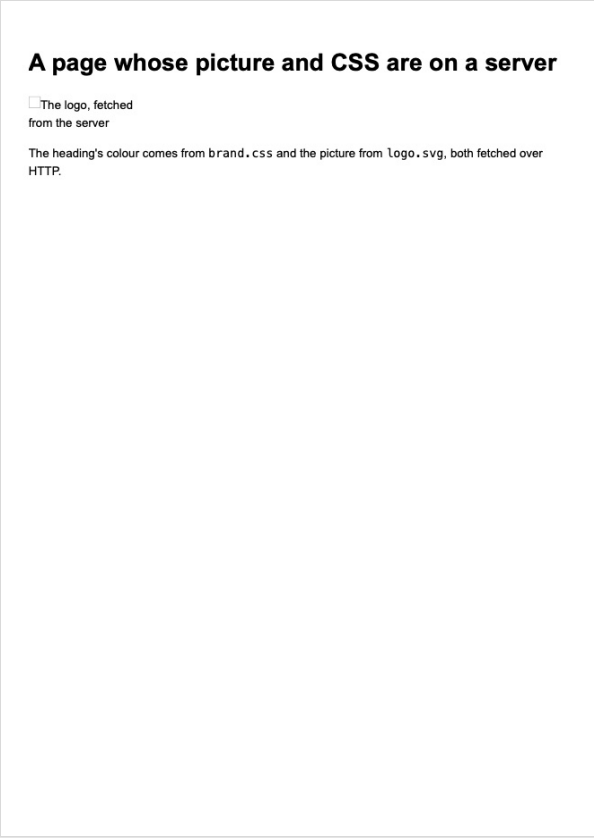
The network is off by default: a page's web addresses are not fetched and noted instead. Allow the hosts a page may load from (`assets` is a server the generator starts on this machine).

Reference: `HtmlPdfOptions.Network`, `HtmlNetwork`, `HtmlPdfOptions.BaseUrl`

▶ The page printed: `network.html`

Off (default)

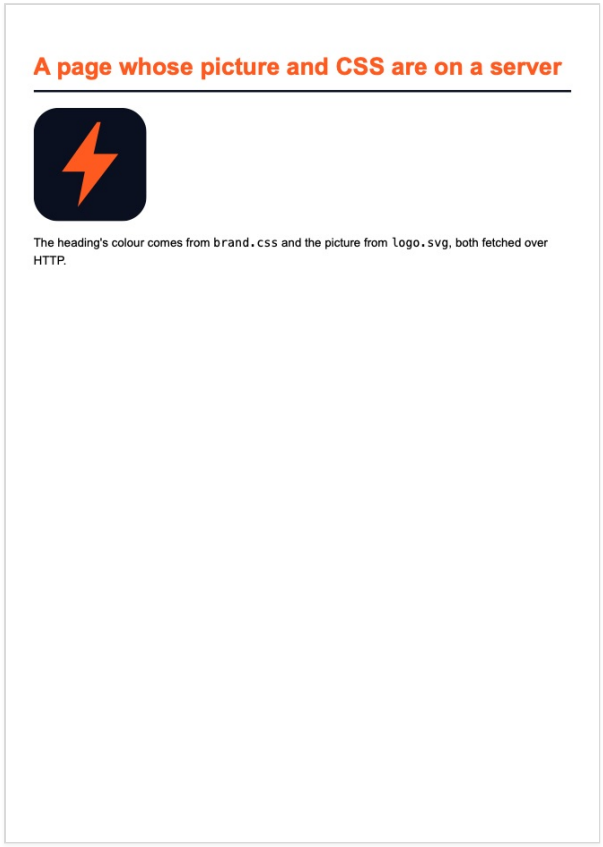
```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { BaseUrl = assets });
```



Pages	1
File	44,119 bytes (43.1 KB) · the PDF
Notes	`Info resourceRefused`: StyleSheet http://127.0.0.1:18431/brand.css: no resource loader was given (only data: URLs load). `Info resourceRefused`: Image http://127.0.0.1:18431/logo.svg: no resource loader was given (only data: URLs load). `Info notImplemented`: A picture that did not load shows its alternative text; Chrome's broken-image icon beside it is laid out but not drawn.

Allow the assets' host

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { BaseUrl = assets, Network = HtmlNetwork.Allow(assets.Host) });
```



Pages 1
File 45,952 bytes (44.9 KB) · the PDF

Pages behind a login

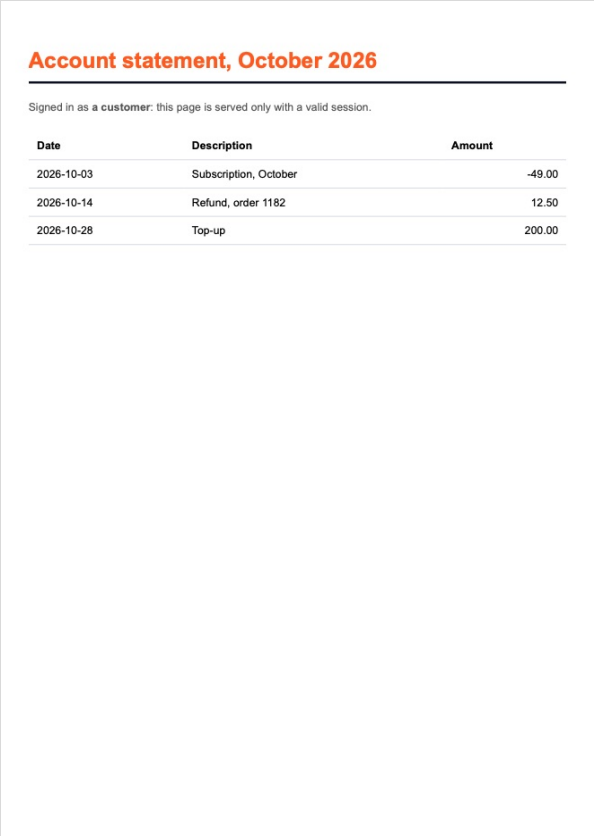
A page your server shows only to signed-in users prints with what a browser would send: a bearer token, a session cookie, or a password for HTTP authentication. Each goes only to the hosts it names, by default the document's own (`assets` is a server the generator starts on this machine; its account pages need one of them).

Reference: `HtmlNetwork.Headers`, `HtmlNetwork.Cookies`, `HtmlNetwork.AcceptCookies`, `HtmlNetwork.Credentials`, `HtmlCookie`

▶ The page printed: `assets/account/statement.html`

A bearer token

```
var result = HtmlToPdf.RenderUrl(new Uri(assets, "account/statement.html"), new
HtmlPdfOptions
{
    Network = HtmlNetwork.Allow(assets.Host).WithBearerToken(accessToken),
});
```



Account statement, October 2026

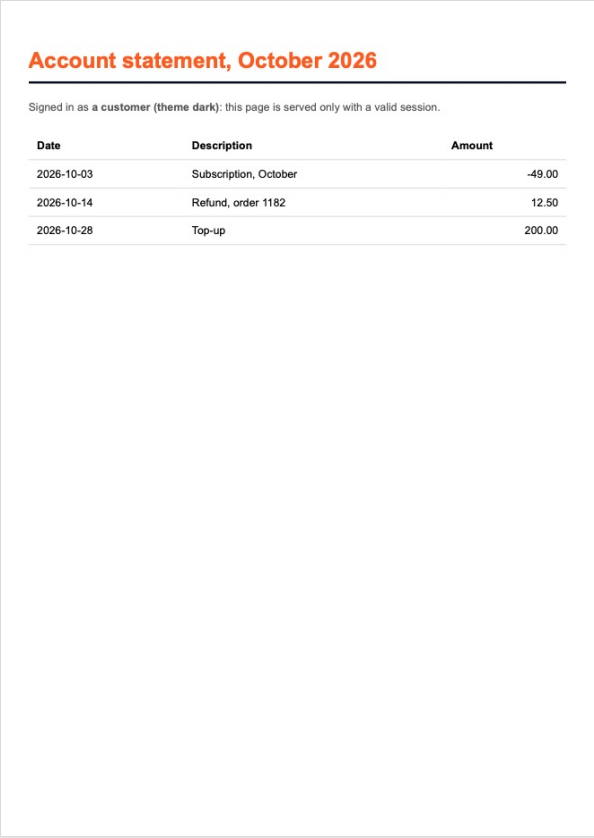
Signed in as **a customer**: this page is served only with a valid session.

Date	Description	Amount
2026-10-03	Subscription, October	-49.00
2026-10-14	Refund, order 1182	12.50
2026-10-28	Top-up	200.00

Pages 1
File 43,239 bytes (42.2 KB) · the PDF
Notes `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

A session cookie

```
var result = HtmlToPdf.RenderUrl(new Uri(assets, "account/statement.html"), new
HtmlPdfOptions
{
    Network = HtmlNetwork.Allow(assets.Host)
        .WithCookie(new HtmlCookie("session", sessionId) { HttpOnly = true })
        .WithCookie(new HtmlCookie("theme", "dark")),
});
```



Account statement, October 2026

Signed in as **a customer (theme dark)**: this page is served only with a valid session.

Date	Description	Amount
2026-10-03	Subscription, October	-49.00
2026-10-14	Refund, order 1182	12.50
2026-10-28	Top-up	200.00

Pages 1
File 44,446 bytes (43.4 KB) · the PDF
Notes `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

PDF compression

The same pages, deflated more or less: the files' sizes below, measured when these pages were made.

Reference: `HtmlPdfOptions.Compression`, `HtmlPdfCompression`

▶ The page printed: `article.html`

None

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Compression =
HtmlPdfCompression.None });
```

Pages 5
File 345,843 bytes (337.7 KB) · the PDF

Fastest (default)

```
var result = HtmlToPdf.Render(html);
```

Pages 5
File 95,383 bytes (93.1 KB) · the PDF

Default

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Compression =
HtmlPdfCompression.Default });
```

Pages 5
File 74,765 bytes (73.0 KB) · the PDF

Smallest

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Compression =
HtmlPdfCompression.Smallest });
```

Pages 5
File 72,832 bytes (71.1 KB) · the PDF

Passwords and permissions

The PDF is encrypted (AES-256): a reader asks for the user password, and without the owner's it may not copy (here).

Reference: `HtmlPdfOptions.Security`, `HtmlPdfSecurity`

▶ The page printed: `article.html`

A user password, no copying

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    Security = new HtmlPdfSecurity { UserPassword = "open-me", OwnerPassword = "owner-secret", PermitCopying = false },
});
```

Pages	5
File	98,636 bytes (96.3 KB), encrypted · the PDF

Title, author and PDF/A

What the PDF says of itself; PDF/A-2b for archives.

Reference: HtmlPdfOptions.Title, HtmlPdfOptions.Author, HtmlPdfOptions.PdfA

▶ The page printed: `article.html`

Title, author, PDF/A

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Title = "Rhône field report",
Author = "Hydrology team", PdfA = true });
```

Pages	5
File	99,689 bytes (97.4 KB), PDF/A · the PDF

Limits

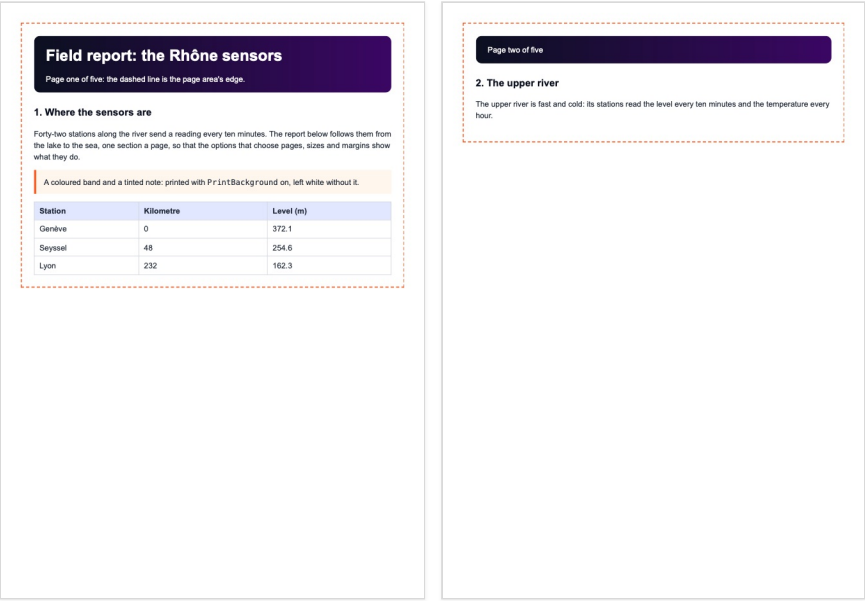
A page held to its limits: here two pages at most. The render stops there, prints what it has, and says so: Completed is false and a note tells which limit.

Reference: `HtmlPdfOptions.Limits`, `HtmlPdfLimits`, `HtmlPdfResult.Completed`

▶ The page printed: `article.html`

MaxPages = 2

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Limits =
HtmlPdfLimits.Default with { MaxPages = 2 } });
Console.WriteLine($"{result.PageCount} pages, completed: {result.Completed}");
```



Pages	2, stopped early (Completed false)
File	74,993 bytes (73.2 KB) · the PDF
Notes	`Warning limitReached`: the document was cut after 2 pages.

Notes: what could not be done

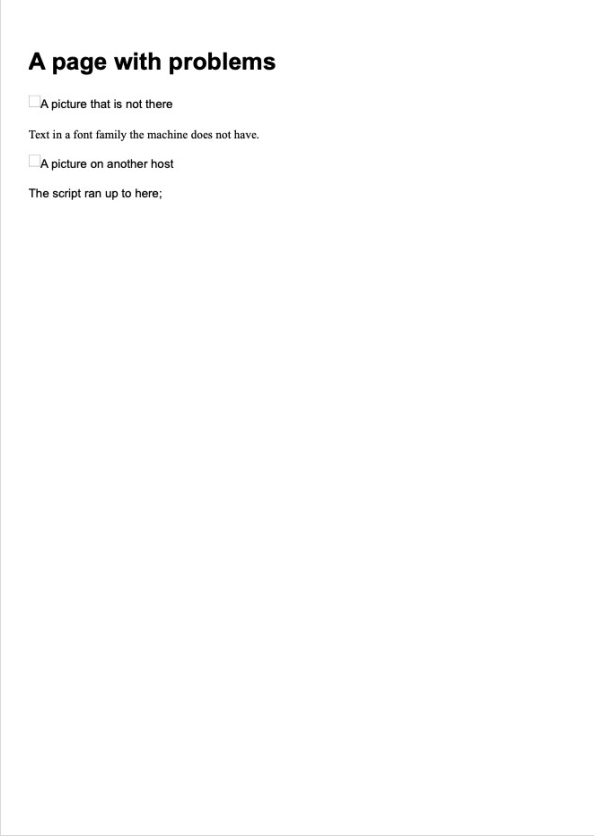
A page's own problems never throw: a picture missing, a host not allowed, a font the machine lacks, a script that throws. Each is a note.

Reference: `HtmlPdfResult.Notes`, `HtmlPdfNote`, `HtmlPdfNoteSeverity`

▶ The page printed: `notes.html`

A page with problems

```
var result = HtmlToPdf.Render(html);
foreach (var note in result.Notes)
    Console.WriteLine($"{note.Severity} {note.Code}: {note.Message}");
```



Pages	1
File	56,265 bytes (54.9 KB) · the PDF
Notes	`Warning scriptError`: ReferenceError: undefinedFunction is not defined | at about:blank#script-25:11:3 | at HostDelegate.<anonymous> (<anonymous>) | at Object.hostCall (ssrs2-dom.js:15562:11) `Warning resourceFailed`: Image "missing-picture.png" is not a URL that resolves against no base URL. `Info resourceRefused`: Image https://cdn.example.com/banner.png: no resource loader was given (only data: URLs load). `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 4 tasks; 0 canvases drawn). `Info notImplemented`: A picture that did not load shows its alternative text; Chrome's broken-image icon beside it is laid out but not drawn. `Info fontFamilyMissing`: The font family "Some Font Nobody Has" is not on this machine: its text goes to the next family.

The browser the page sees

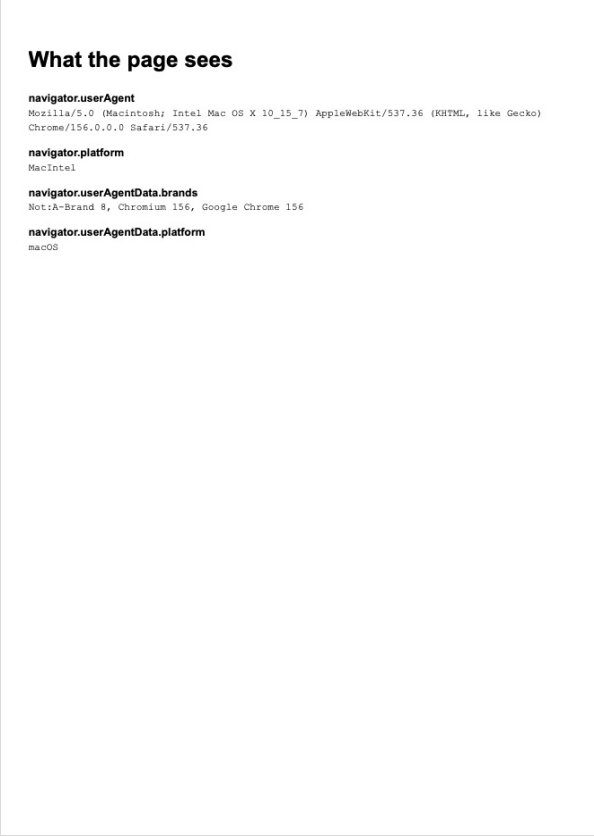
By default a page sees desktop Chrome 156 on the machine's own OS. These pages were made on macOS.

Reference: `HtmlPdfOptions.UserAgent`, `HtmlPdfOptions.UserAgentMetadata`, `HtmlUserAgentMetadata`, `HtmlUserAgentBrand`

▶ The page printed: `user-agent.html`

Default

```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	44,873 bytes (43.8 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

A User-Agent and its metadata

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    UserAgent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML,
like Gecko) Chrome/156.0.0.0 Safari/537.36 InvoiceBot/2.1",
    UserAgentMetadata = HtmlUserAgentMetadata.ChromeOnThisMachine with
    {
        Platform = "Windows",
        Brands = [new("InvoiceBot", "2"), new("Chromium", "156")],
    },
});
```



Pages 1
File 45,030 bytes (44.0 KB) · the PDF
Notes `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

Locale, time zone and clock

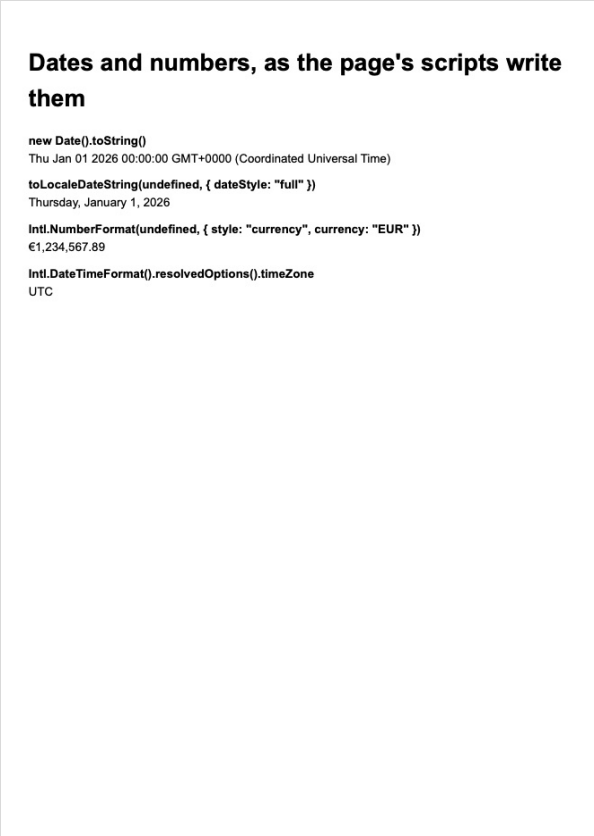
What the page's scripts see of the world: the same values every run, so the same input gives the same PDF.

Reference: `HtmlPdfOptions.Locale`, `HtmlPdfOptions.TimeZone`, `HtmlPdfOptions.Now`

▶ The page printed: `locale.html`

Default

```
var result = HtmlToPdf.Render(html);
```



Dates and numbers, as the page's scripts write them

new Date().toString()
Thu Jan 01 2026 00:00:00 GMT+0000 (Coordinated Universal Time)

toLocaleDateString(undefined, { dateStyle: "full" })
Thursday, January 1, 2026

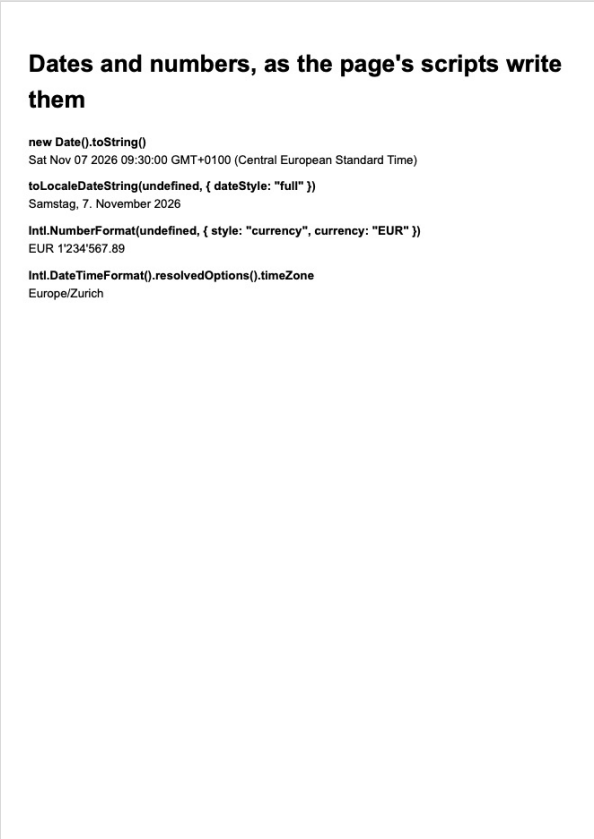
Intl.NumberFormat(undefined, { style: "currency", currency: "EUR" })
€1,234,567.89

Intl.DateTimeFormat().resolvedOptions().timeZone
UTC

Pages	1
File	48,120 bytes (47.0 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

de-CH, Europe/Zurich, a fixed clock

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    Locale = "de-CH",
    TimeZone = "Europe/Zurich",
    Now = new DateTimeOffset(2026, 11, 7, 9, 30, 0, TimeSpan.FromHours(1)),
});
```



Dates and numbers, as the page's scripts write them

new Date().toString()
Sat Nov 07 2026 09:30:00 GMT+0100 (Central European Standard Time)

toLocaleDateString(undefined, { dateStyle: "full" })
Samstag, 7. November 2026

Intl.NumberFormat(undefined, { style: "currency", currency: "EUR" })
EUR 1'234'567.89

Intl.DateTimeFormat().resolvedOptions().timeZone
Europe/Zurich

Pages 1
File 50,717 bytes (49.5 KB) · the PDF
Notes `Info scripts`Ran: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

Installed fonts

By default the machine's fonts are used after the page's own @font-face fonts; off, only the bundled faces. These pages were made on macOS.

Reference: `HtmlPdfOptions.UseInstalledFonts`

▶ The page printed: `fonts.html`

true (default)

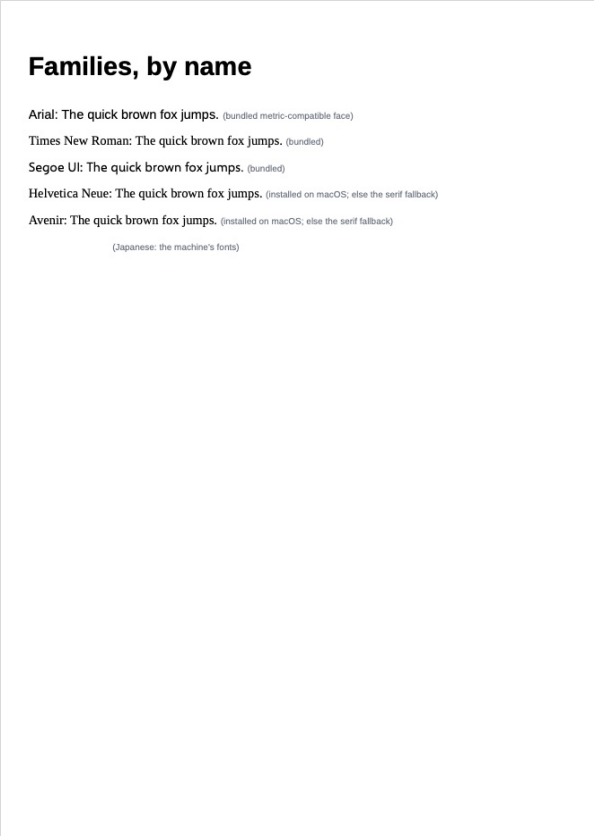
```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	104,493 bytes (102.0 KB) · the PDF
Notes	`Info fontFamilyMissing`: The font family "Segoe UI" is not on this machine: its text goes to the next family.

false: the bundled faces only

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { UseInstalledFonts = false });
```



Pages	1
File	36,473 bytes (35.6 KB) · the PDF
Notes	`Info fontFamilyMissing`: The font family "Helvetica Neue" is not in the font profile "reference": its text goes to the next family. `Info fontFamilyMissing`: The font family "Avenir" is not in the font profile "reference": its text goes to the next family. `Info fontFamilyMissing`: The font family "Hiragino Sans" is not in the font profile "reference": its text goes to the next family. `Warning missingGlyph`: No font of 'Hiragino Sans, serif' nor the fallback has a glyph for a character: drawn as .notdef.

Lists that load as you scroll

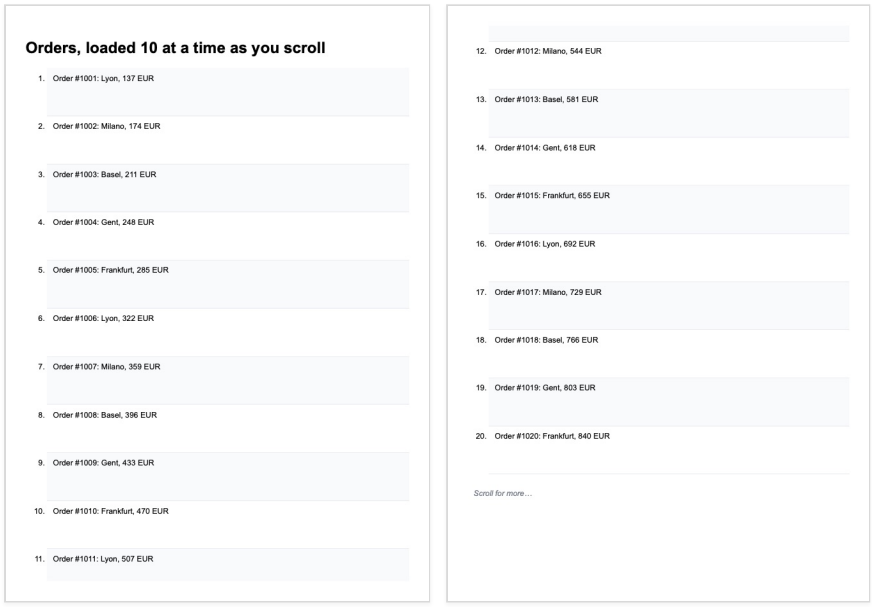
An infinite list loads its next ten orders each time its bottom comes into view. Without a scroll option, Weblitz's IntersectionObserver reports the bottom visible once, so the PDF holds twenty; ScrollThrough scrolls the window down a window height at a time, each step waiting ScrollStepWait of virtual time, so the list loads to its end and all sixty are printed.

Reference: `HtmlPdfOptions.ScrollThrough`, `HtmlPdfOptions.ScrollStepWait`

▶ The page printed: `lazy-list.html`

Default: not scrolled

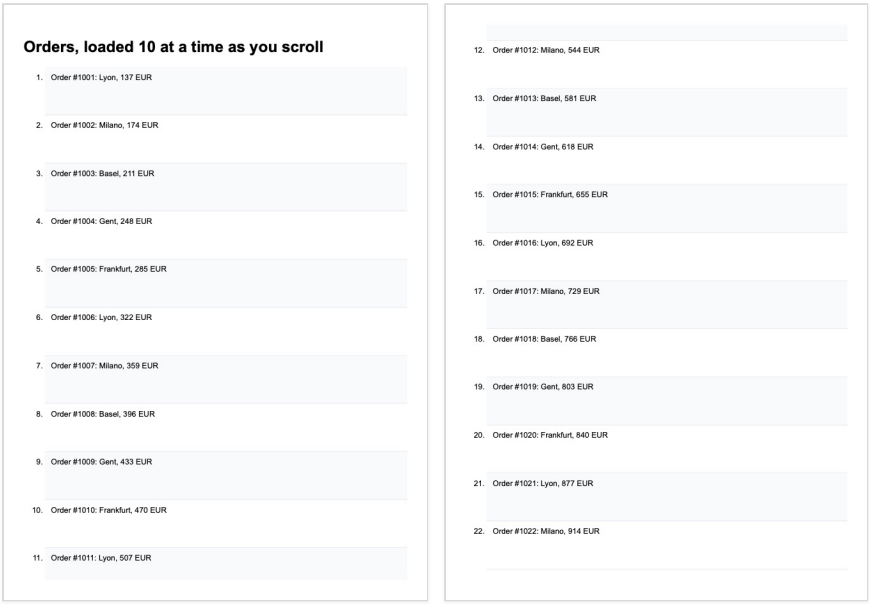
```
var result = HtmlToPdf.Render(html);
```



Pages	2
File	53,524 bytes (52.3 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (133 ms of virtual time, 5 tasks; 0 canvases drawn).

ScrollThrough

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { ScrollThrough = true });
```



Pages 6

File 58,677 bytes (57.3 KB) · the PDF

Notes `Info scriptsRan`: The page's scripts ran (883 ms of virtual time, 40 tasks; 0 canvases drawn).

Scrolling to a section first

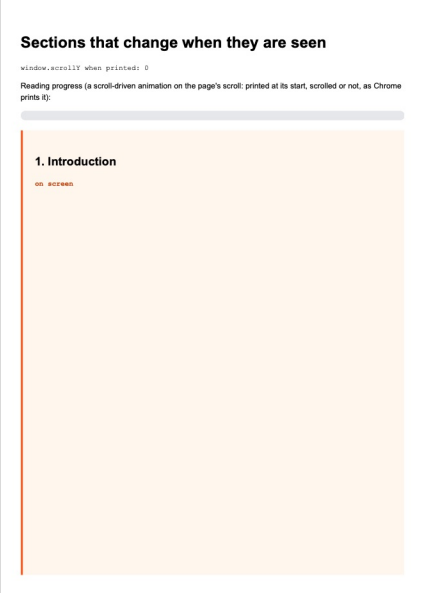
ScrollTo scrolls the window before printing, as a user would before pressing Print: the page's scroll events and IntersectionObserver callbacks see it (here the sections that went off screen, and scrollY). A scroll timeline on the page's own scroll prints at its start either way, as Chrome prints it. The PDF still holds the whole document from its top.

Reference: `HtmlPdfOptions.ScrollTo`

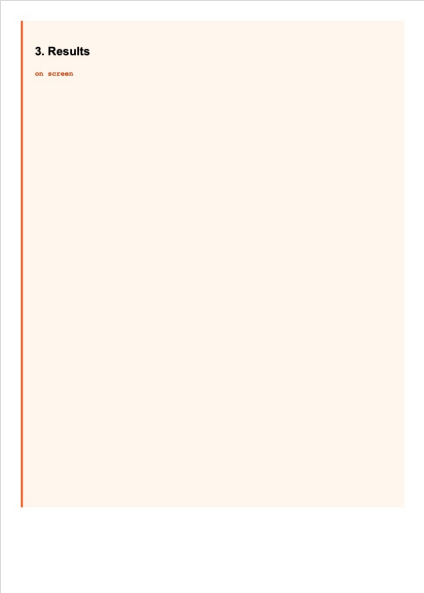
▶ The page printed: `scroll-effects.html`

Default: not scrolled

```
var result = HtmlToPdf.Render(html);
```







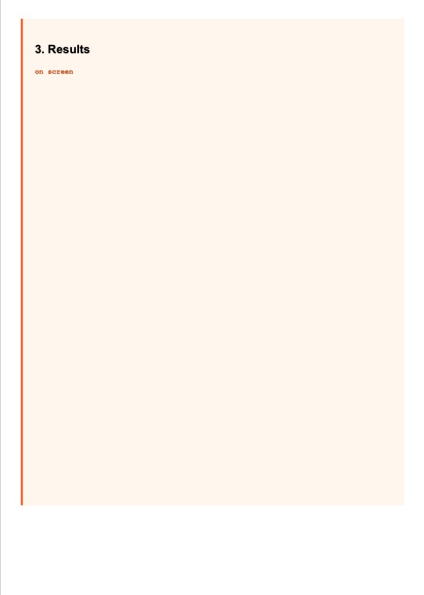
Pages	3
File	69,331 bytes (67.7 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

ScrollTo "#results"

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { ScrollTo = "#results" });
```







Pages	3
File	70,553 bytes (68.9 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (33 ms of virtual time, 4 tasks; 0 canvases drawn).

Printing a part of the page

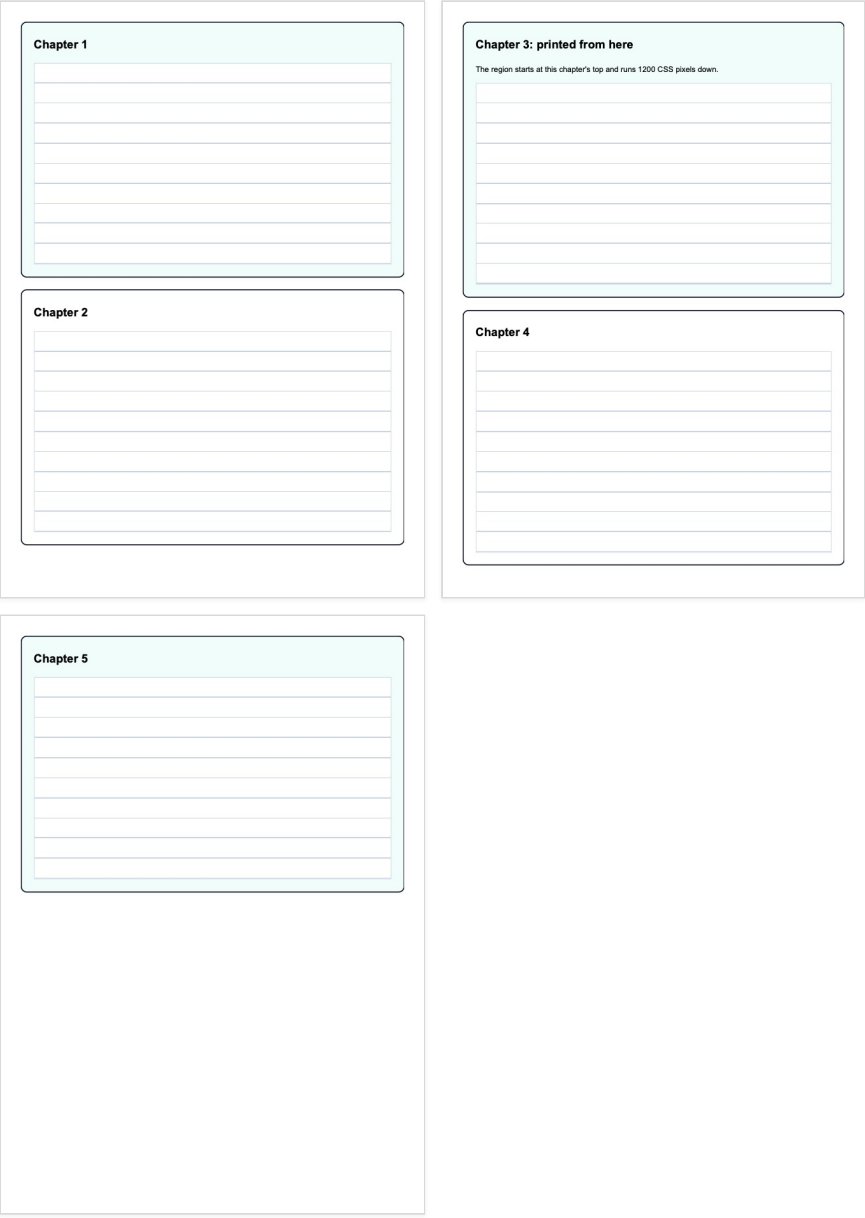
Region prints a part of the document, from a start (the scroll position, a length, an element) for a height; what lies above and below is left out, and the part is laid out on pages as a whole document is.

Reference: `HtmlPdfOptions.Region`, `HtmlPrintRegion`

▶ The page printed: `chapters.html`

Default: the whole document

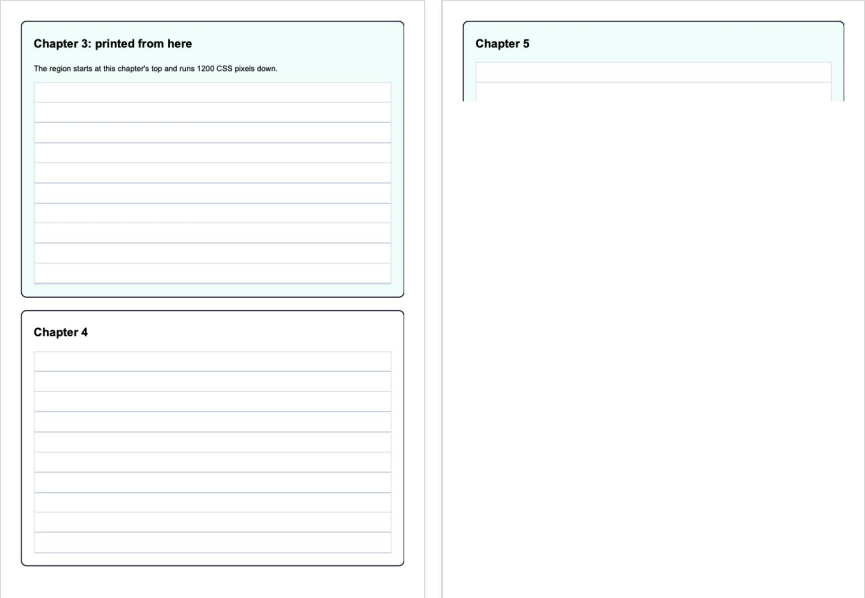
```
var result = HtmlToPdf.Render(html);
```



Pages 3
File 46,933 bytes (45.8 KB) · the PDF

1200 px from #chapter-3

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Region =
HtmlPrintRegion.Of("#chapter-3", "1200px") });
```



Pages 2
File 56,560 bytes (55.2 KB) · the PDF

Chrome's experimental features

Off by default, as stable Chrome prints; on, the features Chrome 156 has behind its "Experimental Web Platform features" flag that Weblitz has built, grid lanes among them.

Reference: `HtmlPdfOptions.ExperimentalWebFeatures`

▶ The page printed: `grid-lanes.html`

Default: as stable Chrome

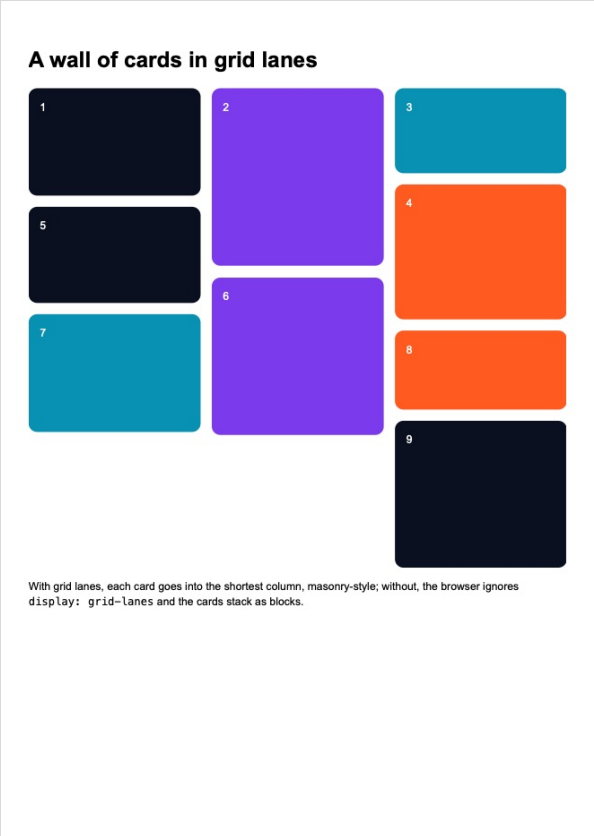
```
var result = HtmlToPdf.Render(html);
```



Pages 2
File 55,360 bytes (54.1 KB) · the PDF

ExperimentalWebFeatures = true

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { ExperimentalWebFeatures = true });
```



Pages 1
File 55,515 bytes (54.2 KB) · the PDF

Printing documentation without its menus

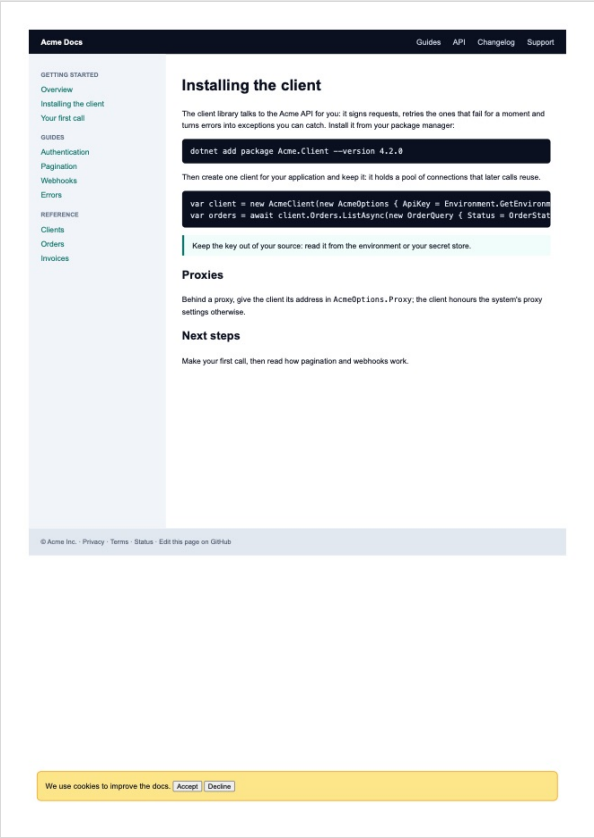
A documentation page without print styles of its own prints its top bar, sidebar, footer and cookie banner. Hide leaves elements out by CSS selector (each a display: none !important rule); Styles adds CSS of your own, here a print style that lets the text take the page's width and wraps long code lines. Both are added as Puppeteer's page.addStyleTag adds CSS before page.pdf(): once the page's scripts have settled, before beforeprint.

Reference: `HtmlPdfOptions.Hide`, `HtmlPdfOptions.Styles`

▶ The page printed: `docs-page.html`

Default: the page as it is

```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	67,355 bytes (65.8 KB) · the PDF
Notes	`Info contentShrunk`: the document is 1052 px wide on a 720 px page: printed at 68.44% (laid out 1052 px wide). `Info wideElement`: reaches 1052 px, past the 720 px page.

Hide and Styles

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    Hide = ["nav", ".sidebar", "footer", "#cookie-banner"],
    Styles = ["@media print { main { max-width: none } pre { white-space: pre-wrap }
}"],
});
```



Installing the client

The client library talks to the Acme API for you: it signs requests, retries the ones that fail for a moment and turns errors into exceptions you can catch. Install it from your package manager:

```
dotnet add package Acme.Client --version 4.2.0
```

Then create one client for your application and keep it: it holds a pool of connections that later calls reuse.

```
var client = new AcmeClient(new AcmeOptions { ApiKey =
Environment.GetEnvironmentVariable("ACME_KEY") });
var orders = await client.Orders.ListAsync(new OrderQuery { Status =
OrderStatus.Open, PageSize = 50 });
```

> Keep the key out of your source: read it from the environment or your secret store.

Proxies

Behind a proxy, give the client its address in `AcmeOptions.Proxy`; the client honours the system's proxy settings otherwise.

Next steps

Make your first call, then read how pagination and webhooks work.

Pages 1

File 58,225 bytes (56.9 KB) · the PDF

Without the consent boxes

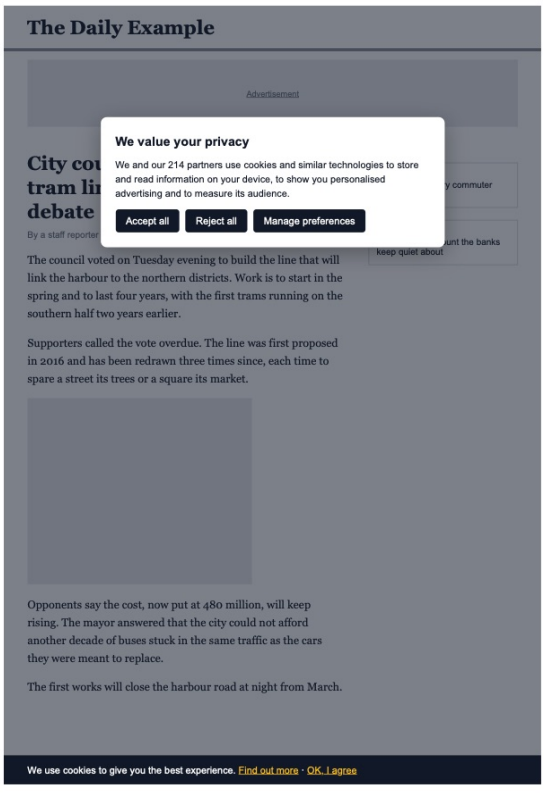
A page from the web often comes with a consent dialog over it and a cookie bar under it. Consent = Hide leaves them out of the print without clicking anything: the page and its consent platform run as they come, no answer is given in the reader's name. It hides the consent platforms' own boxes, the boxes the EasyList Cookie List names, and a box of the page's own (fixed to the window, or a dialog, that speaks of cookies, consent or privacy and holds a button to answer with), with the veil behind it and the lock it puts on the page's scrolling. A consentHidden note says what was hidden.

Reference: `HtmlPdfOptions.Consent`

▶ The page printed: `news-page.html`

Default: the page as it comes

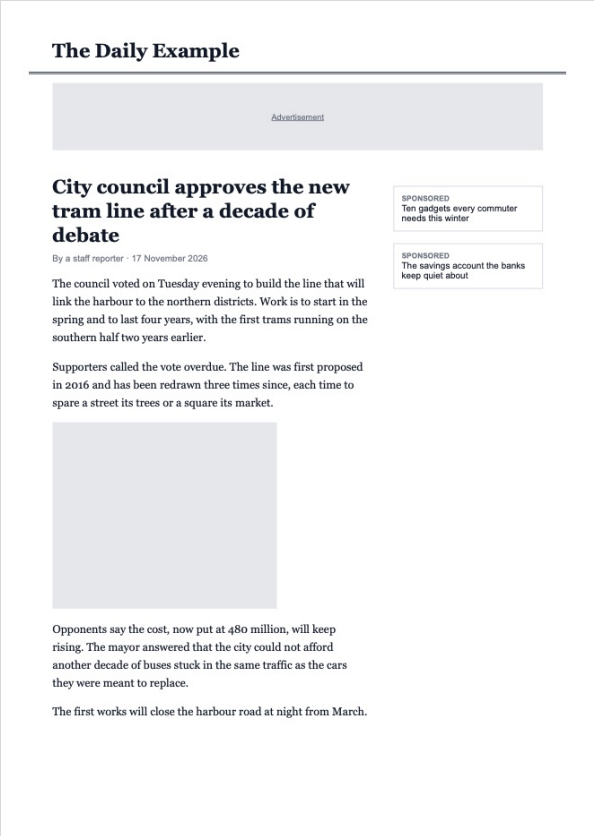
```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	100,219 bytes (97.9 KB) · the PDF
Notes	`Info resourceRefused`: Script https://www.google-analytics.com/analytics.js: no resource loader was given (only data: URLs load). `Info resourceRefused`: Frame https://securepubads.g.doubleclick.net/gampad/adx?slot=inline: no resource loader was given (only data: URLs load). `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

Consent = Hide

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Consent = HtmlConsent.Hide
});
```



Pages	1
File	70,593 bytes (68.9 KB) · the PDF
Notes	`Info resourceRefused`: Script https://www.google-analytics.com/analytics.js: no resource loader was given (only data: URLs load). `Info resourceRefused`: Frame https://securepubads.g.doubleclick.net/gampad/adx?slot=inline: no resource loader was given (only data: URLs load). `Info consentHidden`: Consent boxes are not printed (Consent = Hide): the page's own div#consent, the page's own div#cookie-bar. `Info scriptsRan`: The page's scripts ran (33 ms of virtual time, 6 tasks; 0 canvases drawn).

Without advertisements and trackers

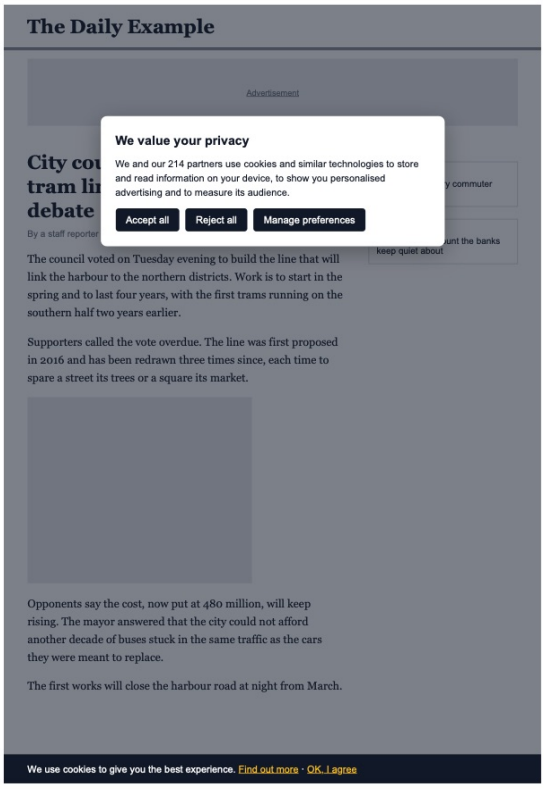
Ads = Block prints a page as an ad blocker shows it, by the filter lists EasyList and EasyPrivacy (the work of the EasyList authors, a snapshot of which the package carries): a request the lists block is not made, an element they hide is not printed, and a frame whose request was refused leaves no empty box. AdFilters adds rules of your own in the same syntax, here for the page's sponsored boxes. An adsBlocked note counts what was left out. Here with Consent = Hide as well.

Reference: `HtmlPdfOptions.Ads`, `HtmlPdfOptions.AdFilters`, `HtmlToPdf.UpdateFilterListsAsync`

▶ The page printed: `news-page.html`

Default: the page as it comes

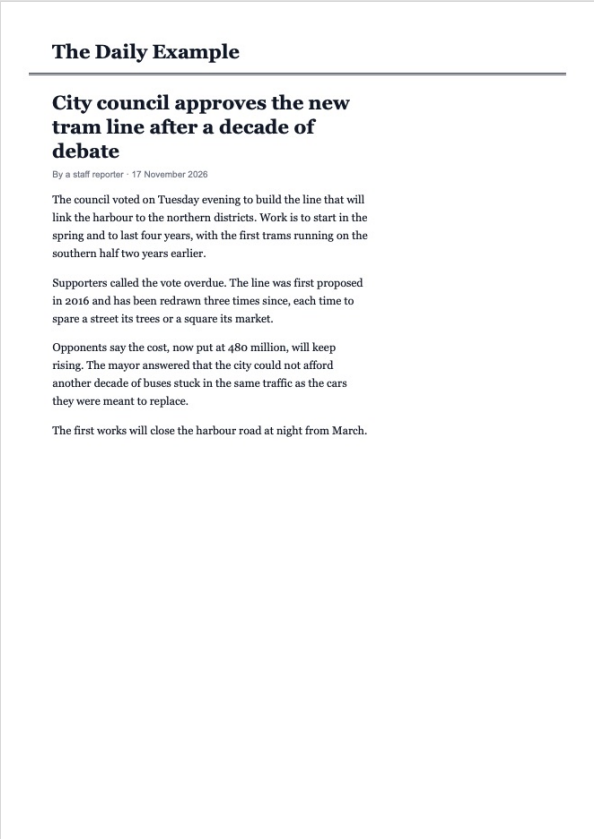
```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	100,219 bytes (97.9 KB) · the PDF
Notes	`Info resourceRefused`: Script https://www.google-analytics.com/analytics.js: no resource loader was given (only data: URLs load). `Info resourceRefused`: Frame https://securepubads.g.doubleclick.net/gampad/adx?slot=inline: no resource loader was given (only data: URLs load). `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

Ads = Block, with a rule of our own

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    Ads = HtmlAds.Block,
    AdFilters = ["##.sponsored"],
    Consent = HtmlConsent.Hide,
});
```



The Daily Example

City council approves the new tram line after a decade of debate

By a staff reporter · 17 November 2026

The council voted on Tuesday evening to build the line that will link the harbour to the northern districts. Work is to start in the spring and to last four years, with the first trams running on the southern half two years earlier.

Supporters called the vote overdue. The line was first proposed in 2016 and has been redrawn three times since, each time to spare a street its trees or a square its market.

Opponents say the cost, now put at 480 million, will keep rising. The mayor answered that the city could not afford another decade of buses stuck in the same traffic as the cars they were meant to replace.

The first works will close the harbour road at night from March.

Pages	1
File	52,292 bytes (51.1 KB) · the PDF
Notes	`Info consentHidden`: Consent boxes are not printed (Consent = Hide): the page's own div#consent, the page's own div#cookie-bar. `Info scriptsRan`: The page's scripts ran (33 ms of virtual time, 6 tasks; 0 canvases drawn). `Info adsBlocked`: Advertisements and trackers are left out (Ads = Block): 2 requests not made, 168 of the lists' hiding rules given to the page.

API reference

Every public type and member of the `WeblitzPdf` namespace (assembly `WeblitzPdf`), as the engine documents them: each member's signature, its default, the values it takes, and where a page of this documentation shows it rendered. Puppeteer's name for an option is given as a note, for those who print with it today.

HtmlToPdf STATIC CLASS

HTML to PDF as Chrome prints it: the page's HTML and CSS laid out, its scripts run once before printing, its pages written as a PDF. Thread-safe: documents render at once on as many threads as the caller uses.

Render STATIC METHOD

```
public static HtmlPdfResult Render(string html, HtmlPdfOptions? options = null)
```

The PDF of `html`. Relative addresses resolve against HtmlPdfOptions.BaseUrl: a folder's (file:) files are read from that folder only, a web address's through the network the options allow; without one, nothing is fetched.

Parameters	`html`: The document's HTML. `options`: How to print it; null: HtmlPdfOptions.Default.
Throws	`System.ArgumentNullException`: `html` is null. `System.ArgumentOutOfRangeException`: The options' HtmlPdfOptions.Scale is outside 0.1 to 2, or HtmlPdfOptions.ScrollStepWait outside 0 to 60 seconds. `System.ArgumentException`: The options contradict themselves: HtmlPdfOptions.PdfA with passwords, an empty HtmlPdfOptions.Format without HtmlPdfOptions.Width and HtmlPdfOptions.Height, an empty HtmlPdfOptions.ScrollTo, a HtmlPdfOptions.Region whose From is empty or whose Height is no length, a network header the fetcher writes itself (Host, Cookie...) or holding a line break, a cookie or credentials that cannot be sent (HtmlNetwork.Headers, HtmlNetwork.Cookies, HtmlNetwork.Credentials). `HtmlRenderException`: No PDF could be made at all (a stage of the engine failed); a page's own errors are notes, never exceptions.
Puppeteer	`page.setContent(html) then page.pdf()`
Rendered	Your first PDF: HTML text, Into a stream

RenderFile STATIC METHOD

```
public static HtmlPdfResult RenderFile(string path, HtmlPdfOptions? options = null)
```

The PDF of the HTML file at `path`: its CSS, pictures, fonts and scripts read from its folder (never outside it), web addresses through the network the options allow.

Throws `System.ArgumentNullException`: `path` is null.
`System.IO.IOException`: The file cannot be read (`System.IO.FileNotFoundException` when it is not there).
`System.ArgumentOutOfRangeException`: The options' HtmlPdfOptions.Scale is outside 0.1 to 2, or HtmlPdfOptions.ScrollStepWait outside 0 to 60 seconds.
`System.ArgumentException`: The options contradict themselves: HtmlPdfOptions.PdfA with passwords, an empty HtmlPdfOptions.Format without HtmlPdfOptions.Width and HtmlPdfOptions.Height, an empty HtmlPdfOptions.ScrollTo, a HtmlPdfOptions.Region whose From is empty or whose Height is no length, a network header the fetcher writes itself (Host, Cookie...) or holding a line break, a cookie or credentials that cannot be sent (HtmlNetwork.Headers, HtmlNetwork.Cookies, HtmlNetwork.Credentials).
`HtmlRenderException`: No PDF could be made at all (a stage of the engine failed); a page's own errors are notes, never exceptions.

Puppeteer `page.goto('file://...') then page.pdf()`

Rendered A file and its folder

RenderUrl STATIC METHOD

```
public static HtmlPdfResult RenderUrl(Uri url, HtmlPdfOptions? options = null)
```

The PDF of the page at `url`, fetched with its resources. The network is the options' (HtmlPdfOptions.Network); when they leave it off, the page's own host is allowed. A page the server answers with an error status (401, 403, 404...) and a page of its own is printed, as Chrome prints it: HtmlPdfResult.HttpStatus and a note with the code documentHttpError tell it apart.

Throws `System.ArgumentNullException`: `url` is null.
`System.ArgumentException`: `url` is not an absolute http or https address (a file goes to HtmlToPdf.RenderFile), or the options contradict themselves (as for HtmlToPdf.Render).
`System.ArgumentOutOfRangeException`: The options' Scale or ScrollStepWait is out of its range.
`HtmlRenderException`: The page could not be loaded (no answer, refused, or an error status with no page), or no PDF could be made at all.

Puppeteer `page.goto(url) then page.pdf()`

Rendered A web address

Render STATIC METHOD

```
public static HtmlPdfResult Render(string html, Stream output, HtmlPdfOptions?
options = null)
```

The PDF of `html` written to `output` (the result's HtmlPdfResult.Pdf is then null).

Throws `System.ArgumentNullException`: `html` or `output` is null. `System.ArgumentOutOfRangeException`: The options' HtmlPdfOptions.Scale is outside 0.1 to 2, or HtmlPdfOptions.ScrollStepWait outside 0 to 60 seconds. `System.ArgumentException`: The options contradict themselves: HtmlPdfOptions.PdfA with passwords, an empty HtmlPdfOptions.Format without HtmlPdfOptions.Width and HtmlPdfOptions.Height, an empty HtmlPdfOptions.ScrollTo, a HtmlPdfOptions.Region whose From is empty or whose Height is no length, a network header the fetcher writes itself (Host, Cookie...) or holding a line break, a cookie or credentials that cannot be sent (HtmlNetwork.Headers, HtmlNetwork.Cookies, HtmlNetwork.Credentials). `HtmlRenderException`: No PDF could be made at all (a stage of the engine failed); a page's own errors are notes, never exceptions.

Puppeteer `page.setContent(html) then page.pdf()`

Rendered Your first PDF: HTML text, Into a stream

RenderAsync STATIC METHOD

```
public static Task<HtmlPdfResult> RenderAsync(string html, HtmlPdfOptions? options =
null, CancellationToken cancellationToken = default)
```

HtmlToPdf.Render on the thread pool. A cancellation stops a render that has not started; one under way runs to its end or to its limits (HtmlScripts.Timeout). The task faults with the exceptions HtmlToPdf.Render throws.

RenderFileAsync STATIC METHOD

```
public static Task<HtmlPdfResult> RenderFileAsync(string path, HtmlPdfOptions?
options = null, CancellationToken cancellationToken = default)
```

HtmlToPdf.RenderFile on the thread pool (cancellation as HtmlToPdf.RenderAsync).

RenderUrlAsync STATIC METHOD

```
public static Task<HtmlPdfResult> RenderUrlAsync(Uri url, HtmlPdfOptions? options =
null, CancellationToken cancellationToken = default)
```

HtmlToPdf.RenderUrl on the thread pool (cancellation as HtmlToPdf.RenderAsync).

`WarmUp` STATIC METHOD

```
public static HtmlWarmUpResult WarmUp(string html, HtmlPdfOptions? options = null,
int renders = 8)
```

Optional: prepares this process for documents like `html` by rendering it `renders` times and discarding the PDFs. Never needed for a correct PDF, and it changes none: what HtmlToPdf.Render returns afterwards is what it returns without it, only sooner. Why it exists: .NET compiles a library's code while it runs, in steps, so a process's first render of a kind of document is several times slower than its later ones and the next dozen are still somewhat slower. Rendering a document of that kind a few times first moves this cost out of the renders that matter: the first real render then takes a fraction of the time, the next few a little less than they would; the speed reached after many renders is the same with and without it.When to call it: once, at start-up or while idle, before the application takes its real work, with a document like those it will print (the same template and options; other kinds of document gain less). A process that prints one document and exits gains nothing from it.What it costs: the time and CPU of those renders, once (seconds for a heavy page); it runs on the calling thread and returns when done (HtmlToPdf.WarmUpAsync runs it on the thread pool). Each warm-up render loads the document's resources and runs its scripts as a real render does: a page that reports to a server reports at each one. Nothing is written anywhere.

Parameters	`html`: A document like those the application will print. `options`: How it will be printed; null: HtmlPdfOptions.Default. `renders`: How many times to render it, 1 to 100. Eight is what our measurements used; more gains little.
Returns	How long the first and the last warm-up render took: the last is close to what a real render will take.
Throws	`System.ArgumentNullException`: `html` is null. `System.ArgumentOutOfRangeException`: `renders` is outside 1 to 100, or an option is out of its range (as for HtmlToPdf.Render). `System.ArgumentException`: The options contradict themselves (as for HtmlToPdf.Render). `HtmlRenderException`: No PDF could be made of the document at all.

WarmUpFile STATIC METHOD

```
public static HtmlWarmUpResult WarmUpFile(string path, HtmlPdfOptions? options =
null, int renders = 8)
```

HtmlToPdf.WarmUp with the HTML file at `path` (read as HtmlToPdf.RenderFile reads it: its CSS, pictures, fonts and scripts from its folder).

Throws `System.ArgumentNullException`: `path` is null.
`System.IO.IOException`: The file cannot be read.
`System.ArgumentOutOfRangeException`: `renders` is outside 1 to 100, or an option is out of its range.
`System.ArgumentException`: The options contradict themselves.
`HtmlRenderException`: No PDF could be made of the document at all.

UpdateFilterListsAsync STATIC METHOD

```
public static Task<HtmlFilterListsResult> UpdateFilterListsAsync(string? folder =
null, bool force = false, CancellationToken cancellationToken = default)
```

Fetches the filter lists as they are published today (EasyList and EasyPrivacy for HtmlPdfOptions.Ads, the EasyList Cookie List for HtmlPdfOptions.Consent) and uses them from then on instead of the snapshot the package carries, which is as old as the package. Optional: call it when your application starts, and again whenever you like; it is the only thing that fetches the lists, no render ever does, and nothing runs in the background. A list whose copy in the folder is younger than the list's own refresh interval is not fetched again (unless `force`). A list that cannot be fetched keeps its copy, or the snapshot, in use: the reason is in HtmlFilterListsResult.Problem, never an exception.

Parameters `folder`: Where the copies are kept, read again by later runs of the application; null: a folder of the user's local application data.
`force`: Fetch them whatever the age of the copies.
`cancellationToken`: Stops the fetch; the task is then cancelled.

Rendered Without advertisements and trackers

WarmUpAsync STATIC METHOD

```
public static Task<HtmlWarmUpResult> WarmUpAsync(string html, HtmlPdfOptions? options
= null, int renders = 8, CancellationToken cancellationToken = default)
```

HtmlToPdf.WarmUp on the thread pool. A cancellation stops it between two of its renders (the one under way runs to its end); the task is then cancelled. It faults with the exceptions HtmlToPdf.WarmUp throws.

HtmlPdfOptions RECORD

What HtmlToPdf prints and how: Puppeteer's `page.pdf()` options under their names (the reference tool's defaults: A4, backgrounds printed, the page's `@page` size preferred), and what a server needs beyond them. A record: `with` makes a variant.

Default STATIC PROPERTY

```
public static HtmlPdfOptions Default { get; }
```

The defaults.

Format PROPERTY

```
public string Format { get; init; }
```

The paper: Letter, Legal, Tabloid, Ledger, A0 to A6 (A4 by default).

Default	`"A4"`
Throws	`System.ArgumentException`: At render: empty while HtmlPdfOptions.Width or HtmlPdfOptions.Height is null.
Puppeteer	`page.pdf({ format })`
Rendered	Paper size and orientation

Width PROPERTY

```
public string? Width { get; init; }
```

The paper's width as a CSS length ("210mm", "8.5in"); with HtmlPdfOptions.Height, wins over HtmlPdfOptions.Format.

Default	`null`
Puppeteer	`page.pdf({ width })`
Rendered	Paper size and orientation

Height PROPERTY

```
public string? Height { get; init; }
```

The paper's height as a CSS length.

Default	`null`
Puppeteer	`page.pdf({ height })`
Rendered	Paper size and orientation

Landscape PROPERTY

```
public bool Landscape { get; init; }
```

The paper turned on its side (its width and height swapped). False by default; the page's `@page size` wins when HtmlPdfOptions.PreferCssPageSize and it gives an orientation.

Default	`false`
Puppeteer	`page.pdf({ landscape })`
Rendered	Paper size and orientation

Margin PROPERTY

```
public HtmlMargins? Margin { get; init; }
```

The page's margins; null: the page's `@page` margins, else Chrome's.

Default	`null`
Puppeteer	`page.pdf({ margin })`
Rendered	Margins

PreferCssPageSize PROPERTY

```
public bool PreferCssPageSize { get; init; }
```

The page's `@page size` wins over HtmlPdfOptions.Format (Chrome's `preferCSSPageSize`).

Default	`true`
Puppeteer	`page.pdf({ preferCSSPageSize })`
Rendered	The page's own @page size

PrintBackground PROPERTY

```
public bool PrintBackground { get; init; }
```

Background colours and pictures printed.

Default	`true`
Puppeteer	`page.pdf({ printBackground })`
Rendered	Backgrounds

`Scale` PROPERTY

```
public double Scale { get; init; }
```

The page's content scaled when printed (Chrome's `scale`): 0.1 to 2, 1 by default; the paper keeps its size.

Default	`1`
Throws	`System.ArgumentOutOfRangeException`: At render: outside 0.1 to 2.
Puppeteer	`page.pdf({ scale })`
Rendered	Scale

`PageRanges` PROPERTY

```
public string? PageRanges { get; init; }
```

The pages printed: "1-5, 8, 11-13"; null: every page.

Default	`null`
Puppeteer	`page.pdf({ pageRanges })`
Rendered	Choosing pages

`DisplayHeaderFooter` PROPERTY

```
public bool DisplayHeaderFooter { get; init; }
```

Chrome's header and footer templates printed in the margins.

Default	`false`
Puppeteer	`page.pdf({ displayHeaderFooter })`
Rendered	Header and footer templates

HeaderTemplate PROPERTY

```
public string? HeaderTemplate { get; init; }
```

The header printed in each page's top margin when HtmlPdfOptions.DisplayHeaderFooter: HTML whose elements of the classes `pageNumber`, `totalPages`, `title`, `url` and `date` get those values. It is a document of its own, as in Chrome: its styles are the template's (a template without a font-size prints at 1px, as in Chrome), no scripts run, nothing is fetched, backgrounds need `print-color-adjust: exact`. Null or empty: Chrome's default header (the date and the title).

Default	`null`
Puppeteer	`page.pdf({ headerTemplate })`
Rendered	Header and footer templates

FooterTemplate PROPERTY

```
public string? FooterTemplate { get; init; }
```

The footer printed in each page's bottom margin when HtmlPdfOptions.DisplayHeaderFooter, as HtmlPdfOptions.HeaderTemplate. Null or empty: Chrome's default footer (the address and "page/pages").

Default	`null`
Puppeteer	`page.pdf({ footerTemplate })`
Rendered	Header and footer templates

BaseUrl PROPERTY

```
public Uri? BaseUrl { get; init; }
```

What relative addresses resolve against for HtmlToPdf.Render.

Default	`null`
Rendered	The network

Scripts PROPERTY

```
public HtmlScripts Scripts { get; init; }
```

The page's scripts and its frames': on, as in Chrome.

Default `HtmlScripts.Default`
Rendered Scripts and the virtual clock

Network PROPERTY

```
public HtmlNetwork Network { get; init; }
```

The network: off by default (nothing is fetched but a file's own folder).

Default `HtmlNetwork.Off`
Rendered A web address, The network

ScrollTo PROPERTY

```
public string? ScrollTo { get; init; }
```

Scroll the window before printing, as a user would before pressing Print (with Puppeteer, a `window.scrollTo` in `page.evaluate` before `page.pdf`): `"top"`, `"bottom"`, a CSS selector (its first match scrolled into view, its top at the window's top) or a length from the document's top in px (`"1200px"`, `"1200"`). Null (the default): not scrolled. It happens once the page's scripts have settled, in screen media, before `beforeprint`: the page's scroll events, scroll positions and IntersectionObserver callbacks see it, and what scripts change is printed. Animations on the root scroll timeline print at their start, scrolled or not, as Chrome prints them. The PDF still holds the whole document from its top, as Chrome's print of a scrolled page does; HtmlPdfOptions.Region prints a part of it. Without scripts, nothing changes.

Default `null`
Throws `System.ArgumentException`: At render: empty or white space (null not to scroll).
Puppeteer `page.evaluate(() ⇒ window.scrollTo(...)) before page.pdf()`
Rendered Scrolling to a section first

ScrollThrough PROPERTY

```
public bool ScrollThrough { get; init; }
```

Scroll the window from the top to the document's end, one window height at a time, before printing: what a page loads as one scrolls (infinite lists, images and charts made when they come into view) is loaded and printed. Each step waits HtmlPdfOptions.ScrollStepWait of the page's virtual time; at most 200 steps. The window then goes to HtmlPdfOptions.ScrollTo, else back to the top.

IntersectionObserver without a scroll option (neither this nor HtmlPdfOptions.ScrollTo): every observed element is reported visible once, wherever it is, so content a page loads as it comes into view (lazy images, charts, list items made by scripts) is loaded and printed, the whole page in the PDF, which is what a PDF is for. Chrome reports only what is in the window, so with Puppeteer such a page prints its first screen unless it is scrolled first. With a scroll option, Weblitz reports real intersections with the window, as Chrome does: set HtmlPdfOptions.ScrollTo to `"top"` to print such a page as Chrome prints it, its first screen's lazy content loaded and the rest as the page left it. Either way the page is laid out and drawn as Chrome draws it.

Default `false`

Rendered Lists that load as you scroll

ScrollStepWait PROPERTY

```
public TimeSpan ScrollStepWait { get; init; }
```

The virtual time each scroll (HtmlPdfOptions.ScrollTo, each step of HtmlPdfOptions.ScrollThrough) gives the page to react: its timers, animation frames, observers and image loads. 0 to 60 seconds; 0.5 s by default.

Default `00:00:00.5000000`

Throws `System.ArgumentOutOfRangeException`: At render: outside 0 to 60 seconds.

Rendered Lists that load as you scroll

Region PROPERTY

```
public HtmlPrintRegion? Region { get; init; }
```

Print only a part of the document (null, the default: all of it): from a start (the window's scroll position, a length or an element) to an optional height; what lies above and below is left out. The part is laid out on pages as a whole document is: @page sizes and margins, page breaks, headers and footers.

Default `null`

Rendered Printing a part of the page

Animations PROPERTY

```
public HtmlPdfAnimations Animations { get; init; }
```

How the page's CSS animations and transitions (and its scripts' `Element.animate()`) are taken while its scripts run: HtmlPdfAnimations.Finished by default, each one styled at its end from its start, so that no time is spent on frames nobody prints; the printed page is the one HtmlPdfAnimations.FrameByFrame prints, Chrome's. HtmlPdfAnimations.None switches CSS animations and transitions off, as the CSS many applications inject into Chrome before printing does.

Weblitz and Chrome differ by default here, on purpose: with HtmlPdfAnimations.Finished a script that reads `getComputedStyle` or an element's box while an animation or transition runs reads its end value, where Chrome's script reads the value at that frame. The events (`transitionend`, `animationend`, `animationiteration`…) are sent at their times, and `Animation` objects (`currentTime`, `playState`, `finished`) read their real times, so a page that waits for an animation to end waits as long as in Chrome. A page whose scripts depend on mid-animation values prints as in Chrome with HtmlPdfAnimations.FrameByFrame.

Default `HtmlPdfAnimations.Finished`

Values `Finished` `FrameByFrame` `None`

Styles PROPERTY

```
public IReadOnlyList<string> Styles { get; init; }
```

CSS of the application's own added to the page before it is printed, as Puppeteer's `page.addStyleTag({ content })` before `page.pdf()` adds it: each text a `<style>` element appended to the document's `<head>` (to its root element when it has no head, as in an SVG or XML document). Its rules are author rules after the page's own style sheets, so they win where specificity and `!important` are equal; they apply in every medium, and a `@media print` block in them only to the print. The elements are added once the page's scripts have settled (after `load`, HtmlScripts.WaitFor and the scroll options), in screen media, before `beforeprint`: the page's load-time scripts do not see or remove them, its `beforeprint` handlers and `getComputedStyle` do. Without scripts (or a page that has none), they are added after the parse. The page's Content-Security-Policy does not apply to them. Empty by default; blank texts are skipped. For example `Styles = ["body { font: 11pt/1.5 Georgia, serif }", "@media print { pre { white-space: pre-wrap } }"]`.

Default	`[]`
Puppeteer	`page.addStyleTag({ content }) before page.pdf()`
Rendered	Printing documentation without its menus

Hide PROPERTY

```
public IReadOnlyList<string> Hide { get; init; }
```

Elements not printed, by CSS selector: each selector becomes its own rule `selector { display: none !important }`, added after HtmlPdfOptions.Styles as they are (once the page's scripts have settled, before `beforeprint`). Being `!important` and last, the rule wins over the page's own `display`, inline styles included, unless the page's is `!important` too: in an inline style (which wins over any style sheet's, as in Chrome) or in a rule of greater specificity. A selector list (`"nav, .sidebar"`) is one rule; a selector that is not valid CSS is left out with a warning in HtmlPdfResult.Notes (code `hideSelectorIgnored`), never an exception, and the others still apply. Empty by default. For example, a documentation page without its navigation, sidebar and cookie banner: `Hide = ["nav", ".sidebar", "#cookie-banner"]`.

Default	`[]`
Puppeteer	`page.addStyleTag({ content: 'nav { display: none !important }' }) before page.pdf()`
Rendered	Printing documentation without its menus

Consent PROPERTY

```
public HtmlConsent Consent { get; init; }
```

What becomes of the consent boxes a page shows (cookie and privacy dialogs, bars and their veils). HtmlConsent.Keep, the default, prints the page as it comes, as a browser does. HtmlConsent.Hide leaves them out of the print: nothing is clicked and no answer is given in the reader's name, the page and its consent platform run as they come and what they show of it is not printed. It hides the boxes of the consent platforms most sites use, by the platforms' own containers; a box of the page's own, found by what it is and says (a box fixed to the window, or a dialog, of a few lines that speaks of cookies, consent or privacy and holds a button to answer with); the veil such a box puts over the page; and the lock it puts on the page's scrolling, which would cut the print to its first page. A box no rule knows stays: name it in HtmlPdfOptions.Hide. A site that shows its content only once consent is given (a cookie wall) prints what it shows without it. What was hidden is said in HtmlPdfResult.Notes (code `consentHidden`).

Default	`HtmlConsent.Keep`
Values	`Keep` `Hide`
Rendered	Without the consent boxes

Ads PROPERTY

```
public HtmlAds Ads { get; init; }
```

What becomes of a page's advertisements and trackers. HtmlAds.Keep, the default, prints the page as a browser does. HtmlAds.Block leaves them out, as an ad blocker does, by the filter lists EasyList and EasyPrivacy: a request they block is not made (fewer requests, less script), an element they hide is not printed, and a frame or picture whose request was refused leaves no empty box. The lists are the snapshot the package carries (in `lists/` beside the assembly, under the lists' own licence), or the copy HtmlToPdf.UpdateFilterListsAsync last fetched; add rules of your own with HtmlPdfOptions.AdFilters. A site that refuses readers who block its advertisements prints what it shows them. What was left out is counted in HtmlPdfResult.Notes (code `adsBlocked`).

Default	`HtmlAds.Keep`
Values	`Keep` `Block`
Rendered	Without advertisements and trackers

AdFilters PROPERTY

```
public IReadOnlyList<string> AdFilters { get; init; }
```

Filter rules of your own, each text a list in the ad blockers' filter syntax (a line a rule: `||ads.example.com^` blocks requests to a host, `example.com##.sponsored` hides elements on a site, `@@` and `#@#` make exceptions). They are added to the lists' when HtmlPdfOptions.Ads is HtmlAds.Block. Empty by default.

Default	`[]`
Rendered	Without advertisements and trackers

UserAgent PROPERTY

```
public string? UserAgent { get; init; }
```

The browser the page sees and the network is told of (Puppeteer's `page.setUserAgent(userAgent, userAgentMetadata)`): navigator.userAgent, appVersion and the User-Agent header. Null: desktop Chrome 156's on the machine's OS (macOS, Windows or Linux).

Default	`null`
Puppeteer	`page.setUserAgent(userAgent)`
Rendered	The browser the page sees

`UserAgentMetadata` PROPERTY

```
public HtmlUserAgentMetadata? UserAgentMetadata { get; init; }
```

What navigator.userAgentData says and the Sec-CH-UA headers send. Null: the machine's Chrome's when HtmlPdfOptions.UserAgent is null, else none (as Chrome, given a User-Agent without metadata).

Default `null`
Puppeteer `page.setUserAgent(userAgent, userAgentMetadata)`
Rendered The browser the page sees

`Automated` PROPERTY

```
public bool Automated { get; init; }
```

Whether the page is told that a program drives its browser: `navigator.webdriver`, true by default, as in Chrome under Puppeteer or Playwright. Analytics and trackers that leave automated browsers out then stay off: fewer requests, less script to run, and a print is not counted as a visit. Some sites show an automated browser a check or less of their page: set it to false to print as an ordinary visitor's browser would (HtmlPdfOptions.UserAgent says desktop Chrome either way).

Default `true`

`Title` PROPERTY

```
public string? Title { get; init; }
```

The PDF's title (the page's `<title>` when null) and author.

Default `null`
Rendered Title, author and PDF/A

`Author` PROPERTY

```
public string? Author { get; init; }
```

The PDF's author (the page's `<meta name="author">` when null).

Default `null`
Rendered Title, author and PDF/A

PdfA PROPERTY

```
public bool PdfA { get; init; }
```

A PDF/A-2b document (archival: fonts embedded, an output intent, XMP metadata). False by default.

Default	`false`
Throws	`System.ArgumentException`: At render: with passwords in HtmlPdfOptions.Security (PDF/A forbids encryption).
Puppeteer	`page.pdf({ tagged }) has no PDF/A`
Rendered	Title, author and PDF/A

Security PROPERTY

```
public HtmlPdfSecurity? Security { get; init; }
```

Passwords and permissions (AES-256); not with HtmlPdfOptions.PdfA.

Default	`null`
Rendered	Passwords and permissions

Compression PROPERTY

```
public HtmlPdfCompression Compression { get; init; }
```

How hard the PDF's streams are deflated: HtmlPdfCompression.Fastest by default, smaller files at the cost of time with HtmlPdfCompression.Default or HtmlPdfCompression.Smallest.

Default	`HtmlPdfCompression.Fastest`
Values	`None` `Fastest` `Default` `Smallest`
Rendered	PDF compression

Locale PROPERTY

```
public string Locale { get; init; }
```

What the page's scripts and `Intl` see: its language, time zone and clock (fixed: the same PDF every time).

Default	`"en-US"`
Rendered	Locale, time zone and clock

`TimeZone` PROPERTY

```
public string TimeZone { get; init; }
```

The time zone the page's scripts see (an IANA name: "Europe/Paris", "America/New_York"; "UTC" by default): `Date`'s local time and offset and `Intl`'s formatting, as Chrome with an emulated time zone.

Default `"UTC"`
Puppeteer `page.emulateTimezone(timezoneId)`
Rendered Locale, time zone and clock

`Now` PROPERTY

```
public DateTimeOffset? Now { get; init; }
```

The clock the page's scripts start at (`Date.now()`, `new Date()`), the PDF's creation date and the Word, Excel and PowerPoint files' dates; then virtual time runs from it. Null: a fixed date (2026-01-01 00:00 UTC) for the scripts and the Office files, so the same input makes the same file.

Default `null`
Rendered Locale, time zone and clock

`UseInstalledFonts` PROPERTY

```
public bool UseInstalledFonts { get; init; }
```

The machine's fonts after the page's own `@font-face` fonts; false: the bundled faces only.

Default `true`
Rendered Installed fonts

`Limits` PROPERTY

```
public HtmlPdfLimits Limits { get; init; }
```

What a page may ask for: input size, pages, resources.

Default `HtmlPdfLimits.Default`
Rendered Limits

`ExperimentalWebFeatures` PROPERTY

```
public bool ExperimentalWebFeatures { get; init; }
```

Chrome's experimental web platform features that Weblitz has built, on, as Chrome 156 lays them out with its "Experimental Web Platform features" flag (default false: Weblitz prints as stable Chrome does, which ignores them). Among them: grid lanes (`display: grid-lanes` and `inline-grid-lanes`, the masonry layout, with `grid-lanes-direction`, `grid-lanes-pack` and `flow-tolerance`), their gap decorations and page and column breaks, `text-decoration-inset`, `text-decoration-skip-spaces`, `text-transform`'s full-width and full-size-kana beside a case keyword, the `flow-start` and `flow-end` alignment keywords, image symbols in `@counter-style`. Scripts see them too (`CSS.supports`, the parsed styles).

Default	`false`
Rendered	Chrome's experimental features

HtmlMargins RECORD

Page margins as CSS lengths ("18mm", "0.5in", "40px"; a bare number is px, as Puppeteer reads it).

HtmlMargins CONSTRUCTOR

```
public HtmlMargins(string? Top = null, string? Right = null, string? Bottom = null,
string? Left = null)
```

Page margins as CSS lengths ("18mm", "0.5in", "40px"; a bare number is px, as Puppeteer reads it).

Parameters
`Top`: The top margin; null: the page's `@page` margin, else Chrome's (about 1 cm).
`Right`: The right margin; null as `Top`.
`Bottom`: The bottom margin; null as `Top`.
`Left`: The left margin; null as `Top`.

Top PROPERTY

```
public string? Top { get; init; }
```

The top margin; null: the page's `@page` margin, else Chrome's (about 1 cm).

Right PROPERTY

```
public string? Right { get; init; }
```

The right margin; null as `Top`.

Bottom PROPERTY

```
public string? Bottom { get; init; }
```

The bottom margin; null as `Top`.

Left PROPERTY

```
public string? Left { get; init; }
```

The left margin; null as `Top`.

`All` STATIC METHOD

```
public static HtmlMargins All(string length)
```

The same margin on the four sides.

HtmlPrintRegion RECORD

The part of the document HtmlPdfOptions.Region prints: from HtmlPrintRegion.From to HtmlPrintRegion.Height below it. What lies above the start and below the end is left out; the part is laid out on pages as the whole document is, so a tall part breaks into pages.

FromScroll STATIC PROPERTY

```
public static HtmlPrintRegion FromScroll { get; }
```

From the window's scroll position (HtmlPdfOptions.ScrollTo, or where the page's scripts scrolled it) to the end.

From PROPERTY

```
public string From { get; init; }
```

Where the part starts: `"scroll"` (the default: the window's scroll position when printing starts), `"top"`, a length from the document's top in px (`"1200px"`, `"1200"`), or a CSS selector (its first match's top). The print's layout is measured, the page's print styles applied: a selector's top is where the element is on paper.

Default `"scroll"`

Height PROPERTY

```
public string? Height { get; init; }
```

How much of the document is printed from HtmlPrintRegion.From, as a CSS length (`"1200px"`, `"20cm"`, `"8in"`, `"600pt"`; a bare number is px). Null (the default): to the document's end.

Default `null`

Throws `System.ArgumentException`: At render: not a length greater than 0.

Of STATIC METHOD

```
public static HtmlPrintRegion Of(string from, string? height = null)
```

From `from` (see HtmlPrintRegion.From), `height` tall (see HtmlPrintRegion.Height; null: to the end).

HtmlScripts RECORD

The page's scripts: run once before printing (V8), timers and animations in virtual time. Its frames' documents run theirs with them, those of another origin too (as in Chrome, their windows out of the page's reach); HtmlScripts.Off runs none, a frame's neither.

Default STATIC PROPERTY

```
public static HtmlScripts Default { get; }
```

Scripts on, with the default timeout (30 s) and wait (5 s of virtual time).

Off STATIC PROPERTY

```
public static HtmlScripts Off { get; }
```

No script runs, a frame's neither: the HTML and CSS as they are written.

Enabled PROPERTY

```
public bool Enabled { get; init; }
```

Whether the page's scripts run (true by default).

Default	`true`
Puppeteer	`page.setJavaScriptEnabled(enabled)`

Timeout PROPERTY

```
public TimeSpan Timeout { get; init; }
```

How long the page's scripts may run, in real (elapsed) time, before they are stopped: 30 s by default. The pages are then printed as the scripts left them, with a `scriptLimit` note, and the result is not HtmlPdfResult.Completed. Raise it for pages whose scripts are slow (large dashboards, heavy charts); `System.Threading.Timeout.InfiniteTimeSpan` sets no limit. Virtual time (timers, animations) is HtmlScripts.WaitFor, not this. var result = HtmlToPdf.RenderUrl(url, new HtmlPdfOptions { Scripts = new HtmlScripts { Timeout = TimeSpan.FromMinutes(2) }, });

Default	`TimeSpan.FromSeconds(30)`
Puppeteer	`page.pdf({ timeout }), 30 s by default too`

WaitFor PROPERTY

```
public TimeSpan WaitFor { get; init; }
```

The virtual time timers and animations run for before printing.

Default `TimeSpan.FromSeconds(5)`

HtmlNetwork RECORD

What the render may fetch from the network: off, some hosts, or any public host (never the machine's own addresses unless named).

Off STATIC PROPERTY

```
public static HtmlNetwork Off { get; }
```

Nothing is fetched from the network (the default): a file's own folder is still read.

Any STATIC PROPERTY

```
public static HtmlNetwork Any { get; }
```

Any public host (never the machine's own or private addresses unless named in HtmlNetwork.AllowedHosts).

AllowedHosts PROPERTY

```
public IReadOnlyList<string> AllowedHosts { get; init; }
```

The hosts the render may fetch from: names ("example.com"), wildcards ("*.example.com"), "*" for any public host. Empty: the network is off.

Default `[]`

ResourceTimeout PROPERTY

```
public TimeSpan ResourceTimeout { get; init; }
```

The time one resource may take (15 s by default); what has not arrived is left out, noted.

Default `TimeSpan.FromSeconds(15)`

TotalTimeout PROPERTY

```
public TimeSpan TotalTimeout { get; init; }
```

The time the whole document's resources may take together (60 s by default).

Default `TimeSpan.FromSeconds(60)`

CacheDirectory PROPERTY

```
public string? CacheDirectory { get; init; }
```

A folder where fetched resources are kept between renders (HTTP's caching rules), or none. A render with HtmlNetwork.Headers, HtmlNetwork.Cookies, HtmlNetwork.AcceptCookies or HtmlNetwork.Credentials neither reads nor writes it: what a signed-in render receives is never handed to another render.

Default `null`

Headers PROPERTY

```
public IReadOnlyList<HtmlRequestHeader> Headers { get; init; }
```

Request headers added to the render's fetches (an `Authorization: Bearer` token, an API key), each sent only to the hosts it names, by default the document's own host: never to a CDN or another site's script unless named. A header is not sent on a redirect to a host it does not name. Host, Content-Length, Cookie (use HtmlNetwork.Cookies) and the other headers the fetcher writes itself are refused.

Default `[]`
Puppeteer `page.setExtraHTTPHeaders(headers)`
Rendered Pages behind a login

Cookies PROPERTY

```
public IReadOnlyList<HtmlCookie> Cookies { get; init; }
```

Cookies the render starts with, as a browser signed in would have them (a session cookie): sent by the cookie rules (domain, path, Secure, SameSite, expiry) with the document, its frames, style sheets, pictures, fonts, scripts and the page's own fetch() and XMLHttpRequest calls to its origin; the page's scripts read them in document.cookie unless HttpOnly. Setting any makes the render's cookies a signed-in user's (HtmlNetwork.AcceptCookies).

Default `[]`
Puppeteer `page.setCookie(...cookies)`
Rendered Pages behind a login

AcceptCookies PROPERTY

```
public bool AcceptCookies { get; init; }
```

Whether the cookies of the render are a signed-in user's: its fetches are then its own, never read from nor written to the shared cache, as with HtmlNetwork.Headers, HtmlNetwork.Cookies or HtmlNetwork.Credentials. Every render keeps the cookies servers set (Set-Cookie, redirects included) and the page's scripts set for the rest of the render and sends them back, as a browser does (a login redirect's session cookie, a bot check's cookie before the page reloads), and nothing is kept between renders: a response that sets a cookie is never shared with other renders, nor one to a request that carried the render's cookies and is private or varies by cookie. True whenever HtmlNetwork.Cookies holds any; false by default.

Default `false`
Rendered Pages behind a login

Credentials PROPERTY

```
public HtmlCredentials? Credentials { get; init; }
```

A user name and password answering a host's HTTP authentication challenge (Basic or Digest, after its 401), for the hosts they name, by default the document's own host; never sent unasked, nor to another host.

Default `null`
Puppeteer `page.authenticate(credentials)`
Rendered Pages behind a login

Allow STATIC METHOD

```
public static HtmlNetwork Allow(string[] hosts)
```

These hosts only ("example.com", "*.example.com").

WithHeader METHOD

```
public HtmlNetwork WithHeader(string name, string value, string[] hosts)
```

This network with a header added (HtmlNetwork.Headers): for `hosts`, else the document's own host.

`WithBearerToken` METHOD

```
public HtmlNetwork WithBearerToken(string token, string[] hosts)
```

This network with `Authorization: Bearer` added: for `hosts`, else the document's own host.

`WithCookie` METHOD

```
public HtmlNetwork WithCookie(HtmlCookie cookie)
```

This network with a cookie added (HtmlNetwork.Cookies).

HtmlRequestHeader RECORD

A request header a render adds to its fetches (HtmlNetwork.Headers).

HtmlRequestHeader CONSTRUCTOR

```
public HtmlRequestHeader(string Name, string Value)
```

A request header a render adds to its fetches (HtmlNetwork.Headers).

Parameters `Name`: The header's name ("Authorization", "X-Api-Key").
`Value`: Its value ("Bearer eyJ...").

Name PROPERTY

```
public string Name { get; init; }
```

The header's name ("Authorization", "X-Api-Key").

Value PROPERTY

```
public string Value { get; init; }
```

Its value ("Bearer eyJ...").

Hosts PROPERTY

```
public IReadOnlyList<string> Hosts { get; init; }
```

The hosts it is sent to ("api.example.com", "*.example.com"); empty: the document's own host only (a document read from a file has none: name the hosts).

HtmlCookie RECORD

A cookie a render starts with (HtmlNetwork.Cookies), as a browser keeps it.

HtmlCookie CONSTRUCTOR

```
public HtmlCookie(string Name, string Value)
```

A cookie a render starts with (HtmlNetwork.Cookies), as a browser keeps it.

Parameters `Name`: Its name.
`Value`: Its value.

Name PROPERTY

```
public string Name { get; init; }
```

Its name.

Value PROPERTY

```
public string Value { get; init; }
```

Its value.

Domain PROPERTY

```
public string? Domain { get; init; }
```

The domain it goes to with its subdomains ("example.com"); null: the document's own host alone.

Path PROPERTY

```
public string Path { get; init; }
```

The path it goes to with the paths under it ("/" by default).

Secure PROPERTY

```
public bool Secure { get; init; }
```

Sent over https: only (and to the machine's own addresses, as Chrome sends it there).

HttpOnly PROPERTY

```
public bool HttpOnly { get; init; }
```

Hidden from the page's scripts (document.cookie, cookieStore).

SameSite PROPERTY

```
public HtmlCookieSameSite SameSite { get; init; }
```

How it goes with requests from other sites (Lax by default).

Values `Lax` `Strict` `None`

Expires PROPERTY

```
public DateTimeOffset? Expires { get; init; }
```

When it expires; null: it lasts the render.

HtmlCookieSameSite ENUM

How a cookie goes with requests from other sites (its SameSite attribute).

Lax VALUE

```
Lax = 0
```

With the site's own requests and the top-level document (browsers' default).

Strict VALUE

```
Strict = 1
```

With the site's own requests only.

None VALUE

```
None = 2
```

With every request, other sites' too; the cookie must be HtmlCookie.Secure.

HtmlCredentials RECORD

A user name and password answering HTTP authentication (HtmlNetwork.Credentials).

HtmlCredentials CONSTRUCTOR

```
public HtmlCredentials(string UserName, string Password)
```

A user name and password answering HTTP authentication (HtmlNetwork.Credentials).

Parameters `UserName`: The user name.
`Password`: The password.

UserName PROPERTY

```
public string UserName { get; init; }
```

The user name.

Password PROPERTY

```
public string Password { get; init; }
```

The password.

Hosts PROPERTY

```
public IReadOnlyList<string> Hosts { get; init; }
```

The hosts that may be answered ("intranet.example.com", "*.example.com"); empty: the document's own host.

HtmlPdfSecurity RECORD

A PDF's passwords and what a reader may do without the owner's (AES-256).

UserPassword PROPERTY

```
public string? UserPassword { get; init; }
```

The password a reader needs to open the PDF; null: it opens without one.

Default `null`

OwnerPassword PROPERTY

```
public string? OwnerPassword { get; init; }
```

The password that lifts the permissions below; null: none set.

Default `null`

PermitPrinting PROPERTY

```
public bool PermitPrinting { get; init; }
```

Whether a reader may print without the owner's password (true by default).

Default `true`

PermitCopying PROPERTY

```
public bool PermitCopying { get; init; }
```

Whether a reader may copy text and pictures without the owner's password (true by default).

Default `true`

PermitModifying PROPERTY

```
public bool PermitModifying { get; init; }
```

Whether a reader may change the document without the owner's password (true by default).

Default `true`

HtmlPdfCompression ENUM

How hard the PDF's streams are deflated: the fastest by default, smaller at the cost of speed.

None VALUE

```
None = 0
```

Streams stored without deflating: the largest files, the fastest writing.

Fastest VALUE

```
Fastest = 1
```

zlib's fastest level (1): the default, the fastest writing of a deflated PDF.

Default VALUE

```
Default = 6
```

zlib's default level (6): smaller files, slower writing.

Smallest VALUE

```
Smallest = 9
```

zlib's best level (9): the smallest files, the slowest writing.

HtmlPdfAnimations ENUM

How a page's CSS animations and transitions are taken while its scripts run (HtmlPdfOptions.Animations).

Finished VALUE

```
Finished = 0
```

The default. While the page's scripts run, an animation or transition that ends is styled as ended from its start, and one that never ends (a spinner) is held at its first moment; at the print each is taken at the page's time, so the PDF is the one HtmlPdfAnimations.FrameByFrame gives. Events and `Animation` objects keep their real times. A script reading a style or a size in the middle of an animation reads its end.

FrameByFrame VALUE

```
FrameByFrame = 1
```

As Chrome runs them: each style a script reads while an animation runs is the animation's value at that frame. Slower on pages with many animated elements.

None VALUE

```
None = 2
```

CSS animations and transitions switched off for every element and pseudo-element, as by `*, *::before, *::after { animation: none !important; transition: none !important }` injected into Chrome (here a user agent rule: it wins over the page's own `!important` and is not among the page's style sheets). Not Chrome's default print: what an animation would show at the print (a fade not ended, an end value kept by `animation-fill-mode: forwards`, a spinner's turn) is not applied, and no animation or transition event is sent, so a page that waits for one to show its content may print without it.

HtmlConsent ENUM

What becomes of the consent boxes a page shows (HtmlPdfOptions.Consent).

Keep VALUE

```
Keep = 0
```

Printed as the page shows them, as a browser prints them.

Hide VALUE

```
Hide = 1
```

Left out of the print. Nothing is clicked or answered.

HtmlAds ENUM

What becomes of a page's advertisements and trackers (HtmlPdfOptions.Ads).

Keep VALUE

```
Keep = 0
```

Loaded and printed as the page has them, as a browser does.

Block VALUE

```
Block = 1
```

Left out by the filter lists: their requests are not made, their elements are not printed.

HtmlPdfLimits RECORD

What a page may ask for (an untrusted page is held to these).

Default STATIC PROPERTY

```
public static HtmlPdfLimits Default { get; }
```

The default limits (each below).

MaxInputBytes PROPERTY

```
public long MaxInputBytes { get; init; }
```

The largest document read (50 MB by default): past it the rest is not parsed and the result is not HtmlPdfResult.Completed.

Default `52,428,800 (50 MB)`

MaxPages PROPERTY

```
public int MaxPages { get; init; }
```

The most pages printed (10,000 by default): the document is cut there, not completed.

Default `10,000`

MaxNodes PROPERTY

```
public int MaxNodes { get; init; }
```

The most DOM nodes, the parser's and the scripts' (1,000,000 by default): past it the parse or the scripts stop, not completed.

Default `1,000,000`

MaxResourceBytes PROPERTY

```
public long MaxResourceBytes { get; init; }
```

The largest resource fetched or read (20 MB by default): a larger one is left out, noted.

Default `20,971,520 (20 MB)`

MaxTotalResourceBytes PROPERTY

```
public long MaxTotalResourceBytes { get; init; }
```

The most bytes of resources a document fetches and reads in all (200 MB by default).

Default `209,715,200 (200 MB)`

HtmlUserAgentMetadata RECORD

What navigator.userAgentData says and the Sec-CH-UA headers send (CDP's Emulation.UserAgentMetadata, Puppeteer's `userAgentMetadata`).

Puppeteer: `Protocol.Emulation.UserAgentMetadata`

Brands PROPERTY

```
public IReadOnlyList<HtmlUserAgentBrand> Brands { get; init; }
```

The brands with their major versions.

Default `[]`

FullVersionList PROPERTY

```
public IReadOnlyList<HtmlUserAgentBrand> FullVersionList { get; init; }
```

The brands with their full versions.

Default `[]`

FullVersion PROPERTY

```
public string? FullVersion { get; init; }
```

uaFullVersion (deprecated in Chrome; Chrome's own when null).

Default `null`

Platform PROPERTY

```
public string Platform { get; init; }
```

"macOS", "Windows", "Linux", "Android"...

Default `""`

PlatformVersion PROPERTY

```
public string PlatformVersion { get; init; }
```

The platform's version ("15.0.0", "10.0.0"); empty: none said.

Default `""`

Architecture PROPERTY

```
public string Architecture { get; init; }
```

"arm", "x86".

Default `""`

Model PROPERTY

```
public string Model { get; init; }
```

The device model, for a mobile device ("Pixel 9"); empty on desktops.

Default `""`

Mobile PROPERTY

```
public bool Mobile { get; init; }
```

Whether the browser says it is a mobile one (navigator.userAgentData.mobile, Sec-CH-UA-Mobile).

Default `false`

Bitness PROPERTY

```
public string Bitness { get; init; }
```

"64", "32".

Default `""`

Wow64 PROPERTY

```
public bool Wow64 { get; init; }
```

Whether a 32-bit browser runs on 64-bit Windows (Sec-CH-UA-WoW64); false elsewhere.

Default `false`

FormFactors PROPERTY

```
public IReadOnlyList<string>? FormFactors { get; init; }
```

"Desktop", "Mobile", "Tablet"...; Chrome's ["Desktop"] when null.

Default `null`

ChromeOnThisMachine STATIC PROPERTY

```
public static HtmlUserAgentMetadata ChromeOnThisMachine { get; }
```

Desktop Chrome 156's on the machine's OS, as the engine says by default.

HtmlUserAgentBrand RECORD

A brand of HtmlUserAgentMetadata: navigator.userAgentData's brands and Sec-CH-UA's.

HtmlUserAgentBrand CONSTRUCTOR

```
public HtmlUserAgentBrand(string Brand, string Version)
```

A brand of HtmlUserAgentMetadata: navigator.userAgentData's brands and Sec-CH-UA's.

Parameters `Brand`: The brand's name ("Google Chrome", "Chromium", "Not:A-Brand").
`Version`: Its version: the major one in HtmlUserAgentMetadata.Brands ("156"), the full one in HtmlUserAgentMetadata.FullVersionList ("156.0.8078.47").

Brand PROPERTY

```
public string Brand { get; init; }
```

The brand's name ("Google Chrome", "Chromium", "Not:A-Brand").

Version PROPERTY

```
public string Version { get; init; }
```

Its version: the major one in HtmlUserAgentMetadata.Brands ("156"), the full one in HtmlUserAgentMetadata.FullVersionList ("156.0.8078.47").

HtmlPdfResult CLASS

A printed document: its PDF, its pages, and what the engine noted.

Pdf PROPERTY

```
public byte[]? Pdf { get; }
```

The document; null when it was written to a stream.

PageCount PROPERTY

```
public int PageCount { get; }
```

The pages printed (after page ranges chose some).

Notes PROPERTY

```
public IReadOnlyList<HtmlPdfNote> Notes { get; }
```

What was left out or could not be drawn: a font missing, a resource refused, a script that threw, content shrunk to fit. A page's own errors are notes, never exceptions, as a browser goes on.

Rendered Notes: what could not be done

Completed PROPERTY

```
public bool Completed { get; }
```

False when a limit or the script timeout stopped the render (the pages printed so far are the PDF).

Rendered Limits

Elapsed PROPERTY

```
public TimeSpan Elapsed { get; }
```

How long the render took, from the call to the PDF written.

`HttpStatus` PROPERTY

```
public int? HttpStatus { get; }
```

The HTTP status the server answered for the document, when it was fetched by its URL (HtmlToPdf.RenderUrl): a success (200), or an error (401, 403, 404…) whose page the server sent and which was printed, as Chrome prints it (a note with the code documentHttpError says so). Null for HTML given as text or read from a file.

HtmlPdfNote RECORD

One thing the engine noted about the document.

HtmlPdfNote CONSTRUCTOR

```
public HtmlPdfNote(string Code, HtmlPdfNoteSeverity Severity, string Message, string?
Element)
```

One thing the engine noted about the document.

Parameters
`Code`: What kind of note: a stable word ("missingGlyph", "resourceFailed", "limitReached", "scriptError", "notImplemented"...) to sort or filter by.
`Severity`: How much it matters to the PDF.
`Message`: What happened, in plain words.
`Element`: Where, when the note is about one element: its path from the root element, CSS-like (the tag, "#id", the first ".class", ":nth-of-type(n)" among siblings of its name, joined by ">": "html>body>div.card:nth-of-type(2)"); null for the document as a whole.

Code PROPERTY

```
public string Code { get; init; }
```

What kind of note: a stable word ("missingGlyph", "resourceFailed", "limitReached", "scriptError", "notImplemented"...) to sort or filter by.

Severity PROPERTY

```
public HtmlPdfNoteSeverity Severity { get; init; }
```

How much it matters to the PDF.

Values `Info` `Warning` `Error`

Message PROPERTY

```
public string Message { get; init; }
```

What happened, in plain words.

`Element` PROPERTY

```
public string? Element { get; init; }
```

Where, when the note is about one element: its path from the root element, CSS-like (the tag, "#id", the first ".class", ":nth-of-type(n)" among siblings of its name, joined by ">": "html>body>div.card:nth-of-type(2)"); null for the document as a whole.

HtmlPdfNoteSeverity ENUM

How much a note matters to the PDF.

Info VALUE

```
Info = 0
```

For information: the PDF is as Chrome's would be (a setting ignored as Chrome ignores it, content shrunk to fit).

Warning VALUE

```
Warning = 1
```

Something is missing or differs (a resource not loaded, a glyph a font lacks, a limit reached).

Error VALUE

```
Error = 2
```

A part could not be made (a script that threw, a stage that failed).

HtmlFilterListsResult CLASS

What HtmlToPdf.UpdateFilterListsAsync did.

Updated PROPERTY

```
public bool Updated { get; }
```

Whether a newer copy of a list was fetched and is now in use.

Versions PROPERTY

```
public IReadOnlyList<string> Versions { get; }
```

The versions of the lists in use, each under its list's title as the list states them ("EasyList 202611172043"); a list that states no version is given by when it was last modified.

Rules PROPERTY

```
public int Rules { get; }
```

How many rules the lists in use hold.

Folder PROPERTY

```
public string Folder { get; }
```

The folder the fetched copies are kept in.

Problem PROPERTY

```
public string? Problem { get; }
```

Why a list could not be fetched, when one could not: the copy or the snapshot stays in use.

HtmlWarmUpResult CLASS

What HtmlToPdf.WarmUp did.

Renders PROPERTY

```
public int Renders { get; }
```

How many times the document was rendered.

First PROPERTY

```
public TimeSpan First { get; }
```

How long the first warm-up render took (in a new process: the slow one).

Last PROPERTY

```
public TimeSpan Last { get; }
```

How long the last warm-up render took: close to what a real render of such a document will take.

Total PROPERTY

```
public TimeSpan Total { get; }
```

How long the whole warm-up took.

HtmlToExcel STATIC CLASS

HTML to Excel: the page's data tables as a workbook, a sheet a table. The page is loaded and its scripts run as for a PDF; then each table that holds data becomes a sheet: its cells at their rows and columns with their spans, numbers, percentages, amounts of money and ISO dates as typed cells (read in the page's language), header rows frozen, fonts, colours, alignments and column widths as the page shows them. A table that only lays the page out is not a sheet (the tables inside it are); a page without tables gives one sheet of its text blocks. Thread-safe.

Render STATIC METHOD

```
public static HtmlExcelResult Render(string html, HtmlExcelOptions? options = null)
```

The workbook of `html` (relative addresses as for HtmlToPdf.Render).

Parameters
`html`: The document's HTML.
`options`: How to read it; null: HtmlExcelOptions.Default.

Throws
`System.ArgumentNullException`: `html` is null.
`System.ArgumentException`: The page options contradict themselves, or one is out of its range (as for HtmlToPdf.Render).
`HtmlRenderException`: No workbook could be made at all; a page's own errors are notes.

RenderFile STATIC METHOD

```
public static HtmlExcelResult RenderFile(string path, HtmlExcelOptions? options =
null)
```

The workbook of the HTML file at `path` (its resources read from its folder, as HtmlToPdf.RenderFile reads them).

Throws
`System.ArgumentNullException`: `path` is null.
`System.IO.IOException`: The file cannot be read.
`System.ArgumentException`: The page options contradict themselves, or one is out of its range.
`HtmlRenderException`: No workbook could be made at all.

RenderUrl STATIC METHOD

```
public static HtmlExcelResult RenderUrl(Uri url, HtmlExcelOptions? options = null)
```

The workbook of the page at `url`, fetched with its resources (the network as for HtmlToPdf.RenderUrl: the page's own host when the options leave it off).

Throws `System.ArgumentNullException`: `url` is null.
`System.ArgumentException`: `url` is not an absolute http or https address, or the page options contradict themselves.
`HtmlRenderException`: The page could not be loaded, or no workbook could be made at all.

Render STATIC METHOD

```
public static HtmlExcelResult Render(string html, Stream output, HtmlExcelOptions?
options = null)
```

The workbook of `html` written to `output` (the result's HtmlExcelResult.Workbook is then null).

Throws `System.ArgumentNullException`: `html` or `output` is null.
`System.ArgumentException`: The page options contradict themselves, or one is out of its range.
`HtmlRenderException`: No workbook could be made at all.

RenderAsync STATIC METHOD

```
public static Task<HtmlExcelResult> RenderAsync(string html, HtmlExcelOptions?
options = null, CancellationToken cancellationToken = default)
```

HtmlToExcel.Render on the thread pool (cancellation as HtmlToPdf.RenderAsync: it stops a render that has not started).

RenderFileAsync STATIC METHOD

```
public static Task<HtmlExcelResult> RenderFileAsync(string path, HtmlExcelOptions?
options = null, CancellationToken cancellationToken = default)
```

HtmlToExcel.RenderFile on the thread pool (cancellation as HtmlToExcel.RenderAsync).

RenderUrlAsync STATIC METHOD

```
public static Task<HtmlExcelResult> RenderUrlAsync(Uri url, HtmlExcelOptions? options
= null, CancellationToken cancellationToken = default)
```

HtmlToExcel.RenderUrl on the thread pool (cancellation as HtmlToExcel.RenderAsync).

HtmlExcelOptions RECORD

How HtmlToExcel reads a page.

Default STATIC PROPERTY

```
public static HtmlExcelOptions Default { get; }
```

The defaults: the page loaded as HtmlPdfOptions.Default loads it.

Page PROPERTY

```
public HtmlPdfOptions Page { get; init; }
```

How the page is loaded and laid out before its tables are read: the PDF's own options (the base address, scripts, the network, HtmlPdfOptions.Consent and HtmlPdfOptions.Ads, HtmlPdfOptions.Hide, the locale, fonts, limits...). The columns are as wide as the page lays its tables out on the options' paper. What only a PDF has is not read: page ranges, headers and footers, the title and author, PDF/A, passwords, compression.

Default `HtmlPdfOptions.Default`

HtmlExcelResult CLASS

A page's tables as a workbook, and what the engine noted.

Workbook PROPERTY

```
public byte[]? Workbook { get; }
```

The workbook, an .xlsx file's bytes; null when it was written to a stream.

Notes PROPERTY

```
public IReadOnlyList<HtmlPdfNote> Notes { get; }
```

What was left out or went wrong on the page (as HtmlPdfResult.Notes).

Completed PROPERTY

```
public bool Completed { get; }
```

False when a limit or the script timeout stopped the page's loading (the workbook holds the page as it was then).

Elapsed PROPERTY

```
public TimeSpan Elapsed { get; }
```

How long it took, from the call to the workbook written.

HttpStatus PROPERTY

```
public int? HttpStatus { get; }
```

The HTTP status the server answered for the page, when it was fetched by its URL (as HtmlPdfResult.HttpStatus); null otherwise.

HtmlToWord STATIC CLASS

HTML to Word: the page as a flowing document a person can edit. The page is loaded and its scripts run as for a PDF; then its content is written in reading order as Word's own blocks: headings in Word's heading styles (the outline and the navigation pane), paragraphs of runs with their font, size, weight, slant, colour and links, bulleted and numbered lists, tables with their spans, fills, borders and repeating header rows, pictures at the size the page shows them, preformatted text line by line. The paper and its margins are the page options'. It is a document to work on, not a picture of the page: positions, columns and backgrounds are not kept. Thread-safe.

Render STATIC METHOD

```
public static HtmlWordResult Render(string html, HtmlWordOptions? options = null)
```

The Word document of `html` (relative addresses as for HtmlToPdf.Render).

Parameters `html`: The document's HTML.
`options`: How to read it; null: HtmlWordOptions.Default.

Throws `System.ArgumentNullException`: `html` is null.
`System.ArgumentException`: The page options contradict themselves, or one is out of its range (as for HtmlToPdf.Render).
`HtmlRenderException`: No document could be made at all; a page's own errors are notes.

RenderFile STATIC METHOD

```
public static HtmlWordResult RenderFile(string path, HtmlWordOptions? options = null)
```

The Word document of the HTML file at `path` (its resources read from its folder, as HtmlToPdf.RenderFile reads them).

Throws `System.ArgumentNullException`: `path` is null.
`System.IO.IOException`: The file cannot be read.
`System.ArgumentException`: The page options contradict themselves, or one is out of its range.
`HtmlRenderException`: No document could be made at all.

`RenderUrl` STATIC METHOD

```
public static HtmlWordResult RenderUrl(Uri url, HtmlWordOptions? options = null)
```

The Word document of the page at `url`, fetched with its resources (the network as for HtmlToPdf.RenderUrl: the page's own host when the options leave it off).

Throws `System.ArgumentNullException`: `url` is null.
`System.ArgumentException`: `url` is not an absolute http or https address, or the page options contradict themselves.
`HtmlRenderException`: The page could not be loaded, or no document could be made at all.

`Render` STATIC METHOD

```
public static HtmlWordResult Render(string html, Stream output, HtmlWordOptions?
options = null)
```

The Word document of `html` written to `output` (the result's HtmlWordResult.Document is then null).

Throws `System.ArgumentNullException`: `html` or `output` is null.
`System.ArgumentException`: The page options contradict themselves, or one is out of its range.
`HtmlRenderException`: No document could be made at all.

`RenderAsync` STATIC METHOD

```
public static Task<HtmlWordResult> RenderAsync(string html, HtmlWordOptions? options
= null, CancellationToken cancellationToken = default)
```

HtmlToWord.Render on the thread pool (cancellation as HtmlToPdf.RenderAsync: it stops a render that has not started).

`RenderFileAsync` STATIC METHOD

```
public static Task<HtmlWordResult> RenderFileAsync(string path, HtmlWordOptions?
options = null, CancellationToken cancellationToken = default)
```

HtmlToWord.RenderFile on the thread pool (cancellation as HtmlToWord.RenderAsync).

RenderUrlAsync STATIC METHOD

```
public static Task<HtmlWordResult> RenderUrlAsync(Uri url, HtmlWordOptions? options =
null, CancellationToken cancellationToken = default)
```

HtmlToWord.RenderUrl on the thread pool (cancellation as HtmlToWord.RenderAsync).

HtmlWordOptions RECORD

How HtmlToWord reads a page.

Default STATIC PROPERTY

```
public static HtmlWordOptions Default { get; }
```

The defaults: the page loaded as HtmlPdfOptions.Default loads it, on A4.

Page PROPERTY

```
public HtmlPdfOptions Page { get; init; }
```

How the page is loaded and laid out before it is written: the PDF's own options (the base address, scripts, the network, HtmlPdfOptions.Consent and HtmlPdfOptions.Ads, HtmlPdfOptions.Hide, the locale, fonts, limits...). The document's paper and margins are theirs (HtmlPdfOptions.Format, HtmlPdfOptions.Landscape, HtmlPdfOptions.Margin, or the page's own @page). What only a PDF has is not read: page ranges, headers and footers, the title and author (the document's title is the page's), PDF/A, passwords, compression.

Default `HtmlPdfOptions.Default`

HtmlWordResult CLASS

A page as a Word document, and what the engine noted.

Document PROPERTY

```
public byte[]? Document { get; }
```

The document, a .docx file's bytes; null when it was written to a stream.

Notes PROPERTY

```
public IReadOnlyList<HtmlPdfNote> Notes { get; }
```

What was left out or went wrong on the page (as HtmlPdfResult.Notes).

Completed PROPERTY

```
public bool Completed { get; }
```

False when a limit or the script timeout stopped the page's loading (the document holds the page as it was then).

Elapsed PROPERTY

```
public TimeSpan Elapsed { get; }
```

How long it took, from the call to the document written.

HttpStatus PROPERTY

```
public int? HttpStatus { get; }
```

The HTTP status the server answered for the page, when it was fetched by its URL (as HtmlPdfResult.HttpStatus); null otherwise.

HtmlToPowerPoint STATIC CLASS

HTML to PowerPoint: the page printed on slides, a slide a printed page, its text as text. The page is loaded, its scripts run and its pages laid out as for a PDF, on paper the size of a slide (HtmlPowerPointOptions.SlideWidth by HtmlPowerPointOptions.SlideHeight); then each page becomes a slide that looks as the page prints: its text as text boxes a person can select, search and edit, a line each at its place with its font, size, weight, slant, colour and links, over one picture of everything else on the page. Text the page turns, clips or draws half transparent stays in the picture. Thread-safe.

Render STATIC METHOD

```
public static HtmlPowerPointResult Render(string html, HtmlPowerPointOptions? options
= null)
```

The deck of `html` (relative addresses as for HtmlToPdf.Render).

Parameters `html`: The document's HTML.
`options`: How to print it; null: HtmlPowerPointOptions.Default.

Throws `System.ArgumentNullException`: `html` is null.
`System.ArgumentException`: A slide size is no length, the page options contradict themselves, or one is out of its range (as for HtmlToPdf.Render).
`HtmlRenderException`: No deck could be made at all; a page's own errors are notes.

RenderFile STATIC METHOD

```
public static HtmlPowerPointResult RenderFile(string path, HtmlPowerPointOptions?
options = null)
```

The deck of the HTML file at `path` (its resources read from its folder, as HtmlToPdf.RenderFile reads them).

Throws `System.ArgumentNullException`: `path` is null.
`System.IO.IOException`: The file cannot be read.
`System.ArgumentException`: A slide size is no length, the page options contradict themselves, or one is out of its range.
`HtmlRenderException`: No deck could be made at all.

RenderUrl STATIC METHOD

```
public static HtmlPowerPointResult RenderUrl(Uri url, HtmlPowerPointOptions? options
= null)
```

The deck of the page at `url`, fetched with its resources (the network as for HtmlToPdf.RenderUrl: the page's own host when the options leave it off).

Throws `System.ArgumentNullException`: `url` is null.
`System.ArgumentException`: `url` is not an absolute http or https address, a slide size is no length, or the page options contradict themselves.
`HtmlRenderException`: The page could not be loaded, or no deck could be made at all.

Render STATIC METHOD

```
public static HtmlPowerPointResult Render(string html, Stream output,
HtmlPowerPointOptions? options = null)
```

The deck of `html` written to `output` (the result's HtmlPowerPointResult.Presentation is then null).

Throws `System.ArgumentNullException`: `html` or `output` is null.
`System.ArgumentException`: A slide size is no length, the page options contradict themselves, or one is out of its range.
`HtmlRenderException`: No deck could be made at all.

RenderAsync STATIC METHOD

```
public static Task<HtmlPowerPointResult> RenderAsync(string html,
HtmlPowerPointOptions? options = null, CancellationToken cancellationToken = default)
```

HtmlToPowerPoint.Render on the thread pool (cancellation as HtmlToPdf.RenderAsync: it stops a render that has not started).

RenderFileAsync STATIC METHOD

```
public static Task<HtmlPowerPointResult> RenderFileAsync(string path,
HtmlPowerPointOptions? options = null, CancellationToken cancellationToken = default)
```

HtmlToPowerPoint.RenderFile on the thread pool (cancellation as HtmlToPowerPoint.RenderAsync).

RenderUrlAsync STATIC METHOD

```
public static Task<HtmlPowerPointResult> RenderUrlAsync(Uri url,
HtmlPowerPointOptions? options = null, CancellationToken cancellationToken = default)
```

HtmlToPowerPoint.RenderUrl on the thread pool (cancellation as HtmlToPowerPoint.RenderAsync).

HtmlPowerPointOptions RECORD

How HtmlToPowerPoint prints a page on slides.

Default STATIC PROPERTY

```
public static HtmlPowerPointOptions Default { get; }
```

The defaults: slides of PowerPoint's own 16:9 size, the page loaded as HtmlPdfOptions.Default loads it.

Page PROPERTY

```
public HtmlPdfOptions Page { get; init; }
```

How the page is loaded, laid out and printed: the PDF's own options (the base address, scripts, the network, HtmlPdfOptions.Consent and HtmlPdfOptions.Ads, HtmlPdfOptions.Margin, HtmlPdfOptions.Scale, HtmlPdfOptions.PageRanges, HtmlPdfOptions.PrintBackground, the locale, fonts, limits...). The paper is the slide: their Format, Width, Height and Landscape are not read (HtmlPowerPointOptions.SlideWidth and HtmlPowerPointOptions.SlideHeight are), but a page's own @page size is, where HtmlPdfOptions.PreferCssPageSize is on. Nor is what only a PDF has: the title and author (the deck's title is the page's), PDF/A, passwords, compression.

Default `HtmlPdfOptions.Default`

SlideWidth PROPERTY

```
public string SlideWidth { get; init; }
```

The slides' width, a CSS length (in, cm, mm, px, pt). The default, 1280px, is the 13.333 inches of PowerPoint's 16:9 slides; 10in with a HtmlPowerPointOptions.SlideHeight of 7.5in gives 4:3 slides.

Default `"1280px"`

`SlideHeight` PROPERTY

```
public string SlideHeight { get; init; }
```

The slides' height, a CSS length. The default, 720px, is the 7.5 inches of PowerPoint's slides.

Default `"720px"`

HtmlPowerPointResult CLASS

A page printed on slides, and what the engine noted.

Presentation PROPERTY

```
public byte[]? Presentation { get; }
```

The deck, a .pptx file's bytes; null when it was written to a stream.

SlideCount PROPERTY

```
public int SlideCount { get; }
```

The slides: the pages printed (after page ranges chose some).

Notes PROPERTY

```
public IReadOnlyList<HtmlPdfNote> Notes { get; }
```

What was left out or could not be drawn (as HtmlPdfResult.Notes).

Completed PROPERTY

```
public bool Completed { get; }
```

False when a limit or the script timeout stopped the render (the slides are the pages printed so far).

Elapsed PROPERTY

```
public TimeSpan Elapsed { get; }
```

How long it took, from the call to the deck written.

`HttpStatus` PROPERTY

```
public int? HttpStatus { get; }
```

The HTTP status the server answered for the page, when it was fetched by its URL (as HtmlPdfResult.HttpStatus); null otherwise.

HtmlToImage STATIC CLASS

HTML to pictures: a PNG, JPEG, WebP or TIFF of the page. The page is loaded and its scripts run as for a PDF; then, as HtmlImageOptions.Capture says, it is drawn as a screen shows it, laid out in a window of HtmlImageOptions.ViewportWidth by HtmlImageOptions.ViewportHeight CSS px in screen media (the whole document in one image, or the window alone), or printed as the PDF is and drawn a page an image. Thread-safe.

Render STATIC METHOD

```
public static HtmlImageResult Render(string html, HtmlImageOptions? options = null)
```

The image of `html` (relative addresses as for HtmlToPdf.Render).

Parameters
`html`: The document's HTML.
`options`: What to draw and how; null: HtmlImageOptions.Default (the whole page as a PNG).

Throws
`System.ArgumentNullException`: `html` is null.
`System.ArgumentException`: An image option is out of its range, or the page options contradict themselves (as for HtmlToPdf.Render).
`HtmlRenderException`: No image could be made at all; a page's own errors are notes.

RenderFile STATIC METHOD

```
public static HtmlImageResult RenderFile(string path, HtmlImageOptions? options =
null)
```

The image of the HTML file at `path` (its resources read from its folder, as HtmlToPdf.RenderFile reads them).

Throws
`System.ArgumentNullException`: `path` is null.
`System.IO.IOException`: The file cannot be read.
`System.ArgumentException`: An image option is out of its range, or the page options contradict themselves.
`HtmlRenderException`: No image could be made at all.

RenderUrl STATIC METHOD

```
public static HtmlImageResult RenderUrl(Uri url, HtmlImageOptions? options = null)
```

The image of the page at `url`, fetched with its resources (the network as for HtmlToPdf.RenderUrl: the page's own host when the options leave it off).

Throws `System.ArgumentNullException`: `url` is null.
`System.ArgumentException`: `url` is not an absolute http or https address, an image option is out of its range, or the page options contradict themselves.
`HtmlRenderException`: The page could not be loaded, or no image could be made at all.

RenderAsync STATIC METHOD

```
public static Task<HtmlImageResult> RenderAsync(string html, HtmlImageOptions?
options = null, CancellationToken cancellationToken = default)
```

HtmlToImage.Render on the thread pool (cancellation as HtmlToPdf.RenderAsync: it stops a render that has not started).

RenderFileAsync STATIC METHOD

```
public static Task<HtmlImageResult> RenderFileAsync(string path, HtmlImageOptions?
options = null, CancellationToken cancellationToken = default)
```

HtmlToImage.RenderFile on the thread pool (cancellation as HtmlToImage.RenderAsync).

RenderUrlAsync STATIC METHOD

```
public static Task<HtmlImageResult> RenderUrlAsync(Uri url, HtmlImageOptions? options
= null, CancellationToken cancellationToken = default)
```

HtmlToImage.RenderUrl on the thread pool (cancellation as HtmlToImage.RenderAsync).

HtmlImageOptions RECORD

How HtmlToImage draws a page.

Default STATIC PROPERTY

```
public static HtmlImageOptions Default { get; }
```

The defaults: the whole page as a PNG, laid out in an 800 by 600 window (Puppeteer's), a pixel a CSS px.

Capture PROPERTY

```
public HtmlImageCapture Capture { get; init; }
```

What is drawn: the whole page (the default), the window, or each printed page.

Default `HtmlImageCapture.FullPage`
Values `FullPage` `Viewport` `Pages`

Type PROPERTY

```
public HtmlImageType Type { get; init; }
```

The file format (Puppeteer's `type`): PNG (the default), JPEG, WebP, or TIFF.

Default `HtmlImageType.Png`
Values `Png` `Jpeg` `WebP` `Tiff`

Quality PROPERTY

```
public int? Quality { get; init; }
```

JPEG's and WebP's quality, 0 to 100 (Puppeteer's `quality`); null: 92.

Default `null`
Throws `System.ArgumentException`: At render: outside 0 to 100, or given for PNG or TIFF (lossless).

ViewportWidth PROPERTY

```
public int ViewportWidth { get; init; }
```

The window's width in CSS px, for HtmlImageCapture.FullPage and HtmlImageCapture.Viewport: what the page is laid out at (its media queries, viewport units, scripts' `innerWidth`). 800 by default, as Puppeteer's; 1 to 16,384.

Default `800`

ViewportHeight PROPERTY

```
public int ViewportHeight { get; init; }
```

The window's height in CSS px (as HtmlImageOptions.ViewportWidth): a full page image is never shorter. 600 by default; 1 to 16,384.

Default `600`

DeviceScaleFactor PROPERTY

```
public double DeviceScaleFactor { get; init; }
```

Image pixels per CSS px (Puppeteer's viewport `deviceScaleFactor`): 1 (the default) draws at 96 dpi, 2 at 192 for a sharp image on a high density screen, 3.125 at 300 for print; 0.1 to 10. It sharpens the image only: the page is laid out as at 1, and its scripts and media queries see a device pixel ratio of 1. An image past 16,384 pixels a side or 100 million in all is cut (a full page) or not drawn (a printed page), noted.

Default `1`

Throws `System.ArgumentException`: At render: outside 0.1 to 10.

Page PROPERTY

```
public HtmlPdfOptions Page { get; init; }
```

How the page is loaded, and for HtmlImageCapture.Pages printed: the PDF's own options (the base address, scripts, the network, HtmlPdfOptions.Consent and HtmlPdfOptions.Ads, HtmlPdfOptions.Hide, styles, the locale, the time zone, fonts, limits…). A screen capture takes neither the paper, the margins, the scale, the page ranges, the headers and footers nor the region (a window has none) and always draws backgrounds. Nor is what only a PDF has read: the title and author, PDF/A, passwords, compression.

Default `HtmlPdfOptions.Default`

HtmlImageType ENUM

The image's file format.

Png VALUE

```
Png = 0
```

PNG, lossless; its resolution written in it.

Jpeg VALUE

```
Jpeg = 1
```

JPEG at HtmlImageOptions.Quality.

WebP VALUE

```
WebP = 2
```

WebP at HtmlImageOptions.Quality.

Tiff VALUE

```
Tiff = 3
```

TIFF, lossless: one file holding every image (every printed page for HtmlImageCapture.Pages).

HtmlImageCapture ENUM

What HtmlToImage draws of a page.

FullPage VALUE

```
FullPage = 0
```

The whole document as a screen shows it, in one image: laid out in the window, the image as wide and as tall as the document scrolls (at least the window), as Puppeteer's `fullPage: true` captures it.

Viewport VALUE

```
Viewport = 1
```

The window alone (HtmlImageOptions.ViewportWidth by HtmlImageOptions.ViewportHeight), as Puppeteer's screenshot without `fullPage`.

Pages VALUE

```
Pages = 2
```

The page printed as HtmlToPdf prints it (paper, margins, print media, page ranges, headers and footers from HtmlImageOptions.Page), an image a page.

HtmlImageResult CLASS

A page drawn by HtmlToImage.

Images PROPERTY

```
public IReadOnlyList<byte[]> Images { get; }
```

The images, each a file's bytes in HtmlImageOptions.Type: one for a screen capture, one a printed page for HtmlImageCapture.Pages (after page ranges chose some), and one TIFF file holding them all for HtmlImageType.Tiff.

PageCount PROPERTY

```
public int PageCount { get; }
```

The pages drawn: 1 for a screen capture, the pages printed (after page ranges chose some) for HtmlImageCapture.Pages.

Notes PROPERTY

```
public IReadOnlyList<HtmlPdfNote> Notes { get; }
```

What was left out or could not be drawn (as HtmlPdfResult.Notes).

Completed PROPERTY

```
public bool Completed { get; }
```

False when a limit or the script timeout stopped the render (the images are of what was drawn so far).

Elapsed PROPERTY

```
public TimeSpan Elapsed { get; }
```

How long it took, from the call to the images written.

`HttpStatus` PROPERTY

```
public int? HttpStatus { get; }
```

The HTTP status the server answered for the page, when it was fetched by its URL (as HtmlPdfResult.HttpStatus); null otherwise.

HtmlRenderException CLASS

No PDF could be made at all (the input over its limit, a stage that failed).

HtmlRenderException CONSTRUCTOR

```
public HtmlRenderException(string message)
```

The exception with what went wrong.

Parameters `message`: Why no PDF could be made, with the engine's error notes.

License STATIC CLASS

Your SSRS2 licence key, which covers Weblitz: the same as `Ssrs2.License`, under Weblitz's name. Without a key, every page carries a trial watermark.

IsActive STATIC PROPERTY

```
public static bool IsActive { get; }
```

Whether a valid licence is active.

LicensedTo STATIC PROPERTY

```
public static string? LicensedTo { get; }
```

The licensed company, or null when not licensed.

ExpiryDate STATIC PROPERTY

```
public static DateTime? ExpiryDate { get; }
```

The licence's expiry date, or null when not licensed.

UpdatesUntil STATIC PROPERTY

```
public static DateTime? UpdatesUntil { get; }
```

The last release date a licence covers for good, or null (not licensed, or a key that simply expires).

Tier STATIC PROPERTY

```
public static string? Tier { get; }
```

The licence's tier, or null when not licensed.

`Features` STATIC PROPERTY

```
public static IReadOnlyList<string> Features { get; }
```

The features the active licence names.

`Activate` STATIC METHOD

```
public static void Activate(string licenseKey)
```

Activates the licence key (an `SSRS2-...` key). Call it once at application startup.

Throws `SSRS2.Licensing.LicenseException`: The key is invalid or has expired.

`TryActivate` STATIC METHOD

```
public static bool TryActivate(string licenseKey, out string error)
```

Activates the licence key without throwing: false and the reason when it is refused.

`TryRead` STATIC METHOD

```
public static bool TryRead(string licenseKey, out LicenseInfo? info, out string
error)
```

Reads a licence key without activating it: whether it is genuine and has not expired, and what it says.

`HasFeature` STATIC METHOD

```
public static bool HasFeature(string feature)
```

Whether the active licence names `feature`.

Node.js

The npm package `weblitzpdf` is Weblitz for Node.js: HTML, a file or a web address to a PDF, an Excel, Word or PowerPoint file, or a picture, printed as Chrome prints them, with no browser to install, start or drive. The same package renders RDLC and RDL reports and Markdown. Its options are Puppeteer's `page.pdf()` options under the same names.

Install

```
npm install weblitzpdf
```

Version 2.0.0-rc.2, a prerelease of 2.0. Node.js 22 or later, on Linux with glibc (x64, Arm64), macOS (Arm64, x64) and Windows (x64, Arm64). npm installs the engine for your platform only, as an optional dependency (one of @weblitzpdf/linux-x64, @weblitzpdf/linux-arm64, @weblitzpdf/darwin-arm64, @weblitzpdf/darwin-x64, @weblitzpdf/win32-x64, @weblitzpdf/win32-arm64): it holds the engine and the .NET runtime it runs on, so there is nothing else to install and no .NET on the machine is needed.

Installing with `--omit=optional` leaves the engine out, and so does a `node_modules` copied from another platform (a Mac's into a Linux container): run `npm ci` in the image. A call then fails with the code `workerMissing`, naming the package it misses.

Alpine and other musl systems are not supported yet: use a glibc image, `node:22-bookworm-slim` for example.

On Linux the engine needs fontconfig and ICU, which slim images lack: `apt-get install -y --no-install-recommends libfontconfig1 libicu72` (the ICU package's number follows the system's release).

Your first PDFs

```
import { readFile, writeFile } from 'node:fs/promises';
import { pathToFileURL } from 'node:url';
import { renderHtml, renderUrl } from 'weblitzpdf';   // or: const {
renderHtml, renderUrl } = require('weblitzpdf');

// HTML text
const invoice = await renderHtml('<h1>Invoice 42</h1><p>...</p>', {
format: 'A4', printBackground: true });
await writeFile('invoice.pdf', invoice.file);

// A web page, after its scripts ran; path writes the file too, as
Puppeteer's
await renderUrl('https://example.com/report', {
  path: 'report.pdf',
  margin: { top: '18mm', right: '18mm', bottom: '20mm', left: '20mm'
},
});

// A file on disk: its CSS, pictures, fonts and scripts are read from
its folder
const letters = await renderHtml(await
readFile('templates/letters.html', 'utf8'), {
  baseUrl: pathToFileURL('templates/').href,
});

// What could not be drawn or loaded is said, never thrown
for (const note of invoice.notes) console.log(`${note.severity}
${note.code}: ${note.message}`);
```

A render answers `{ file, contentType, extension, pages, notes, elapsedMs, trial }`, `file` a Buffer. Beside Puppeteer's options (`format`, `width`, `height`, `landscape`, `margin`, `preferCSSPageSize`, `printBackground`, `scale`, `pageRanges`, `displayHeaderFooter`, `headerTemplate`, `footerTemplate`, `path`) the engine has its own: `baseUrl`, `output`, `scripts`, `network`, `title`, `author`, `pdfA`, `compression`, `locale`, `timeZone`, `now`, `useInstalledFonts`, `styles`, `hide`, `consent: 'hide'` and `ads: 'block'`. They are .NET's options under JavaScript's names.

Excel, Word, PowerPoint and pictures

```
const workbook = await renderUrl('https://example.com/prices', {
output: 'xlsx' });   // or 'docx', 'pptx'
const shot = await renderUrl('https://example.com', { output: 'png',
viewport: { width: 1280, height: 720, deviceScaleFactor: 2 } });
const pages = await renderHtml(html, { output: 'png', capture:
'pages', format: 'A4' });   // pages.file is a ZIP
```

`output: 'png'` (or `'jpeg'`, `'webp'`, `'tiff'`) draws the page instead of printing it: pictures of a page. The Office files are described in Excel, Word and PowerPoint.

What a page loads

A page loads nothing from the network by default but, for `renderUrl`, its own host. `network` opens more, with headers, cookies and credentials for pages behind a login; each goes only to the hosts it names (`allowedHosts: ['*']` allows any host):

```
await renderUrl('https://app.example.com/reports/42', {
  network: {
    allowedHosts: ['app.example.com', 'cdn.example.com'],
    headers: [{ name: 'Authorization', value: `Bearer ${token}`,
hosts: ['app.example.com'] }],
    cookies: [{ name: 'session', value: sessionId, domain:
'app.example.com', secure: true, httpOnly: true }],
  },
});
```

Reports and Markdown

The same package renders SSRS2's reports: `renderReport`, `checkReport`, `markdownTo` and `importReport`. `@ssrs2/ssrs2` (`npm install @ssrs2/ssrs2`) has the same functions for projects that only render reports; it runs on `weblitzpdf` at the same version.

```
import { renderReport, checkReport } from 'weblitzpdf';

const pdf = await renderReport({ path: 'Reports/Invoice.rdlc' }, {
  format: 'PDF',   // or WORDOPENXML, EXCELOPENXML, PPTX, IMAGE, PNG,
HTML5, CSV, XML (docx, xlsx...)
  datasets: { Items: [{ Description: 'Widget', Price: 104.99, Qty: 1
}] },
  parameters: { InvoiceNumber: '42' },
});

const check = await checkReport({ path: 'Reports/Invoice.rdlc' });
// its problems, without rendering it
if (!check.ok) console.log(check.notes);
```

Workers, timeouts and errors

The engine runs in worker processes, started on the first call and kept warm; idle workers do not keep Node.js running. `configure()` sets the shared pool's options before that, and `new Engine(options)` makes another pool, with the same methods and `close()`.

```
import { configure, Engine } from 'weblitzpdf';

configure({ workers: 2, timeout: 30_000 });   // milliseconds

const engine = new Engine({ workers: 4, concurrency: 2, maxQueue: 100
});
const result = await engine.renderHtml(html, { timeout: 5_000, signal:
AbortSignal.timeout(10_000) });
await engine.close();
```

A failed call throws a `WeblitzError` whose `code` says why: `badRequest`, `tooLarge`, `timeout`, `refused`, `invalidReport`, `renderFailed`, `busy`, `cancelled`, `workerCrashed`, `workerMissing`, `startFailed`, `protocolMismatch`. A worker that crashes fails only the calls it was running; the next call starts another.

On a server

By default the workers have the settings of the .NET library on your own machine: a page may open any host, read local files through a `file:` base URL and run without a time limit. A server that prints pages other people send should start its workers with SSRS2's server settings: a page then loads nothing but a URL's own host, private addresses and `file:` are refused, report code is always restricted, and a render stops at 2,000 pages or 120 seconds.

```
configure({ env: { SSRS2_WORKER_POLICY: 'service' } });

// changed as JSON over those settings: 500 pages at most, and pages
may open cdn.example.com
configure({
  env: {
    SSRS2_WORKER_POLICY: 'service',
    SSRS2_WORKER_SETTINGS: JSON.stringify({ maxPages: 500, network: {
allowedHosts: ['cdn.example.com'] } }),
  },
});
```

Fonts

The engine brings faces drawn for Arial, Helvetica, Times New Roman, Courier New, Calibri, Cambria and Segoe UI (Latin, Greek, Cyrillic and Hebrew). Arabic, Chinese, Japanese, Korean, Thai, the Indic scripts and emoji are drawn with the machine's own

fonts: macOS and Windows have them, a Linux container usually does not (`apt-get install -y fonts-noto-core fonts-noto-cjk fonts-noto-color-emoji`). The fonts guide

Licence

The SSRS2 licence covers Weblitz and SSRS2's reports. Give the key as the `SSRS2_LICENSE` environment variable or as `configure({ license: 'SSRS2-...' })`. Without one, PDF and image pages carry a trial watermark and a result's `trial` is true. Buy a licence

Python

The PyPI package `weblitz` is Weblitz for Python: HTML, a file or a web address to a PDF, an Excel, Word or PowerPoint file, or a picture, printed as Chrome prints them, with no browser to install, start or drive. The same package renders RDLC and RDL reports and Markdown. Its options are Playwright's `page.pdf()` options under their Python names.

Install

```
pip install --pre weblitz
```

Version 2.0.0rc2, a prerelease of 2.0, so pip is asked for prereleases: Python's spelling of 2.0.0-rc.2, the version of every package. Python 3.9 or later, on Linux with glibc (x64, Arm64), macOS (Arm64, x64) and Windows (x64, Arm64): a wheel each, holding the engine and the .NET runtime it runs on, so there is nothing else to install. Alpine and other musl systems are not supported yet; on a slim Linux image, add fontconfig and ICU (`libfontconfig1` and `libicu72`, the ICU package's number following the system's release).

Your first PDFs

```
import weblitz

# HTML text
result = weblitz.render_html("<h1>Invoice 42</h1><p>...</p>",
format="A4", print_background=True)
result.save("invoice.pdf")

# A web page, after its scripts ran; path writes the file too
weblitz.render_url("https://example.com/report", path="report.pdf",
                   margin={"top": "18mm", "right": "18mm", "bottom":
"20mm", "left": "20mm"})

# A file on disk: its CSS, pictures, fonts and scripts are read from
its folder
from pathlib import Path
letters =
weblitz.render_html(Path("templates/letters.html").read_text(),

base_url=Path("templates").resolve().as_uri() + "/")

# What could not be drawn or loaded is said, never raised
for note in result.notes:
    print(note.severity, note.code, note.message)
```

Every call answers a `Result` (`file`, `content_type`, `extension`, `pages`, `notes`, `trial`, and `save(path)`) or raises `WeblitzError`, whose `code` says why. The options are .NET's options under Python's names: `format`, `margin`, `landscape`, `print_background`, `page_ranges`, `header_template`, `base_url`, `output`, `scripts`, `network`, `locale`, `time_zone`, `now`…

Excel, Word, PowerPoint and pictures

```
weblitz.render_url("https://example.com/prices",
output="xlsx").save("prices.xlsx")   # or "docx", "pptx"
weblitz.render_url("https://example.com", output="png",
                   viewport={"width": 1280, "device_scale_factor": 2},
path="example.png")
pages = weblitz.render_html(html, output="png", capture="pages",
format="A4")    # pages.file is a ZIP
```

`output="png"` (or `"jpeg"`, `"webp"`, `"tiff"`) draws the page instead of printing it: pictures of a page.

What a page loads

A page loads nothing from the network by default but, for `render_url`, its own host. `network` opens more, with headers and cookies for pages behind a login, each going only to the hosts it names:

```
weblitz.render_url("https://app.example.com/reports/42", network={
    "allowed_hosts": ["app.example.com", "cdn.example.com"],
    "headers": [{"name": "Authorization", "value": f"Bearer {token}",
"hosts": ["app.example.com"]}],
})
```

Reports and Markdown

The same package renders SSRS2's reports: `render_report`, `check_report`, `markdown_to` and `import_report`. A dataset's rows are a list of dicts or a pandas or polars DataFrame. `ssrs2` (`pip install --pre ssrs2`) has the same functions for projects that only render reports; it runs on `weblitz`.

```
from pathlib import Path
import weblitz

rows = [{"Description": "Widget", "Price": 104.99, "Qty": 1}]
weblitz.render_report(Path("Invoice.rdlc"), datasets={"Items": rows},
path="invoice.pdf")
weblitz.render_report(Path("Invoice.rdlc"), format="docx", datasets=
{"Items": dataframe})

problems = weblitz.check_report(Path("Invoice.rdlc"))   # its
problems, without rendering it
```

Workers, asyncio and timeouts

Calls go to a pool of worker processes kept warm, started on first use; any thread may make them. `weblitz.configure(...)` sets its options; `weblitz.Engine` is a pool of your own and `weblitz.AsyncEngine` the same for asyncio, where cancelling the awaiting task cancels the render. Times are in milliseconds.

```
weblitz.configure(workers=2, timeout=30_000)

async with weblitz.AsyncEngine(workers=4) as engine:
    result = await engine.render_html(html, format="A4",
timeout=5_000)
```

On a server

A server that prints pages other people send should start its workers with SSRS2's server settings: a page then loads nothing but a URL's own host, private addresses and `file:` base URLs are refused, report code is always restricted, and a render stops at 2,000 pages or 120 seconds. `SSRS2_WORKER_SETTINGS` changes them, as JSON (the Node.js page shows it).

```
weblitz.configure(env={"SSRS2_WORKER_POLICY": "service"})
```

Licence

The SSRS2 licence covers Weblitz and SSRS2's reports. Give the key as `license=` or as the environment's `SSRS2_LICENSE`. Without one, PDF and image pages carry a trial watermark and a result's `trial` is true. Buy a licence

Java

The Maven package `com.weblitzpdf:weblitzpdf` is Weblitz for Java: HTML, a file or a web address to a PDF, an Excel, Word or PowerPoint file, or a picture, printed as Chrome prints them, with no browser to install, start or drive. The same package renders RDLC and RDL reports and Markdown. Its options are Playwright's `Page.PdfOptions` under the same names.

Install

The Java packages are in Weblitz's own Maven repository, `https://maven.weblitzpdf.com`, not in Maven Central. Add the repository and the dependency:

```
<repositories>
    <repository>
        <id>weblitzpdf</id>
        <url>https://maven.weblitzpdf.com</url>
    </repository>
</repositories>

<dependencies>
    <dependency>
        <groupId>com.weblitzpdf</groupId>
        <artifactId>weblitzpdf</artifactId>
        <version>2.0.0-rc.2</version>
    </dependency>
</dependencies>
```

Version 2.0.0-rc.2, a prerelease of 2.0, the version of every package. Java 17 or later, on Linux with glibc (x64, Arm64), macOS (Arm64, x64) and Windows (x64, Arm64), with no other dependency. The engine runs in a worker process that comes as a jar of its own for each platform, `com.weblitzpdf:weblitzpdf-worker:2.0.0-rc.2:<platform>` (linux-x64, linux-arm64, osx-arm64, osx-x64, win-x64, win-arm64): it holds the engine and the .NET runtime it runs on, so there is nothing else to install and no .NET on the machine is needed. Its first use copies it into the user's cache, which takes a second or two once.

Maven adds the worker of the machine it builds on by itself. A build for another machine names it, and so does Gradle:

```
<!-- Maven: an application built on a Mac for a Linux server -->
<dependency>
    <groupId>com.weblitzpdf</groupId>
    <artifactId>weblitzpdf-worker</artifactId>
    <version>2.0.0-rc.2</version>
    <classifier>linux-x64</classifier>
    <scope>runtime</scope>
</dependency>

// Gradle (Kotlin)
repositories { maven { url = uri("https://maven.weblitzpdf.com") } }
dependencies {
    implementation("com.weblitzpdf:weblitzpdf:2.0.0-rc.2")
    runtimeOnly("com.weblitzpdf:weblitzpdf-worker:2.0.0-rc.2:linux-x64")
}
```

Without the worker a call fails with the code `workerMissing`, naming the jar to add. Alpine and other musl systems are not supported yet; on a slim Linux image, add fontconfig and ICU (`libfontconfig1` and `libicu72`, the ICU package's number following the system's release).

Your first PDFs

```
import com.weblitzpdf.*;
import java.nio.file.Files;
import java.nio.file.Path;

// HTML text
Result invoice = Weblitz.renderHtml("<h1>Invoice 42</h1><p>... </p>",
        new HtmlOptions().setFormat("A4").setPrintBackground(true));
invoice.save(Path.of("invoice.pdf"));

// A web page, after its scripts ran
Weblitz.renderUrl("https://example.com/report",
        new HtmlOptions().setMargin(new
Margin().setTop("18mm").setRight("18mm").setBottom("20mm").setLeft("20
mm")))
       .save(Path.of("report.pdf"));

// A file on disk: its CSS, pictures, fonts and scripts are read from
its folder
Result letters =
Weblitz.renderHtml(Files.readString(Path.of("templates/letters.html"))
,
        new
HtmlOptions().setBaseUrl(Path.of("templates").toUri().toString()));

// What could not be drawn or loaded is said, never thrown
for (Note note : invoice.notes()) System.out.println(note.severity() +
" " + note.code() + ": " + note.message());
```

A render answers a `Result` (`file()`, `contentType()`, `extension()`, `pages()`, `notes()`, `elapsedMs()`, `trial()`, and `save(path)`) or throws a `WeblitzException`, whose `code()` says why. The options are .NET's options under Java's names: `setFormat`, `setMargin`, `setLandscape`, `setPrintBackground`, `setPageRanges`, `setHeaderTemplate`, `setBaseUrl`, `setOutput`, `setScripts`, `setNetwork`, `setLocale`, `setTimeZone`, `setNow`…

Excel, Word, PowerPoint and pictures

```
Weblitz.renderUrl("https://example.com/prices", new
HtmlOptions().setOutput("xlsx"))   // or "docx", "pptx"
       .save(Path.of("prices.xlsx"));
Weblitz.renderUrl("https://example.com", new
HtmlOptions().setOutput("png")
        .setViewport(new Viewport(1280,
720).setDeviceScaleFactor(2))).save(Path.of("example.png"));
Result pages = Weblitz.renderHtml(html, new
HtmlOptions().setOutput("png").setCapture("pages").setFormat("A4"));
// a ZIP
```

`setOutput("png")` (or `"jpeg"`, `"webp"`, `"tiff"`) draws the page instead of printing it: pictures of a page.

What a page loads

A page loads nothing from the network by default but, for `renderUrl`, its own host. `setNetwork` opens more, with headers and cookies for pages behind a login, each going only to the hosts it names:

```
Weblitz.renderUrl("https://app.example.com/reports/42", new
HtmlOptions().setNetwork(new Network()
        .setAllowedHosts("app.example.com", "cdn.example.com")
        .addHeader("Authorization", "Bearer " + token,
"app.example.com")));
```

Reports and Markdown

The same package renders SSRS2's reports: `renderReport`, `checkReport`, `markdownTo` and `importReport`. A dataset's rows are Java's own values: records, beans, maps, lists and streams. `net.ssrs2:ssrs2` (`net.ssrs2:ssrs2:2.0.0-rc.2`) has the same functions for projects that only render reports; it runs on `com.weblitzpdf:weblitzpdf`.

```
import net.ssrs2.Ssrs2;   // or Weblitz's functions of the same names

record Item(String description, BigDecimal price, int qty) {}

Ssrs2.renderReport(Path.of("Reports/Invoice.rdlc"), new
ReportOptions()
        .setFormat("PDF")               // or WORDOPENXML,
EXCELOPENXML, PPTX, IMAGE, PNG, HTML5, CSV, XML
        .setDataset("Items", List.of(new Item("Widget", new
BigDecimal("104.99"), 1)))
        .setParameter("InvoiceNumber", "42"))
     .save(Path.of("invoice.pdf"));

CheckResult check =
Ssrs2.checkReport(Path.of("Reports/Invoice.rdlc"));   // its problems,
without rendering it
```

Workers, futures and timeouts

Calls go to a pool of worker processes kept warm, started on first use; any thread may make them. `Weblitz.configure(...)` sets its options; `new Engine(...)` is a pool of your own, which `close()` ends. Every method has an `...Async` twin answering a `CompletableFuture`, and cancelling it cancels the render. Times are in milliseconds.

```
Weblitz.configure(new
EngineOptions().setWorkers(2).setTimeout(30_000));

try (Engine engine = new Engine(new EngineOptions().setWorkers(4))) {
    Result result = engine.renderHtml(html, new
HtmlOptions().setTimeout(5_000));
    CompletableFuture<Result> later = engine.renderHtmlAsync(html, new
HtmlOptions());
}
```

On a server

A server that prints pages other people send should start its workers with SSRS2's server settings: a page then loads nothing but a URL's own host, private addresses and `file:` base URLs are refused, report code is always restricted, and a render stops at

2,000 pages or 120 seconds. `SSRS2_WORKER_SETTINGS` changes them, as JSON (the Node.js page shows it).

```
Weblitz.configure(new EngineOptions().setEnv("SSRS2_WORKER_POLICY",
"service"));
```

Licence

The SSRS2 licence covers Weblitz and SSRS2's reports. Give the key as `setLicense(...)` on the engine's options or as the environment's `SSRS2_LICENSE`. Without one, PDF and image pages carry a trial watermark and a result's `trial()` is true. Buy a licence

HTTP server and Docker

For every other language, and for a service of your own: the server's Docker image puts Weblitz and SSRS2's reports behind one HTTP API. A page or an address to PDF, Excel, Word, PowerPoint or a picture; a report with its rows as JSON to any of SSRS2's formats; a report's problems; Markdown to a deck or a document; another tool's report to RDLC. The file it answers is the file the .NET packages write with the same options and data.

The image

```
docker run --rm -p 8080:8080 -e SSRS2_LICENSE
ghcr.io/antoinegou/ssrs2-weblitz-server:2.0.0-rc.2
```

`ghcr.io/antoinegou/ssrs2-weblitz-server:2.0.0-rc.2`, public on GitHub's container registry, for linux/amd64 and linux/arm64, answers everything; the two smaller editions below answer a part. It carries the Noto fonts for the scripts and emoji the packages do not bring, listens on port 8080 and runs as the image's own user, never root; put TLS in front of it. Each request runs in a worker process of the server's, so a report whose code never ends, or brings its process down, costs that worker and not the server. `GET /health` answers 503 while the engines warm up, then 200.

The image comes in three editions from the same code, each public, for linux/amd64 and linux/arm64:

Image	Edition	Answers	Status
`ghcr.io/antoinegou/ssrs2-server:2.0.0-rc.2`	reports	/v1/report, /v1/check, /v1/markdown, /v1/import	published
`ghcr.io/antoinegou/weblitz-server:2.0.0-rc.2`	html	/v1/html	published
`ghcr.io/antoinegou/ssrs2-weblitz-server:2.0.0-rc.2`	full	every /v1 route	published

From 2.0.0-beta.11 on, `ssrs2-server` answers reports only: a service printing HTML takes `weblitz-server`, or `ssrs2-weblitz-server` for both. An operation an edition does not have answers 404 with the code `unknownOp`, and `GET /v1/info` says its `edition`.

A page and a report

```
# a page as it is, with the engine's defaults
curl -H 'Content-Type: text/html' --data-binary @page.html
http://localhost:8080/v1/html -o page.pdf

# with options: Puppeteer's page.pdf() names, and the engine's own
curl -H 'Content-Type: application/json' \
     -d '{"html": "<h1>Hello</h1>", "options": {"format": "A4",
"margin": "15mm", "printBackground": true}}' \
     http://localhost:8080/v1/html -o hello.pdf

# a web page as a picture
curl -H 'Content-Type: application/json' \
     -d '{"url": "https://example.com", "output": "png", "options":
{"viewport": {"width": 1280, "deviceScaleFactor": 2}}}' \
     http://localhost:8080/v1/html -o example.png

# a report with its rows: the request as a field, the definition as a
file
curl -F 'request={"format": "PDF", "datasets": {"Items":
[{"Description": "Widget", "Price": 9.5, "Qty": 2}]}}' \
     -F report=@Invoice.rdlc http://localhost:8080/v1/report -o
invoice.pdf

# Markdown to a PowerPoint deck; another tool's report to RDLC
curl -F 'request={"format": "PPTX", "layout": "slides"}' -F
deck=@deck.md http://localhost:8080/v1/markdown -o deck.pptx
curl -F file=@Orders.frx http://localhost:8080/v1/import -o
Orders.rdlc
```

Endpoints

Request	What it does
`POST /v1/html`	A page (`html`) or an address (`url`) to `pdf` (the default), `xlsx`, `docx`, `pptx`, `png`, `jpeg`, `webp` or `tiff` (`output`)
`POST /v1/report`	A report (`definition`) to a `format`, with `datasets`, `parameters`, `deviceInfo`, `subreports`
`POST /v1/check`	A report's problems as notes, no file
`POST /v1/markdown`	Markdown to a deck (`PPTX`) or a document (`PDF`, `WORDOPENXML`)
`POST /v1/import`	Another tool's report, or a document, to RDLC
`GET /v1/info`	Versions, operations, formats, licence and platform

Request	What it does
`GET /health`	`503` while the engines warm up, then `200`

A request is JSON (camelCase names; a field the request does not have is refused, so a typo is not ignored), or multipart with a `request` field holding the JSON and the files beside it, or, for `/v1/html`, a page sent as `text/html`. Every request may give `timeoutMs`, which the server's limit caps.

What it answers

The file as the body, with its `Content-Type`, and the result in headers: `SSRS2-Pages`, `SSRS2-Notes` (how many notes), `SSRS2-Trial` (true without a licence key), `SSRS2-Elapsed-Ms`, and `SSRS2-Result`, the whole result as one line of JSON. With `Accept: application/json` the answer is the result as JSON, the file in base64: ask for it when the notes matter. A page's or a report's own problems are notes, never errors.

An error is `{"ok": false, "error": {"code", "message"}}` with its status: 400 `badRequest`, 401 `unauthorized`, 403 `refused` (a private address, a `file:` base URL, a network the server does not allow), 404 `unknownOp`, 413 `tooLarge`, 422 `invalidReport`, 500 `renderFailed` or `workerCrashed`, 503 `busy`, 504 `timeout`.

Settings

Setting	
`SSRS2_LICENSE`	The licence key. Without it, PDF and image pages carry the trial watermark.
`SSRS2_SERVER_TOKEN`	When set, every `/v1` request carries `Authorization: Bearer <token>`; `/health` stays open.
`Server__Workers`	Worker processes the requests run in (1 by default).
`Service__Concurrency`	Renders at once, across the workers (half the processors by default).
`Service__MaxTimeoutMs`	The longest a request may take (120,000 by default).
`Service__Network__AllowedHosts__0`	Hosts a page may load beyond its own address's, one setting each.

Settings come from the environment, a double underscore for each level. The defaults are for callers the server does not know: a page given as `html` loads nothing from the network and an address loads only from its own host, unless `Service__Network__AllowedHosts` opens more; private addresses and `file:` base URLs are refused; report code is restricted; a worker sees none of the server's secrets. Without a token the server answers anyone who reaches it: keep it on a private network then.

Pictures of a page

The same page, drawn instead of printed: a PNG, JPEG, WebP or TIFF. The page is loaded and its scripts run as for a PDF; then it is drawn as a screen shows it, or printed as the PDF is and drawn a page an image. Since 2.0.0-beta.10, in every package.

What is drawn

Capture	The image
The whole page (the default)	The document as a screen shows it, in one image: laid out in a window of 800 by 600 CSS px in screen media, the image as wide and as tall as the document scrolls. Puppeteer's `fullPage: true`.
The window	The window alone, as a screenshot without `fullPage`.
Each printed page	The page printed as the PDF prints it (paper, margins, print media, page ranges, headers and footers), an image a page.

Formats. PNG (the default, lossless), JPEG and WebP at a quality from 0 to 100 (92 when none is given), TIFF (lossless, one file holding every image).

The window. Its width and height, 1 to 16,384 CSS px: what the page is laid out at (its media queries, viewport units, its scripts' `innerWidth`).

Sharpness. The device scale factor is image pixels per CSS px: 1 draws at 96 dpi, 2 at 192 for a high density screen, 3.125 at 300 for print (0.1 to 10). It sharpens the image only: the page is laid out as at 1, and its scripts and media queries see a device pixel ratio of 1.

Size. An image past 16,384 pixels a side or 100 million in all is cut (a whole page) or not drawn (a printed page), and a note says so.

The page's options. The base address, scripts, the network, consent boxes and advertisements, styles, the locale and time zone, fonts and limits are the PDF's. A screen capture always draws backgrounds and takes neither paper, margins, scale, page ranges nor headers and footers.

The licence. Without a key, pictures carry the trial watermark, as a PDF's pages do.

.NET

`HtmlToImage` sits beside `HtmlToPdf`, with the same `Render`, `RenderFile` and `RenderUrl` and their `Async` twins. `HtmlImageOptions` holds `Capture`, `Type`, `Quality`, `ViewportWidth`, `ViewportHeight`, `DeviceScaleFactor` and `Page`, the PDF's own options. The result's `Images` are the files' bytes, one for a screen capture, one a printed page.

```
using WeblitzPdf;

// The whole page, sharp on a high density screen
var shot = HtmlToImage.RenderUrl(new Uri("https://example.com"), new
HtmlImageOptions
{
    ViewportWidth = 1280,
    DeviceScaleFactor = 2,
});
File.WriteAllBytes("example.png", shot.Images[0]);

// Each printed page of an A4 document as a JPEG
var pages = HtmlToImage.RenderFile("templates/letters.html", new
HtmlImageOptions
{
    Capture = HtmlImageCapture.Pages,
    Type = HtmlImageType.Jpeg,
    Quality = 85,
    Page = new HtmlPdfOptions { Format = "A4" },
});
for (var i = 0; i < pages.Images.Count; i++)
    File.WriteAllBytes($"page-{i + 1}.jpg", pages.Images[i]);
```

The result also has `PageCount`, `Notes` (what was left out or could not be drawn), `Completed` (false when a limit or the script timeout stopped the render), `Elapsed` and, for an address, `HttpStatus`.

Node.js

`output` names the format, with Puppeteer's screenshot options: `capture: 'viewport'` (or `fullPage: false`) for the window, `capture: 'pages'` for the printed pages, answered as a ZIP of `page-1.png`, `page-2.png`… (a TIFF holds them all in one file).

```
import { renderUrl, renderHtml } from 'weblitzpdf';

const shot = await renderUrl('https://example.com', { output: 'png',
viewport: { width: 1280, height: 720, deviceScaleFactor: 2 }, path:
'example.png' });
const pages = await renderHtml(html, { output: 'jpeg', quality: 85,
capture: 'pages', format: 'A4' });   // pages.file is a ZIP
```

Python

```
import weblitz

weblitz.render_url("https://example.com", output="png",
                   viewport={"width": 1280, "device_scale_factor": 2},
path="example.png")
pages = weblitz.render_html(html, output="jpeg", quality=85,
capture="pages", format="A4")   # pages.file is a ZIP
```

The names are Playwright's: `capture="viewport"` or `full_page=False` for the window.

Java

```
import com.weblitzpdf.*;

Weblitz.renderUrl("https://example.com", new
HtmlOptions().setOutput("png")
        .setViewport(new Viewport(1280,
720).setDeviceScaleFactor(2))).save(Path.of("example.png"));
Result pages = Weblitz.renderHtml(html, new
HtmlOptions().setOutput("jpeg").setQuality(85)
        .setCapture("pages").setFormat("A4"));   // pages.file() is a
ZIP
```

The names are Playwright's: `setCapture("viewport")` or `setFullPage(false)` for the window.

HTTP

```
curl -H 'Content-Type: application/json' \
     -d '{"url": "https://example.com", "output": "png", "options":
{"viewport": {"width": 1280, "deviceScaleFactor": 2}}}' \
     http://localhost:8080/v1/html -o example.png
```

`/v1/html` of the server takes `output` `png`, `jpeg`, `webp` or `tiff`, and `capture`, `quality` and `viewport` in its `options`; printed pages answer a ZIP.

Printing like Chrome

Weblitz prints a page the way Chrome's "Save as PDF" does: the page is first loaded as on a screen, its scripts run, then it is printed. Knowing the order explains what a page sees and when.

From HTML to PDF

Load. The HTML is parsed, its stylesheets, pictures and fonts are loaded (from the file's folder, or the network if you allow it), and the page is laid out in *screen* media.

Scripts. The page's scripts run once, at the width and height of the printed page area (the paper less its margins), as they would in the window Chrome prints from. Timers and animations run in virtual time, up to HtmlScripts.WaitFor.

beforeprint. The page's `beforeprint` handlers run: a page can change itself for print there.

Print media. The media switches to `print` (`@media print` rules apply) and the document is laid out on pages: the paper, the margins, `@page` rules, page breaks.

The PDF is written, then `afterprint` runs (it changes nothing in the PDF).

The scripts example shows it: text added as the page loads, by a timer, and on `beforeprint`.

Defaults, beside Puppeteer's

The options carry Puppeteer's `page.pdf()` names, so a call moved over from Puppeteer reads the same. Four defaults differ: three chosen to print as the engine is measured against Chrome, and one so that the PDF holds the whole page:

Format is A4 (Puppeteer's is Letter).

PrintBackground is true: backgrounds are printed (Puppeteer's default leaves them out).

PreferCssPageSize is true: a page's own `@page` size wins over Format.

Lazy content is printed. Without a scroll option, a page's IntersectionObserver reports every element it observes visible, so lazy pictures, charts drawn when they come into view and items added as the list is reached are all in the PDF. Chrome, and so Puppeteer, reports only what is in the first screen and prints the rest as the page left it: placeholders, spinners, empty boxes. Set ScrollTo to `"top"` to print as Chrome does (Scrolling).

Set them in the options if your pages expect Puppeteer's.

Videos

A `<video>` prints its poster, or the frame at its current time once it plays, as Chrome prints it. Playback runs on the page's virtual time: a page that plays a video, then prints, shows the frame at that time. VP8, VP9 and AV1 are decoded by Weblitz itself. H.264 and HEVC, the usual codecs of `.mp4` files, are decoded with the machine's own codec, which Weblitz does not ship: macOS has it; Windows desktop editions have it, and Windows Server once its Media Foundation feature is added (`Install-WindowsFeature Server-Media-Foundation`; HEVC needs Microsoft's HEVC Video Extensions); Linux needs FFmpeg's libavcodec (`apt-get install -y libavcodec61`,

or your release's version; `apk add ffmpeg-libavcodec` on Alpine) or, for H.264, Cisco's OpenH264. Without one, such a video prints its poster, and the render's notes say which codec to install.

What it is not

Weblitz prints; it does not browse. There is no window to click or type in: content a page shows only after a user's clicks needs to be in its HTML (or made by its scripts) to print. Pages behind a login print with what a browser would send, headers and bearer tokens, cookies, Basic and Digest (pages behind a login), and the scroll options scroll the window before printing (scrolling). Pages are printed once, in their state after the scripts' virtual time.

Fonts

Text is drawn with the first font that has it, in this order:
the page's own fonts, loaded with `@font-face` from its folder (or the network, if allowed);
the machine's installed fonts, by family name;
the faces in the package: metric-compatible fonts for Arial, Helvetica, Times New Roman, Courier New, Calibri, Cambria and Segoe UI (Latin, Greek, Cyrillic and Hebrew letters), so a page naming them lays out with the same widths everywhere.
With UseInstalledFonts false, the machine's fonts are left out: only the page's fonts and the bundled faces draw, and the same input gives the same PDF on every machine. The fonts example shows both on a Mac: families the Mac has (Helvetica Neue, Avenir, Japanese) fall to the next family without its fonts.
What a missing font does
A family nobody has goes to the next one in the list, as in Chrome, and a character no font has is drawn as an empty box. Both are reported as notes (`fontFamilyMissing`, `missingGlyph`): check them when a page's text looks wrong.
Linux containers
On Linux, Weblitz needs fontconfig, which most systems have but the official .NET Docker images lack: without it a render stops with an error that says what to install. On an image with `apt`:

```
RUN apt-get update && apt-get install -y --no-install-recommends
libfontconfig1 && rm -rf /var/lib/apt/lists/*
```

A container image has few fonts, often none. The bundled faces cover Latin, Greek, Cyrillic and Hebrew text in the families above; Arabic, Chinese, Japanese, Korean, Thai, the Indic scripts and emoji need fonts installed in the image. The Noto families cover them; with fontconfig, on an image with `apt`:

```
RUN apt-get update && apt-get install -y --no-install-recommends \
    libfontconfig1 fonts-noto-core fonts-noto-cjk fonts-noto-color-
emoji && rm -rf /var/lib/apt/lists/*
```

On Alpine, add the `gcompat` package too (Weblitz's script engine is built for glibc) and `fontconfig`:

```
RUN apk add --no-cache gcompat fontconfig font-noto font-noto-arabic
font-noto-cjk font-noto-emoji
```

Or ship the fonts with your pages: `@font-face` fonts in a page's folder are used first on every machine.

Pages, headers and footers

The paper

The paper is Format (A4 by default; Letter, Legal, Tabloid, Ledger, A0 to A6), or Width and Height as CSS lengths, turned with Landscape. A page's own `@page { size: ... }` wins by default (PreferCssPageSize). See paper sizes.

Margins

Margin sets them as CSS lengths (`HtmlMargins.All("15mm")`, or each side). Without it, the page's `@page` margins apply, else Chrome's default margins. See margins. Scale lays the page out as on larger or smaller paper (0.1 to 2): see scale.

Page breaks

CSS decides where pages break: `break-before`, `break-after` and `break-inside: avoid` on elements, `orphans` and `widows` on paragraphs; a table's header rows are printed again on each page it runs over. PageRanges keeps some pages only ("2-3, 5"): see choosing pages.

Headers, footers and page numbers

The page's own CSS can print them in the margins, with `@page` margin boxes; `counter(page)` and `counter(pages)` give the numbers:

```
@page {
  size: A4;
  margin: 18mm 15mm 20mm;
  @bottom-right { content: "Page " counter(page) " of "
counter(pages); font: 9pt Arial; color: #6b7280; }
  @top-left { content: "Quarterly report"; font: 9pt Arial; }
}
```

Or Chrome's header and footer templates, as Puppeteer prints them (DisplayHeaderFooter, HeaderTemplate, FooterTemplate): HTML printed in the margins of each page, its elements of the classes `pageNumber`, `totalPages`, `title`, `url` and `date` filled in. A template that sets no font size prints in Chrome's tiny default size: set one. Give the margins room for them: see the example.

```
var options = new HtmlPdfOptions
{
    DisplayHeaderFooter = true,
    Margin = new HtmlMargins("22mm", "12mm", "20mm", "12mm"),
    FooterTemplate = "<div style='font-size:9px; width:100%; text-
align:right; padding:0 12mm'>Page <span class='pageNumber'></span> of
<span class='totalPages'></span></div>",
};
```

Scripts and the virtual clock

A page's scripts run once, before printing, in V8 (Chrome's JavaScript engine, built into the package): pages built by React or Vue, charts drawn by Chart.js, tables filled from the page's own data. The scripts of the page's frames run with them. HtmlScripts.Off runs none.

The virtual clock

Timers and animations run in virtual time: a page that waits two seconds for a timer does not make the render wait two seconds. The page gets WaitFor of virtual time (5 seconds by default) before it is printed; a timer set for later does not run. Timeout is different: the real processor time scripts may take (30 seconds by default) before they are stopped and the page printed as it is. See scripts and the clock.

```
var options = new HtmlPdfOptions
{
    Scripts = HtmlScripts.Default with { WaitFor =
TimeSpan.FromSeconds(10), Timeout = TimeSpan.FromSeconds(5) },
};
```

Animations and transitions

A print shows one moment of a page, so the frames of an animation in between are work nobody sees. By default (Animations = `HtmlPdfAnimations.Finished`) Weblitz takes each CSS animation, transition and `Element.animate()` at its end while the page's scripts run, and one that never ends, a spinner, at its first moment. The printed page is the one Chrome prints. The events (`transitionend`, `animationend`) are sent at their times and `Animation` objects read their real times, so a page that waits for an animation to end waits as long as in Chrome.

One difference from Chrome is on purpose: a script that reads a style or a size in the middle of an animation reads its end value, where Chrome's reads the value at that frame.

`HtmlPdfAnimations.FrameByFrame` runs them as Chrome does, for a page whose scripts depend on values read in mid-animation. It takes longer on pages with many animated elements.

`HtmlPdfAnimations.None` switches CSS animations and transitions off, as the rule `* { animation: none !important; transition: none !important }` many applications inject into Chrome before printing. No animation or transition event is sent, so a page that waits for one to show its content may print without it.

```
var options = new HtmlPdfOptions { Animations =
HtmlPdfAnimations.FrameByFrame };
```

The same PDF every time

What scripts read of the world is fixed by the options: Locale (en-US), TimeZone (UTC) and Now; `Date` and `Intl` both read the time zone. Give them, and the same input prints the same PDF: see locale and clock.

Canvas text and web fonts

A chart drawn on a canvas writes its text with the fonts loaded when it draws. Draw after your web fonts are loaded, the weight you use included:

```
Promise.all([document.fonts.load("400 12px 'Your Font'"),
document.fonts.ready]).then(drawCharts);
```

The browser a page sees

A page sees desktop Chrome 156 on the machine's own system (`navigator.userAgent`, `navigator.userAgentData`, and the User-Agent header when the network is on). UserAgent and UserAgentMetadata change it: see the browser a page sees.

The page is also told that a program drives its browser: `navigator.webdriver` is true, as in Chrome under Puppeteer or Playwright. Analytics and trackers that leave automated browsers out then stay off, so a render makes fewer requests, runs less script, and is not counted as a visit. Some sites show an automated browser a check, or less of their page: Automated = `false` prints as an ordinary visitor's browser would. The user agent says desktop Chrome either way.

Since WeblitzPdf 2.0.0-beta.4; in 2.0.0-beta.3 and before, `navigator.webdriver` is false.

Network and security

Nothing leaves the machine by default

Render fetches nothing: relative addresses resolve against BaseUrl, a folder's files are read from that folder only.

RenderFile reads the file's folder, and nothing outside it.

RenderUrl allows the page's own host when the network is left off.

Anything else is refused and noted (`resourceRefused`). Allow hosts with HtmlNetwork:

```
var options = new HtmlPdfOptions
{
    Network = HtmlNetwork.Allow("cdn.example.com", "*.example.org")
with
    {
        ResourceTimeout = TimeSpan.FromSeconds(10),
        CacheDirectory = "/var/cache/weblitz",
    },
};
```

HtmlNetwork.Any allows any public host; the machine's own addresses are never reached unless named. See the network example.

Pages behind a login, and error pages

A page behind a login prints with what it needs: headers, a bearer token, cookies, Basic or Digest credentials (see pages behind a login). Cookies are kept by site as Chrome keeps them, with the Public Suffix List: example.co.uk and other.co.uk are two sites, and a cookie a server sets for a whole public suffix (`Domain=co.uk`) is refused.

When the server answers with an error status (401, 403, 404…) and a page of its own, that page is printed, as Chrome prints it. HttpStatus gives the status and a note says so (`documentHttpError`): check it before sending the PDF on. An error with no page throws HtmlRenderException.

```
var result = HtmlToPdf.RenderUrl(new
Uri("https://intranet.example.com/statement"), options);
if (result.HttpStatus is not 200)
    throw new InvalidOperationException($"The server answered
{result.HttpStatus}.");
```

Limits for pages you don't trust

HtmlPdfLimits bounds what a page may ask for: its size, its nodes, its pages, each resource and all of them; scripts are bounded by HtmlScripts.Timeout. A render that reaches a limit stops, prints what it has, returns Completed false, and notes which limit (`limitReached`): see limits. A page's own errors are notes, never exceptions; an exception (HtmlRenderException) means no PDF could be made at all.

Limits are a guard, not a sandbox. HTML from people you don't trust is best printed in a process or container of its own, without secrets and without network access beyond what the pages need.

Protected PDFs

Security encrypts the PDF (AES-256) with a user password, an owner password, and what a reader may do without the owner's: print, copy, modify. Not with PdfA: PDF/A forbids encryption. See passwords and permissions.

Many documents

HtmlToPdf is static and thread-safe: one call per document, on as many threads as you use. There is nothing to start, pool or keep alive: no browser process, no page to recycle.

Many documents at once

```
Parallel.ForEach(invoices, new ParallelOptions {
MaxDegreeOfParallelism = Environment.ProcessorCount }, invoice ⇒
{
    var result = HtmlToPdf.Render(invoice.Html, options);
    File.WriteAllBytes(invoice.Path, result.Pdf!);
});
```

In a web application, RenderAsync runs a render on the thread pool. A cancellation stops a render that has not started; one under way runs to its end or its limits.

Large documents

Render(html, stream) writes the PDF to a stream (a file, a response) instead of returning its bytes.

Compression trades size for time: Fastest by default, Smallest for files kept long. See the sizes.

MaxPages and the other limits keep a runaway page from filling memory.

Warming up

.NET compiles an application's code as it runs, so a process's first renders of a kind of document are slower than its later ones. HtmlToPdf.WarmUp renders a document a few times and writes nothing, so that this cost is paid before the renders that matter. It is optional: Weblitz never warms up by itself, and nothing runs in the background.

```
// Once, at start-up, with a document like those the application
prints
var warm =
HtmlToPdf.WarmUp(File.ReadAllText("templates/invoice.html"), options);
Console.WriteLine($"first render {warm.First.TotalMilliseconds:0} ms,
last {warm.Last.TotalMilliseconds:0} ms");
```

Call it once, at start-up or while idle, with the template and the options of the real work: other kinds of document gain less. A process that prints one document and exits gains nothing from it.

It costs the time and processor of its renders (eight unless you say otherwise), once, on the calling thread. WarmUpAsync runs it on the thread pool; WarmUpFile takes a file. Each of its renders loads the document's resources and runs its scripts as a real render does: a page that reports to a server reports at each one.

The result says how long the first and the last of them took: the last is close to what a real render will take.

Speed

Speed figures will be published here once measured on a quiet machine, with the date and the method.

Scrolling and printing a part

Lazy content prints, with no option to set. Without a scroll option (neither ScrollTo nor ScrollThrough), a page's IntersectionObserver reports every element it observes visible, once, wherever it sits on the page. So what a page loads as it comes into view, lazy pictures, charts drawn when they are reached, items its scripts add, is loaded and printed: the whole page is in the PDF. Chrome reports only what is in its window, so with Puppeteer such a page prints its first screen unless a script scrolls it first.

To print as Chrome does, with only the first screen's lazy content loaded, set a scroll option: `new HtmlPdfOptions { ScrollTo = "top" }`. The observer then reports real intersections with the window, and what lies below the first screen prints as the page left it, as in Chrome's "Save as PDF" or Puppeteer's `page.pdf()`. This is a choice, not a fidelity setting: either way the page is laid out and drawn as Chrome draws it.

Some pages want real scrolling: a list that loads more each time its bottom comes into view, effects that follow the scroll, a page opened at a section. Four options scroll the window before printing, as a user would before pressing Print, or print a part of the page. With a scroll option set, Weblitz reports real intersections with the window, as Chrome does.

ScrollThrough: scroll from the top to the end, a window height at a time, so what loads on the way is printed (200 steps at most).

ScrollTo: scroll to `"top"`, `"bottom"`, an element (a CSS selector) or a length, once the scripts have settled; the page's scroll events and IntersectionObserver callbacks see it. A scroll timeline on the page's own scroll prints at its start, scrolled or not, as Chrome prints it.

ScrollStepWait: the virtual time each scroll gives the page to react (0.5 s by default, 0 to 60 s).

Region: print a part of the document (HtmlPrintRegion), from the scroll position, a length or an element, for a height; laid out on pages as a whole document is.

Scrolling alone never crops: the PDF holds the whole document from its top, as Chrome's print of a scrolled page does. Only Region leaves parts out.

The examples below show both: the infinite list prints twenty orders by default, as its observer reports its bottom once, and all sixty with ScrollThrough.

Lists that load as you scroll

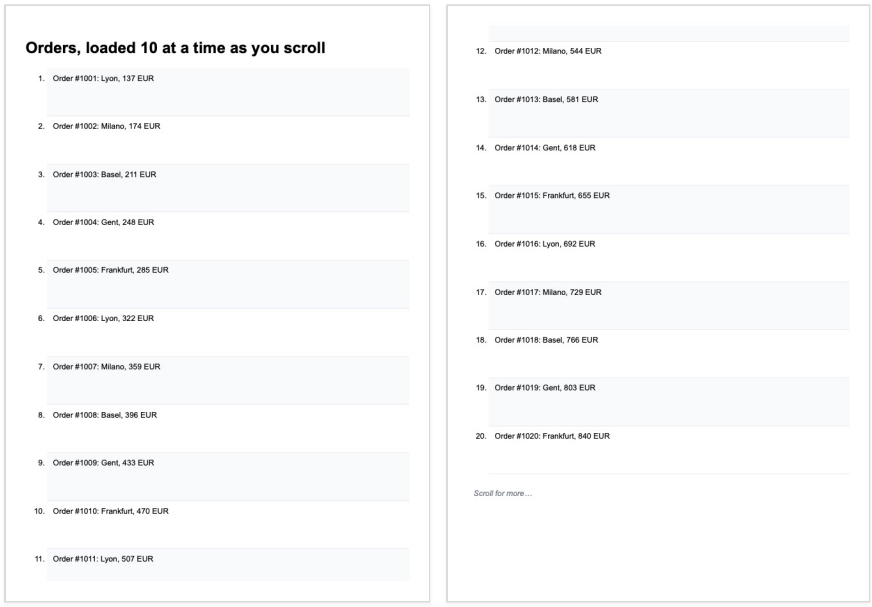
An infinite list loads its next ten orders each time its bottom comes into view. Without a scroll option, Weblitz's IntersectionObserver reports the bottom visible once, so the PDF holds twenty; ScrollThrough scrolls the window down a window height at a time, each step waiting ScrollStepWait of virtual time, so the list loads to its end and all sixty are printed.

Reference: `HtmlPdfOptions.ScrollThrough`, `HtmlPdfOptions.ScrollStepWait`

▶ The page printed: `lazy-list.html`

Default: not scrolled

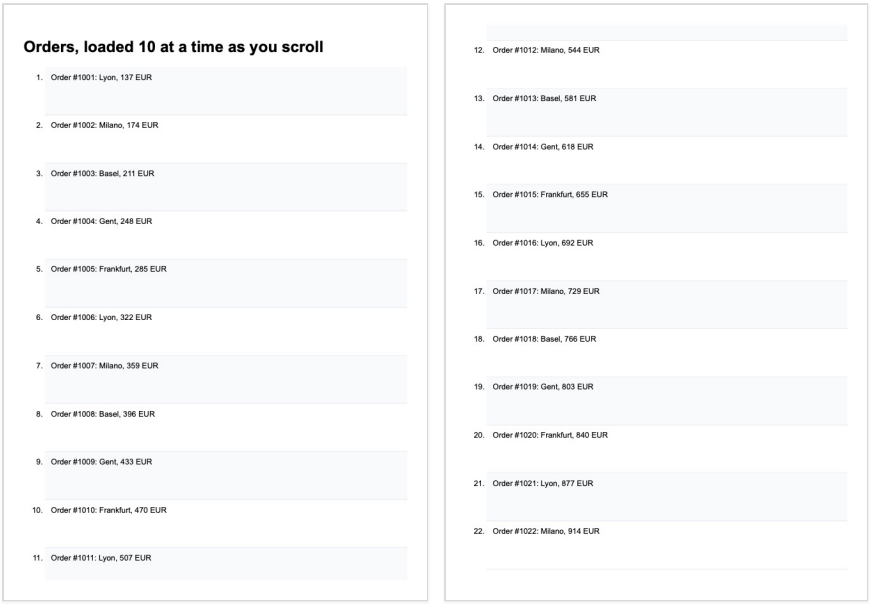
```
var result = HtmlToPdf.Render(html);
```



Pages	2
File	53,524 bytes (52.3 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (133 ms of virtual time, 5 tasks; 0 canvases drawn).

ScrollThrough

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { ScrollThrough = true });
```



Pages 6

File 58,677 bytes (57.3 KB) · the PDF

Notes `Info scriptsRan`: The page's scripts ran (883 ms of virtual time, 40 tasks; 0 canvases drawn).

Scrolling to a section first

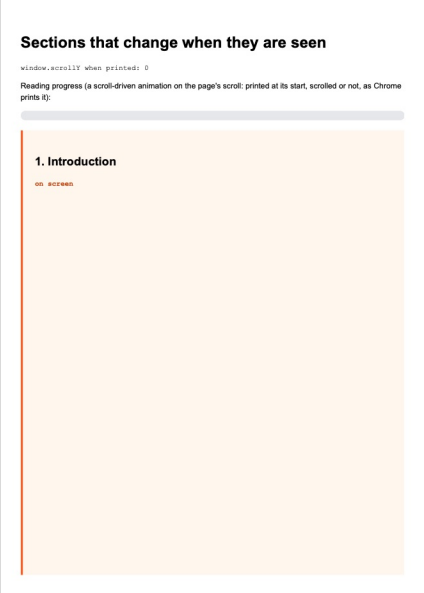
ScrollTo scrolls the window before printing, as a user would before pressing Print: the page's scroll events and IntersectionObserver callbacks see it (here the sections that went off screen, and scrollY). A scroll timeline on the page's own scroll prints at its start either way, as Chrome prints it. The PDF still holds the whole document from its top.

Reference: `HtmlPdfOptions.ScrollTo`

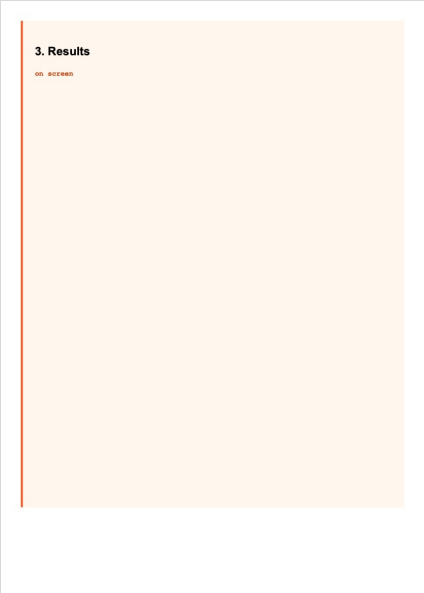
▶ The page printed: `scroll-effects.html`

Default: not scrolled

```
var result = HtmlToPdf.Render(html);
```







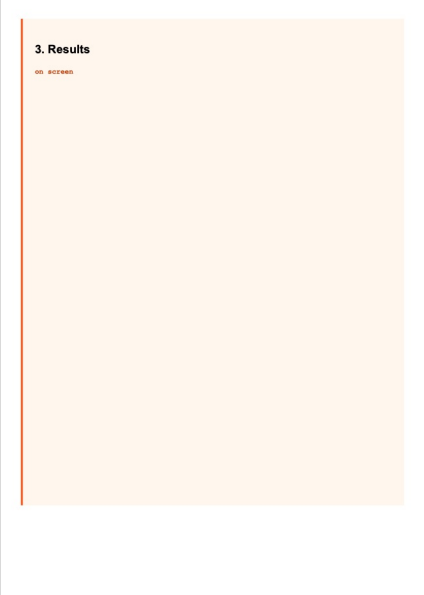
Pages	3
File	69,331 bytes (67.7 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

ScrollTo "#results"

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { ScrollTo = "#results" });
```







Pages	3
File	70,553 bytes (68.9 KB) · the PDF
Notes	`Info scriptsRan`: The page's scripts ran (33 ms of virtual time, 4 tasks; 0 canvases drawn).

Printing a part of the page

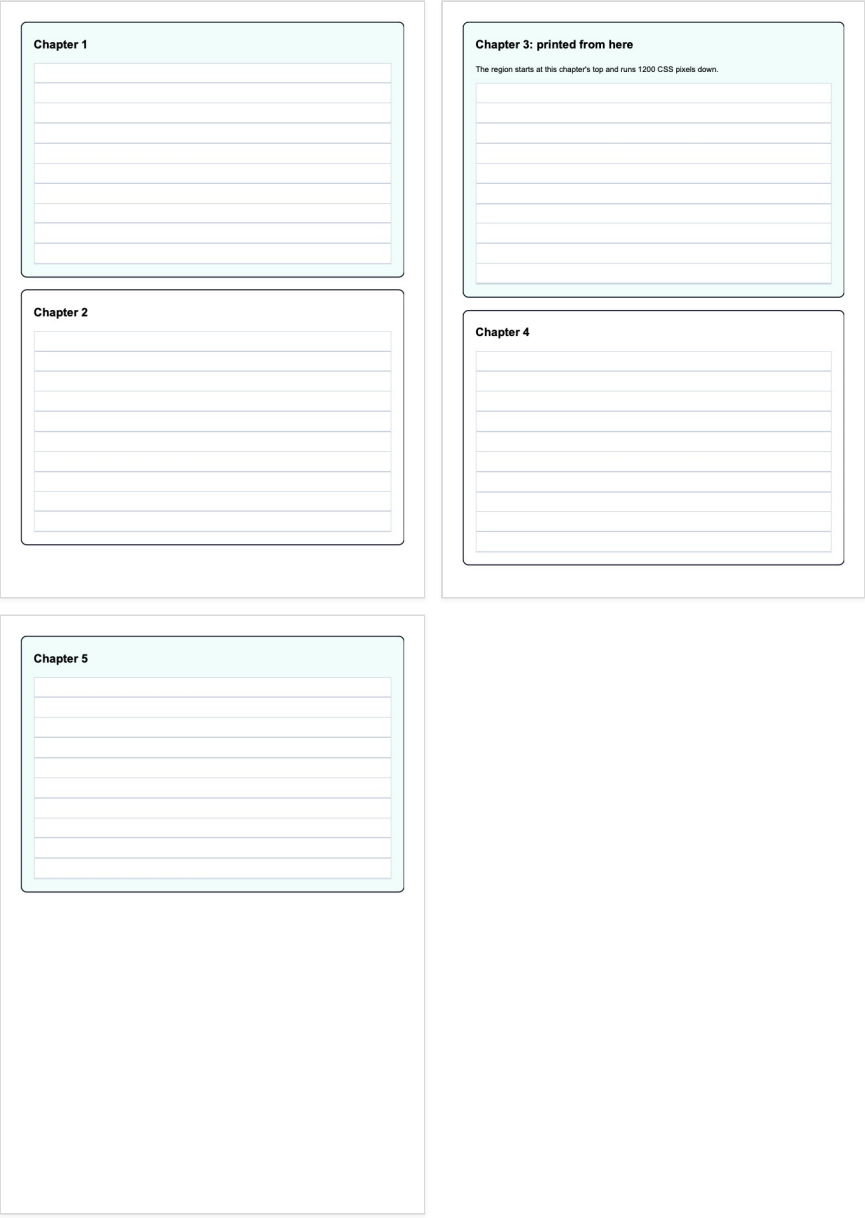
Region prints a part of the document, from a start (the scroll position, a length, an element) for a height; what lies above and below is left out, and the part is laid out on pages as a whole document is.

Reference: `HtmlPdfOptions.Region`, `HtmlPrintRegion`

▶ The page printed: `chapters.html`

Default: the whole document

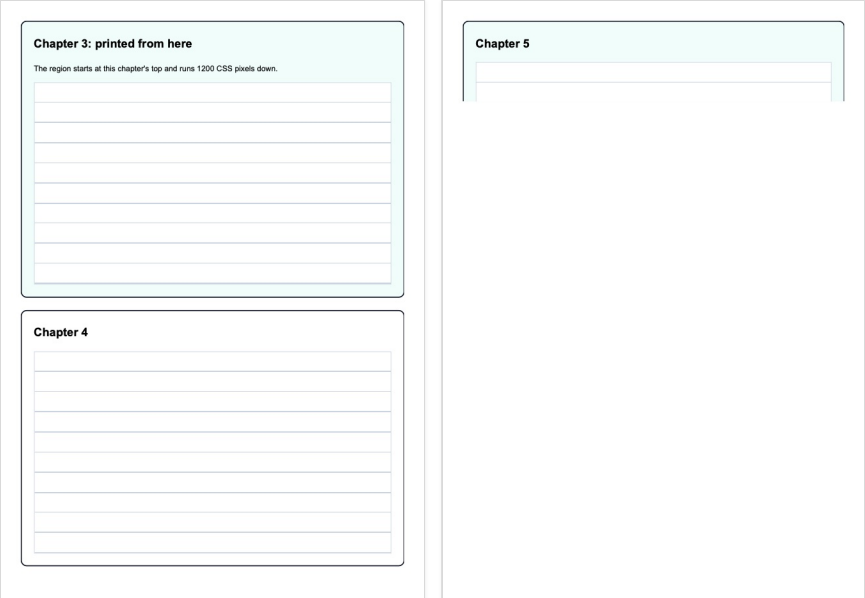
```
var result = HtmlToPdf.Render(html);
```



Pages 3
File 46,933 bytes (45.8 KB) · the PDF

1200 px from #chapter-3

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Region =
HtmlPrintRegion.Of("#chapter-3", "1200px") });
```



Pages 2
File 56,560 bytes (55.2 KB) · the PDF

Consent boxes and advertisements

A page fetched from the web seldom comes alone: a consent dialog lies over it, a cookie bar under it, and advertisements between its paragraphs. By default Weblitz prints all of it, as a browser does. Two options leave it out. Both are off unless you set them, and neither clicks anything.

Since WeblitzPdf 2.0.0-beta.4.

Consent boxes

Consent = `HtmlConsent.Hide` leaves consent boxes out of the print. No button is pressed and no answer is given in the reader's name: the page and its consent platform run as they come, and what they show of it is not printed. It hides:

the boxes of the consent platforms most sites use (OneTrust, Sourcepoint, Cookiebot, Didomi, Usercentrics, CookieYes and others), by the platforms' own containers;
what the EasyList Cookie List names, a list of thousands of sites' cookie notices kept by the EasyList authors;
a box of the page's own: one fixed to the window, or a dialog, of a few lines that speaks of cookies, consent or privacy and holds a button to answer with;
the veil such a box puts over the page, and the lock it puts on the page's scrolling, which would cut the print to its first page.

What was hidden is said in the result's notes (code `consentHidden`), each of the page's own boxes named as a selector names it. A box no rule knows stays: name it in Hide. A site that shows its content only once consent is given prints what it shows without it.

Advertisements and trackers

Ads = `HtmlAds.Block` prints a page as an ad blocker shows it, by the filter lists EasyList and EasyPrivacy:

a request the lists block is not made: fewer requests, less script to run;
an element they hide is not printed;
a frame or a picture whose request was refused leaves no empty box.

The notes count what was left out (code `adsBlocked`). AdFilters adds rules of your own in the lists' syntax, a line a rule: `||ads.example.com^` blocks the requests to a host, `example.com##.sponsored` hides elements on a site, `@@` and `#@#` make exceptions. A site that refuses readers who block its advertisements prints what it shows them.

The lists, and keeping them fresh

The filter lists are the work of the EasyList authors (easylist.to). The package carries a snapshot of EasyList, EasyPrivacy and the EasyList Cookie List as plain files in `lists/` beside your application, unchanged, with their notice and licences in `lists/NOTICE.txt`. The snapshot is as old as the package, and the lists change every day.

HtmlToPdf.UpdateFilterListsAsync() fetches today's lists and uses them from then on. Call it when your application starts:

```
var lists = await HtmlToPdf.UpdateFilterListsAsync();   // optional, at start
```

It is the only thing that fetches the lists: no render ever does, and nothing runs in the background. The copies are kept in a folder (one of your own if you name it) and read again by later runs; a list is not fetched again before its own refresh interval has passed, and one that cannot be fetched leaves the previous copy, or the snapshot, in use, with the reason in the result.

Without the consent boxes

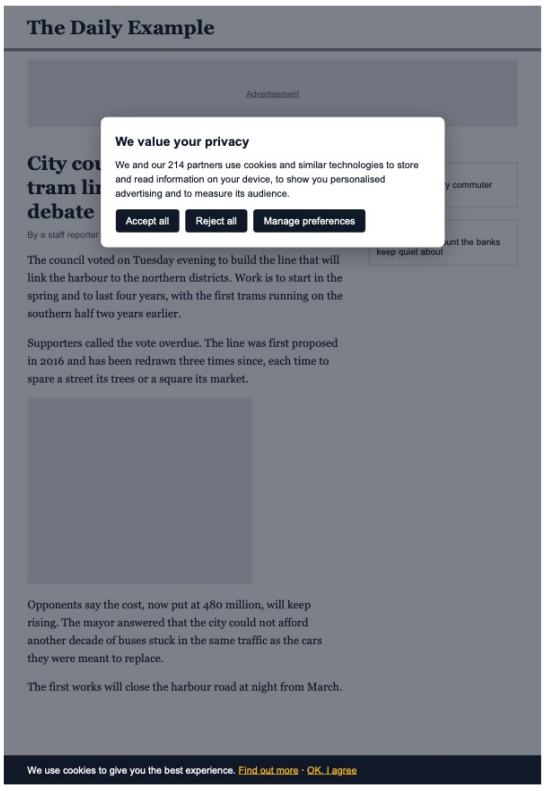
A page from the web often comes with a consent dialog over it and a cookie bar under it. Consent = Hide leaves them out of the print without clicking anything: the page and its consent platform run as they come, no answer is given in the reader's name. It hides the consent platforms' own boxes, the boxes the EasyList Cookie List names, and a box of the page's own (fixed to the window, or a dialog, that speaks of cookies, consent or privacy and holds a button to answer with), with the veil behind it and the lock it puts on the page's scrolling. A consentHidden note says what was hidden.

Reference: `HtmlPdfOptions.Consent`

▶ The page printed: `news-page.html`

Default: the page as it comes

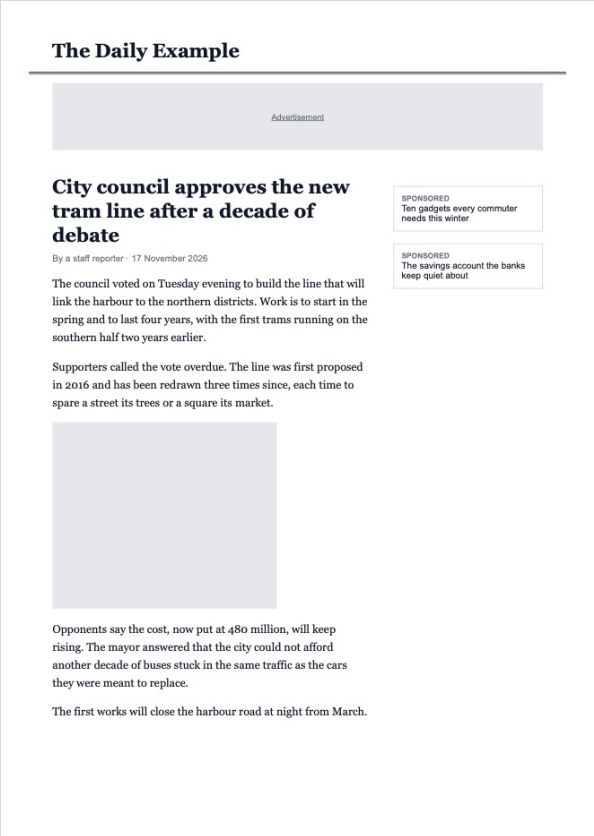
```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	100,219 bytes (97.9 KB) · the PDF
Notes	`Info resourceRefused`: Script https://www.google-analytics.com/analytics.js: no resource loader was given (only data: URLs load). `Info resourceRefused`: Frame https://securepubads.g.doubleclick.net/gampad/adx?slot=inline: no resource loader was given (only data: URLs load). `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

Consent = Hide

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions { Consent = HtmlConsent.Hide
});
```



Pages	1
File	70,593 bytes (68.9 KB) · the PDF
Notes	`Info resourceRefused`: Script https://www.google-analytics.com/analytics.js: no resource loader was given (only data: URLs load). `Info resourceRefused`: Frame https://securepubads.g.doubleclick.net/gampad/adx?slot=inline: no resource loader was given (only data: URLs load). `Info consentHidden`: Consent boxes are not printed (Consent = Hide): the page's own div#consent, the page's own div#cookie-bar. `Info scriptsRan`: The page's scripts ran (33 ms of virtual time, 6 tasks; 0 canvases drawn).

Without advertisements and trackers

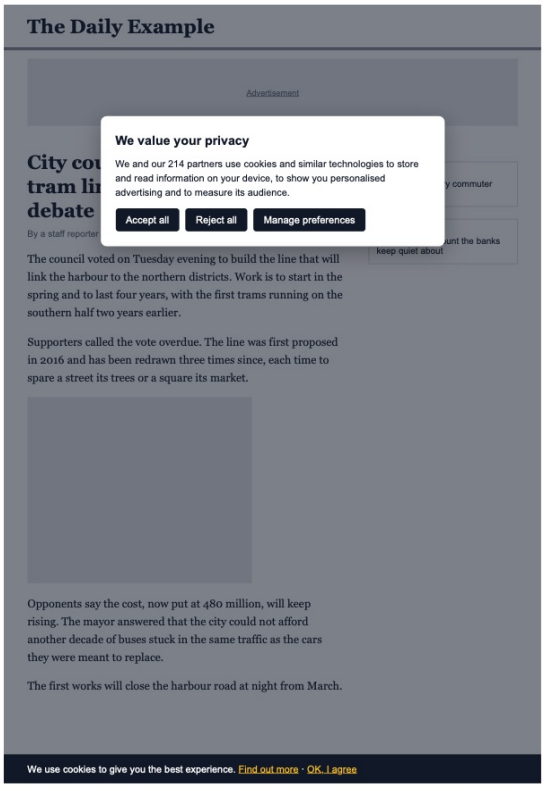
Ads = Block prints a page as an ad blocker shows it, by the filter lists EasyList and EasyPrivacy (the work of the EasyList authors, a snapshot of which the package carries): a request the lists block is not made, an element they hide is not printed, and a frame whose request was refused leaves no empty box. AdFilters adds rules of your own in the same syntax, here for the page's sponsored boxes. An adsBlocked note counts what was left out. Here with Consent = Hide as well.

Reference: `HtmlPdfOptions.Ads`, `HtmlPdfOptions.AdFilters`, `HtmlToPdf.UpdateFilterListsAsync`

▶ The page printed: `news-page.html`

Default: the page as it comes

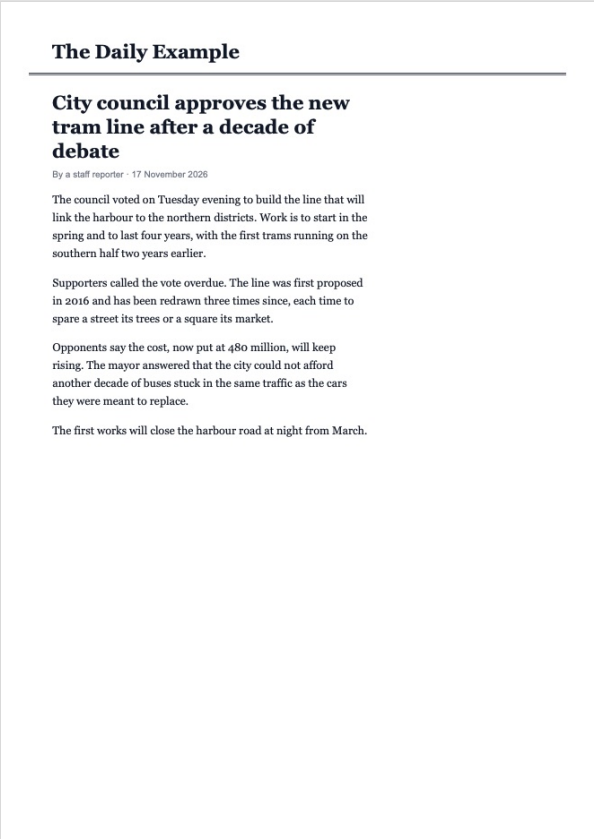
```
var result = HtmlToPdf.Render(html);
```



Pages	1
File	100,219 bytes (97.9 KB) · the PDF
Notes	`Info resourceRefused`: Script https://www.google-analytics.com/analytics.js: no resource loader was given (only data: URLs load). `Info resourceRefused`: Frame https://securepubads.g.doubleclick.net/gampad/adx?slot=inline: no resource loader was given (only data: URLs load). `Info scriptsRan`: The page's scripts ran (17 ms of virtual time, 2 tasks; 0 canvases drawn).

Ads = Block, with a rule of our own

```
var result = HtmlToPdf.Render(html, new HtmlPdfOptions
{
    Ads = HtmlAds.Block,
    AdFilters = ["##.sponsored"],
    Consent = HtmlConsent.Hide,
});
```



The Daily Example

City council approves the new tram line after a decade of debate

By a staff reporter · 17 November 2026

The council voted on Tuesday evening to build the line that will link the harbour to the northern districts. Work is to start in the spring and to last four years, with the first trams running on the southern half two years earlier.

Supporters called the vote overdue. The line was first proposed in 2016 and has been redrawn three times since, each time to spare a street its trees or a square its market.

Opponents say the cost, now put at 480 million, will keep rising. The mayor answered that the city could not afford another decade of buses stuck in the same traffic as the cars they were meant to replace.

The first works will close the harbour road at night from March.

Pages	1
File	52,292 bytes (51.1 KB) · the PDF
Notes	`Info consentHidden`: Consent boxes are not printed (Consent = Hide): the page's own div#consent, the page's own div#cookie-bar. `Info scriptsRan`: The page's scripts ran (33 ms of virtual time, 6 tasks; 0 canvases drawn). `Info adsBlocked`: Advertisements and trackers are left out (Ads = Block): 2 requests not made, 168 of the lists' hiding rules given to the page.

Excel, Word and PowerPoint

The same page, loaded the same way, written as an Office file instead of a PDF. Three classes stand beside HtmlToPdf, with the same `Render`, `RenderFile` and `RenderUrl` and their `Async` twins: HtmlToExcel, HtmlToWord and HtmlToPowerPoint. Nothing else to install: the files are written by Weblitz itself, with no Office on the machine.

Since WeblitzPdf 2.0.0-beta.7; a workbook's pictures and a deck's tables since 2.0.0-beta.8. Tell us what your pages need.

Which one

The three are not three copies of the page. Each keeps something else of it:

Excel keeps the data. The page's tables become sheets whose numbers are numbers.

Word keeps the content. The page becomes a flowing document that someone can go on writing in.

PowerPoint keeps the look. Each printed page becomes a slide that shows it as it prints, with its text as text.

When the page must look exactly as it prints and nobody will change it, a PDF is still the answer.

Excel: the page's tables

```
var result = HtmlToExcel.RenderUrl(new
Uri("https://example.com/prices"));
File.WriteAllBytes("prices.xlsx", result.Workbook!);
```

Each table that holds data is a sheet, named after its caption, its `aria-label` or the heading before it:

cells at their rows and columns, with their row and column spans as merged cells;

numbers, percentages and amounts of money as typed cells, read in the page's language (`1,250.50` in English, `1.250,50` in German) and shown with the page's decimals, thousands separators and currency; ISO dates and `<time datetime>` as dates;

what only looks like a number stays text: a code with leading zeros, a telephone number, a number of more than 15 digits, a date written a local way (03/04/2026 is two different days);

header rows frozen at the top; fonts, colours, alignments and column widths as the page shows them; a cell that is one link keeps it.

A cell's pictures are over the cell, where the page shows them (a country's flag beside its name), with their alternative text. A table that only lays the page out is not a sheet: the tables inside it are. A page without any table gives one sheet of its text, a block a row.

Word: a document to work on

```
var result = HtmlToWord.Render(html);
File.WriteAllBytes("page.docx", result.Document!);
```

The page's content in reading order, as Word's own blocks:

headings in Word's heading styles, so the navigation pane and a table of contents find them;

paragraphs whose runs keep their font, size, weight, slant, colour, underline and links; bulleted and numbered lists as Word's lists;
tables with their spans, fills, borders and a header row that repeats on each page; pictures at the size the page shows them, with their alternative text, SVG, AVIF and WebP ones too; preformatted text line by line.
The paper and its margins are the page options' (Format, Landscape, Margin, or the page's own `@page`).
It is not a picture of the page: positions, columns and backgrounds are not kept; a menu the page lays out in a row is one line, its items apart by tabs. Hide leaves out what the document does not need.
PowerPoint: the printed pages as slides

```
var result = HtmlToPowerPoint.RenderFile("report.html");
File.WriteAllBytes("report.pptx", result.Presentation!);   //
result.SlideCount slides
```

The page is laid out and broken into pages as for a PDF, on paper the size of a slide, and each page is a slide:
its text as text boxes, a paragraph a box with the page's line breaks, at the place, size, font, weight, slant and colour the page prints it with: it can be selected, searched and changed;
its tables as PowerPoint tables: columns and rows as the page lays them out, each cell's text where the page prints it, numbers flush right, over the page's own borders and backgrounds;
a link's text as a box of its own, whose click follows the link in a slide show (in a table, the text's own click);
everything else (pictures, backgrounds, borders, charts) as one picture under the text.
Slides are PowerPoint's own 16:9 size unless SlideWidth and SlideHeight say otherwise; a page's own `@page` size is the slide's, as it is the paper's in a PDF. A page break in the page (`break-before: page`) starts a slide.

```
var options = new HtmlPowerPointOptions { SlideWidth = "10in",
SlideHeight = "7.5in" };   // 4:3
```

The text is in the page's fonts: a reader who does not have one of them sees that text in another font, a little wider or narrower. Text the page turns, clips or draws half transparent stays in the slide's picture, as it prints.

The same options as a PDF

Each class has its own options, HtmlExcelOptions, HtmlWordOptions and HtmlPowerPointOptions. Their Page is how the page is loaded and laid out: the PDF's own HtmlPdfOptions, so everything the other guides say about scripts, the network, fonts, consent boxes and advertisements holds here.

```
var page = new HtmlPdfOptions { Network = HtmlNetwork.Any, Ads = HtmlAds.Block, Consent = HtmlConsent.Hide };
var result = HtmlToWord.RenderUrl(new Uri("https://example.com/news"), new HtmlWordOptions { Page = page });
```

What only a PDF has is not read: PDF/A, passwords, compression, the title and author (the file's title is the page's). The results carry the same notes, `Completed`, `Elapsed` and `HttpStatus` as a PDF's.

Without a licence key the three files are written in full and carry no trial watermark; a PDF and a picture carry one.