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`.