Table of Contents

1. PDF generation

`prodockit.pdf` builds a standalone PDF from your Zensical site - the kind of downloadable, submittable document professional and academic reports commonly need alongside the website itself. It reads the same `zensical.toml` your site already has, so there's nothing new to learn or configure beyond a couple of optional settings.

1.1 Requirements

The PDF is built via Pandoc and WeasyPrint, so both need to be installed and on your `PATH`:

```
pip install weasyprint
```

then follow Pandoc's own install instructions for your platform (e.g. `brew install pandoc` on macOS).

Back-of-book indexes additionally need `pymupdf` - `pip install prodockit[index]` (or plain `pip install pymupdf`) - but only if you actually turn `pdf_include_index` on; every other feature on this page needs nothing beyond Pandoc/WeasyPrint above.

1.2 Quick start

From your project root (wherever `zensical.toml` lives):

```
prodockit pdf
```

That's it. Every page in your `nav` is rendered, in order, into `docs/site_documentation.pdf` (or wherever `extra.pdf_output` says - see below), complete with a generated table of contents, chapter numbering, and your site's own admonitions, code blocks, tables, and Mermaid diagrams.

If a build fails (a missing `pandoc`, a WeasyPrint error, and so on), the command prints the underlying error and exits with a non-zero status, rather than silently producing a broken or missing file.

1.3 Configuration

Everything is read from your project's own `zensical.toml` - nothing is passed on the command line beyond, optionally, which config file to use:

```
prodockit pdf --config-file zensical.toml   # -f for short;
this is the default
```

1.3.1 Building a single file

To build a PDF from just one markdown file - a single chapter, say, rather than the whole site - pass `--markdown-file` (`-m` for short), a path relative to `docs_dir`:

```
prodockit pdf --markdown-file chapter1.md   # -m for short
```

This ignores `nav` entirely and renders only that page. Everything else - fonts, page size, margins, `heading_numbering`, and so on - still comes from `zensical.toml` exactly as it would for a full build. The output defaults to that file's own name with a `.pdf` extension inside `docs_dir` (e.g. `docs/chapter1.pdf`) instead of `site_documentation.pdf`, unless `pdf_output` is set, in which case that always wins.

Most of what the PDF needs, it already gets from settings your site likely has for other reasons: `site_name`, `copyright`, `repo_url`, `docs_dir`, `theme.font.text` / `.code`, `theme.icon.admonition`, and `extra_css` - your site's own stylesheet(s) are passed straight through, so a `@media print` rule (e.g. hiding a website-only "Download PDF" link/button, since WeasyPrint always renders in print mode) applies in the PDF too. The rest lives under `[project.extra]`, all optional:

Setting	Default	What it does
`pdf_output`	`"<docs_dir>/site_documentation.pdf"`	Where the PDF is written.
`pdf_copyright`	falls back to `copyright`	Overrides `copyright` for the PDF's own footer only - see Copyright text.
`pdf_page_size`	`"A4"`	Any WeasyPrint-supported CSS page size (`"Letter"`, ...).
`pdf_margin_top` / `_right` / `_bottom` / `_left`	`"2cm"` each	Page margins, as CSS lengths.
`pdf_double_sided`	`false`	Duplex-printing layout - see Double-sided (duplex) printing.

Setting	Default	What it does
`pdf_margin_inner` / `_outer`	`"2cm"` each	Spine-side/fore-edge margins, used instead of `pdf_margin_left` / `_right` when `pdf_double_sided` is on.
`pdf_header_footer_font_size` / `_color` / `_divider_color`	`"10pt"` / `"#555555"` / `"#e2e8f0"`	Running header/footer styling.
`heading_numbering`	`true`	Chapter/appendix numbering on headings and captions.
`reference_style`	`"european"`	`"european"` (tight, single-line citation entries) or `"global"` (double-spaced, hanging indent - the common APA/MLA/Chicago style).
`pdf_include_table_of_contents`	`true`	Whether to generate and insert a table of contents.
`pdf_table_of_contents_title`	`"Table of Contents"`	That page's own heading text.
`pdf_include_index`	`false`	A back-of-book index from every `\index{Term}` marker - see Index (pdf-only). Requires the optional `pymupdf` dependency.
`pdf_index_title`	`"Index"`	That page's own heading text.
`pdf_mmdc_bin`	auto-detected	Path to a mermaid-cli `mmdc` binary, for pre-rendering Mermaid diagrams. Diagrams are left unrendered if none is found.

Setting	Default	What it does
`pdf_tex2svg_script` / `pdf_math_dir`	auto-detected	A local MathJax `tex2svg`-style Node script, for pre-rendering TeX math (WeasyPrint has no JS engine to run MathJax client-side). Formulas are left as literal text if none is found.
`pdf_source_bundle`	`false`	Bundle this repository's own source code into a separate `source_bundle.pdf` - see Bundling source code into a PDF. Only runs for a full, nav-driven build - never for a `--markdown-file`-scoped one.
`pdf_extra_css`	none	A list of `docs_dir`-relative stylesheet paths, same shape as `extra_css` above but meant *only* for the PDF - e.g. a rule that would look wrong on the live website, or one overriding something `extra_css` itself sets (concatenated after it, so it wins the cascade).

A page's own front matter `is_appendix: true` gives it letter-based numbering ("A", "A.1", ...) instead of numeric, matching prodockit.headings' own `appendix_attr` convention. A page's own front matter `recto_title: "Short Title"` overrides its running header text from the *next* page onward - see Double-sided (duplex) printing. Your `nav`'s index page can also use `{WORDCOUNT}` / `{REPOURL}` / `{RELEASE}` / `{{ site_name }}` markers - see Cover page markers.

1.3.2 Web-only / PDF-only content

Mark any block or inline element `{.web-only}` (via `attr_list`) for content meant only for the live website - a "Download PDF" link/button is the common case, since linking to the very PDF you're already reading doesn't make sense once it's embedded in that PDF. `{.pdf-only}` is the opposite: content meant

only for the PDF, e.g. an automated word count or release tag on a cover page that only makes sense in a standalone document.

```
[Download this page as PDF](chapter1.pdf){.web-only}

Word count: {WORDCOUNT}{.pdf-only}
```

`.web-only` needs no configuration - `prodockit.pdf`'s own generated CSS always hides it, in every build, whether you're using `prodockit pdf` or calling `build_pdf()` directly. `.pdf-only` is the one half prodockit can't provide automatically (its own CSS has no reach into your live website), so add this one line to your project's own website stylesheet:

```
.pdf-only {
  display: none !important;
}
```

(see this project's own `docs/stylesheets/extra.css` for a working example). If your project doesn't yet use `.pdf-only` for anything, there's nothing to add until it does.

1.3.3 Copyright text

`copyright` (a native Zensical setting, not one of prodockit's own - see Building a single file above) feeds both the live website's own footer *and* the PDF's running footer, by default - whatever you set once shows, unchanged, in both places.

`pdf_copyright` (under `[project.extra]`) overrides it for the PDF's own footer only, leaving the website's copyright text completely untouched either way. Unset by default, so an existing project's PDF and website keep matching unless you deliberately add it - useful when you want the PDF to show something the website version wouldn't make sense showing (or vice versa), without having to keep two near-identical strings in sync by hand.

Both `copyright` and `pdf_copyright` accept a real HTML fragment, the same as Zensical's own website-side `copyright` setting already does - a real `<a href="../...">` link renders as a real, clickable link in the PDF too, not flattened to plain text. Use a real `<br>` for a forced line break. The obvious use: crediting the tools your report was built with, on their own second line, without touching the copyright/licence text itself:

```
[project.extra]
pdf_copyright = 'Author: Jane Doe. Licensed under the MIT
License.<br>Made with <a href="https://
```

```
zensical.org/">Zensical</a> and <a href="https://
buckwem.github.io/prodockit-extensions/">prodockit</a>.'
```

Under the hood, this isn't a generated CSS `content: "..."` string the way most other running header/footer values are (e.g. `site_name`) - a CSS content string can't contain a real link at all. Instead, prodockit writes `copyright`/`pdf_copyright` once as a real (hidden until the footer clones it) HTML element, and the PDF's own generated CSS pulls it into the footer on every page via CSS Paged Media's `position: running()`/`content: element()` - confirmed directly against real WeasyPrint output. You don't need to know any of this to use the setting; it only matters if you're digging into why the PDF's footer behaves differently from a plain `content` string elsewhere in `prodockit.pdf.css`.

The equivalent website-side credit (if your project wants a "Made with Zensical and *X*" line on the live site too) isn't a prodockit setting at all - prodockit has no reach into the website's own Jinja partials. It's a Zensical theme override instead: with `custom_dir` set (see Zensical's own docs), drop your own `overrides/partials/copyright.html` based on the bundled version, adding a second credit line after the existing "Made with Zensical" one.

1.3.4 Cover page markers

Drop any of these literal strings into your `nav`'s index page - typically a cover page, e.g. wrapped in `{.pdf-only}` as in the example above - and `prodockit pdf` substitutes a real value once that page's HTML exists, no configuration needed:

Marker	Becomes
`{WORDCOUNT}`	The site-wide word count (the same value a `{{ word_count }}` website macro variable would show), so a submission's PDF and its live website page never disagree.
`{REPOURL}`	The git-detected repo URL (the same value `{{ repo_url }}` gives a website macro).
`{RELEASE}`	The latest published GitHub/GitLab release tag (e.g. `v1.2.0`). The *whole line* containing this marker is dropped instead if there isn't one - most projects never publish a release at all, so nothing shows a bare `"Release: "` label by default.
`{{ site_name }}`	Your project's own `site_name`, substituted literally - `prodockit pdf` never evaluates Jinja, so the exact same `{{ site_name }}` text a website macro variable uses works here too, one line of markdown for both outputs.

Skipped entirely for a `--markdown-file`-scoped build, or if your `nav` has only one page - there's no separate "cover" vs "content" to compute a word count from either way.

1.3.5 Sideways tables

A table too wide for a portrait page - a wide reference table, say - can be printed sideways instead: wrap it (and its own caption) in `<div class="prodockit-table-rotated" markdown="1">`, using `md_in_html` (the `markdown="1"` is required - without it, the table inside is left as literal, unconverted text):

```
<div class="prodockit-table-rotated" markdown="1">

**A wide reference table**

| ID {: width="15%" } | Description {: width="70%" } | Due {:
width="15%" } |
|---|---|---|
| 1 | ... | Q1 |
```

The table prints on its own landscape-sized page(s) - same configured page size, width/height swapped - spanning multiple pages with its header row repeated exactly like any other table (see prodockit.tables for the `width` syntax above, which works exactly the same way here). A page break is always forced immediately before and after the block, so it never shares a page with anything else.

This isn't a CSS `transform: rotate()` - confirmed directly, that clips a table to a single page instead of splitting it, and pushes its heading row and first few rows off-page entirely, before any of this was written. Instead, WeasyPrint lays the table out normally, unrotated, on its own landscape page; a rotation is applied afterwards, directly on the finished PDF's own per-page display flag, once WeasyPrint is done - see `prodockit.pdf.rotate` for that step, always run automatically as the last part of a build (a no-op if nothing used `prodockit-table-rotated`).

This is PDF-only - the same wrapped table renders as a completely normal, unrotated table on the live website, the same way `.web-only` content elsewhere in this project only ever affects one of the two outputs.

1.3.6 Double-sided (duplex) printing

Set `pdf_double_sided = true` under `[project.extra]` for a document meant to be printed and bound on both sides - a book or handbook, rather than a web-printed report. Left-hand (verso) and right-hand (recto) pages mirror their header/footer content and page margins, and every numbered heading starts its own recto page:

```
[project.extra]
pdf_double_sided = true
pdf_margin_inner = "3cm"   # spine side - wider, to leave room
```

```
for binding
pdf_margin_outer = "1.5cm" # fore-edge (outer) side
```

`pdf_margin_inner` / `pdf_margin_outer` replace `pdf_margin_left` / `_right` once `pdf_double_sided` is on - the "inner" (spine) side is the left margin on a recto page but the right margin on a verso page, and vice versa for "outer" (fore-edge), so a single pair of settings covers both without you having to think about which physical side is which for any given page. `pdf_margin_top` / `_bottom` are unaffected either way.

Every corner of the running header/footer mirrors between recto and verso, keeping the chapter title and page number on the outer, fore-edge corner and the site name/copyright on the inner, spine-side corner, whichever physical side that happens to be for a given page - confirmed directly, by rendering a real double-sided document and inspecting facing pages, that this is how it actually looks.

Every numbered heading (chapter start) also always starts on its own recto page - a blank page is inserted automatically if the previous chapter ended on an odd page, exactly like the blank pages you'd expect at the start of each chapter in a real printed book. This needs no configuration; it's part of what `pdf_double_sided` turns on.

A `prodockit-table-rotated` landscape page's own rotation direction also alternates by its own final page position once `pdf_double_sided` is on - 270 degrees (anticlockwise) on a recto page, 90 (clockwise) on a verso page - since the spine sits on the opposite physical side either way, and the rotation has to compensate to keep the landscape content's own top edge facing the fore-edge rather than the spine. With `pdf_double_sided` off, every rotated page always rotates 270 degrees, as before this option existed.

A page's own front matter `recto_title: "Short Title"` overrides that page's own running header text with a shorter title, from the *next* page onward (the heading's own page still shows its full title) - handy when a chapter's real title is too long to comfortably fit the running header:

```
---
recto_title: "Ch. 1"
---

# Chapter One: A Rather Long Title That Wouldn't Fit In A
Running Header
```

This setting is meaningful whether or not `pdf_double_sided` is on - the running chapter title appears in the header either way, just in a different corner.

1.3.7 Bundling source code into a PDF

Set `pdf_source_bundle = true` under `[project.extra]` to also produce

`source_bundle.pdf` - a separate PDF, in the top-level project directory (not `docs_dir`), containing every text/source file `.gitignore` doesn't exclude, one file per page:

```
[project.extra]
pdf_source_bundle = true
```

This is unrelated to the rest of `prodockit.pdf` - there's no Markdown involved at all, just this repository's own raw source files, so it skips Pandoc entirely and hands a small, self-contained HTML document straight to WeasyPrint. Only runs for a full, nav-driven build (`prodockit pdf` with no `--markdown-file`) - a single-page build has no reason to also rebuild a whole-repository artifact every time.

Every file is rendered in 8pt Courier with wrapped lines (a genuinely long line wraps rather than running off the page or getting cut off), starting on its own page, with a running header (the project's own `site_name` on the left, that page's own file path on the right) and a "Page N of M" footer. Which files are included is decided by `.gitignore` - both already-tracked files and untracked-but-not-yet-added ones count, as long as `.gitignore` doesn't exclude them - and by content (anything that isn't valid UTF-8 text, e.g. an image or compiled binary, is silently skipped, not by file extension).

A few classes of vendored, never-author-written content are always excluded too, regardless of `.gitignore` - none of this is a project-level setting, so there's no `zensical.toml` knob to opt any of it back in:

- Any directory literally named `.icons` (e.g. a `custom_icons` directory, per pymdownx.emoji's own convention).
- Any directory literally named `styles` (e.g. a Vale `StylesPath`, holding downloaded rule packs).
- Common dependency lockfiles by exact file name - `package-lock.json`, `npm-shrinkwrap.json`, `yarn.lock`, `pnpm-lock.yaml`, `Pipfile.lock`, `poetry.lock`, `Cargo.lock`.

These are typically *tracked* (needed for the site/PDF to build at all), so `.gitignore` alone can't keep a vendored icon pack's hundreds of SVGs, a Vale style pack, or a multi-thousand-line lockfile out of the bundle.

Scripting this outside `prodockit pdf` (e.g. from a different build tool)? See `prodockit.pdf.source_bundle` below.

1.3.8 Back-of-book index

A traditional, two-column back-of-book index, generated from every `\index{Term}` marker (the `prodockit.index` extension) via `pdf_include_index` / `pdf_index_title` - PDF-only, there's no equivalent on the live website. Marking terms, turning the setting on, and what the generated

page itself looks like are all covered together in Index (pdf-only), since (unlike every other feature on this page) marking and generation are two different extensions - see that page for the full syntax and worked examples. If you're scripting your own build pipeline rather than using `prodockit pdf` / `zensical.toml`, see `prodockit.pdf.index` below for exactly how its two-pass build works.

1.3.9 Python API

If you're scripting your own build pipeline and want to call into `prodockit.pdf` directly rather than through `zensical.toml`, `build_pdf()` is a one-call function you can import instead of shelling out to the `prodockit` command:

```
from prodockit.pdf import Page, build_pdf

build_pdf(
    [
        Page(docs_rel_path="index.md",
html=rendered_html["index.md"], is_index=True),
        Page(docs_rel_path="chapter1.md",
html=rendered_html["chapter1.md"]),
        Page(docs_rel_path="chapter2.md",
html=rendered_html["chapter2.md"]),
    ],
    "dist/report.pdf",
    site_name="My Report",
    copyright_text="Copyright 2026 Jane Doe",
)
```

`Page.html` is a page's own already-rendered HTML (not yet fixed up for Pandoc - `build_pdf` does that internally for you); `Page.docs_rel_path` is that page's path relative to your docs directory, e.g. `"starthere/installtooling.md"`. Mark exactly one page `is_index=True` if you have a dedicated cover/title page - its headings are treated as decorative rather than real chapters. `build_pdf` raises `prodockit.pdf.PdfBuildError` with the underlying error message attached if the build fails.

`build_pdf`

```
build_pdf(
    pages: list[Page],
    output_path: str,
    *,
    docs_dir: str = "docs",
    extra_css: str = "",
```

```
    repo_url: str = "",
    admonition_icon_config: dict[str, Any] | None = None,
    icon_registry: dict[str, str] | None = None,
    render_mermaid: Callable[[str], str | None] | None = None,
    main_font: str = "Inter",
    mono_font: str = "JetBrains Mono",
    copyright_text: str = "",
    site_name: str = "",
    page_size: str = "A4",
    margin_top: str = "2cm",
    margin_right: str = "2cm",
    margin_bottom: str = "2cm",
    margin_left: str = "2cm",
    double_sided: bool = False,
    margin_inner: str = "2cm",
    margin_outer: str = "2cm",
    header_footer_font_size: str = "10pt",
    header_footer_color: str = "#555555",
    header_footer_divider_color: str = "#e2e8f0",
    reference_style_global: bool = False,
    reference_spacing_european: str = "-0.8em",
    reference_indent_global: str = "1.27cm",
    reference_spacing_global: str = "2em",
    heading_numbering_enabled: bool = True,
    mathjax_available: bool = False,
    math_dir: str | None = None,
    tex2svg_script: str = "",
    include_table_of_contents: bool = True,
    table_of_contents_title: str = "Table of Contents",
    work_dir: str | None = None,
    keep_work_dir: bool = False,
) → None
```

Its two required arguments are the page list and `output_path` (where the PDF gets written - any path, absolute or relative; its parent directory must already exist). Everything else mirrors the `zensical.toml` settings above one-for-one, plus a few options only meaningful from Python:

Content

- `docs_dir`: your project's docs root, used to resolve each page's own relative image/link references.
- `extra_css`: your own website stylesheet's content, layered underneath the CSS `build_pdf` generates, so its own `!important` rules can still override a website-only style that doesn't make sense in a paginated PDF.

- `admonition_icon_config` / `icon_registry`: give an admonition its own icon in the PDF (Zensical's admonition HTML has none by default outside a website) - see prodockit.pdf.icons.
- `render_mermaid`: called with each Mermaid diagram's own source text, should return an image path/`data:` URI or `None` if rendering failed - see prodockit.pdf.mermaid for a ready-made renderer.

Working files

`pandoc` needs a few intermediate files on disk (the concatenated HTML, the generated Lua filter, the compiled CSS) - written under `work_dir` if given, or a fresh temporary directory otherwise (always cleaned up regardless of `keep_work_dir` - there's no path left to inspect afterwards). `keep_work_dir=True` with an explicit `work_dir` leaves those files in place, useful for checking exactly what Pandoc/WeasyPrint received when a build succeeds but the PDF looks wrong.

`Page`

```
@dataclass
class Page:
    docs_rel_path: str
    html: str
    is_index: bool = False
    is_appendix: bool = False
    recto_title: str | None = None
```

One page to include in the PDF. `html` is this page's own already-rendered HTML (not yet fixed up for Pandoc - `build_pdf` does that for you). `is_appendix` gives this page's first heading a letter instead of a number ("A", "A.1", ...) if you enable `heading_numbering_enabled`. `recto_title`, if given, overrides this page's own running header text from the next page onward - see Double-sided (duplex) printing.

`PdfBuildError`

Raised by `build_pdf` when the underlying `pandoc` invocation fails. `returncode` and `stderr` are attached for logging or troubleshooting.

1.3.10 Building your own pipeline

If `build_pdf`'s shape doesn't fit how your project is put together - e.g. you want to fix up and inspect each page's HTML individually before concatenating them yourself, or drive `pandoc` with your own extra arguments - the pieces `build_pdf` is built from are all directly importable too:

`prodockit.pdf.html`

Fixes up one page's rendered HTML for Pandoc's own reader/writer quirks - the biggest piece, and what `build_pdf` calls per page internally.

`prodockit.pdf.lua`
Generates the Pandoc Lua filter (chapter numbering, caption ordering, tab reconstruction, math).

`prodockit.pdf.css`
Generates the compiled CSS a paginated PDF needs on top of your website's own stylesheet.

`prodockit.pdf.icons`
Resolves an admonition type to its accent-coloured icon SVG.

`prodockit.pdf.mermaid`
Pre-renders a Mermaid diagram to a static SVG via `mermaid-cli`.

`prodockit.pdf.html`

```
fix_up_page_html(
    html: str,
    *,
    current_docs_rel_path: str,
    docs_dir: str,
    page_anchor_map: dict[str, str],
    is_index: bool = False,
    is_appendix: bool = False,
    repo_url: str = "",
    admonition_icon_config: dict[str, Any] | None = None,
    icon_registry: dict[str, str] | None = None,
    render_mermaid: Callable[[str], str | None] | None = None,
) → str
```

Applies every HTML fixup a page needs - a raw inline `<svg>` (icons, emoji, diagrams) doesn't survive Pandoc's HTML-to-HTML round trip through to WeasyPrint at all; Pandoc's `Para` AST node has no attribute field, silently dropping any `id` / `class` a `<p>` carries; a caption meant to appear *before* its table/figure always ends up *after* it once Pandoc's own HTML writer re-serializes the page; a multi-page site concatenated into one PDF document needs cross-page links rewritten to in-document anchors; and more.

A few standalone helpers, used once up front across every page rather than from within `fix_up_page_html` itself:

Function	Purpose
`build_page_anchor_map(md_files)`	Maps each page to a deterministic in-document anchor id, for cross-page link rewriting.

Function	Purpose
`build_virtual_page_map(md_files)`	Same mapping, keyed by Zensical's own clean-URL "virtual directory" path instead of the raw filename.
`virtual_page_path(docs_rel_path)`	The clean-URL virtual directory a single page's path maps to.
`to_base64_data_uri(img_src, base_dir)`	Resolves a (possibly relative) image src to an absolute path and returns it as a base64 `data:` URI.

`prodockit.pdf.lua`

```
build_lua_filter(
    heading_numbering_enabled: bool,
    mathjax_available: bool,
    math_dir: str,
    tex2svg_script: str,
) → str
```

Generates the complete Pandoc Lua filter source as a string - write it to a file and pass it to Pandoc with `--lua-filter=`.

`prodockit.pdf.css`

```
build_css(
    main_font: str,
    mono_font: str,
    copyright_text: str,
    site_name: str,
    page_size: str = "A4",
    margin_top: str = "2cm",
    margin_right: str = "2cm",
    margin_bottom: str = "2cm",
    margin_left: str = "2cm",
    double_sided: bool = False,
    margin_inner: str = "2cm",
    margin_outer: str = "2cm",
    header_footer_font_size: str = "10pt",
    header_footer_color: str = "#555555",
    header_footer_divider_color: str = "#e2e8f0",
    reference_style_global: bool = False,
    reference_spacing_european: str = "-0.8em",
    reference_indent_global: str = "1.27cm",
```

```
    reference_spacing_global: str = "2em",
) → str
```

Generates the complete compiled CSS a PDF needs, layered on top of your own website stylesheet. Covers running header/footer boxes, footnotes anchored via `float: footnote`, and page-break tuning for headings, paragraphs, tables, code blocks, figures/captions, admonitions, tabbed sets, and grid cards - every rule here exists because a plausible-looking print CSS rule forced a real, confirmed blank-page gap or an orphaned heading in WeasyPrint specifically.

`prodockit.pdf.icons`

```
admonition_icon_svg(
    adm_type: str,
    admonition_icon_config: dict[str, Any] | None,
    icon_registry: dict[str, str],
) → str | None
```

Resolves an admonition type (note, warning, tip, ...) to its accent- coloured icon SVG markup. `discover_icon_dirs()` / `build_icon_registry()` find and index the Material/Zensical/FontAwesome `.icons` directories your project's own icon shortcodes already draw from.

`prodockit.pdf.mermaid`

```
render_mermaid_diagram(
    diagram_source: str,
    mmdc_bin: str,
    output_dir: str,
    index: int,
    timeout: int = 60,
) → str | None
```

Pre-renders one Mermaid diagram's source to a static SVG via a local mermaid-cli install, since WeasyPrint has no JS engine to run Mermaid.js client-side.

`prodockit.pdf.source_bundle`

```
build_source_bundle(
    output_path: str = "source_bundle.pdf",
    *,
    root: str = ".",
    report_name: str = "",
    page_size: str = "A4",
```

```
    work_dir: str | None = None,
    keep_work_dir: bool = False,
) → int
```

Unrelated to the rest of `prodockit.pdf` - see Bundling source code into a PDF above for what this builds; `pdf_source_bundle` is the same function called for you. Returns how many files ended up in the bundle. `root` is both where `git ls-files --cached --others --exclude-standard` looks for files to include and where a relative `output_path` is written to. `work_dir` / `keep_work_dir` mirror `build_pdf()`'s own pair - a place to put (and optionally keep) the intermediate HTML `weasyprint` actually renders, handy when the output looks wrong. Raises `prodockit.pdf.source_bundle.SourceBundleError` (the underlying `git` / `weasyprint` exit code and stderr attached, where applicable) if either invocation fails.

`prodockit.pdf.index`

```
mark_index_terms(html: str) → tuple[str, list[str],
list[bool]]
extract_term_pages(pdf_path: str, occurrence_count: int) →
dict[int, int | None]
build_index_entries(
    terms: list[str],
    occurrence_pages: dict[int, int | None],
    code_flags: list[bool] | None = None,
) → dict[str, IndexEntry]
render_index_content(entries: dict[str, IndexEntry], level: int
= 1) → str
format_pages(pages: list[int]) → str
```

The three main pieces `build_pdf()`'s own `include_index` calls, in order, for its two-pass build - see Index (pdf-only) for the feature itself, and this module's own docstring for why a real two-pass build (rather than CSS's own `target-counter()`) is what backs it. `mark_index_terms()` finds every `[prodockit.index](extensions/index-terms.md)` `<span class="index">` and inserts a unique, near-invisible text marker after each occurrence, returning the terms found in order - a flat `"Term"` or, for a hierarchical `\index{Parent!Child}`, `"Parent!Child"` - alongside whether each one was a code-styled term. `extract_term_pages()` needs the optional `pymupdf` dependency (only imported here, so only required if you actually call this function) - raises a plain `ModuleNotFoundError` with a clear install message if it isn't installed. `build_index_entries()` builds `mark_index_terms()`'s flat/hierarchical paths (plus its own `code_flags`) into a nested tree of `IndexEntry(display: str, pages: list[int], children: dict[str,`

`IndexEntry], code: bool = False)` nodes, alphabetised ignoring leading punctuation (see Index (pdf-only) above). `render_index_content()` walks that tree, grouping top-level entries under a bold letter heading per first letter (`build_css()`'s own compiled CSS lays the whole thing out in two columns), indenting each deeper level, and wrapping a code-styled entry's own text in `<code>` - matching a traditional printed book's own back-of-book index - using `format_pages()` to collapse each entry's own consecutive pages into en-dash ranges.

1.4 Limitations and workarounds

`prodockit.pdf` pipes your site's own rendered HTML through Pandoc and WeasyPrint to produce the PDF - two tools with their own reader/writer quirks and no JS engine, quite different from a browser rendering your live website. This section documents the confirmed limitations that shape `prodockit.pdf.html` / `.lua` / `.css`, and the workaround each one gets, so a project hitting unexpected PDF output has somewhere to check *why* before assuming it's a bug.

No JS engine (WeasyPrint can't run client-side JS)

- Mermaid diagrams: no JS engine to run Mermaid.js client-side → each ```` ```mermaid ```` fence is pre-rendered to a static SVG via `mermaid-cli` before Pandoc ever sees it (see `prodockit.pdf.mermaid`).
  - Mermaid's default node/edge labels are HTML `<foreignObject>` content, which WeasyPrint's SVG renderer can't display (text silently vanishes) → `htmlLabels` is forced off, so Mermaid emits plain SVG `<text>` / `<tspan>` labels instead.
- Math (`$...$` / `$$...$$`, `pymdownx.arithmatex`): no JS engine to run MathJax client-side → each formula is pre-rendered to a static SVG via a Lua filter `Math()` handler piping to a `tex2svg` script (see `prodockit.pdf.lua`).
  - `arithmatex`'s *generic*-mode math (`<div class="arithmatex">` / `<span class="arithmatex">`) has no native Math AST node in Pandoc's *HTML* reader (unlike its *markdown* reader, which recognises `$...$` as a real Math node) → matched by CSS class in dedicated `Div()` / `Span()` Lua handlers instead of the `Math()` function.
- A live site's own header repo widget (release/version info) fetches it client-side via JS; Pandoc/WeasyPrint has no JS engine to do the same → a project embedding similar info in a PDF cover page needs to fetch and substitute it directly before the page ever reaches `build_pdf()`.

Multi-page → single-document concatenation

- A link that resolves fine on a website (a separate page) has nothing to point at once every page is concatenated into one PDF - Pandoc treats it as a link to an external file at whatever absolute path the PDF happened to be built from → rewritten to in-document anchors instead (see

`build_page_anchor_map()` / `build_virtual_page_map()` in `prodockit.pdf.html`).

- Local image/file references can't depend on relative paths resolving correctly from wherever Pandoc happens to run in a standalone document → base64-embedded as `data:` URIs directly in the HTML (see `to_base64_data_uri()`).
- A PDF has no light/dark toggle to make the mkdocs-material/Zensical `#only-light` / `#only-dark` / `#gh-light-mode-only` / `#gh-dark-mode-only` image convention meaningful → the `#only-dark` / `#gh-dark-mode-only` half of a pair is dropped entirely rather than embedded, since `to_base64_data_uri()`'s resulting `data:` URI has no trace of the fragment left for any stylesheet to hide it by (this used to leave both halves of every such pair permanently visible, stacked one after the other).
- A CSS `url()` reference (e.g. in your own website stylesheet, passed via `extra_css`) resolved relative to wherever Pandoc runs is meaningless (and can leak a local file path) to anyone reading the PDF → a project passing its own CSS through `extra_css` should rewrite any such reference to a stable, absolute URL (e.g. the file's canonical GitHub/ GitLab "blob" URL) before handing it to `build_pdf()`.

Raw `<svg>` doesn't survive Pandoc's HTML→HTML round trip through to WeasyPrint at all (confirmed directly, isolated test) - affects admonition icons, grid-card title icons, and pre-rendered Mermaid diagrams alike → every `<svg>` is converted to a base64 `data:` URI `<img>` instead (see `prodockit.pdf.icons`).

Content tabs (`pymdownx.blocks.tab`): each tab's label renders as an inline `<label>` sibling with no block boundary between them; Pandoc's HTML reader merges adjacent inline-level siblings with no block boundary into one `Plain` block, collapsing every label in a tabbed-set into one unseparated run of text with no way to recover the boundary afterward in a Lua filter → each `<label>` is rewritten into its own `<p>` *before* Pandoc's reader ever sees it, then the Lua filter reconstructs the `tabbed-set` / `tabbed-labels` / `tabbed-content` structure into a `tabbox` shape (see `prodockit.pdf.lua`).

Figure/table captions in "prepend" position: Pandoc's `Figure` AST node stores `Caption` and content as two separate, independently-typed fields rather than ordered children reflecting DOM position, and Pandoc's own HTML writer always re-emits a `Figure`'s `<figcaption>` *after* its content when serializing back to HTML - confirmed directly (a `<figcaption>` placed first in source HTML still comes out last from Pandoc's own HTML writer), discarding "prepend" positioning entirely regardless of input order → any figure/table whose caption comes first is retagged from `<figure>` to `<div>` before Pandoc parses it (a `Div`'s children *are* emitted in original document order), with the `<figcaption>` unwrapped into the div's first child block.

Pandoc's native `Para` AST node has no attribute field at all (unlike `Div` / `Header` / `CodeBlock` / `Table` / `Figure`, which all carry one) - confirmed: a `<p id="..." class="...">` comes out the other end as a bare `Para` with both the

`id` and the `class` silently gone, with no error. This is exactly the shape every `attr_list` citation/acronym/glossary definition takes (see prodockit.citations/ prodockit.glossary) → any `<p>` carrying an `id` or `class` is retagged to a `<div>` instead (which Pandoc's reader does preserve attributes on).

Lightbox-wrapped images: an `<a class="glightbox">` wrapping an `<img>` resolves its `href` one directory level differently than the `<img>`'s own `src` (an artifact of Zensical's URL cleaning), which Pandoc/WeasyPrint then fails to resolve as a broken link → the lightbox `<a>` is unwrapped, leaving just the `<img>`.

Embedded `<iframe>` (e.g. a YouTube video): left as-is, produces a stray unwanted heading in the compiled PDF (WeasyPrint attempts to fetch the iframe's `src`, and something in that response ends up parsed as real page content) → replaced with a link-styled reference to the video instead - a static PDF can't embed a live video player regardless.

No Jinja evaluation: Pandoc/`prodockit.pdf` never evaluates Jinja - a `{{ site_name }}` placeholder that resolves via macro evaluation on the live site (see prodockit.zensical_macros) is left as literal text unless a project substitutes it directly in its own page HTML before handing it to `build_pdf()`.

No `.md-typeset` wrapper: unlike a Zensical website, Pandoc's HTML output has no `.md-typeset` wrapper element, so website CSS rules scoped to `.md-typeset ...` (reference/acronym/glossary spacing, a `.screenshot` class, and so on) never match in the PDF → `prodockit.pdf.css` duplicates the relevant rules as plain, unscoped selectors instead (see `prodockit.pdf.css`).

Footnotes: Pandoc's default behaviour collects every footnote in the whole document into one section at the very end of the PDF, rather than at the bottom of the page it's referenced on like a printed book → a Lua filter handler replaces each footnote reference with an inline span styled via CSS `float: footnote` instead (see `prodockit.pdf.lua` / `prodockit.pdf.css`).

WeasyPrint's CSS Grid support is too limited to trust for an actual side-by-side multi-column layout → a Zensical grid-cards block renders as one full-width stacked box per row instead of a real grid.

`<figcaption>` centering doesn't extend to its sibling `<img>`: WeasyPrint's UA stylesheet centers `<figcaption>` text by default via `text-align`, but that doesn't affect the sibling image, which stays left-aligned and visibly misaligned under its own caption → explicit CSS centers the whole figure/wrapping element instead.

Two-space vs. four-space nested-list indentation discrepancy: Pandoc's markdown reader nests a sub-list at just 2-space indentation (no 4-space requirement), unlike Python-Markdown's stricter 4-space rule. Only relevant if you write markdown by hand for a *separate* Pandoc-only input rather than feeding `prodockit.pdf` your already-rendered HTML (the normal, documented path) -

the HTML-based pipeline sidesteps this entirely, since Pandoc's HTML reader has no such indentation rule to begin with.

General "every markdown extension needs its own bespoke translation" limitation, and why `prodockit.pdf` avoids it: Pandoc is a completely different parser from Python-Markdown/Zensical, so a pipeline built around hand-translating each markdown feature into a Pandoc-compatible dialect needs a new bespoke translation for every extension a project enables (admonitions, tabs, grid cards, captions, `attr_list` spans, `{% if %}` conditionals, and so on) - fragile, and it grows without bound. `prodockit.pdf` sidesteps this by feeding Pandoc your site's own already- rendered HTML (via `zensical.markdown.render.render()`, the same pipeline that builds your live website) instead of raw markdown - Pandoc's own HTML reader already understands standard HTML correctly, with no per-feature translation needed. The fixups documented above are what's left after that: genuine gaps in Pandoc/WeasyPrint's own HTML handling, not gaps in markdown-dialect translation.

`prodockit.bibliography` is a partial exception to this pattern, worth flagging explicitly: resolving `\citebib{id}` / `\bibliography` itself calls out to a *separate*, independent `pandoc --citeproc` invocation at markdown-render time (see prodockit.bibliography) - unrelated to, and already finished well before, `prodockit.pdf`'s own `pandoc --pdf-engine=weasyprint` call below. By the time `prodockit.pdf` sees the page, citations and the reference list are already resolved, ordinary HTML - `id`-bearing `<div>`s and `<a>` links like any other content - so none of the fixups documented above apply to it specially; a build using both ends up invoking Pandoc twice, for two entirely unrelated reasons.

1.5 Status

No formal, versioned public API stability contract yet (see prodockit-extensions#7).