prodockit

A docs-as-code workflow for professional and academic documentation, built on Zensical.

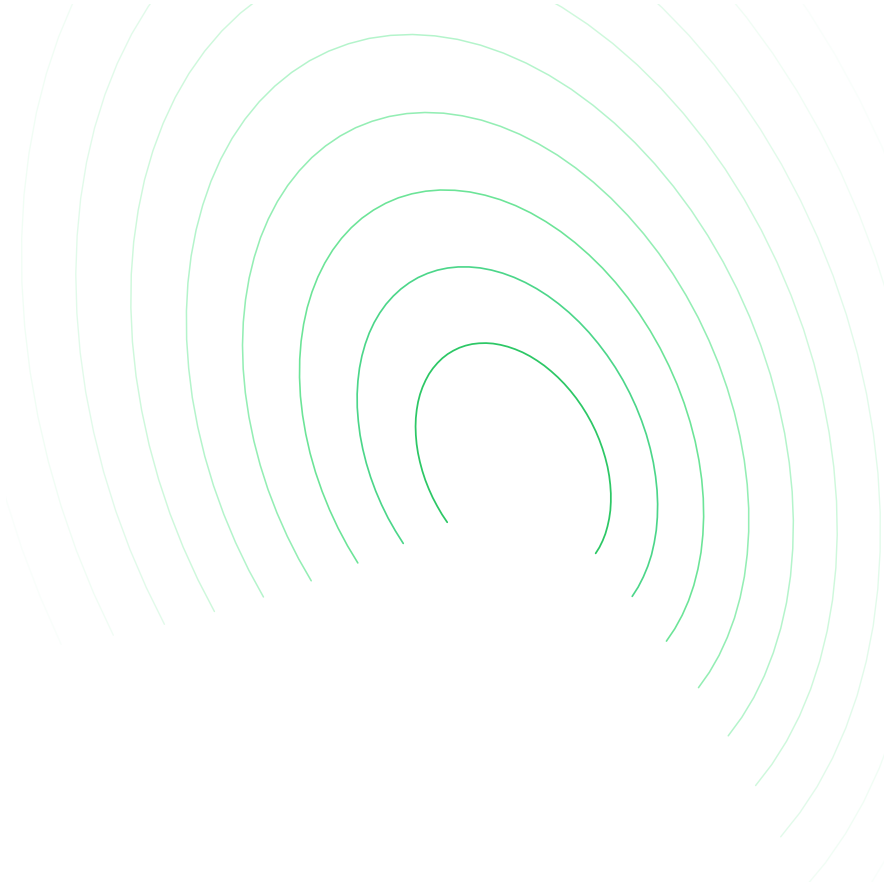

Table of Contents

1. Introduction

prodockit is a family of extensions for Zensical needed for professional and academic documentation - the pieces a report, dissertation, or technical document commonly needs that Zensical doesn't provide out of the box: section cross-references, bibliography/citation handling, a glossary, and a Pandoc/WeasyPrint PDF pipeline for the downloadable, submittable document these usually need alongside the website itself.

Most of prodockit is Python-Markdown extensions, that are enabled in `zensical.toml`. prodockit.pdf is a command-line tool instead, since a PDF build pipeline isn't a Markdown syntax extension - no Python required, it reads the same `zensical.toml` too. In addition, there are a set of website macros to help use the features of prodockit.

It's a kit for professional documentation, built on Zensical's own Markdown and Pandoc/WeasyPrint PDF pipeline.

1.1 Extensions

Extension	Description
prodockit.headings	Gives every heading an id and a hierarchical section number ("1", "1.1", "1.2", "2", ...).
prodockit.refs	`\ref{id}` section cross-references, resolving to the target's current number - similar in spirit to LaTeX's `\ref`.
prodockit.citations	Define a source once, cite it by key anywhere with `\cite{id}` - auto-generates the bracketed, linked citation text.
prodockit.glossary	Define a term once (an acronym expansion, a glossary entry), insert it by id anywhere with `\gls{id}` - similar in spirit to LaTeX's `glossaries` package.
prodockit.tables	Percentage or fixed column widths on a table, via a `width` attribute already attachable to a header cell with `attr_list`.
prodockit.bibliography	An alternative to `prodockit.citations`: define sources in a BibTeX/BibLaTeX `.bib` file and format `\citebib{id}` /the reference list in any Citation Style Language style, via Pandoc's own `--citeproc`.

1.2 PDF generation

Module	Description
prodockit.pdf	Builds a standalone PDF from your site, via Pandoc and WeasyPrint. Run `prodockit pdf` from your project root - everything is read from your own `zensical.toml`.

```
prodockit pdf
```

See PDF generation for the `zensical.toml` settings it reads, and for the Python API if you're scripting your own build pipeline instead.

1.3 Website macros

Module	Description
prodockit.zensical_macros	Jinja variables/macros for Zensical's macros plugin: a site-wide word count, the git-detected repository URL, chapter/appendix numbering that continues across pages, and reference/acronym/glossary spacing that matches `prodockit.pdf`'s own PDF output.

```
[project.markdown_extensions.zensical.extensions.macros]
modules = ["prodockit.zensical_macros"]
```

See Website macros for the full variable/macro list.

1.4 Quick example

```
import markdown

html = markdown.markdown(
    """
# Introduction

See \\ref{background} for context.\\cite{skou2023}

## Background

...

Skoulikari, A. (2023) *Learning Git*.
{: #skou2023 data-cite-text="Skoulikari, 2023" }
```

```
""",
    extensions=["attr_list", "prodockit.headings",
"prodockit.refs", "prodockit.citations"],
)
```

`\ref{background}` resolves to `1.1` - the current section number of the heading it points to - and `\cite{skou2023}` resolves to `[Skoulikari, 2023]`, linked to that source's own paragraph. Both stay correct if headings/sources are reordered, since resolution happens fresh on every conversion.

1.5 Status

Early, but functional. `prodockit.headings`, `prodockit.refs`, `prodockit.citations`, `prodockit.glossary`, `prodockit.tables`, `prodockit.bibliography`, `prodockit.pdf`, and `prodockit.zensical_macros` are implemented and tested. `prodockit.citations` itself still doesn't auto-generate a full references list from structured bibliographic data (see prodockit.citations) - prodockit.bibliography is the alternative that does, via a `.bib` file and Pandoc's own `--citeproc`. None of `prodockit.bibliography` / `prodockit.pdf` / `prodockit.zensical_macros` has a formal, versioned public API yet (see prodockit-extensions#7). See the Release Notes for what's landed so far.

2. Installation

2.1 From PyPI

```
pip install prodockit
```

`prodockit` depends on `Markdown` (>= 3.4), `zensical`, and `beautifulsoup4` (>= 4.12, used by `prodockit.pdf` - see PDF generation).

2.2 Enabling an extension

Each prodockit extension is registered as a standard Python-Markdown extension under the `markdown.extensions` entry point group, so it can be enabled by name, the same way you'd enable a built-in extension like `toc` or a `pymdownx` one:

```
import markdown

html = markdown.markdown(
    text,
    extensions=["prodockit.headings", "prodockit.refs",
"prodockit.citations", "prodockit.glossary"],
)
```

Or, for a Zensical project, in `zensical.toml` alongside the built-in and `pymdownx` extensions. Unlike `pymdownx`'s and Zensical's own namespaces, Zensical doesn't hoist a nested `prodockit.headings` table into that dotted extension name, so each one needs a quoted key instead:

```
[project.markdown_extensions."prodockit.headings"]
[project.markdown_extensions."prodockit.refs"]
[project.markdown_extensions."prodockit.citations"]
[project.markdown_extensions."prodockit.glossary"]
```

See each extension's own page for its options and for how to share a registry across multiple pages of a site build:

- prodockit.headings
- prodockit.refs
- prodockit.citations
- prodockit.glossary

`prodockit.pdf` is different: it isn't a Python-Markdown extension (no

`markdown.extensions` entry point, nothing to add to `zensical.toml`) - it's a plain function library for a separate PDF-generation build step. See PDF generation for how it's used.

2.3 Development install

```
git clone https://github.com/buckwem/prodockit-extensions
cd prodockit-extensions
python -m venv .venv
source .venv/bin/activate
pip install -e ".[dev]"
pytest
```

`zensical` is a core dependency, so `zensical serve` is available as soon as `pip install -e ".[dev]"` finishes - no extra step needed to build these docs locally.

3. Headings

`prodockit.headings` gives every heading in a document an `id` and a hierarchical section number, and records both - along with the heading's text and level - in a shared registry other prodockit extensions build on (e.g. prodockit.refs, which resolves `\ref{id}` by looking an id up in exactly this registry). You can also enable it on its own if you just want ids/numbers on your headings without cross-references.

3.1 Quick start

Enable it in `zensical.toml`:

```
[project.markdown_extensions."prodockit.headings"]
```

and every heading gets a hierarchical number automatically - an `h1` is a top-level counter ("1", "2", ...), an `h2` nests under the nearest preceding `h1` ("1.1", "1.2", ...), and so on down through `h6`:

Markdown

```
# Introduction

## Background

## Scope

# Method
```

Result

Heading	id	number
Introduction	`introduction`	`1`
Background	`background`	`1.1`
Scope	`scope`	`1.2`

Heading	id	number
Method	`method`	2

`prodockit.headings` doesn't render the number into the heading text itself - only into the registry above, which prodockit.refs then renders inline wherever you write `\ref{id}`. Numbers are recomputed from scratch on every conversion, so reordering headings always produces correct numbers on the next build - there's no stored/stale numbering state.

3.1.1 Appendices

Flag a page's front matter with `is_appendix: true` to give it letter-based numbering instead of the normal numeric sequence - `"A"`, `"A.1"`, `"A.1.1"` - once you've enabled continuous numbering (see Reference below). An appendix page doesn't consume a number from the numeric sequence at all, so pages after it aren't left with a gap. Letters are assigned sequentially in nav order - the first `is_appendix` page becomes `"A"`, the second `"B"`, and so on, independent of how many numbered pages come before them.

For example, with this nav order:

```
nav = [
  {"Install tooling" = "installtooling.md"},
  {"Glossary" = "glossary.md"},
  {"References" = "references.md"},
]
```

and `glossary.md` flagged as an appendix:

```
<!-- glossary.md -->
---
is_appendix: true
---

# Glossary

## Terms {: #terms }
```

a prodockit.refs reference to `Terms` from another page:

```
<!-- references.md -->
# References
```

```
See \ref{terms} for defined terms.
```

renders to (a link to `glossary.md#terms`, shown here as a code block since `glossary.md` isn't a real page on *this* site):

```
<p>See <a class="prodockit-ref" href="glossary.md#terms">A.1</
a> for defined terms.</p>
```

`Glossary`'s own `h1` becomes `"A"` (the first appendix page in nav) and its `Terms` subheading becomes `"A.1"` - and `References`, the page after it, still gets the next plain number in the numeric sequence (`"2"`, not `"3"`), exactly as if the appendix page had never consumed one. Only meaningful under Zensical; ignored otherwise.

3.1.2 Unnumbered headings

A heading with an `unnumbered` class - e.g. a cover page or title slide - still gets an id, but is skipped when computing section numbers, so it doesn't consume a counter position:

```
# Cover Page {: .unnumbered }

# Introduction
```

`Introduction` above is still numbered `1`, as if `Cover Page` weren't there at all.

3.2 Reference

3.2.1 Continuous numbering across pages (Zensical)

By default, numbering is per-document: every page's own `h1` starts back at 1, regardless of what came before it in a multi-page build. Set `numbering="continuous"` to make `h1` numbering carry on from wherever the previous nav page left off instead:

```
[project.markdown_extensions."prodockit.headings"]
numbering = "continuous"
```

e.g. if page one ends with `h1` number `"3"`, page two's first `h1` becomes `"4"`, not `"1"` again. This is what makes a `\ref{id}` link to a heading on a *different* page (see prodockit.refs) show the same number that's actually displayed on that page.

Once enabled, a page whose front matter sets `is_appendix: true` is numbered with a letter instead - see Appendices above for a worked example.

3.2.2 Ids

An id comes from one of, in order of precedence:

1. An explicit id set via `attr_list`, e.g. `# Introduction {: #custom-id }`.
2. Python-Markdown's own `toc` extension, which `prodockit.headings` enables automatically (with its defaults) if you haven't already enabled it yourself - so if you *have* configured `toc` (e.g. with `permalink: true`), that configuration is left untouched and reused.
3. A minimal built-in slugify fallback, used only if `toc` is somehow not registered at all (this should not normally happen, since `prodockit.headings` enables it).

3.2.3 Options

Option	Type	Default	Description
`source`	`str`	`""`	Identifier for the current document (e.g. its file path). Used to scope this document's entries in the registry, and to safely clear/replace them on a rebuild of the same document.
`registry`	`IdRegistry | None`	a new `IdRegistry()`	Share one registry across multiple documents/ conversions - see below. Passed as a constructor keyword, not a string-based config value (Python-Markdown's config system can't carry arbitrary Python objects safely).
`numbering`	`"per-document" | "continuous"`	`"per-document"`	`"continuous"` makes `h1` numbering carry on across pages in Zensical nav order, instead of restarting at 1 on every page - see above. Only meaningful under Zensical; ignored otherwise.
`appendix_attr`	`str`	`"is_appendix"`	Front matter flag name marking a page for letter-based numbering ("A", "A.1", ...) instead of the normal numeric sequence, when `numbering="continuous"`.

3.2.4 Sharing a registry across a multi-page build

To resolve cross-page references, every page in a build needs to write into - and read from - the *same* `IdRegistry` instance, each scoped by its own, distinct `source`.

Under Zensical, this happens automatically with no configuration - see prodockit.refs for details: `prodockit.headings` detects Zensical's per-page context (each page gets its own fresh `Markdown` instance) and uses it to derive `source` from the page's own path, sharing one registry across the whole build. This auto-detection only activates when you *haven't* set an explicit `registry` or `source` yourself, and has no effect at all outside Zensical.

For any other tool, construct a registry yourself and pass it to every page's extension instance, along with that page's own `source`:

```
import markdown
from prodockit.headings import HeadingsExtension
from prodockit.util import IdRegistry

registry = IdRegistry()

for path, text in pages:
    html = markdown.markdown(
        text,
        extensions=[HeadingsExtension(registry=registry,
source=path)],
    )
```

A duplicate id registered from two *different* sources raises `prodockit.util.DuplicateIdError` here - re-converting the *same* source (e.g. a live-reload dev server) is safe and expected; its previous entries are cleared first. (Zensical's automatic sharing above uses the same registry, but logs a warning and keeps the first registration instead of raising - appropriate for a best-effort default rather than a setup you configured deliberately.)

> ⚠ **The 'keeping the first' winner isn't stable across builds**
>
> A heading name shared across two or more pages (e.g. every page having its own "Quick start"/"Options"/"Syntax" section - common in a set of parallel extension/module docs) produces a "collides with ... - keeping the first" warning under Zensical's automatic sharing above - but *which* page's registration actually wins isn't something you can rely on: Zensical doesn't render pages in a guaranteed stable order, confirmed directly by running `zensical build` repeatedly against identical source and observing the

> reported winner change from one run to the next. Anything depending on that id - a `\ref{id}` /a hand- typed anchor link - can silently point at the wrong page's heading depending on which build produced it.
>
> The fix is to give every colliding heading its own explicit, unique id via `attr_list`, rather than leaving it to whichever page happens to register first:
>
> ```
> ## Quick start {: #refs-quick-start }
> ```
>
> A page-prefixed slug (`<page>-<heading>`) is a simple, collision-proof convention - this project's own documentation uses exactly this scheme throughout (every extension page shares several heading names with the others), so its own markdown source is a worked example if you want to see it applied across a whole site.

3.2.5 Looking up the same numbers from your own build tooling

`prodockit.headings.prescan(appendix_attr="is_appendix")` returns `(start_counts, appendix_letters)` - both `dict[str, ...]` keyed by nav-relative page path - the exact same pre-scan `HeadingsExtension` itself uses internally for `numbering="continuous"`. A consuming project's own build tooling can call this directly to stay in sync automatically, rather than re-deriving the same page-order/heading-count logic a second, independent way - e.g. a template macro that emits a presentational CSS counter-reset per page, matching whatever number `prodockit.headings` computes for that page's first heading (see prodockit.zensical_macros). Returns `None` outside a Zensical build.

3.2.6 CSS hooks

`prodockit.headings` doesn't add any class of its own to a heading - only an `id` (see above), the class(es) already on the heading (e.g. `unnumbered`), and whatever the numbers themselves feed into via the registry (typically consumed by prodockit.refs, or by a template's own build tooling via `prescan()` above to drive presentational CSS). There is currently no `prodockit-heading`-style class to hook a stylesheet onto directly.

4. Refs

`prodockit.refs` adds a `\ref{id}` cross-reference syntax - similar in spirit to LaTeX's `\ref` - that resolves to the *current* section number of the heading with that id. It depends on the id/number registry that prodockit.headings builds; enabling `prodockit.refs` on its own transparently enables `prodockit.headings` too, with matching defaults, so a single document works with no extra configuration.

4.1 Quick start

Enable it in `zensical.toml`:

```
[project.markdown_extensions."prodockit.refs"]
```

then reference any heading's id with `\ref{id}`:

```
# Introduction {: #intro }

See \ref{intro} for background.

## Background
```

renders to:

5. Introduction

See 1 for background.

5.1 Background

Because the number is looked up fresh on every conversion, it stays correct even if sections are added, removed, or reordered - you never have to manually renumber a cross-reference. Referencing a heading defined on a *different* page (see Multi-page builds below) links to that page directly (e.g. `other.md#intro`, which Zensical rewrites into the correct clean URL) rather than a bare `#intro` fragment, which would only work if the target happened to be on the same page.

5.1.1 Forward references

A reference to a heading defined *later* in the same document resolves correctly:

```
See \ref{background} below.

## Background {: #background }
```

5.1.2 Unresolved references

`\ref{id}` renders the `unresolved` marker (`??` by default) instead of a number when:

- `id` doesn't exist in the registry at all - e.g. a typo, or a reference to a heading in a page that hasn't been converted yet in a multi-page build (the same way an undefined LaTeX `\ref` shows `??` until a later compilation pass).
- `id` exists but belongs to a heading marked `unnumbered` (see prodockit.headings) - it's still a valid link target in this case, just without a number to show.

```
# Cover Page {: .unnumbered }

See \ref{cover-page}.
```

renders `\ref{cover-page}` as `??`, linked to `#cover-page`.

5.2 Reference

5.2.1 Syntax

```
\ref{<id>}
```

`<id>` is the target heading's id - either one you set explicitly via `attr_list` (`# Introduction {: #intro }`), or the one `toc` derived automatically from the heading text (see prodockit.headings for the exact precedence).

`\ref{...}` is recognised the same way Python-Markdown's own inline syntax is - meaning it's protected inside inline code spans and fenced code blocks, so it's safe to show as a literal example:

````
Type `\ref{intro}` to reference a section.

```

\ref{intro}
```
````

Neither of the two shown above is resolved; both render the literal text.

5.2.2 Options

Option	Type	Default	Description
`unresolved`	`str`	`"??"`	Text rendered when `id` doesn't resolve to a numbered heading.
`source`	`str`	`""`, auto-detected under Zensical	Identifier for the current document (e.g. its path) - used only to decide whether a resolved target is on this same page (bare `#id`) or a different one (a real link to it). Doesn't affect resolution itself.
`registry`	`IdRegistry | None`	discovered from a sibling `prodockit.headings`, or a new one	Share one registry across multiple documents - see below. Passed as a constructor keyword, not a string-based config value.

5.2.3 Multi-page builds

Under Zensical: automatic

Under Zensical, cross-page references work with no extra configuration - just enable both extensions in `zensical.toml` as usual:

```
[project.markdown_extensions."prodockit.headings"]
[project.markdown_extensions."prodockit.refs"]
```

Zensical builds each page with its own, fresh `Markdown` instance, so `prodockit.headings` detects this (via Zensical's own per-page context) and transparently shares one registry across every page of the build, keyed by each page's own path - no explicit `registry` / `source` needed.

Referencing a heading on a page built later works too, in a single `zensical build` pass: `prodockit.headings` pre-scans every page in the current build's nav for its headings' ids and section numbers before any page has actually been converted, the same way prodockit.citations pre-scans for citation definitions. Pages aren't necessarily built in nav order - or even all within one shared Python process - so without this a cross-page `\ref{id}` could resolve on one build and fall back to `unresolved` (`??`) on the next, from the same unchanged source (see prodockit-extensions#54). A page that *is* converted in the same process supersedes its own pre-scanned entries with the real ones from the parsed document, so the pre-scan only ever fills a gap.

Two pages that happen to share an identically-titled heading (e.g. both have their own "Overview" section) don't fail the build: the *first* one built keeps that id, and the collision is logged as a warning rather than raised as an error - give one of them an explicit id via `attr_list` (`## Overview {: #api-overview }`) to disambiguate and make both referenceable.

Under other tools: manual

Outside Zensical, give `prodockit.headings` and `prodockit.refs` the *same* `IdRegistry` on every page yourself, converting pages in the order cross-references should become resolvable in:

```
import markdown
from prodockit.headings import HeadingsExtension
from prodockit.refs import RefsExtension
from prodockit.util import IdRegistry

registry = IdRegistry()

for path, text in pages:
    html = markdown.markdown(
        text,
        extensions=[
            HeadingsExtension(registry=registry, source=path),
            RefsExtension(registry=registry, source=path),
```

```
        ],
    )
```

Give `RefsExtension` the same `source=path` as `HeadingsExtension` - without it, every resolved link is treated as cross-page (harmless, just not the minimal same-page form for a reference that happens to target its own page).

Here, a genuine id collision between two different `source`s *does* raise `prodockit.util.DuplicateIdError` rather than warning - a deliberately shared registry means you're expected to notice and fix a collision, unlike the best-effort automatic Zensical case above.

5.2.4 CSS hooks

`prodockit.refs` always sets a class on the `\ref{id}` link it renders - resolved or not - so a stylesheet has a stable hook either way:

State	Class
Resolved	`prodockit-ref`
Unresolved	`prodockit-ref prodockit-ref-unresolved`

An unresolved reference (see Unresolved references above) still gets a `class` either way; style `prodockit-ref-unresolved` distinctly (e.g. a warning colour) to make a broken cross-reference visually obvious without inspecting the page source. No `data-*` attribute is left in the rendered output - the internal `data-prodockit-ref` placeholder attribute used during resolution is always stripped before the page is rendered.

6. Citations or References

`prodockit.citations` lets you define a source once and cite it by key from anywhere in a build, instead of hand-typing a bracketed link at every citation site.

> **Looking for an auto-generated bibliography instead?**
>
> `prodockit.citations` is one of two alternatives for citing sources - this one resolves against a hand-authored paragraph you write yourself, once. If you'd rather define sources in a BibTeX/BibLaTeX `.bib` file and have the reference list generated for you, in any citation style, see prodockit.bibliography and its comparison of both approaches.

6.1 Quick start

Enable it in `zensical.toml`:

```
[project.markdown_extensions."prodockit.citations"]
```

Define a source's paragraph with an id and a short display text via `attr_list`, then cite it from anywhere with `\cite{id}`:

```
Git is a tool used to manage version control.\cite{skou2023}

Skoulikari, A. (2023) *Learning Git: A Hands-On and Visual
Guide to the
Basics of Git*. Sebastopol, CA: O'Reilly Media.
{: #skou2023 .reference data-cite-text="Skoulikari, 2023" }
```

renders to:

Git is a tool used to manage version control.[Skoulikari, 2023]

Skoulikari, A. (2023) *Learning Git: A Hands-On and Visual Guide to the Basics of Git*. Sebastopol, CA: O'Reilly Media.

`[Skoulikari, 2023]` is linked directly to the source's own paragraph (e.g. `references.md#skou2023`, or `#skou2023` if cited from that same page) - Zensical rewrites that into the correct clean URL for the citing page's own location, the same way a hand-typed `[text](references.md#skou2023)` link already gets rewritten. Unlike hand-typing that link yourself, you never have to work out the relative path to the references page, retype the display text, or fix every citation site if the display text needs to change - it's defined once.

Multiple comma-separated keys join into one bracket: `\cite{skou2023,chacon2014}` → `[Skoulikari, 2023; Chacon and Straub, 2014]`.

Unlike prodockit.glossary's `\gls{id}`, which inserts a term's own registered text in place, `\cite{id}` *generates* new bracketed citation text around a link - closer to a bibliography citation than a glossary/acronym expansion.

6.1.1 Forward references

A citation to a source defined *later* in the same document resolves correctly:

```
See \cite{skou2023} above.

Skoulikari, A. (2023) *Learning Git*.
{: #skou2023 data-cite-text="Skoulikari, 2023" }
```

6.1.2 Unresolved citations

A key that doesn't resolve to a definition renders the `unresolved` marker (`?` by default) in place of that one entry - the rest of a multi-key citation still resolves normally:

```
\cite{skou2023,does-not-exist}
```

renders `[Skoulikari, 2023; ?]` - the unresolved entry has no link, unlike a resolved one.

6.2 Reference

6.2.1 Syntax

Defining and citing are bundled into one extension, unlike prodockit.headings/prodockit.refs: a definition is useless without somewhere to cite it, so there's no independently useful "just defining" half to split out.

Defining a source

Any block element - typically a paragraph - with both an `id` and a `data-cite-text` attribute becomes a citable source:

```
Chacon, S. and Straub, B. (2014) *Pro Git*. 2nd edn. New York:
Apress.
```

```
{: #chacon2014 .reference data-cite-text="Chacon and Straub,
2014" }
```

`data-cite-text` is the short text rendered at each citation site - it's stripped from the rendered output (it's internal bookkeeping, not meant to be visible), while `id` stays, since citations link straight to it.

Citing a source

```
\cite{<id>}
\cite{<id1>,<id2>,...}
```

Like prodockit.refs, `\cite{...}` is recognised the same way Python-Markdown's own inline syntax is, so it's protected inside inline code spans and fenced code blocks:

````
Type `\cite{skou2023}` to cite a source.

```

\cite{skou2023}
```
````

Neither of the two shown above is resolved; both render the literal text.

6.2.2 Options

Option	Type	Default	Description
`source`	`str`	`""`	Identifier for the current document (e.g. its file path). Used to scope this document's own citation definitions in the registry.
`unresolved`	`str`	`"?"`	Text rendered for a `\cite{id}` key that doesn't resolve to a definition.
`registry`	`CitationRegistry | None`	discovered automatically, or a new one	Share one registry across multiple documents - see below. Passed as a constructor keyword, not a string-based config value.

6.2.3 Multi-page builds

Under Zensical: automatic

Under Zensical, citing a source defined on a *different* page (the common case - a references page separate from the pages that cite it) works with no extra configuration, the same way prodockit.refs shares its registry across pages: `prodockit.citations` detects Zensical's per-page context and shares one registry across the whole build automatically.

```
[project.markdown_extensions."prodockit.citations"]
```

Citing a source before it's defined works too - the common case, since a references page is usually cited from earlier chapters but kept at the *end* of nav as an appendix. Normally that's a forward reference to a page `zensical build`'s single, one-shot pass hasn't rendered yet (unlike `zensical serve`'s live-reload, which eventually rebuilds every page at least once) - `prodockit.citations` avoids this by pre-scanning every page in the current Zensical build's nav for citation definitions (reading raw file text directly, not waiting for Python-Markdown to parse each one) before any single page has actually been converted, the same way LaTeX needs multiple compilation passes to resolve a `\cite` used before its `\bibitem` - except here it happens automatically, within one `zensical build` invocation.

Two different sources that happen to share the same key don't fail the build: the first one scanned keeps that key, and the collision is logged as a warning rather than raised as an error.

Under other tools: manual

Outside Zensical, share a `CitationRegistry` yourself, the same way as prodockit.headings:

```
import markdown
from prodockit.citations import CitationsExtension
from prodockit.util import CitationRegistry

registry = CitationRegistry()

for path, text in pages:
    html = markdown.markdown(
        text,
        extensions=[CitationsExtension(registry=registry,
source=path)],
    )
```

A genuine key collision between two different `source`s raises `prodockit.util.DuplicateIdError` here, rather than warning - a deliberately shared registry means you're expected to notice and fix it.

6.2.4 What this doesn't do (yet)

`prodockit.citations` covers citation-key management - the "define once, cite anywhere" part. It doesn't auto-generate the references page's listing itself from structured bibliographic data (author/year/title/publisher/URL fields) the way a full BibTeX-style tool would - the reference entry's own text (as shown in the examples above) is still hand-authored prose, just like today. See prodockit.bibliography for the alternative that does exactly this, from a `.bib` file, in any citation style - and for the tradeoffs between the two approaches.

6.2.5 CSS hooks

`prodockit.citations` emits three hooks - one on the outer wrapper, one on each individual key's own link:

Element	State	Class
Outer `<span>` wrapping the whole `\cite{...}` citation	always	`prodockit-cite`
Each key's own `<a>`	resolved	`prodockit-cite-resolved`
Each key's own `<a>`	unresolved	`prodockit-cite-unresolved`

An unresolved key's `<a>` has no `href` (see Unresolved citations above) - style `prodockit-cite-unresolved` distinctly (e.g. a muted colour, no underline) to make a missing citation visually obvious without inspecting the page source. No `data-*` attribute is left in the rendered output - the internal `data-prodockit-cite` placeholder attribute used during resolution, and the `data-cite-text` attribute marking a definition, are both always stripped before the page is rendered.

7. Acronyms and Glossary

`prodockit.glossary` lets you define a term once - an acronym expansion, a glossary definition, anything with a short name and a longer explanation - and insert it by id from anywhere in a build, instead of hand-typing a link around the term's own text at every use.

7.1 Quick start

Enable it in `zensical.toml`:

```
[project.markdown_extensions."prodockit.glossary"]
```

Define a term's paragraph with an id and its display text via `attr_list`, then insert it from anywhere with `\gls{id}`:

```
This site uses \gls{css} to control appearance.

**CSS** - Cascading Style Sheets.
{: #css .acronym data-term="CSS" }
```

renders to:

This site uses CSS to control appearance.

CSS - Cascading Style Sheets.

`CSS` is linked directly to the term's own page (e.g. `acronyms.md#css`, or `#css` if used from that same page) - Zensical rewrites that into the correct clean URL for the citing page's own location, the same way a hand-typed `[CSS](acronyms.md#css)` link already gets rewritten.

Unlike prodockit.citations's `\cite{id}`, which *generates* new bracketed citation text (`\cite{id}` → `[Skoulikari, 2023]`), `\gls{id}` inserts the term's *own* registered text in place - closer to LaTeX's `glossaries` package (`\gls{key}` expands to the term's own name) than to a citation. One shared registry covers both acronym entries and glossary entries (and anything else you want to define a short term for) - they're the same kind of thing, an id with a short display text, just conventionally organised across two differently-named pages (see Acronyms and Glossary: one registry, two pages below).

7.1.1 Forward references

A `\gls{id}` pointing at a term defined *later* in the same document resolves correctly, the same way prodockit.refs/prodockit.citations do:

```
See \gls{css} above.

**CSS** - Cascading Style Sheets.
{: #css data-term="CSS" }
```

7.1.2 Unresolved references

An id that doesn't resolve to a definition renders the `unresolved` marker (`?` by default), unlinked:

```
\gls{does-not-exist}
```

renders `?`, with no link.

7.2 Acronyms and Glossary: one registry, two pages

A common convention splits acronym expansions (a short form → long form) and glossary entries (a term → its definition) across two separate pages. `prodockit.glossary` doesn't need to know which page is "acronyms" and which is "glossary" - both are just term definitions in the same registry, so a `\gls{id}` resolves the same way regardless of which page defines it:

```
<!-- acronyms.md -->
**CSS** - Cascading Style Sheets.
{: #css .acronym data-term="CSS" }
```

```
<!-- glossary.md -->
**Cascading Style Sheets** - The language used to control
appearance.
{: #css-def .glossary data-term="Cascading Style Sheets" }
```

7.2.1 Cross-links between entries: use a plain link, not `\gls{id}`

Linking an acronym entry to its own glossary counterpart (and vice versa) is a "see also" cross-reference, not a term insertion - the link text needs to say something like "see the glossary", not repeat the term itself. `\gls{id}` always inserts the *term's own registered text*, so it's the wrong tool here: `\gls{css-def}` renders `Cascading Style Sheets`, so `See \gls{css-def} for what this means` would read *"See Cascading Style Sheets for what this means"* - it resolves, but loses the "go look elsewhere" cue the word "glossary" gives the reader.

Use a plain, hand-typed Markdown link instead, exactly as you would for any other page-to-page cross-reference - it's understood natively by both outputs, and

doesn't need `prodockit.refs` / `prodockit.citations` / `prodockit.glossary` at all:

```
<!-- acronyms.md -->
**CSS** - Cascading Style Sheets. See the [glossary]
(glossary.md#css-def) for what this means in practice.
{: #css .acronym data-term="CSS" }
```

```
<!-- glossary.md -->
**Cascading Style Sheets** - The language used to control
appearance. See the [Acronyms](acronyms.md#css) entry for the
expansion.
{: #css-def .glossary data-term="Cascading Style Sheets" }
```

As a rule of thumb: reach for `\gls{id}` when the term's own name belongs at that point in the sentence, and a plain link when the link text needs to say something else entirely - a "see also", a page name, or any other custom wording.

7.3 Reference

7.3.1 Syntax

Like prodockit.citations, defining and inserting are bundled into one extension: a definition is useless without somewhere to use it.

Defining a term

Any block element - typically a paragraph - with both an `id` and a `data-term` attribute becomes usable with `\gls{id}`:

```
**GUI** - Graphical User Interface.
{: #gui .acronym data-term="GUI" }
```

`data-term` is the text inserted at each `\gls{id}` site - it's stripped from the rendered output (it's internal bookkeeping, not meant to be visible), while `id` stays, since references link straight to it.

Using a term

```
\gls{<id>}
```

Unlike `\cite{...}`, `\gls{...}` only ever takes a single id - there's no multi-term/bracketed form, since inserting a term's own text doesn't compose the way a citation list does.

Like prodockit.refs/prodockit.citations, `\gls{...}` is recognised the same way Python-Markdown's own inline syntax is, so it's protected inside inline code spans and fenced code blocks:

````
Type `\gls{css}` to insert a term.

```
\gls{css}
```
````

Neither of the two shown above is resolved; both render the literal text.

7.3.2 Options

Option	Type	Default	Description
`source`	`str`	`""`	Identifier for the current document (e.g. its file path). Used to scope this document's own term definitions in the registry, and to build a correct link when a `\gls{id}` target lives on a different page.
`unresolved`	`str`	`"?"`	Text rendered for a `\gls{id}` that doesn't resolve to a definition.
`registry`	`GlossaryRegistry | None`	discovered automatically, or a new one	Share one registry across multiple documents - see below. Passed as a constructor keyword, not a string-based config value.

7.3.3 Multi-page builds

Under Zensical: automatic

Under Zensical, referencing a term defined on a *different* page (the common case - Acronyms/Glossary appendix pages separate from the pages that use them) works with no extra configuration, the same way prodockit.citations shares its registry across pages:

```
[project.markdown_extensions."prodockit.glossary"]
```

Using a term before it's defined works too, the same way as `prodockit.citations`: `prodockit.glossary` pre-scans every page in the current Zensical build's nav for term definitions before any page has actually been

converted, so a term used from an early chapter but defined on an Acronyms/Glossary page kept at the end of nav resolves correctly within a single `zensical build` pass.

Two different sources that happen to define the same id don't fail the build: the first one scanned keeps that id, and the collision is logged as a warning rather than raised as an error.

Under other tools: manual

Outside Zensical, share a `GlossaryRegistry` yourself, the same way as prodockit.citations:

```
import markdown
from prodockit.glossary import GlossaryExtension
from prodockit.util import GlossaryRegistry

registry = GlossaryRegistry()

for path, text in pages:
    html = markdown.markdown(
        text,
        extensions=[GlossaryExtension(registry=registry,
source=path)],
    )
```

A genuine id collision between two different `source`s raises `prodockit.util.DuplicateIdError` here, rather than warning - a deliberately shared registry means you're expected to notice and fix it.

7.3.4 CSS hooks

`prodockit.glossary` always sets a class on the `\gls{id}` link it renders - resolved or not - so a stylesheet has a stable hook either way:

State	Class
Resolved	`prodockit-gls`
Unresolved	`prodockit-gls prodockit-gls-unresolved`

An unresolved id's `<a>` has no `href` (see Unresolved references above) - style `prodockit-gls-unresolved` distinctly (e.g. a warning colour) to make a missing term visually obvious without inspecting the page source. No `data-*` attribute is left in the rendered output - the internal `data-prodockit-gls` placeholder attribute used during resolution, and the `data-term` attribute marking a definition, are both always stripped before the page is rendered.

8. Tables

`prodockit.tables` gives a table column a percentage or fixed width, via a `width` attribute already attachable to a header cell with `attr_list` - no new syntax to learn.

8.1 Quick start

Enable it in `zensical.toml`:

```
[project.markdown_extensions."prodockit.tables"]
```

then attach a `width` to any header cell you want to size.

8.1.1 Percentages that add up to 100%

Give every column an explicit percentage and they're used exactly as written:

Markdown

```
| Name {: width="25%" } | Description {: width="50%" } | Due
{: width="25%" } |
|---|---|---|
| Headings | Heading ids and section numbers | Q1 |
| Refs | Cross-references, resolved by number | Q2 |
```

Result

Name	Description	Due
Headings	Heading ids and section numbers	Q1
Refs	Cross-references, resolved by number	Q2

8.1.2 Percentages that don't add up to 100%

Leave a column unannotated and it takes whatever's left over, split evenly if more than one column is left unannotated - the same "standard algorithm" any HTML

table with a `<colgroup>` and `table-layout: fixed` already uses to size an unspecified column, not something `prodockit.tables` computes itself:

Markdown

```
| Name {: width="20%" } | Description | Due {: width="15%" }
|
|---|---|---|
| Headings | Heading ids and section numbers | Q1 |
| Refs | Cross-references, resolved by number | Q2 |
```

Result

Name	Description	Due
Headings	Heading ids and section numbers	Q1
Refs	Cross-references, resolved by number	Q2

`Name` and `Due` get the widths given; `Description`, left unannotated, takes the remaining 65%. A column left unannotated in a table with no `width` anywhere at all is completely untouched, though - only a table with at least one `width` gets a `<colgroup>`.

8.1.3 Fixed widths for every column

A CSS length works the same way as a percentage - useful for a column that should stay a constant size regardless of how wide the table's container ends up being. Every column can be given one, with none left to share remaining space:

Markdown

```
| Icon {: width="60px" } | Description {: width="200px" } |
Format {: width="100px" } |
|---|---|---|
| :material-file-pdf-box: | A downloadable PDF | PDF |
```

```
| :material-file-document: | A Markdown source file |
Markdown |
```

Result

Icon	Description	Format
PD	A downloadable PDF	PDF
	A Markdown source file	Markdown

8.1.4 Mixing percentages and fixed widths

Percentage and fixed-length columns can appear in the same table - CSS's own table layout algorithm sizes both correctly at once, the same way it would for any hand-written `<colgroup>`. A column can still be left unannotated here too, sharing whatever's left over the same way as the percentage-only case above:

Markdown

```
| # {: width="40px" } | Name {: width="50%" } | Description
|
|---|---|---|
| 1 | prodockit.headings | Heading ids and section numbers |
| 2 | prodockit.tables | Column widths on a table |
```

Result

#	Name	Description
1	prodockit.headings	Heading ids and section numbers
2	prodockit.tables	Column widths on a table

8.1.5 Left-aligning a header

By default a header cell is centred and a body cell is left-aligned - the browser's own default styling for `<th>` / `<td>`, unrelated to `prodockit.tables`. To left-align a header too, use Python-Markdown's own column-alignment syntax - a `:` on the left side of that column's own dashes in the separator row - which applies to the header *and* every body cell in that column alike, and combines with `width` on the same header cell with no conflict:

Markdown

```
| Name {: width="30%" } | Description |
|:---|---|
| Headings | Heading ids and section numbers |
| Refs | Cross-references, resolved by number |
```

Result

Name	Description
Headings	Heading ids and section numbers
Refs	Cross-references, resolved by number

`:---:` / `---:` center- or right-align a column the same way - see Python-Markdown's own `tables` docs for the full syntax. This isn't a `prodockit.tables` feature; it's documented here because it's the natural companion to `width` when sizing a column, not something `prodockit.tables` needs to reimplement.

8.2 Reference

8.2.1 Syntax

Builds on Python-Markdown's own `tables` extension (auto-enabled if not already present, the same way prodockit.refs auto-enables prodockit.headings).

Attach `width` to a header cell (`th`), not a body cell - a column's width is a property of the whole column, so it's declared once, on the heading:

```
| Column {: width="<css-length>" } | ... |
|---|---|
```

`<css-length>` is any valid CSS width value - a percentage (`"30%"`) or a fixed length (`"120px"`, `"4cm"`, `"3em"`, ...) - passed through to the generated `<colgroup>` as-is, with no validation of its own; an invalid value behaves exactly as it would in any other hand-written CSS, since `prodockit.tables` doesn't parse or interpret it beyond that.

8.2.2 CSS hooks

A table with at least one `width`-attributed header cell gets a `<colgroup>` (one `<col>` per column, `style="width: ..."` set only on the columns that had one) inserted as its first child, and `class="prodockit-table-sized"` on the `<table>` itself:

Element	Condition	Hook
`<table>`	at least one header cell has `width`	`class="prodockit-table-sized"`
`<col>`	that column's header cell had `width`	`style="width: <value>;"`
`<col>`	that column's header cell had no `width`	none - left for `table-layout: fixed` to size

The `width` attribute itself is always stripped from the `<th>` once read - it isn't meant to also linger on the header cell.

`prodockit-table-sized` only *marks* a table as sized - it isn't styled by `prodockit.tables` itself. A stylesheet needs to apply `table-layout: fixed` itself for the `<colgroup>` widths (and the "share what's left" behaviour) to take effect at all. Under Zensical's Material-based theme specifically, its own default table styling (border, padding, alternating rows) is scoped `.md-typeset table:not([class])` - confirmed directly in Zensical's bundled CSS - so a table carrying `prodockit-table-sized` (or any other class) gets **none** of it, not just no width control:

```
.md-typeset table.prodockit-table-sized {
  table-layout: fixed;
  width: 100%;
  border-collapse: collapse;
  border: 1px solid #555555;
}
.md-typeset table.prodockit-table-sized th,
.md-typeset table.prodockit-table-sized td {
  border: 1px solid #555555;
  padding: 0.9375em 1.25em;
  vertical-align: top;
```

```
}
.md-typeset table.prodockit-table-sized th {
  font-weight: 700;
}
```

prodockit.pdf already includes the equivalent rule in its own generated CSS (as well as its own table border/padding, unaffected by this theme-specific quirk), so a sized table works in the PDF with no extra configuration; a project's own website theme needs the CSS above added itself (see this project's own `docs/stylesheets/extra.css` for a working example) - `prodockit.tables` doesn't ship a bundled website stylesheet the way `prodockit.pdf` ships one for the PDF path.

A table with no `width`-attributed header cells at all is left completely untouched - no `<colgroup>`, no `prodockit-table-sized` class - so enabling `prodockit.tables` has no effect on any table that doesn't use it.

9. Bibliography

`prodockit.bibliography` is an alternative to prodockit.citations: define your sources once in a BibTeX/BibLaTeX `.bib` file, cite them by key with `\citebib{id}` from anywhere in a build, and get a fully formatted, sorted reference list generated for you - in any Citation Style Language (CSL) style (APA, IEEE, Harvard, Vancouver, and hundreds more), the same open, actively-maintained style ecosystem Zotero/Mendeley/EndNote already use.

Uses its own `\citebib{id}` syntax, distinct from `prodockit.citations`' `\cite{id}` - see Comparing the two approaches below for the full tradeoffs.

9.1 Requirements

Pandoc needs to be installed and on `PATH` - **including for a project that never builds a PDF at all**, unlike every other prodockit extension, which needs nothing beyond Python-Markdown itself:

```
brew install pandoc   # or see https://pandoc.org/
installing.html
```

See How it works below for why.

9.2 Quick start

Enable it in `zensical.toml`, pointing `bib_file` at your own `.bib` file (a path relative to wherever `zensical build` / `zensical serve` is run from - typically your project root):

```
[project.markdown_extensions."prodockit.bibliography"]
bib_file = "references.bib"
```

```
←!── references.bib ──→
@book{chacon2014,
  author    = {Chacon, Scott and Straub, Ben},
  title     = {Pro Git},
  edition   = {2},
  year      = {2014},
  publisher = {Apress}
}
```

Cite it from anywhere with `\citebib{id}`:

```
Git is a distributed version control system
\citebib{chacon2014}.
```

renders to (default style shown; see Choosing a citation style below):

```
<p>Git is a distributed version control system <span
class="prodockit-bib-cite"><a href="references.md#ref-
chacon2014">(Chacon and Straub 2014)</a></span>.</p>
```

9.2.1 The reference list

Put a bare `\bibliography` marker, alone on its own paragraph, wherever you want the complete, formatted reference list to appear - typically a dedicated References page, kept at the end of `nav` as an appendix, the same convention `prodockit.citations`' own hand-authored references pages already use:

```
# References

\bibliography
```

Every entry in `bib_file` appears, in the order your chosen CSL style sorts them (alphabetically by default) - not just the ones actually cited, the same way LaTeX's `\nocite{*}` includes every `.bib` entry regardless of whether it's cited in the document. A `\citebib{id}` written on any page links directly to its own entry here, adjusted for that page's own directory depth - a real Zensical clean-URL link like `prodockit.refs` / `prodockit.citations` already build, not a bare `#id` fragment that would 404 from a different page.

`\bibliography` takes two optional parameters narrowing this down to just what's actually cited, and/or a different `.bib` file than the configured default - see Multiple sections: References and Bibliography below.

9.2.2 Multiple sections: References and Bibliography

A style guide often distinguishes two different lists:

- **References** (or "Works Cited") - a strict list of only the sources actually cited in the text.
- **Bibliography** - a broader list: everything cited, *plus* background reading the author wants to list even though it's never individually cited inline.

`\bibliography` takes two optional, positional parameters for exactly this:

```
\bibliography{<file>}{<true|false>}
```

- `<file>` - which `.bib` file *this* marker draws from. Leave it empty (`{}`) or omit it entirely to use the configured `bib_file`.
- `<true|false>` - `true` restricts this marker's list to only entries actually `\citebib{}`-cited somewhere in the build; `false` (the default, and the only behaviour before these parameters existed) keeps every entry, cited or not.

Write two markers to get both sections from the same `.bib` file:

```
<!-- references.md -->
# References

\bibliography{}{true}
```

```
<!-- bibliography.md -->
# Bibliography

\bibliography{}{false}
```

or from two different files, if background/further-reading sources live separately from the ones actually cited:

```
<!-- references.md -->
# References

\bibliography{references.bib}{true}
```

```
<!-- bibliography.md -->
# Bibliography

\bibliography{background.bib}
```

A `\citebib{id}` links to whichever marker's page actually lists that entry, based on which `.bib` file defines it - in the common single-file case (one bare `\bibliography`, as in Quick start above), that's still just the one page, exactly as before. `csl_style` stays a single, extension-wide setting - only the file and cited-only flag are per-marker.

9.2.3 Choosing a citation style

Point `csl_style` at any `.csl` file - your institution's own house style, or one of

the thousands available from the Citation Style Language project (the same repository Zotero/Mendeley pull styles from). This project's own docs, and `prodockit-template` / `prodockit-userguide` 's hand-authored references, already follow Cite Them Right Harvard - `harvard-cite-them-right.csl` reproduces that exact style automatically:

```
[project.markdown_extensions."prodockit.bibliography"]
bib_file = "references.bib"
csl_style = "harvard-cite-them-right.csl"
```

renders (confirmed directly against the same `.bib` file used above):

```
<p>Git is a distributed version control system <span
class="citation">(Chacon and Straub, 2014)</span>.</p>
...
<div id="ref-chacon2014" class="csl-entry">
Chacon, S. and Straub, B. (2014) <em>Pro git</em>. 2nd edn. New
York: Apress. Available at: <a href="https://git-scm.com/
book">https://git-scm.com/book</a>.
</div>
```

- exactly the format already hand-typed throughout this project's own reference lists, just generated instead. Leaving `csl_style` unset uses Pandoc's own default (a Chicago author-date style) instead. Confirmed directly: the exact same `.bib` file, with only `csl_style` changed, produces correctly (and very differently) formatted output - author-date parenthetical citations and a hanging-indent reference list for APA, numbered `[1]` citations and a numbered list for IEEE, and so on - with no other configuration.

9.2.4 Unresolved citations

A key that doesn't resolve to a `.bib` entry renders the `unresolved` marker (`?` by default), unlinked:

```
\citebib{does-not-exist}
```

renders `?`, with no link.

9.3 Reference

9.3.1 Syntax

```
\citebib{<id>}
```

Only a single key is supported - unlike `prodockit.citations`' `\cite{id1,id2,...}`, a multi-key citation isn't matched by this extension's own syntax at all (falls through as literal text, a visible, honest "not supported" rather than a silently wrong result) - see Comparing the two approaches for why.

Like prodockit.citations, `\citebib{...}` is recognised the same way Python-Markdown's own inline syntax is, so it's protected inside inline code spans and fenced code blocks.

```
\bibliography
\bibliography{<file>}
\bibliography{<file>}{<true|false>}
```

`<file>` and `<true|false>` are both optional - see Multiple sections: References and Bibliography above for what they do and when to use them. Put the marker alone on its own paragraph/line.

9.3.2 Options

Option	Type	Default	Description
`bib_file`	`str`	`"references.bib"`	Path to a BibTeX/BibLaTeX `.bib` file, relative to wherever `zensical build` / `zensical serve` (or your own script) is run from.
`csl_style`	`str`	`""` (Pandoc's own default)	Path to a Citation Style Language (`.csl`) file.
`unresolved`	`str`	`"?"`	Text rendered for a `\citebib{id}` key that doesn't resolve to a `.bib` entry.
`source`	`str`	`""`, auto-detected under Zensical	Identifier for the current document, used to build a correct link from `\citebib{id}` to `\bibliography`'s own page.

9.3.3 CSS hooks

Element	Condition	Hook
`<span>` wrapping a resolved `\citebib{id}`	always	`class="prodockit-bib-cite"`
`<span>` wrapping an unresolved `\citebib{id}`	always	`class="prodockit-bib-cite prodockit-bib-cite-unresolved"`

Element	Condition	Hook
Each generated reference-list entry	always	`class="csl-entry reference"`

Every generated reference-list entry also gets `class="reference"` (in addition to Pandoc's own `csl-entry`) - matching the class `prodockit.citations`' own hand-authored entries already use, so `prodockit.zensical_macros`' `reference_style()` / prodockit.pdf's own `reference_style` setting apply uniformly, whether an entry was hand-typed or generated.

9.4 How it works

Citation/bibliography formatting is delegated entirely to Pandoc's own `--citeproc` (confirmed directly: a plain `.bib` file plus a chosen `.csl` style produces correctly formatted, sorted output with no custom code at all) rather than reimplemented here - CSL processing (sorting, disambiguation, locale-specific formatting) is a mature-tool-sized problem, the same reasoning prodockit.pdf already gives for why it feeds Pandoc real HTML instead of hand-translating every markdown feature.

Zensical renders your site as usual, but each time this extension resolves a `\citebib{id}` or `\bibliography` marker it shells out to `pandoc --citeproc`, once per distinct citation and once per generated list, each memoized for the rest of the build - Pandoc never sees, and has no part in rendering, anything else on the page:

```
flowchart LR
    bib[".bib file<br>.csl style"]
    md["Markdown source<br>\citebib{id} / \bibliography"]
    ext["prodockit.bibliography<br>(Python-Markdown
extension)"]
    pandoc["pandoc --citeproc"]
    web["Zensical<br>(live website)"]
    pdf["prodockit.pdf<br>(WeasyPrint PDF)"]

    bib ⟶ ext
    md ⟶ ext
    ext -- "subprocess call,<br>memoized per build" ⟶ pandoc
    pandoc -- "formatted citation /<br>reference list HTML" ⟶
ext
    ext ⟶ web
    ext ⟶ pdf
```

9.5 Comparing the two approaches

Both extensions solve the same problem - cite a source by key, get a formatted reference list - but make a fundamentally different tradeoff about where the formatted text comes from. They can be enabled together in the same build without conflict (this project's own docs do, to demonstrate both side by side), though a typical single project only needs one.

	prodockit.citations	prodockit.bibliography
Source of truth	A hand-typed paragraph, once, tagged `data-cite-text`	A `.bib` file entry
Reference list	You write it, by hand, in full	Generated automatically
Citation style	Whatever you typed - one style, fixed	Any CSL style, swappable via one setting
Multi-key citations (`\cite{a,b}`)	Yes - each key individually linked	Not supported (falls through as literal text)
External dependencies	None	`pandoc` on `PATH`, even without a PDF build
Editing a reference	Edit the prose by hand, on the references page	Edit the `.bib` entry once, everywhere it's cited updates
Separate References/ Bibliography sections	Not built in - would need two hand-authored lists kept in sync manually	Built in - `\bibliography{<file>}{<true|false>}` generates a strict cited-only list and/or a broader everything-included list, see Multiple sections

Where `prodockit.citations` fits best: a short reference list, a house style unlikely to ever change, or a project that doesn't want a `pandoc` dependency for its website build at all (only for its optional PDF, via prodockit.pdf, which already needs `pandoc` anyway).

Where `prodockit.bibliography` fits best: a longer, actively-maintained reference list; needing to match a specific institution's CSL style (or switch between several, e.g. a thesis needing IEEE for one chapter's publications list and Harvard for the rest of the document - one `.bib` file, several instances with different `csl_style` values); or wanting a citation added anywhere in the document to also add it to the reference list, correctly formatted, with no hand-typing at all.

9.5.1 What this project's own template and user guide currently do

Neither has adopted `prodockit.bibliography` yet - both prodockit-template

and prodockit-userguide use `prodockit.citations`' hand-authored approach exclusively: a `references.md` page listing each source as its own hand-typed paragraph, tagged `{: #id .reference data-cite-text="..." }`. This is a reasonable, deliberate choice for both - each is a short, relatively static reference list (around ten entries), and neither wants a `pandoc` dependency added to its website build purely for citations (`prodockit-userguide` in particular has no PDF build at all today, so `pandoc` would otherwise be an entirely new requirement). A project outgrowing that - a longer, frequently-updated bibliography, or needing to match a specific CSL style - is exactly the case `prodockit.bibliography` is built for instead.

9.6 Status

New, less battle-tested than `prodockit.citations` - no formal, versioned public API stability contract yet (see prodockit-extensions#7).

10. Index (pdf-only)

`prodockit.index` marks a term inline, wherever it's actually discussed, for a PDF-only back-of-book index (an alphabetised list of terms with the page number(s) they appear on) - the term both displays exactly as written and is marked for indexing in one go, no separate "definition" step needed anywhere else, unlike prodockit.citations/prodockit.glossary. Marking and generating the index are both covered on this page - PDF-only (there's no equivalent on the live website, where readers use Zensical's own site search instead), but the actual index page is built by prodockit.pdf, via the `prodockit.pdf.index` module - see `prodockit.pdf.index` if you're scripting your own build pipeline rather than using `prodockit pdf` / `zensical.toml`.

10.1 Requirements

Marking a term needs nothing beyond the extension itself. Actually *generating* the index page additionally needs the optional `pymupdf` dependency (a term's own page number can only be known once WeasyPrint has already laid the PDF out once - see Generating the index below for why):

```
pip install prodockit[index]   # or plain: pip install pymupdf
```

10.2 Quick start

Enable it in `zensical.toml`:

```
[project.markdown_extensions."prodockit.index"]
```

then mark a term with `\index{Term}`:

Markdown

```
A \index{widget} is the basic unit of work.

Later, this \index{widget} gets combined with a
\index{gadget}.
```

Result

A widget is the basic unit of work.

Later, this widget gets combined with a gadget.

Every marked term renders inline exactly as written - `\index{widget}` becomes plain "widget" text, nothing more, on the live website. The marker only has an effect on the *PDF*, and only once `pdf_include_index` is on - see Generating the index below for the generated index page itself.

The same term marked more than once merges into one index entry (case-insensitively - "Widget" and "widget" become one entry, keeping whichever casing was used first; "widgets" is still a separate entry from "widget" - no plural/singular normalisation).

10.3 Generating the index

Set `pdf_include_index = true` under `[project.extra]` in `zensical.toml` for a traditional, two-column back-of-book index - terms grouped under a bold letter heading (A, B, C, ...), each followed by the page number(s) it appears on - appended as its own page(s) at the very end of the document:

```
[project.extra]
pdf_include_index = true
pdf_index_title = "Index"   # optional - that page's own
heading text
```

The `\index{widget}` / `\index{gadget}` example above renders to an index page like:

```
Index

G
Gadget, 3

W
Widget, 1, 3
```

with its own page list deduplicated and sorted; consecutive pages collapse into an en-dash range (`67–70`) rather than listing every page individually, and non-consecutive pages/ranges are comma-separated (`64, 175`) - standard back-of-book index convention.

A term is alphabetised (and letter-grouped) ignoring any leading punctuation - `--set-upstream option (git push)` and `-u option (git branch)` are filed under **S** and **U** respectively (matching where "set-upstream"/"u" itself would sort), not lumped into a separate "symbols" section, the same way a technical book's own index treats command-line options.

This is the one feature in `prodockit.pdf` that genuinely needs a two-pass build: a term's own page number can only be known once WeasyPrint has already laid the whole PDF out once - confirmed directly, before settling on this design, that CSS's own `target-counter()` *can* resolve a page number in a single pass, but can't deduplicate two markers that land on the same page (nothing on the Python side can know that without already knowing the layout) - accepted as a real limitation and not used here, in favour of a genuinely clean, deduplicated index. See `prodockit.pdf.index` for exactly how the two passes work, if you're scripting your own build pipeline.

10.4 Sub-entries

`\index{Parent!Child!Grandchild}` - `!` separates up to three levels in practice, matching LaTeX `makeidx`'s own long-established `\index{primary!secondary!tertiary}` convention - nests a term under another, the same way a printed book's own index groups related entries together (e.g. "staging area" with "adding"/"modified" indented beneath it) rather than listing every term as one flat alphabetical run:

Markdown

```
Now generate the \index{Git!ssh keys} to use for
authentication.
```

Result

Now generate the ssh keys to use for authentication.

Only the *last* segment displays inline (`ssh keys` above) - wherever the term is actually mentioned in your prose - the earlier segments (`Git`) are only ever used to build the generated index's own nesting, and never appear inline themselves. Renders a nested entry like:

```
G
```

```
Git
    ssh keys, 13, 89
```

A parent with no marker of its own anywhere (like `Git` above) still gets its own line - a bare category label with no trailing page list - so its children have somewhere to nest under.

10.5 Code-styled terms

Backticks around the *last* segment mark a command or other code term - it displays inline in a real `<code>` element instead of plain text, and the generated index entry renders the same way:

Markdown

```
Run \index{`git commit`} to save your changes.
```

Result

Run `git commit` to save your changes.

Combine this with sub-entries by putting the backticks around just the last segment - nesting a code-styled `git commit` entry under a plain `Git` one:

```
\index{Git!`git commit`}
```

Both render an index entry in your document's own monospace font, matching how the term already displays inline:

```
G

Git
    git commit, 13, 89
```

> **Showing this syntax as literal example text**
>
> Unlike a plain `\index{Term}` (protected by a real code span, the same way this page's own examples above are written), the code-styled pattern has to

run *before* Python-Markdown's own backtick handling, so it can recognise its own inner backticks - a side effect is that wrapping the whole call in inline backticks doesn't protect it the way it does for the plain syntax. A fenced code block (as every example on this page already uses) still works, since it's stashed before any inline pattern - this one included - ever runs.

10.6 Linked terms

A term can be a markdown link - the same "not exempted from later inline passes" behaviour that lets `\index{*emphasised*}` work also resolves a link, rather than leaving `[Text](url)` as literal text:

Markdown

```
\index{[Git](https://git-scm.com/)} is a version control system.
```

Result

Git is a version control system.

`prodockit.pdf.index`'s own `mark_index_terms()` still extracts the plain "Git" text correctly either way - it already strips a nested `<code>`/`<em>` the same way, via BeautifulSoup's `get_text()`.

⚠ **`attr_list` (`{target="_blank"}`) doesn't combine cleanly**

This project's own convention for an external link - `[Git](https://git-scm.com/){target="_blank"}` - doesn't work wrapped in `\index{}`: `attr_list` attaches the attribute to whichever element comes immediately before it, which after `\index{}` processes is the *outer* `<span class="index">`, not the link itself - a `target` on a `<span>` does nothing, silently losing the "open in a new tab" behaviour. Putting the attribute *inside* the `\index{}` call doesn't work either, since its own `{`/`}` braces confuse `\index{}`'s own regex, which stops at the first `}` it finds. If you need `target="_blank"` on a linked term, use a raw inline `<a>` tag instead, which needs no `attr_list` at all:

```
\index{<a href="https://git-scm.com/" target="_blank">Git</a>} is a version control system.
```

10.7 CSS hooks

A flat `\index{Term}` renders as `<span class="index">Term</span>` - no other class or attribute, and nothing left behind to strip. A hierarchical `\index{Parent!Child}` additionally carries the full path on `data-index-term` (`<span class="index" data-index-term="Parent!Child">Child</span>`), read by `prodockit.pdf.index` to build the nested index - harmless for a project's own CSS, which would only ever need to target the shared `.index` class either way. A code-styled term further carries `data-index-code="true"` and wraps its own text in a real `<code>` element (`<span class="index" data-index-code="true" data-index-term="Term"><code>Term</code></span>`) - picking up whatever `code {}` styling your project already has, on the website and in the generated index alike, with nothing extra to configure. Unlike `prodockit.refs` / `prodockit.citations` / `prodockit.glossary`, there's no resolved/unresolved state to distinguish (nothing to resolve - a term either renders or the whole page fails to parse), so no extra styling hook is needed by default. A project wanting to visually highlight indexed terms on the live website can target `.index` directly.

Why a Markdown extension, not `attr_list`?

Every other prodockit extension's own marker syntax builds on `attr_list` - `prodockit.index` needs a real Markdown extension instead because plain `attr_list` can't wrap arbitrary inline text in a span on its own: confirmed directly, `[Term]{.index}` is left as literal text (`attr_list` only reaches inline content already wrapped in something it recognises - emphasis, a link, code - each of which would force an unwanted visual side effect just to mark a term). Raw inline HTML (`<span class="index">Term</span>`) works today with no extension at all, but is exactly the "disrupts normal writing flow" outcome a good marker syntax should avoid.

11. 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.

11.1 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.

11.2 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.

11.3 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
```

11.3.1 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.

11.3.2 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.

11.3.3 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.

11.3.4 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.

11.3.5 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.

11.3.6 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.

11.3.7 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.

11.3.8 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.

11.3.9 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.

11.3.10 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.

11.4 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.

11.5 Status

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

12. Website macros

`prodockit.zensical_macros` provides a handful of Jinja variables and macros for Zensical's own macros plugin - the pieces a professional/academic report's website commonly wants that aren't specific to any one project: a site-wide word count, the git-detected repository URL, chapter/appendix numbering that continues across pages, and reference/acronym/glossary list spacing that matches `prodockit.pdf`'s own PDF output.

12.1 Quick start

Add it alongside your own project's `macros.py` (which keeps anything genuinely project-specific - a custom macro, institution branding, and so on):

```
[project.markdown_extensions.zensical.extensions.macros]
module_name = "macros"
modules = ["prodockit.zensical_macros"]
```

Zensical's macros plugin loads `module_name` and every entry in `modules`, merging all of their variables/macros into the same Jinja environment - so a project with no macros of its own can drop `module_name` / `macros.py` entirely and just use:

```
[project.markdown_extensions.zensical.extensions.macros]
modules = ["prodockit.zensical_macros"]
```

12.2 Variables

Variable	Description
`{{ word_count }}`	Prose word count across every nav page except the first (assumed to be the cover page) and any page flagged `exclude_from_word_count: true` in its own front matter - a comma-formatted string (e.g. `"9,971"`).
`{{ repo_url }}`	The fully-qualified `https://` URL for the current checkout's git `origin` remote (converted from `git@host:path.git` SSH syntax, with any embedded CI credentials stripped) - `""` if there's no git remote configured.
`{{ site_name }}`	`project.site_name` from `zensical.toml`.

12.3 Macros

Macro	Description
`{{ heading_counter_reset(page) }}`	Place near the top of every page - continues chapter/section numbering (and the matching sidebar numbering) across pages, from this page's position in nav. See below.
`{{ reference_style() }}`	Place once near the top of a references page - controls `.reference` paragraph spacing. See below.
`{{ acronym_style() }}`	Place once near the top of an acronyms page - matches `reference_style()`'s default spacing.
`{{ glossary_style() }}`	Place once near the top of a glossary page - matches `reference_style()`'s default spacing.

12.3.1 `heading_counter_reset(page)`

Emits a `<style>` block that continues heading numbering from wherever the *previous* page left off, using `prodockit.headings.prescan()` - the single source of truth for what number/letter a page actually gets, so this always matches what `\ref{}` resolves to for a heading on this page. Nothing else needs to change when pages are reordered or headings are added/removed.

Set `project.extra.heading_numbering = false` in `zensical.toml` to turn numbering off entirely (content and sidebar) across the whole site. A page flagged `is_appendix: true` in its own front matter gets letter-based numbering instead - "Appendix A", "A.1", "A.1.1" - matching `prodockit.headings`' own `appendix_attr` default.

12.3.2 `reference_style()` / `acronym_style()` / `glossary_style()`

Controls list-entry spacing, driven by the same `project.extra.*` settings `prodockit.pdf` reads for the PDF, so both outputs stay in sync from one configured value:

Setting	Default	What it does
`reference_style`	`"european"`	`"european"`: single line spacing throughout, no indent, entries close together. `"global"`: single line spacing within each entry, double spacing *between* entries, with a hanging indent on wrapped lines (the common APA/MLA/Chicago style). Only `reference_style()` /the References page switches look - acronyms/glossary always use the tight "european" spacing.
`reference_spacing_european`	`"-0.8em"`	Gap between entries, "european" style - also used unconditionally for the acronym/glossary lists.
`reference_indent_global`	`"1.27cm"`	Hanging indent on wrapped lines, "global" style.
`reference_spacing_global`	`"2em"`	Gap between entries, "global" style.

12.4 Status

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

13. Release Notes

13.1 0.10.5 (2026-07-25)

Fixed a real bug found by reproducing it directly: a cross-page `\ref{id}` under `zensical build` depended on whether the page defining `id` happened to be rendered before the page referencing it, in the same Python process - `zensical build` renders pages neither in nav order nor necessarily all in one process, so this was pure luck. Reproduced locally on a 3-page site: the previous release left 1-5 references as `??` per build, varying from one otherwise-identical build to the next (only 2 of 12 clean builds fully resolved).

`prodockit.headings` now pre-scans every nav page's headings (ids, levels, section numbers) into the shared registry before any page is converted - the same idea `prodockit.citations` / `prodockit.glossary` already use for their own cross-page definitions - so resolution no longer depends on build order. A page actually rendered in this process still supersedes its own pre-scanned entries with the real ones. 20 consecutive clean builds now produce byte-identical, fully-resolved output; `prodockit-template`'s entire built site is byte-identical before and after this change.

Also fixed a second bug found while testing the above: extension order isn't guaranteed, so `prodockit.refs` can construct its own default `HeadingsExtension` and trigger the pre-scan with per-document numbering before a project's configured `numbering = "continuous"` instance runs - silently showing a cross-page reference's number one step behind (`1.1` instead of `2.1`) roughly 1 build in 12. The pre-scan now reruns if a differently-configured instance appears.

Fixes #54. See #99 for a related, separate limitation found along the way (this pre-scan can go stale under `zensical serve`'s live-reload - not fixed here).

13.2 0.10.4 (2026-07-25)

- Added `CONTRIBUTING.md` and a `.github/pull_request_template.md`, adapted from prodockit-template's own - library-specific setup (`pip install -e ".[dev]"`, `pytest` / `ruff` / `mypy`, the real-`pandoc` requirement for `prodockit.bibliography`'s own tests) rather than the template's assignment-writing framing. Linked from README.md's Development section.
- Docs: added an admonition to `headings.md` documenting a real gap this project's own docs hit directly while enabling every prodockit extension on its own site (#87) - Zensical's automatic cross-page id sharing only warns (rather than raising) on a heading name shared across pages, and build order isn't guaranteed stable, so the "keeping the first" winner can change

between builds. Documents the fix - an explicit, page-prefixed id via `attr_list` - pointing to this project's own docs as a worked example.

No code changes.

13.3 0.10.3 (2026-07-25)

Docs: several extension pages mixed "why it was built this way" design rationale into their opening paragraph, ahead of the practical "what it does"/"how to use it" content most readers want first - moved that reasoning further down each page instead:

- `bibliography.md`: trimmed the intro to the core value proposition, moved the Pandoc-delegation rationale and architecture diagram out of Requirements (now just what to install) into a new "How it works" section after Reference, and folded the "can be enabled alongside prodockit.citations" note into Comparing the two approaches.
- `citations.md` / `glossary.md`: moved the "why bundled into one extension, unlike headings/refs" rationale from the intro into their own Syntax section.
- `tables.md`: moved the "auto-enables Python-Markdown's own tables extension" implementation note from the intro into Syntax.
- `index-terms.md`: moved the "why a Markdown extension, not attr_list" rationale from the intro to the end of CSS hooks, wrapped in its own admonition naming the subject explicitly.

`headings.md` / `refs.md` were already lean and needed no changes. No syntax, behaviour, or code changes.

13.4 0.10.2 (2026-07-25)

Docs: `extensions/index-terms.md` described the live website's search as generic "browser/Ctrl-F search" - updated to point at Zensical's own built-in site search instead, a more accurate and discoverable description of how a reader actually finds a term on the live website. No code changes.

13.5 0.10.1 (2026-07-24)

Docs: `prodockit.index`'s marking syntax and `prodockit.pdf`'s back-of-book index *generation* were split across two pages (`extensions/index-terms.md` and `pdf.md` respectively), even though the marker is useless without turning `pdf_include_index` on and vice versa - `prodockit.bibliography`'s own docs already combine marking and generation into one page. Moved the generation content into `extensions/index-terms.md` as a new "Generating the index" section, merged the per-feature rendered-output examples into their existing marking sections instead of duplicating them, and renamed the page from "Index terms" to "Index (pdf-only)" now that it covers the whole feature. `pdf.md`

keeps only a short pointer, matching how `prodockit.bibliography` is treated there. No code changes.

13.6 0.10.0 (2026-07-24)

`prodockit.bibliography`'s `\bibliography` marker now takes two optional, positional parameters - `\bibliography{<file>}{<true|false>}` - so a project can generate both a strict **References** section (only sources actually `\citebib{}`-cited in the text) and a broader **Bibliography** section (every entry, including background reading that's never individually cited) in one build, from the same or different `.bib` files:

```
<!-- references.md -->
\bibliography{}{true}
```

```
<!-- bibliography.md -->
\bibliography{background.bib}
```

Bare `\bibliography` is completely unchanged - fully backward compatible, no breaking change. A `\citebib{id}` citation now cross-links to whichever marker's page actually defines that entry (via a new, lightweight `.bib` entry-key discovery helper, not a CSL reimplementation) rather than assuming a single global bibliography page - the common single-file case is unaffected. See Multiple sections: References and Bibliography for the full syntax and worked examples.

Fixes #89.

13.7 0.9.0 (2026-07-24)

Breaking: `copyright` / `pdf_copyright` are now a real HTML fragment, rendered as a real DOM element in the PDF's running footer via CSS Paged Media's `position: running()` / `content: element()`, instead of being escaped into a CSS `content: "..."` string. This is what makes a real `<a href="../...">` link inside either value survive as a real, clickable link in the PDF - on every page, not just wherever the source element itself sits - matching how Zensical's own website-side `copyright` setting already works. Use a real `<br>` for a forced line break; the `\A` CSS-escape trick added in 0.8.0 only ever worked for a plain string, not real markup, and no longer applies - update any existing `pdf_copyright` using it to a real `<br>` instead.

`prodockit.pdf.css.build_css()` no longer takes a `copyright_text` parameter (`site_name` is unaffected, still a plain CSS content string) - no formal, versioned public API surface yet for `prodockit.pdf` (see prodockit-extension#7), so this is an acceptable break at this stage.

This project's own cover page (`docs/index.md`'s hero subtitle) no longer hyperlinks the word "Zensical" - it stays as plain text, matching this project's own PDF footer now crediting Zensical/prodockit with real links instead of the cover page doing it via a website-only, PDF-invisible link.

13.8 0.8.1 (2026-07-24)

Docs: this project's own docs site and PDF were missing the "Made with Zensical and prodockit" credit line that `overrides/partials/ copyright.html` / `pdf_copyright` (new in 0.8.0) already give a downstream project - added both here too, via a new `overrides/partials/ copyright.html` for the website and `extra.pdf_copyright` in `zensical.toml` for the PDF, so this site credits itself the same way a project built with it does. No library code changed.

13.9 0.8.0 (2026-07-24)

New `pdf_copyright` setting: `project.copyright` (a plain, native Zensical setting) already feeds the footer of both the website and the PDF by default - `pdf_copyright` is an opt-in override for the PDF's footer only, for a project that wants its PDF footer to say something different from its website's (e.g. adding a "Made with Zensical and prodockit" credit line only to the downloadable PDF, not the live site). Write a forced line break in either setting with a literal `\A` inside a TOML *literal* string (`'''...'''`) - see Copyright text for the full mechanism and why a literal string is required.

Also fixed a real, previously-undocumented rendering gap found while building this: a `\A` forced line break inside a `content` string only actually renders as a line break under `white-space: pre-line` - under WeasyPrint's default `white-space: normal` it silently collapsed to a plain space instead. Both the single-sided and double-sided verso copyright footer boxes now set `white-space: pre-line` so the forced break always works as expected.

13.10 0.7.1 (2026-07-24)

This project's own documentation site now enables every prodockit extension (`prodockit.headings`, `prodockit.refs`, `prodockit.citations`, `prodockit.glossary`, `prodockit.bibliography`, in addition to the `prodockit.tables` / `prodockit.index` already enabled) via `zensical.toml`, dogfooding the full set rather than just the two used to build this site previously.

Doing so surfaced a real bug: Zensical does not render pages in a stable order between builds, so a heading name shared across two or more pages (e.g. "Quick start", "Syntax", "Options") non-deterministically resolves its id collision differently from one `zensical build` to the next - confirmed by running repeated clean builds and observing the reported "keeping the first" winner change between runs. Fixed by giving every colliding heading across the docs an explicit, unique, page-prefixed id via `attr_list` (e.g. `## Quick start {: #refs-quick-`

`start }`), rather than relying on build order at all. No library code changed - this is a docs-content-only fix, and not something a project sharing a heading name across its own pages will normally have to think about, since a one-off name collision is far less likely there than in this project's consciously-parallel per-extension documentation structure.

13.11 0.7.0 (2026-07-24)

Breaking: `prodockit.bibliography` now uses its own `\citebib{id}` syntax instead of `\cite{id}`. Previously it registered the same `\cite{id}` pattern `prodockit.citations` uses, at the same inline-pattern priority - enabling both extensions together left it undefined which one actually resolved a given `\cite{...}` occurrence. Renaming `prodockit.bibliography`'s own syntax removes the conflict entirely: both extensions can now be enabled in the same build with no interference, each citing its own sources by its own marker. A project still using `prodockit.bibliography` on its own needs to update every `\cite{id}` in its source to `\citebib{id}` - the old syntax no longer resolves.

13.12 0.6.8 (2026-07-21)

`build_pdf_from_zensical_config()` (what `prodockit pdf` runs) now supports cover page markers, so a project no longer needs its own custom Python just to fill in a cover page's word count/repo URL/release tag - found via `prodockit-template`, whose `build_pdf.py` had grown to nearly nothing except this one piece:

- `{WORDCOUNT}` - the site-wide word count (the same value a `{{ word_count }}` website macro shows).
- `{REPOURL}` - the git-detected repo URL.
- `{RELEASE}` - the latest published GitHub/GitLab release tag - the whole line is dropped instead if there isn't one.
- `{{ site_name }}` - substituted literally, since `prodockit pdf` never evaluates Jinja.

All four are opt-in by literally writing the marker in your `nav`'s index page - no new `zensical.toml` setting needed. See Cover page markers.

Also new: `pdf_extra_css`, a stylesheet meant *only* for the PDF (e.g. a rule that would look wrong on the live website), concatenated after `extra_css` - the same `["stylesheets/print.css"]` role a project's own custom PDF-build script might have hardcoded outside `zensical.toml` entirely before, now expressible as ordinary configuration.

Also fixed two real bugs found while building this:

- `extra_css`'s (and now `pdf_extra_css`'s) own relative `url(...)` references (e.g. a light/dark logo swap or a header background image) were passed through unresolved, pointing nowhere once compiled into the PDF's

own temporary work directory - now resolved and base64-embedded, matching how a local `<img>` reference already was.
- `copyright` / `site_name` were passed straight into `build_pdf()`'s generated CSS `content: "..."` string with no escaping at all - `project.copyright` is commonly a triple-quoted TOML string spanning multiple lines, and a raw embedded newline (or a literal `"`) silently broke the whole generated rule, dropping the running header/footer entirely with no error. Both are now collapsed to one line and escaped before being passed through.

13.13 0.6.7 (2026-07-21)

Fixed `prodockit.pdf.html.fix_up_page_html()` permanently embedding *both* halves of a `#only-light` / `#only-dark` (or GitHub's `#gh-light-mode-only` / `#gh-dark-mode-only`) image pair in a PDF, stacked one after the other, instead of just one - found via `prodockit-template`'s own cover page hero graphic showing twice. A PDF has no light/dark toggle to make that convention meaningful, but `to_base64_data_uri()` already strips anything from `#` onward before resolving the file (to find the right one), so the resulting `data:` URI has no trace of the fragment left for any stylesheet to hide either half by. The `#only-dark` / `#gh-dark-mode-only` half is now dropped entirely rather than embedded.

13.14 0.6.6 (2026-07-21)

- Docs: the cover page hero graphic (`docs/assets/cover-hero-*.svg`) used a different colour palette in light mode (blue) than in dark mode (green) - recoloured the light variant to match dark exactly, so the hero reads the same regardless of theme. The "Download PDF" button also picked up this same green, rather than the theme's default primary colour.
- `prodockit.pdf.css`'s back-of-book index letter-group headings (`h2.prodockit-index-letter` - the "A", "B", "C" separators) were hardcoded to the hero graphic's *old* light-theme blue - updated to match the now-green hero, which a PDF always shows regardless of a project's own website light/dark toggle.
- No functional (Python package behaviour) changes beyond the index letter colour.

13.15 0.6.5 (2026-07-21)

Extends the 0.6.4 always-excluded-directory mechanism in `prodockit.pdf.source_bundle` to two more classes of vendored, never student-written content:

- Any directory literally named `styles` - a Vale `StylesPath` (conventionally named this way) holds downloaded rule packs (e.g. the Microsoft, proselint, and Readability style guides), typically tracked for offline/CI builds rather than gitignored.

- Common dependency lockfiles by exact file name - `package-lock.json`, `npm-shrinkwrap.json`, `yarn.lock`, `pnpm-lock.yaml`, `Pipfile.lock`, `poetry.lock`, `Cargo.lock` - machine-generated by a package manager, never hand-written, and often thousands of lines each.

Neither is project-configurable, matching 0.6.4's `.icons` exclusion: a project can't reach for the same knob to narrow what a bundle archives.

Also fixes `source_bundle.pdf`'s running header naming the wrong file: a file's own last page could show the *next* file's name instead of its own, because the invisible marker that sets the header text had no page break of its own - only the following content did - so it rendered on the tail end of the previous file's last page. The break now moves onto the marker itself, so the string-set and the page it applies to always agree.

13.16 0.6.4 (2026-07-21)

`prodockit.pdf.source_bundle` now always excludes any directory literally named `.icons` (e.g. a `custom_icons` directory, per `pymdownx.emoji`'s own convention) from `source_bundle.pdf`, regardless of `.gitignore` - found via `prodockit-template`, whose own vendored icon packs (`overrides/.icons/bootstrap`, `overrides/.icons/gitlab` - together ~2,500 unused SVGs) turned a source bundle meant to hold a student's own report content into a 3,000-page dump of unreferenced vendor assets. `.gitignore` alone can't fix this, since these directories are typically *tracked* (needed for the site/PDF to build at all) - deliberately not a project-configurable setting, so a project can't reach for the same knob to exclude something that should actually be archived.

13.17 0.6.3 (2026-07-20)

Bug fixes found by an in-depth test-coverage review of the extensions and PDF pipeline test suites - each paired with a new regression test.

- Fixed `prodockit pdf`'s CLI command showing a raw, unhandled traceback instead of a clean `Error: ...` message when `pdf_source_bundle` was enabled and the underlying `git`/`weasyprint` invocation failed - `SourceBundleError` wasn't in the CLI's caught exception tuple.
- Fixed `prodockit.headings` numbering a skipped heading level (e.g. h1 followed directly by h3, or a document starting below h1) with a literal "0" segment (e.g. "1.0.1") - a shallower level with no heading of its own yet is now treated as an implicit first one instead.
- Fixed `prodockit.pdf.mermaid` letting an uncaught `OSError`/`PermissionError` (e.g. a non-executable `mmdc` binary) escape instead of failing just that one diagram gracefully.
- Fixed `prodockit.pdf.source_bundle` crashing the whole bundle build on a file that's valid UTF-8 in the first 8 KiB sniffed to decide "is this text?" but not further in - now skipped like any other binary file instead.

- Fixed `prodockit.pdf`'s generated Lua filter producing broken syntax if a configured math/tex2svg path contained a quote or backslash - both are now escaped.
- Fixed `prodockit.pdf.build_pdf()` having no timeout on the underlying `pandoc`/WeasyPrint invocation, so a hang (e.g. a pathological CSS layout) could block the whole build indefinitely - added a `pandoc_timeout` parameter (default 30 minutes).
- Fixed a back-of-book index term nested more than three levels deep rendering with no extra indent at all, since the generated CSS only defines an indent step up to level 3 - now clamped to the deepest available indent instead.
- Substantially expanded test coverage across the extensions and PDF pipeline test suites (shared registries, cross-page linking, malformed input, table/index edge cases, icon/rotation/CSS edge cases) - no other functional changes.

13.18 0.6.2 (2026-07-20)

- Docs: fixed a real bug found by checking the live site after 0.6.1 - four spots in `docs/extensions/index-terms.md`/`docs/pdf.md` (plus two more in this same changelog) tried to show the code-styled `\index{}` syntax as literal example text using *inline* backticks around a hierarchical, code-styled path. Confirmed directly this doesn't work the way it does for the plain syntax - the code-styled pattern has to run before Python-Markdown's own backtick handling (see 0.6.0's own entry below), so inline backticks don't protect it, and the live site was rendering a raw internal Python-Markdown placeholder string instead of the intended literal text. Moved each one to a fenced code block (already documented as the safe way to show this syntax) or reworded to avoid the literal example entirely.
- No functional changes.

13.19 0.6.1 (2026-07-20)

- Docs: `prodockit.index` (new in 0.6.0) was missing from `README.md` - and so from PyPI's own project page - entirely: added it to the "Status" line and the extensions table, and mentioned `pdf_include_index` alongside `prodockit.pdf`'s other PDF-only features. Also added it to `pyproject.toml`'s own `description` (PyPI's summary line) and `src/prodockit/__init__.py`'s module docstring, both of which had the same gap.
- No functional changes.

13.20 0.6.0 (2026-07-20)

- New `prodockit.index` extension: mark a term inline with `\index{Term}` for a traditional, PDF-only back-of-book index (browser/Ctrl-F search covers this on the live website, so there's no equivalent there) - the term displays

inline exactly as written and is marked for indexing in one go, no separate "definition" step. Needed its own extension rather than the usual `attr_list` marker convention every other prodockit extension uses - confirmed directly plain `attr_list` can't wrap arbitrary inline text in a span on its own.
    - **Sub-entries**: `\index{Parent!Child!Grandchild}` (up to three levels deep in practice, matching LaTeX `makeidx`'s own `\index{primary!secondary!tertiary}` convention) nests related entries together instead of listing every term flat.
    - **Code-styled terms**: backticks around the last segment - or, combined with sub-entries, around just the last segment of a hierarchical path - mark a command/code term: it displays inline in a real `<code>` element, and the generated index entry renders the same way.
    - A term can be a markdown link or contain nested emphasis/code - confirmed directly neither needs special handling, since a term isn't exempted from Python-Markdown's own later inline-pattern passes the way `\ref{id}` / `\cite{id}` / `\gls{id}` are.
- New `prodockit.pdf.index`: the two-pass build (a term's own page number can only be known once WeasyPrint has already laid the PDF out once) behind `pdf_include_index` / `pdf_index_title` (both off/unset by default) - a traditional, two-column, letter-headed index page (matching this project's own cover page hero graphic colour), alphabetised ignoring leading punctuation (so `--set-upstream option` files under "S", not a separate symbols section), with consecutive pages collapsed into an en-dash range (`67–70`). Requires the new optional `pymupdf` dependency - `pip install prodockit[index]`.
- Fixed a real bug found while writing tests: code-styling a non-last segment of a hierarchical term - never a supported combination, but this shouldn't have corrupted anything either - used to leak a raw Python-Markdown internal stash placeholder into the generated index instead of failing gracefully - a real rendered PDF would have shown a nonsense category label instead of "Git".

13.21 0.5.0 (2026-07-19)

- New `prodockit.pdf.source_bundle`: bundles every text/source file `.gitignore` doesn't exclude into a separate `source_bundle.pdf` at a project's own top-level directory - 8pt Courier, wrapped lines, each file starting its own page, a running header (`site_name` on the left, that page's own file path on the right), and a "Page N of M" footer. Off by default; set `pdf_source_bundle = true` under `[project.extra]` to turn it on. Independent of the rest of `prodockit.pdf` - there's no Markdown involved, so it skips Pandoc entirely and hands a small, self-contained HTML document straight to WeasyPrint. File discovery shells out to `git ls-files --cached --others --exclude-standard` rather than reimplementing `.gitignore`'s own matching rules; text/ binary filtering is content-based, not by file extension.

- Docs: this site's own header now shows a PDF download icon next to "view" (an `overrides/partials/actions.html` override, linking to that page's own per-page PDF) instead of a "Download this page as PDF" text link at the top of the page - removed from every page that had one. Since the new icon is template markup rather than Markdown content, it also no longer shows up inside the PDF itself (no `.web-only` CSS trick needed, unlike the link it replaces).

13.22 0.4.2 (2026-07-19)

- Docs: matched more of this site's own theme config to prodockit-userguide's - the header's repo link now shows the actual GitHub logo instead of Zensical's default Git icon; the "View source of this page" button now shows an eye icon instead of a generic file icon; every admonition (e.g. the "tip" callout in `citations.md`) now uses the same custom FontAwesome icon set userguide uses instead of Zensical's own bundled defaults - this also feeds into `prodockit.pdf`'s own admonition icons, so PDF output picks it up too; added the matching theme features userguide already had (`content.tabs.link` in particular actually affects this project's own tabbed content); and swapped the palette toggle icons to match userguide's own light/dark convention.
- No functional (Python package) changes.

13.23 0.4.1 (2026-07-18)

- Docs: reworked this site's own chrome to match prodockit-userguide's - a new split hero cover page ("Home"), reusing that project's own logo/ favicon/ illustration assets; top-level nav moved to a top tab bar with the right-hand page TOC merged into the left sidebar instead; the previous cover page's own prose moved to a new "Introduction" page.
- Fixed a real bug found along the way: `zensical.toml`'s own `copyright` was a triple-quoted, multi-line TOML string - `prodockit.pdf` substitutes it verbatim into a CSS `content: "..."` string for the PDF's running footer, and the embedded newline silently broke that declaration, dropping the whole footer with no error (this site's own PDF footer had no copyright text at all). Fixed to a single-line string, and switched `&copy;` to a literal `©` character - a CSS content string doesn't decode HTML entities either.
- Fixed the deploy workflow missing a per-page PDF build step for the new Introduction page (its own "Download this page as PDF" link 404'd), and that page's leftover PDF link (still the old cover page's, pointing at the whole-site PDF) to the same per-page convention every other content page already uses.
- No functional (Python package) changes.

13.24 0.4.0 (2026-07-18)

- New `pdf_double_sided` option: a duplex-printing layout for book/handbook-style documents printed and bound on both sides. Verso (left-hand) and recto (right-hand) pages mirror their header/footer content and page margins (new `pdf_margin_inner` / `pdf_margin_outer`, replacing `pdf_margin_left` / `_right` in this mode) via CSS Paged Media's `@page :left` / `:right` selectors - chapter title and page number always on the outer, fore-edge corner; site name and copyright always on the inner, spine-side corner, whichever physical side that is for a given page. Every numbered heading now starts its own recto page (`break-before: recto`, auto-inserting a blank page as needed - confirmed directly this needs no Python-side page-counting logic at all), and a `prodockit-table-rotated` landscape page's own rotation direction now alternates by its final page position (270 degrees on recto, 90 on verso - the spine sits on the opposite physical side either way).
- New `recto_title` front matter key: overrides a 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 - confirmed directly this is a consequence of CSS `string()`'s "first value on this page wins" default policy) - useful for a chapter title too long to fit comfortably in the header, with or without `pdf_double_sided`.
- Off by default: a single-sided build is completely unchanged.

13.25 0.3.1 (2026-07-18)

- Docs: renamed `glossary.md`'s heading to "Acronyms and Glossary" and `citations.md`'s to "Citations or References" (and their matching nav labels); added a flow diagram to `bibliography.md`'s Requirements section (and fixed a real, unrelated gap found along the way - this docs site had no Mermaid `custom_fences` config at all, so a plain ```` ```mermaid ```` fence never rendered as a diagram anywhere on the site); switched the citation-style example to `harvard-cite-them-right.csl`; added an admonition pointing from `prodockit.citations` to `prodockit.bibliography`; and noted `prodockit.bibliography`'s own independent Pandoc invocation in `prodockit.pdf`'s "Limitations and workarounds".
- Docs: updated `README.md` (and so PyPI's own project page description) to include `prodockit.tables` / `prodockit.bibliography`, and to mention sideways tables/ `.web-only` / `.pdf-only` under PDF generation - it had gone stale since both extensions shipped in 0.3.0.
- No functional changes.

13.26 0.3.0 (2026-07-18)

- New `prodockit.bibliography` extension: an alternative to `prodockit.citations` for a `.bib`-backed reference list instead of a

hand-authored one. Define sources in a BibTeX/BibLaTeX `.bib` file, cite them with the same `\cite{id}` syntax, and get the resolved citation text and a full, auto-generated reference list formatted in any Citation Style Language (CSL) style (APA, IEEE, Harvard, ...) via Pandoc's own `--citeproc` - confirmed directly against real Pandoc output rather than reimplementing citation formatting, and rejected an actual LaTeX/biblatex toolchain as a new hard dependency along the way. Makes `pandoc` a required dependency for this extension specifically, including for a website-only build with no PDF. New `docs/extensions/bibliography.md` includes a "References and Bibliography" comparison of this, `prodockit.citations`, and what `prodockit-template` / `prodockit-userguide` currently do.
- New sideways (90-degree anticlockwise) tables in the PDF: wrap a table and its own caption in `<div class="prodockit-table-rotated" markdown="1">` to print it on its own landscape-sized page(s), spanning multiple pages with a repeated heading row exactly like any other table. Confirmed directly that a CSS `transform: rotate()` doesn't work for this (clips the table to one page and loses its heading row) - the actual rotation is applied afterwards via a `/Rotate` post-process on the finished PDF (new `prodockit.pdf.rotate` module, new `pypdf` dependency).
- `.web-only` content is now hidden in every PDF build automatically, via `prodockit.pdf.css`'s own always-included stylesheet - no project-side CSS needed any more. `.pdf-only` is documented as a one-line, centrally-sourced snippet instead (`prodockit` has no way to reach into a project's own website stylesheet), in a new "Web-only / PDF-only content" section in the PDF generation docs.

13.27 0.2.0 (2026-07-18)

- New `prodockit.tables` extension: gives a table column a percentage or fixed width via a `width` attribute already attachable to a header cell with `attr_list` - no new syntax. Column-width distribution beyond what's explicitly given is left to CSS's own `table-layout: fixed` algorithm rather than computed in Python. Ships with the matching CSS in `prodockit.pdf`'s generated stylesheet, and documents the equivalent rule a project's own website theme needs (see the new Tables docs page).
- New `prodockit pdf --markdown-file` / `-m` option: builds a PDF from a single markdown file instead of the whole `nav`, using the same `zensical.toml` settings as a full build.
- `prodockit.pdf`'s generated table CSS now draws a full grey 0.5pt grid - outer border and internal row *and* column lines (there was previously no line between columns at all) - and reads a project's own `extra_css` (from `zensical.toml`), so a project-specific `@media print` rule (e.g. hiding a website-only "Download PDF" link/button) also applies in the PDF.
- `prodockit.citations`: a resolved `\cite{id}` link now always gets `class="prodockit-cite-resolved"` (previously no class at all), matching

`prodockit.refs` / `prodockit.glossary`'s existing convention of a stable class for both the resolved and unresolved case.
- Docs: added a "CSS hooks" section to `refs.md` / `citations.md` / `glossary.md` (`headings.md` already had one), documenting every class/attribute each extension itself emits; replaced the docs site's "edit this page" link with "view this page" (a `content.action.view` link to the raw source rather than a GitHub edit form); added a whole-site PDF download button on the front page and a per-page download link on every other page, both built via the new `--markdown-file` option above.
- Fixed `prodockit.__version__` reporting a stale `"0.10.0"` (left over from before the `zendoc` → `prodockit` rename) instead of matching this package's actual, `pyproject.toml`-declared version.

13.28 0.1.1 (2026-07-17)

- Docs: reworded the package intro on the docs site and README (dropped the `pymdown-extensions` comparison, added a mention of the website macros and a one-line "kit for professional documentation" summary) - no functional changes.

13.29 0.10.0 (2026-07-15)

- New `prodockit.zensical_macros`: Jinja variables/macros for Zensical's own macros plugin - `{{ word_count }}` (site-wide prose word count, excluding the cover page and any page flagged `exclude_from_word_count: true`), `{{ repo_url }}` (git-detected repository URL), `{{ site_name }}`, and `heading_counter_reset(page)` / `reference_style()` / `acronym_style()` / `glossary_style()` macros. Add it alongside a project's own `macros.py` via `zensical.toml`'s `modules = ["prodockit.zensical_macros"]` - or use it alone if the project has no macros of its own.
- New `prodockit.wordcount`: the generic prose word-count utility (`count_words()` / `compute_word_count()`) behind both `prodockit.pdf`'s `{WORDCOUNT}`-style cover-page use and `prodockit.zensical_macros`' `{{ word_count }}` - previously duplicated independently by each downstream project needing both.
- New `prodockit.settings`: `flatten_nav()`, `heading_numbering_enabled()`, and `reference_style_values()` - the `project.extra.*` reading shared by `prodockit.pdf.config` and `prodockit.zensical_macros`, so the two agree on one set of fallback defaults instead of each hand-maintaining its own copy. `prodockit.pdf.config.build_pdf_from_zensical_config()` now uses these too (previously inlined), and its `pdf_math_dir` setting is now created automatically if configured to a directory that doesn't already exist (matching the auto-detected default's existing behaviour).

13.30 0.9.0 (2026-07-15)

- New `prodockit pdf` command: builds a complete PDF with no Python required, reading everything - nav, docs directory, fonts, page size, and all PDF-specific settings - from the project's own `zensical.toml`, the same way `zensical build` / `zensical serve` do. Installing `prodockit` now registers a `prodockit` console script (`pip install prodockit` is enough - no separate build script to write). See the new `prodockit.pdf.config` module (`build_pdf_from_zensical_config()`) for the config-to-`build_pdf()` orchestration this wraps: nav-tree flattening, per-page `is_appendix` front-matter detection, and auto-detection of a local `mmdc` (Mermaid) binary and MathJax `tex2svg` script, so a typical project needs no extra configuration beyond what it likely already has.
- `build_pdf()` gained `include_table_of_contents` / `table_of_contents_title` parameters (both used automatically by `prodockit pdf`): a generated table of contents is now inserted by default, right after a cover page if one is marked `is_index=True`, or at the very start otherwise.
- Rewrote the PDF generation docs page around the `prodockit pdf` command as the primary, and for most projects only necessary, way to use `prodockit.pdf` - `build_pdf()` and the individual pipeline pieces are now documented as the advanced, scripting-your-own-pipeline path.

13.31 0.8.0 (2026-07-15)

- New `prodockit.pdf.build_pdf()`: a one-call convenience wrapper around the rest of `prodockit.pdf` - hand it a list of already-rendered pages (`prodockit.pdf.Page`) and where to write the PDF, and it fixes up each page's HTML, generates the Lua filter and CSS, concatenates everything, and runs `pandoc`/WeasyPrint for you. Takes `output_path` (the PDF's own destination path) plus font/page-size/margin/header-footer/reference- style/ numbering/math parameters, all with sensible defaults. Raises the new `prodockit.pdf.PdfBuildError` (with the underlying `pandoc` exit code and stderr attached) if the build fails, rather than failing silently. `prodockit.pdf.html` / `.lua` / `.css` / `.icons` / `.mermaid` remain directly importable if you need more control over how the pieces fit together.
- Rewrote the PDF generation docs page around `build_pdf()` as the primary documented way to use `prodockit.pdf`, leading with a short, practical quick-start example rather than the implementation-level detail of how Pandoc/WeasyPrint's own quirks are worked around (that detail is still there, now further down, for anyone who wants it).

13.32 0.7.0 (2026-07-15)

- New `prodockit.pdf`: a Pandoc/WeasyPrint pipeline for building a standalone PDF from Zensical-rendered HTML - not a Python-Markdown extension (no `markdown.extensions` entry point), a plain function library, since a PDF build pipeline isn't a Markdown syntax extension:
  - `prodockit.pdf.html`: `fix_up_page_html()` and link/anchor/image helpers
  - fixes up one page's already-rendered HTML for Pandoc's own reader/writer quirks (attribute loss on `<p>`, raw `<svg>` not surviving the round trip to WeasyPrint, footnote/caption structural mismatches, cross-page link rewriting for a concatenated multi-page PDF, and more).
  - `prodockit.pdf.lua`: `build_lua_filter()` - chapter/appendix numbering, caption chapter-prefix numbering, tabbed-set reconstruction, and MathJax pre-rendering, generated as a parameterized Lua filter.
  - `prodockit.pdf.css`: `build_css()` - the compiled CSS a PDF needs on top of a project's own website stylesheet, including WeasyPrint-specific page-break tuning for headings, paragraphs, tables, code blocks, figures/captions, admonitions, and grid cards.
  - `prodockit.pdf.icons` / `prodockit.pdf.mermaid`: admonition icon resolution and Mermaid diagram pre-rendering, as standalone helpers.
- Fixed a real bug found while writing tests: the iframe→"Watch Video" admonition link builder stripped the video id from every single conversion (a replace-then-split ordering removed the just-added `?v=...` too) - now produces a working YouTube watch link.
- No formal, versioned public API surface yet (see prodockit-extension#7) - import whatever's needed directly, the same informal way as the rest of this package.
- New dependency: `beautifulsoup4` (>= 4.12).
- Broadened the package's own description: prodockit is now framed as a family of extensions for Zensical needed for professional and academic documentation, rather than "Python-Markdown extensions" specifically - `prodockit.pdf` isn't one, and the framing was due to broaden anyway now that PDF generation is in scope alongside cross-references/citations/glossary.

13.33 0.6.0 (2026-07-14)

- `prodockit.headings`: new `numbering="continuous"` option (Zensical only) - `h1` numbering carries on from wherever the previous nav page left off, instead of restarting at 1 on every page. Fixes `\ref{id}` showing the wrong number for a heading on a different page (it previously always showed that page's own per-document number, not the number actually displayed on the page - see zendoc-template#89).
- New `appendix_attr` option (default `is_appendix`): a page whose front matter sets this flag is numbered with a letter instead - "A", "A.1", "A.1.1" - and

doesn't consume a number from the numeric sequence, so later pages aren't left with a gap. Letters are assigned sequentially in nav order.
- New public `prodockit.headings.prescan(appendix_attr="is_appendix")` function: returns the same `(start_counts, appendix_letters)` pre-scan `HeadingsExtension` uses internally, for a consuming project's own build tooling (e.g. a template macro driving a presentational CSS counter-reset) to stay in sync automatically rather than re-deriving the same page-order/heading-count logic independently.

13.34 0.5.1 (2026-07-14)

- `prodockit.glossary`: a resolved `\gls{id}` now always renders with `class="prodockit-gls"` (previously it had no class at all), matching `prodockit.refs`' always-present base class. The unresolved case now renders `class="prodockit-gls prodockit-gls-unresolved"` (previously just `prodockit-gls-unresolved`, missing the base class), so a stylesheet has one stable hook (`.prodockit-gls`) regardless of resolution state, with `.prodockit-gls-unresolved` layered on top only when needed.

13.35 0.5.0 (2026-07-14)

- New `prodockit.glossary` extension: define a term once via `attr_list` (an id plus a `data-term` short display string), then insert it by id from anywhere with `\gls{id}`, which resolves to the term's own text, linked to its definition - e.g. `\gls{css}` → `CSS`. Unlike `prodockit.citations`' `\cite{id}` (which generates new bracketed citation text), `\gls{id}` inserts the term's own registered text in place - closer to LaTeX's `glossaries` package.
- One shared `GlossaryRegistry` covers both acronym-style and glossary-style entries - they're the same kind of thing (an id with a short display text), so acronym and glossary pages can reference each other, or be referenced from any other page, with no special wiring.
- Supports forward references within a document, an `unresolved` marker (`?` by default) for an unknown id, and the same automatic Zensical cross-page registry sharing and nav pre-scan (for citing/using a term before its defining page has been converted) that `prodockit.citations` got in 0.4.0.
- Refactored the nav pre-scan logic (previously private to `prodockit.citations`) into a shared, generic `prodockit._zensical.preseed_attr_from_nav` helper, since `prodockit.glossary` needed the identical scan.

13.36 0.4.0 (2026-07-14)

Fixes found migrating a real multi-page site's references page to

`prodockit.citations` for real - all discovered by actually building a real multi-page site, not just single-document tests:

- **Fixed a real correctness bug**: `prodockit.refs` / `prodockit.citations` were emitting a bare `#id` fragment for *every* resolved link, including a cross-page one - which only works by coincidence in a single concatenated PDF document, but 404s on an actual multi-page website (an `#id` fragment only navigates within the *current* page). Both now emit a real relative link (e.g. `references.md#id`, correctly adjusted for the citing page's own directory depth) when the target is on a different page, which Zensical already knows how to rewrite into the right clean URL - the same way a hand-typed cross-page Markdown link already works.
- New: `prodockit.citations` pre-scans every page in a Zensical build's nav for citation definitions before any page is actually converted, so citing a source *before* it's defined - the common case, since a references page is usually kept at the end of nav as an appendix - resolves correctly in a single `zensical build` pass, rather than only working from `zensical serve`'s live-reload. New `CitationRegistry.preseed()` method backs this; a real registration always supersedes a preseeded stub.
- `RefsExtension` gained a `source` option (mirroring `HeadingsExtension`'s), needed for the same-page-vs-cross-page link decision above.
- Fixed the nav pre-scan matching a citation-definition attr_list example shown literally inside a fenced code block in documentation - it now skips fenced content, the same protection `CitationDefTreeprocessor` already gets for free from the real Python-Markdown parser.

13.37 0.3.0 (2026-07-14)

- New `prodockit.citations` extension: define a source once via `attr_list` (an id plus a `data-cite-text` short display string), then cite it by key from anywhere with `\cite{id}` (or `\cite{id1,id2,...}` for multiple), auto-generating a bracketed, linked citation - `[Skoulikari, 2023]` - instead of hand-typing the link and text at every citation site.
- Supports forward references within a document, an `unresolved` marker (`?` by default) for an unknown key, and the same automatic Zensical cross-page registry sharing (with soft-fail on key collisions) that `prodockit.headings` / `prodockit.refs` got in 0.2.0.
- Auto-generating the references page's own listing from structured bibliographic data isn't built yet - see the extension's docs for the current scope.
- Fixed the `zensical.toml` installation examples in the docs: nested `[project.markdown_extensions.prodockit.headings]` tables don't work (Zensical only hoists the `pymdownx` / `zensical` namespaces that way) - the quoted-key form (`[project.markdown_extensions."prodockit.headings"]`) is required.

13.38 0.2.0 (2026-07-14)

- `prodockit.headings` / `prodockit.refs` now share their registry automatically under Zensical, without any explicit `registry` / `source` configuration: each extension detects Zensical's per-page rendering context and derives a stable `source` from the page's own path, fixing cross-page `\ref{id}` references not resolving.
- A heading id collision across two different sources, when detected via this automatic Zensical sharing, now logs a warning and keeps the first registration instead of raising `DuplicateIdError` - so two unrelated pages that happen to share a heading title (e.g. both have an "Overview" section) no longer break the build. Explicitly-shared registries (the manual multi-page pattern) still raise on a collision, unchanged.
- Fixed an extension-ordering bug: `prodockit.headings` and `prodockit.refs` now find and share each other's registry regardless of which order they're listed in - previously, only `prodockit.headings`-then-`prodockit.refs` worked reliably, and Zensical's own TOML-to-extension-list conversion doesn't preserve list order at all.

13.39 0.1.0 (2026-07-14)

Initial release.

- `prodockit.headings`: heading ids and hierarchical section numbering, backed by a shared `IdRegistry`.
- `prodockit.refs`: `\ref{id}` section cross-references, resolving to the target's current section number, including forward references within a document and across a shared registry.
- Documentation site built with Zensical, published at buckwem.github.io/prodockit-extension.

14. License

```
MIT License

Copyright (c) 2026 Mark Buckwell and contributors

Permission is hereby granted, free of charge, to any person
obtaining a copy
of this software and associated documentation files (the
"Software"), to deal
in the Software without restriction, including without
limitation the rights
to use, copy, modify, merge, publish, distribute, sublicense,
and/or sell
copies of the Software, and to permit persons to whom the
Software is
furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be
included in all
copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO
EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES
OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE,
ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER
DEALINGS IN THE
SOFTWARE.
```