Grimoire

Build a documentation website, and a printable PDF, from a directory of Markdown files.

Made with <3 by mplx <jennifer@mplx.dev>

2026-08-12

Grimoire

Build a documentation website - and a printable PDF - from a directory of Markdown files. What mdBook, MkDocs, and similar tools do - written in Jennifer (https://jennifer-lang.dev/).

> **A grimoire is a book of magical knowledge** - astrological rules, lists of angels and demons, spells, and instructions for making talismans - copied and recopied across Europe from the late Middle Ages into the eighteenth century. The best known are the *Clavicula Salomonis* , the *Grimorium Verum* , and the *Grand Grimoire* .
>
> The word comes from Old French *gramaire* , which is also the root of *grammar* and of *glamour* : a book of rules that, to anyone who could not read it, looked like sorcery. This one only makes websites.

Point it at a directory of Markdown. If that directory holds a `SUMMARY.md` it is used as the book outline, in the mdBook shape; if it does not, the outline is derived from the directory tree, the way MkDocs does. The output is a self-contained static site - themed, searchable, with a colour-mode selector - plus, on request, the whole book as one paginated PDF.

```
./grimoire init my-book     # scaffold a book
./grimoire build            # build the site
./grimoire build --pdf      # site plus the printable book
./grimoire pdf              # the printable book on its own
./grimoire serve            # build, then preview on :8080
./grimoire themes           # list the built-in themes
```

`grimoire` is a Jennifer script with a shebang; the program itself is `src/main.j` . It runs from any working directory and through any symlink, so either put the checkout on your `PATH` or link the launcher into a directory already on it, and the leading `./` goes away:

```
ln -s "$PWD/grimoire" ~/.local/bin/grimoire
```

A built site is a directory of files that works served from a web root, served from a subdirectory, or **opened straight off the disk over file://** - search included. Nothing is fetched from anywhere unless you opt in, and the only setting that reaches off the machine at all is `[highlightjs]` , which is off by default.

> **Read this manual as a single PDF (grimoire.pdf).** Every page in one paginated file, with a clickable outline in your reader's bookmark panel and `page/total` in the footer - handy for reading offline. It is built from these same pages on every build, so it never drifts from the site.

What it does

- **Two outline styles.** `SUMMARY.md` with part headings, nested entries, prefix and suffix chapters, drafts, and separators; or no `SUMMARY.md` at all, in which case the directory tree becomes the outline.
- **Anchors that match mdBook and GitHub exactly** , so hand-written cross-references survive a migration. `### REPL` (`cmd/jennifer/repl.go`) anchors at `#repl-cmdjenniferreplgo` , not `#repl-cmd-jennifer-repl-go` .
- **Links rewritten** : `x` becomes `guide/syntax.html#anchor` ; a directory `README.md` folds onto its `index.html` .
- **Client-side search** over per-section records, so a hit lands on the paragraph rather than the top of a long chapter. Opens with `/` or `Ctrl-K` , arrow keys to move, `Enter` to open. No search library: the index and the scorer are both Grimoire's own, and the index loads as a script rather than a `fetch` so it works over `file://` .
- **Ten themes** , each with a light and a dark palette.
- **A mandatory dark mode.** Every theme ships both palettes; the selector in the top bar offers light, dark, and follow-the-system, and the choice is stamped on the document before the first paint, so there is no white flash on navigation.

- **Syntax highlighting in two layers.** `[highlight]` alone highlights Jennifer **while the site is built** - no CDN, no JavaScript, nothing to load, and it works with scripting off. `[highlightjs]` additionally pulls highlight.js from a configurable CDN for the other languages. Both are off by default, and the second does nothing without the first.
- **A logo** beside the title: an SVG is inlined, so it inherits the colour mode and costs no extra request.
- **A printable book** : every chapter in one PDF, with a cover page, chapters starting on fresh pages, a nested bookmark outline, and document metadata. It wears the book's theme too - heading bars, table headers, code panels, and the tint and rule on a blockquote all come from the theme's light palette.
- **Chapters render in parallel** , one task per CPU, with the work split longest-chapter-first. With `--pdf` , the PDF is laid out *alongside* the site rather than after it.
- **Deterministic output.** The same input produces byte-identical files - including the search index, whose order does not depend on how the work was split across jobs.

Where to go next

Commands	every subcommand and flag
Configuration	grimoire.toml, key by key
Themes	the ten themes, with screenshots, and how to write one
Docker	running from the official image, and the Grimoire image
Internals	the source layout, and what Grimoire does to the Markdown
Performance	where the time goes, and why it scales the way it does

Requirements

Jennifer 0.25.0 or newer.

`build` , `pdf` , `init` , and `themes` run on both the default binary and `jennifer-tiny` ; `serve` needs the default binary, since `jennifer-tiny` stubs `httpd` . To pick the embeddable interpreter, invoke it directly rather than through the shebang:

```
jennifer-tiny run ./grimoire build
```

License

LGPL-3.0-only. Copyright (C) 2026 mplx <jennifer@mplx.dev>.

Using Grimoire

Commands

Five commands. Every flag has a long form; the common ones have a short form too, and `--help` on any command prints the same table.

```
grimoire --help          # the command list
grimoire build --help    # one command's flags
grimoire --version
```

A flag always wins over `grimoire.toml` , so a one-off build needs no edit to the file.

grimoire build

Render the site.

Flag	Default	What it does
-c, --config PATH	grimoire.toml	which configuration file to read
-s, --src DIR	from config	the directory of Markdown to build
-o, --out DIR	from config	where to write the site
-t, --theme NAME	from config	theme to use; grimoire themes lists them
-m, --mode MODE	from config	first-visit colour mode: auto, light, dark
--pdf	off	also render the book to PDF
--no-search	off	skip the search index and the search UI
-j, --jobs N	0	chapters to render in parallel; 0 is one per CPU
-v, --verbose	off	report each chapter as it is rendered
-q, --quiet	off	print nothing on success

```
grimoire build
grimoire build --theme nordic --out /tmp/preview
grimoire build --pdf --jobs 4
```

The exit status is `1` when the outline names a chapter with no file behind it - the rest of the book still builds, and the missing entries are reported on stderr.

grimoire pdf

Render only the PDF, skipping the site.

Flag	Default	What it does
-c, --config PATH	grimoire.toml	which configuration file to read
-s, --src DIR	from config	the directory of Markdown to build
-o, --out DIR	from config	where to write the PDF
-v, --verbose	off	report each chapter as it is laid out
--output FILE	from config	PDF filename, relative to the output directory
--paper SIZE	from config	a4 or letter

```
grimoire pdf
grimoire pdf --paper letter --output manual.pdf
```

grimoire serve

Build, then serve the result on a local address until interrupted.

Flag	Default	What it does
-c, --config PATH	grimoire.toml	which configuration file to read
-s, --src DIR	from config	the directory of Markdown to build
-o, --out DIR	from config	which directory to serve
-v, --verbose	off	report each chapter as it is rendered
-a, --addr ADDR	127.0.0.1:8080	address to listen on
--no-build	off	serve what is already there, without rebuilding

```
grimoire serve
grimoire serve --addr 0.0.0.0:9000 --no-build
```

The server runs a small pool of accept loops, because a browser asks for the page, the stylesheet, the runtime, and (on the first search) the index in parallel; a single-threaded loop would serialise them. It needs the default `jennifer` binary - `jennifer-tiny` stubs `httpd` and says so.

grimoire init [dir]

Write a starter book - `grimoire.toml` , a `SUMMARY.md` , and three chapters - into `dir` , or into the current directory if none is given.

```
grimoire init my-book
```

Existing files are never overwritten. Running it twice is safe: the second run reports each file it kept, which also makes it a way to add the pieces you deleted back.

grimoire themes

List the built-in themes with a one-line description of each. See Themes for what they look like.

Verbose output

`--verbose` names each chapter as it goes, which is how you find the one that is slow or throwing:

```
$ grimoire build --verbose --jobs 1
building docs -> site (theme grimoire, 13 chapters, 1 job)
  render  index.md  ->  index.html
  render  guide/syntax.md  ->  guide/syntax.html
  ...
  assets  stylesheet, runtime, search index
  copied  2 files from docs
```

Chapters render in parallel, so with the default `--jobs` the lines arrive in the order chapters *finish* , not the order they are listed. Pass `--jobs 1` when you want outline order.

`--verbose` and `--quiet` are not exclusive: verbose adds progress, quiet suppresses the closing summary. Passing both gives progress and no summary.

Configuration

`grimoire.toml` sits next to the book, and every key in it is optional - a directory of Markdown files builds with no configuration file at all. Point `--config` elsewhere to use a different one.

Two rules hold throughout:

- **A command-line flag wins over the file** , so a one-off build needs no edit.
- **A key of the wrong type keeps its default** rather than failing the build, and an unknown key is ignored. A half-written table degrades to the defaults instead of aborting a build that would otherwise have succeeded.

The whole file

Every key, with its default:

```
[book]
title = "Documentation"
description = ""
authors = []
authorsLabel = "Written by"
language = "en"
src = "docs"

[build]
out = "site"
jobs = 0                # 0 = one render task per CPU

[html]
theme = "grimoire"
mode = "auto"           # auto | light | dark
tocDepth = 3
sectionNumbers = true
footer = "Rendered with <a href=\"...\">Grimoire</a>"
repoUrl = ""
repoLabel = "Source"
editUrl = ""
favicon = ""
logo = ""

[highlight]
enabled = false

[highlightjs]
enabled = false
cdn = "https://cdnjs.cloudflare.com/ajax/libs/highlight.js/11.11.1"
style = "github"
styleDark = "github-dark"
languages = ["bash", "go", "json", "yaml", "xml", "ini", "nginx"]

[search]
enabled = true
bodyChars = 1200

[pdf]
enabled = false
output = "book.pdf"
paper = "a4"            # a4 | letter
bookmarkLevel = 3
pageNumbers = false
footerLeft = ""
titlePage = true
```

[book]

Key	Type	Default	Meaning
title	string	"Documentation"	shown in the sidebar and in every page title; also the PDF cover and Title

description	string	""	emitted as a description meta tag, and as the PDF Subject
authors	list of string	[]	the author meta tag, the PDF Author field, and the reader-facing credit
authorsLabel	string	"Written by"	what introduces those names where a reader sees them; "" prints them alone
language	string	"en"	the BCP 47 tag on the html element
src	string	"docs"	the directory of Markdown to build, relative to the working directory

`src` is where the outline comes from. If it holds a `SUMMARY.md` , that file is the outline; if not, the directory tree is walked instead.

`authors` is used two ways, and only one of them is labelled. The `author` meta tag and the PDF `Author` field want the names alone, because those are read by software. The page footer and the PDF cover want a credit, because a name standing on its own says nothing about what it is - so `authorsLabel` goes in front of it:

```
authors = ["Ada Lovelace", "Grace Hopper"]
authorsLabel = "Built with love by"   # -> Built with love by Ada Lovelace, Grace Hopper
authorsLabel = ""                     # -> Ada Lovelace, Grace Hopper
```

[build]

Key	Type	Default	Meaning
out	string	"site"	where the site is written
jobs	int	0	chapters rendered in parallel; 0 means one task per CPU

The output directory is created if it is missing. Files already there are left alone unless the build writes over them, so an unrelated file in `site/` survives - but so does a chapter you deleted from the book, which is worth knowing before publishing.

`jobs` above the chapter count simply idles the extra workers. See Performance for what raising it actually buys.

[html]

Key	Type	Default	Meaning
theme	string	"grimoire"	one of the ten themes; an unknown name falls back to grimoire
mode	string	"auto"	the colour mode a first-time reader gets: auto, light, dark
tocDepth	int	3	deepest heading level in the per-page contents; clamped to 1-6
sectionNumbers	bool	true	number the chapters in the sidebar
footer	string	the Grimoire credit	HTML placed in the page footer; "" for no footer
repoUrl	string	""	a source-repository link in the top bar; "" for none
repoLabel	string	"Source"	the label on that link
editUrl	string	""	an edit-this-page URL with a {path} slot; "" for none
favicon	string	""	a favicon path, copied into the site
logo	string	""	a logo shown beside the title, relative to src

`mode` only decides the **first** visit. Once a reader touches the selector, their choice is remembered and this setting no longer applies to them. Whatever it resolves to is stamped on the document before the first paint, so

navigating a dark site never flashes white.

`editUrl` substitutes the chapter's source path for `{path}` :

```
editUrl = "https://github.com/me/book/edit/main/docs/{path}"
```

`logo` pointing at an SVG gets **inlined** into the page, so it inherits the colour mode through `currentColor` and costs no extra request; any other format is linked as an `img` . The path is resolved against `src` , and `../` out of it is fine:

```
logo = "../assets/wordmark.svg"
```

`footer` is emitted **verbatim** , so it can carry HTML:

```
footer = "&copy; 2026 Example Ltd &middot; <a href=\"/imprint\">Imprint</a>"
```

That is deliberate - the footer is your own configuration, not untrusted input. Everything else that renders text, including the book and chapter titles, goes through the Markdown renderer and is escaped.

[highlight] and [highlightjs]

Highlighting comes in two layers, and they are two tables because they have very different consequences.

`[highlight]` is the switch; `[highlightjs]` describes the CDN layer, so every key that only means something to highlight.js lives there.

Key	Type	Default	Meaning
highlight.enabled	bool	false	the master switch. On its own: the built-in Jennifer highlighter
highlightjs.enabled	bool	false	additionally load highlight.js from a CDN
highlightjs.cdn	string	cdnjs 11.11.1	the CDN base URL, with no trailing slash
highlightjs.style	string	"github"	the highlight.js stylesheet used in light mode
highlightjs.styleDark	string	"github-dark"	the stylesheet used in dark mode
highlightjs.languages	list of string	["bash", "go", "json", "yaml", "xml", "ini", "nginx"]	extra highlight.js language packs to load

[highlight] alone highlights Jennifer code **while the site is built** . The spans are in the HTML that gets written, so there is no CDN, no JavaScript, and nothing to load; it works with scripting off and over `file://` . This is the layer to enable for a book that must make no third-party requests.

[highlightjs] on top pulls highlight.js from `cdn` and highlights the languages the built-in highlighter does not know, loading the `languages` packs and swapping `style` and `styleDark` with the mode selector. Blocks Grimoire already highlighted are marked so highlight.js leaves them alone.

With `highlightjs.enabled` = `false` , the other three keys do nothing - there is no stylesheet to choose and no grammar to fetch - which is exactly why they sit in this table rather than beside the master switch.

`highlight.enabled` is the master switch:

highlight	highlightjs	Result
off	off	no highlighting (the default)
on	off	Jennifer, at build time; nothing fetched
on	on	Jennifer at build time, everything else from the CDN
off	on	no highlighting, and the build says so

The last row is a contradiction rather than an intent, and it resolves to off: a book that says "no highlighting"

should not start making third-party requests because a second table was left enabled. The build reports the combination on stderr instead of silently picking one of the two readings.

The default CDN URL pins a version rather than tracking latest. A documentation build should render the same today and in a year, and a silent major-version bump on a CDN is exactly the kind of change that breaks a language grammar.

[search]

Key	Type	Default	Meaning
enabled	bool	true	build the search index and ship the search UI
bodyChars	int	1200	body text kept per indexed section; floored at 120

Indexing is per **section** , not per page, so a hit lands on the paragraph rather than at the top of a long chapter. `bodyChars` trades index size against how deep into a section a match can still be found. `--no-search` turns it off for a single build without touching the file.

[pdf]

Key	Type	Default	Meaning
enabled	bool	false	grimoire build also renders the PDF
output	string	"book.pdf"	the PDF path, relative to build.out
paper	string	"a4"	a4 or letter
bookmarkLevel	int	3	bookmark headings down to this level; 0 disables the outline
pageNumbers	bool	false	print page/total at the outside edge of every page footer
footerLeft	string	""	a template for the other side of that footer; "" leaves it empty
titlePage	bool	true	open the book with a title page; off starts it at the first chapter

`enabled` is what `--pdf` sets, and `grimoire pdf` renders the PDF regardless of it. Expect the PDF to dominate the build - see Performance.

`pageNumbers` needs the total page count, which does not exist until the whole book has been laid out, so the build renders the document and *then* stamps the footer on every page. The number is drawn inside the bottom margin in the theme's muted colour - a footer is for placing yourself, not for reading.

`footerLeft` fills the other side, and carries two slots read from the book's own git checkout:

{version}	the tag the build sits on, with any leading v removed
{commit}	the short commit id - only when there is no tag

Exactly one of the two is ever filled, which is what lets a single template cover a release and a working build. This manual uses:

```
footerLeft = "Grimoire {version} Manual {commit}"
```

giving `Grimoire 1.0.0 Manual` on a tagged commit and `Grimoire Manual 0e173c1` on anything else. The gap the empty slot leaves is closed before the line is drawn. Outside a git checkout - or where git is unavailable, including on `jennifer-tiny` , which ships no `os/exec` - both slots come back empty and the rest of the template still prints.

`titlePage` off drops the cover entirely and starts the PDF at the first chapter, for a book that would rather supply its own front matter as a prefix chapter.

A worked example

The `grimoire.toml` in this repository builds these pages - Grimoire renders its own documentation:

```
[book]
title = "Grimoire"
description = "Build a documentation website, and a printable PDF, from a
directory of Markdown files."
authors = ["mplx <jennifer@mplx.dev>"]
authorsLabel = "Written by"
language = "en"
src = "docs"

[build]
out = "site"

[html]
theme = "grimoire"
mode = "auto"
footer = 'Rendered with <a href="https://grimoire.jennifer-lang.dev/">Grimoire</
a>, by itself.'
repoUrl = "https://github.com/jennifer-language/grimoire"
repoLabel = "Source"
editUrl = "https://github.com/jennifer-language/grimoire/edit/main/docs/{path}"
favicon = "favicon.ico"

[highlight]
enabled = true

[pdf]
enabled = true
output = "grimoire.pdf"
paper = "a4"
pageNumbers = true
footerLeft = "Grimoire {version} Manual {commit}"
titlePage = true
```

Note the single-quoted `footer` : TOML's literal strings take no escapes, which makes an HTML attribute far easier to write than the `\"` of a basic string.

Themes

Ten themes ship with Grimoire. Every one defines a **light and a dark palette** - that is not optional in the theme model, so no theme can render a broken dark mode - plus a type stack, a corner radius, and the measure of the text column.

```
grimoire themes                  # the list, with one line each
grimoire build --theme nordic    # try one without editing the config
```

```
[html]
theme = "nordic"
mode = "auto"     # what a first-time reader gets: auto | light | dark
```

Each screenshot below is the same page of this manual - the command reference, light on the left and dark on the right - captured at 1440px wide, where the three-column layout is showing. Regenerate them all with `scripts/screenshots.sh` .

The gallery

Alphabetically. `grimoire` is the default; the rest are equals.

carbon

Graphite and teal, designed dark-first: the dark palette came first and the light one is its inverse rather than the other way round. Built for long technical reference.

carbon (screenshots/carbon.png)

ember

High-contrast neutral with an orange accent, near-square corners, and the widest measure at 860px - the most screen-native of the ten.

ember (screenshots/ember.png)

grimoire

Warm parchment and ink with a rust accent, and the only theme with serif headings over a sans body. The default.

grimoire (screenshots/grimoire.png)

ivy

Cream paper and forest green, with a **serif body** and a narrow 700px measure. The one to pick when the book is mostly prose.

ivy (screenshots/ivy.png)

meridian

Deep navy ink on a white page, blue-tinted panels, a dark steel-blue accent, and a navy dark mode. The product-handbook register: sober, corporate, unfussy.

meridian (screenshots/meridian.png)

nordic

The Nord palette - arctic blue-greys, a frost-blue accent, and a dark mode that is deliberately not black. Low glare and very even contrast, which suits reading for an hour more than it suits a screenshot.

nordic (screenshots/nordic.png)

obsidian

Cool slate with a violet accent, sans throughout. The dark mode is the one this theme is really about.

obsidian (screenshots/obsidian.png)

orchid

Mauve-tinted paper with a violet-pink accent, a plum dark mode, and the roundest corners of the set. Friendly and light.

orchid (screenshots/orchid.png)

sepia

Yellowed paper and deep coffee brown, full serif. The gentlest of the ten on a bright screen, and the only one whose dark mode is dark *brown* - old leather rather than the near-black the others use.

sepia (screenshots/sepia.png)

terminal

Monospace everywhere, phosphor green on near-black, corners almost square. A console in a browser.

terminal (screenshots/terminal.png)

At a glance

Theme	Body	Headings	Radius	Measure
carbon	sans	sans	5px	800px
ember	sans	sans	3px	860px
grimoire	sans	serif	7px	760px
ivy	serif	serif	4px	700px
meridian	sans	sans	6px	790px
nordic	sans	sans	6px	780px
obsidian	sans	sans	6px	780px
orchid	sans	sans	8px	770px
sepia	serif	serif	5px	700px
terminal	mono	mono	2px	820px

The PDF follows the theme

The printable book is themed too. Heading bars and the table header band are drawn from the selected theme's **light** palette - paper is white, so the dark palette would print as slabs of toner - with heading bars taking the accent mixed toward white, deepest at level one. So `terminal` prints with green heading bars and `sepia` with brown ones, and the hierarchy still reads at a glance.

How a theme is built

A theme is one file under `src/themes/` , and it is almost entirely a colour table. The layout rules live in a single constant stylesheet that only ever reads `var(--gr-*)` custom properties, so a theme never restates a rule - it supplies values.

```
export func theme() {
    return palette.Theme{
        name: "nordic",                       # the value for html.theme
        label: "Nordic",
        description: "Arctic blue-grey ...",  # the line `grimoire themes` prints
        light: palette.palette(...),          # fourteen colours
        dark: palette.palette(...),           # fourteen more
        fontBody: palette.sans(),
        fontHeading: palette.sans(),
        fontMono: palette.mono(),
        radius: 6,
        contentWidth: 780
    };
}
```

`palette.palette` is positional, taking the fourteen colours in the order the `Palette` struct declares them, which keeps a theme file readable as a table:

#	Field	What it paints
1	bg	the page background
2	surface	raised surfaces: sidebar, code blocks, cards
3	surfaceAlt	hover and zebra-stripe fills
4	border	hairline rules and outlines
5	text	body text
6	muted	secondary text: captions, breadcrumbs, metadata
7	heading	heading text
8	accent	links, the active chapter, focus rings
9	accentHover	the accent under a pointer
10	onAccent	text drawn on an accent fill
11	codeText	code text
12	codeBg	inline-code and code-block background
13	selection	the text-selection highlight
14	shadow	the shadow colour, including its alpha

Any CSS colour notation works; the shipped themes use hex for opaque colours and `rgba()` for `selection` and `shadow` .

Three font helpers - `palette.sans()` , `palette.serif()` , `palette.mono()` - return stacks that each end in a generic family, so a machine with none of the named faces still renders the intended shape. A theme is free to pass its own stack string instead.

Adding one

1. Copy an existing file in `src/themes/` and fill in the two palettes.
2. `import` it in `src/theme.j` and add `yours.theme()` to `all()` .

That is the whole registration. `grimoire themes` , the `--theme` flag, the config validation, and the PDF colours all read from `all()` , so nothing else needs to hear about it.

```
jennifer fmt --write src/themes/yours.j
./grimoire build --theme yours
```

Regenerating the gallery

`scripts/screenshots.sh` rebuilds every image in `docs/screenshots/` :

```
scripts/screenshots.sh              # all ten
scripts/screenshots.sh nordic ivy   # just these
```

It needs `chromium` , ImageMagick, and a Jennifer interpreter on `PATH` (override with `CHROMIUM` , `MAGICK` , `JENNIFER`). The book is built **once** , because the HTML is identical for every theme - only `assets/grimoire.css` differs, so the loop swaps the stylesheet with `scripts/theme-css.j` rather than running ten builds. Each theme is then captured twice, with the colour mode pinned in `localStorage` before the first paint, and the pair is composited side by side.

`PAGE` , `SRC` , `OUT` , and the four size variables at the top of the script override the defaults, so pointing the gallery at a different page or a different book is a variable, not an edit:

```
PAGE=index.html SRC=docs OUT=/tmp/shots scripts/screenshots.sh grimoire
```

Docker

From a source checkout, with no image build

The official Jennifer image already carries the interpreter and every module Grimoire imports, so a checkout runs as-is. The `:dev` tag is deliberate - Grimoire requires Jennifer 0.25.0 and `:latest` is still 0.24.0, so drop the tag once 0.25.0 is released:

```
# From the Grimoire checkout: build the book mounted at /work.
docker run --rm --user "$(id -u):$(id -g)" \
    -v "$PWD:/work" \
    ghcr.io/jennifer-language/jennifer:dev run ./grimoire build --pdf
```

The base image runs as `nonroot` , so `--user "$(id -u):$(id -g)"` is what lets it write into the bind mount as you rather than as a stranger. The launcher is handed to the interpreter rather than executed, because the image may have no `/usr/bin/env` to resolve a shebang with.

To build a book that lives somewhere else, mount both:

```
docker run --rm --user "$(id -u):$(id -g)" \
    -v "$PWD:/grimoire" -v "/path/to/my-book:/work" \
    ghcr.io/jennifer-language/jennifer:dev run /grimoire/grimoire build
```

The Grimoire image

For everyday use there is a prebuilt image with Grimoire baked in, so the book is the only thing you mount:

```
docker run --rm --user "$(id -u):$(id -g)" \
    -v "$PWD:/work" ghcr.io/jennifer-language/grimoire build --pdf

docker run --rm -p 8080:8080 \
    -v "$PWD:/work" ghcr.io/jennifer-language/grimoire serve --addr 0.0.0.0:8080
```

`serve` binds `127.0.0.1` by default, which inside a container means nothing outside can reach it - hence the explicit `--addr 0.0.0.0:8080` above.

The entrypoint is Grimoire itself, so everything after the image name is a Grimoire command; `--entrypoint sh` if you want a shell instead.

Build it yourself with the `Dockerfile` at the repository root:

```
docker build -t grimoire .
docker build --build-arg JENNIFER_TAG=static -t grimoire:static .   # smaller base
```

Nothing is compiled - Grimoire is Jennifer source, so the image is a copy and an entrypoint on top of `ghcr.io/jennifer-language/jennifer` .

`JENNIFER_TAG` defaults to `dev` for the version reason above, and becomes `latest` when 0.25.0 ships. The `static` line will work then too - the distroless variant tracks the release, so today it is still 0.24.0 and there is no `dev-static` to stand in for it.

The CI pipeline

`.github/workflows/docker.yml` builds that image on every push and pull request.

It builds a **single-arch image first and smoke-tests it** - `--version` , `themes` , and a real book built end to end with `--pdf` , asserting the HTML, the PDF, and the stylesheet all landed - and only then builds the multi-arch (`linux/amd64` + `linux/arm64`) manifest and pushes it to GHCR. Testing before the expensive emulated arm64 build is the point of the two-stage shape: a broken image fails in a minute rather than after the whole matrix.

The test runs the image the way a reader would, with `--user "$(id -u):$(id -g)"` and a bind-mounted

book, so a permissions regression in the image is caught by CI rather than by the first person to try it.

A pull request runs the build and the test but never publishes. Pushes to the default branch and tags publish to `ghcr.io/jennifer-language/grimoire` .

Reference

Internals

Source layout

```
grimoire            the launcher (a Jennifer script with a shebang)
Dockerfile          Grimoire on top of the official Jennifer image
grimoire.toml       the configuration that builds docs/ into site/
src/
  main.j            the CLI
  build.j           the build: render, write, report; parallel across chapters
  config.j          grimoire.toml -> Config, with defaults for everything
  summary.j         SUMMARY.md parser, and the directory-tree fallback
  content.j         Markdown -> HTML: anchors, link rewriting, code blocks
  highlight.j       the built-in, build-time Jennifer syntax highlighter
  layout.j          the page shell: top bar, sidebar, contents, pager, search
  palette.j         the theme model and the stylesheet generator
  theme.j           the theme registry
  themes/*.j        the ten shipped themes
  assets.j          the client runtime (mode selector, search, copy buttons)
  assets/           vendored: the Jennifer highlight.js grammar
  search.j          the search index
  pdfbook.j         the printable build
  serve.j           the local preview server
  util.j            slugs, paths, text helpers
scripts/
  screenshots.sh    regenerate the theme gallery
  theme-css.j       write one theme's stylesheet to a path
docs/               this documentation, and the book this repository builds
```

`grimoire` is a Jennifer program with a `#!/usr/bin/env -S jennifer run` shebang - the same language as the rest of the tool - and it is deliberately three lines of work. `src/main.j` is the program; the launcher only says where Grimoire is installed and hands over the command line:

```
exit main.run(path.join(path.dir(os.ARGS[0]), "src"), os.ARGS);
```

That app directory is how the build finds the assets Grimoire ships - the highlight.js grammar - without ever consulting the working directory. `main.j` is a module rather than a script, so it takes the directory as an argument: modules hold no mutable state in Jennifer, so there is nowhere for a program-wide value like this to sit except a parameter.

Two things make that work from anywhere. The interpreter resolves the import relative to the launcher's own file, so the working directory never matters. And the launcher runs `os.ARGS[0]` through `fs.realpath` before taking its directory, so a symlink on `PATH` - which is how a command normally gets installed - finds the assets beside the **real** file rather than beside the link:

```
ln -s "$PWD/grimoire" ~/.local/bin/grimoire
```

An unresolvable path falls back to the invocation path. A book still builds then; only the bundled highlight.js grammar would be missed, and the build says so when it is.

Notes on the Markdown

Rendering goes through the `markdown` module's document tree rather than its `toHtml`, because a documentation site needs more than the plain translation: stable heading anchors, `.md` links rewritten to `.html`, code blocks wrapped with a language tag and a copy button, and scroll containers around tables.

The walk hands each block kind to its own renderer, and two of them are worth calling out:

- An **html_block** is emitted **verbatim** , because writing one is a deliberate act by the book's author. It is the single exception: everything on the page that comes from anywhere else is escaped, and every link goes through `html.safeUrl` .
- A **page_break** renders as nothing. It is a directive for the printable build, and has nothing to draw on a web page.

Inline spans nest, so the children of a `**...**` , an emphasis, or a link label are rendered as the nodes they are - which is what keeps **json.Value** bold *and* monospaced, and what gets a link inside bold its `.md` rewritten to `.html` like any other link.

Anchors

Heading anchors follow **GitHub and mdBook exactly** , which is what lets a book migrate without rewriting its cross-references. Punctuation is *dropped* rather than folded to a dash, so

```
### REPL (`cmd/jennifer/repl.go`)
```

anchors at `#repl-cmdjenniferreplgo` , not `#repl-cmd-jennifer-repl-go` . Getting this wrong is quiet - the page still builds, and only the links break - so it is worth stating: whitespace becomes a dash, runs of it are preserved, letters and digits (including non-ASCII) are kept, and everything else vanishes. A repeated heading gets a `-1` , `-2` suffix in document order.

Links are rewritten to match: `x` becomes `guide/syntax.html#anchor` , and a directory's `README.md` folds onto its `index.html` , with the fragment carried through untouched.

Search

The index is built from **sections** , not pages: a heading and the body text under it up to `search.bodyChars` , so a hit lands on the paragraph rather than the top of a long chapter. Both the index and the scorer are Grimoire's own - there is no search library to load - and the index is delivered as a **script** rather than fetched, because `fetch` on a `file://` page is blocked by every browser and a book that cannot be read off a USB stick is a book with a dependency it does not need.

Records are grouped per chapter and reassembled in outline order after the parallel render, so the index is byte-identical no matter what `--jobs` was.

The PDF

The printable build assembles every chapter into one Markdown document and hands it to the `pdf` module, with a cover page, a nested bookmark outline, and every chapter opening a page of its own.

That last part takes two mechanisms, because chapters are not all at the same level. A chapter outside the parts keeps its level-one heading, and the layout breaks the page at every level-one heading - which is also what gives the cover a page to itself. A chapter **under a part** is demoted one level so the part heading can own the top of the outline, and a demoted heading no longer breaks anything; those chapters ask for the break explicitly with a `<!-- pagebreak -->` directive, which the module parses into a `page_break` node. The chapter that opens a part is the exception: the part heading has just broken the page, and a second break would leave the part title alone on a sheet.

Writing the directive as an HTML comment is deliberate - the same combined source still renders as HTML, where a browser shows nothing at all.

It picks up the book's theme throughout. Heading bars, the table header band, the panel behind a code block, and the tint and rule on a blockquote are all drawn from the selected theme's **light** palette - paper is white, so the dark palette would print as slabs of toner - with heading bars taking the accent mixed toward white, deepest at level one. So `terminal` prints with green heading bars and a green quote rule, `sepia` with brown ones, and the hierarchy still reads at a glance.

The tool credit lives in the document metadata rather than on the title page, in `Creator` and `Producer` , where

a reader's document properties show it.

Print differs from the site in two deliberate ways:

- **Raw HTML is dropped** , by the layout. A hand-written block has no rendering on paper, and left in place the Jennifer introduction's inline SVG wordmark typesets as a page and a half of path data before the reader reaches a sentence.
- **Links are resolved for paper.** A cross-reference is not clickable in print, so it reads as its label alone, while an external URL keeps its address in parentheses.

Beyond those, the print path passes each line through untouched, **indentation included** - which matters more than it sounds. Reflowing a paragraph here, by gathering its lines and trimming each one, would strip the indent from a continuation line, and an indented continuation that loses its indent stops belonging to its list item and becomes a stranded paragraph between the items. The layout reflows paragraphs itself, so there is nothing to gain by trying.

Characters the standard-14 fonts cannot encode are transliterated here (`->` to `->` , box drawing to `-` and `|`) before the layout sees them. The module would substitute a single ? for each, which is correct but says less: `->` says what the arrow said. So Grimoire spends a transliteration table and leaves `unencodable` as the last resort for the rest.

What the layout does, and what it does not

Grimoire leans on the `markdown` module for more than the parse, and it is worth knowing where the line falls - several things that look like they need code here do not:

thematic_break, html_block	a rule and a raw block are parsed, not guessed at from source
nested inline spans	json.Value keeps its code formatting
indented list continuations	a soft-wrapped item stays one item
autolinks	<https://example.com> is a link node
pdf.foldLine in the layout	a long code line folds to the column by itself
quoteFill, quoteRule, codeFill, codeBorder	themed panels in print
creator, producer, unencodable	metadata, and the encoding fallback
page_break / <!-- pagebreak -->	a demoted chapter can still open a page

One thing the printable build cannot do: running headers and a "page N of M" footer. `pdf.setHeader` / `setFooter` and the `%page%` / `%pages%` placeholders exist, but `markdown.renderPdf` returns bytes rather than a `pdf.Document` , so the document-level hooks are out of reach from the Markdown path. Having them would mean Grimoire laying the book out page by page itself.

Building this repository

`grimoire.toml` here builds `docs/` - the pages you are reading - so the repository is its own worked example, and a change to the renderer shows up in the next build of this manual:

```
./grimoire build        # the site and the PDF, into site/
./grimoire serve        # read it at http://127.0.0.1:8080/
```

`[pdf] enabled` is on, so a plain `build` produces `site/grimoire.pdf` as well - which is what the download link on the introduction points at. Turn it off and that link goes dead.

`.github/workflows/pages.yml` runs exactly that build on every push, in the official Jennifer image, and publishes `site/` to GitHub Pages. It asserts the pieces a reader needs - the landing page, the stylesheet, the search index, a non-empty PDF, and the link between the last two - so an empty publish fails in CI rather than on the live site. A pull request builds and checks without publishing.

`docs/CNAME` is what binds the custom domain, and it needs no special handling: it is a non-Markdown file under

`src` , so the ordinary asset copy puts it at `site/CNAME` . It has to be **in the site** rather than only in the repository settings, because a Pages deployment from an artifact publishes exactly what the artifact holds - a build that dropped the file would quietly unbind the domain. The pipeline asserts its contents for that reason.

Sources are formatted with `jennifer fmt` and clean under `jennifer lint` :

```
jennifer fmt --write src/*.j src/themes/*.j scripts/*.j
jennifer lint src/*.j src/themes/*.j scripts/*.j
```

Possible extensions

Not built, but the shape is there for them: a link checker over the resolved outline (the build already knows every output path and anchor), a `--watch` rebuild loop for `serve` , and multi-language books.

Performance

Measured on a 155-chapter, 2.3 MiB book (the official Jennifer 0.24.0 documentation), on an 8-core Ryzen 7 5800X. The wall-clock figures are from bare metal; the phase split below was measured in a VM on the same part, so read that as proportions rather than absolutes.

	wall clock
site, -j 1	~47 s
site, -j 16	~12 s
site + PDF	~118 s

Two things shape those numbers.

The site scales to roughly 4x, not 8x

Total CPU stays flat as jobs rise, so the limit is not CPU. It is that chapter sizes span two orders of magnitude - 135 KiB against a 7 KiB median - and **a chapter cannot be split** : it is one parse and one render.

Work is handed out heaviest-first, each chapter going to whichever worker is least loaded at that moment. That is the greedy longest-processing-time rule, and on this book it does noticeably better than the obvious alternatives:

Split	Busiest worker, -j 8	-j 16
round-robin over the outline	336 KiB	195 KiB
heaviest-first, least-loaded	211 KiB	135 KiB

At `-j 8` that is an exactly even split - 211 KiB is the total divided by eight, so no arrangement does better. At `-j 16` it is the size of the single largest chapter, which is the floor by definition. Past that, more jobs cannot help; the one 135 KiB chapter decides when the build ends.

Sorting first was tried and is *worse* at high job counts, which is the counter-intuitive part: it packs the small chapters neatly but leaves nothing left to balance the big ones against.

The PDF dominates, and it is serial

Where the ~2 minutes go:

Phase	Time	Where
assembling the combined Markdown	5.5 s	Grimoire
parsing it	38.6 s	the markdown module
laying out 506 pages	79.7 s	the markdown module

95% of it is inside the module, and single-threaded. Grimoire starts the PDF at the same time as the chapter render, so `--pdf` costs roughly `max(site, pdf)` rather than their sum - but the PDF is the floor. Getting under it means fixing the module, not Grimoire.

Re-derive the split at any time with the profiler:

```
jennifer profile ./grimoire pdf --src <your-book> --out /tmp/pdfout
```

Practical notes

- **--jobs 0** (the default) is one task per CPU and is almost always the right answer. Raising it past the point above buys nothing; lowering it to `1` is useful for reading `--verbose` output in outline order.
- **--no-search** saves very little. Indexing rides along with the render that has already parsed the chapter.
- **grimoire pdf** skips the site entirely when the PDF is all you want, but since the two overlap, `build --pdf` costs about the same and gives you both.
- **Output is deterministic** at any job count, so a build is reproducible and diffable regardless of how the work

was split.