TiefDownConverter Documentation

Tiefseetauchner et al.

January 05, 2026

Contents

1. Welcome to TiefDown

> This document prescribes and describes how to use and work with TiefDown Projects. Crucially, it does not document TiefDownLib or TiefDownConverter. Check out the corresponding documentation from the main page: https://tiefseetauchner.github.io/TiefDownConverter°.

TiefDown is a project format for converting structured markdown inputs into deterministic, structured outputs. A TiefDown project consists of markdown folders, a manifest, and templates, which are processed through a defined conversion pipeline to produce documents such as PDF or EPUB.

Document conversion pipelines tend to grow complex quickly when combining multiple inputs, templates, metadata, and processors. TiefDown exists to make these pipelines explicit, reproducible, and manageable.

TiefDown allows full control over the conversion process, from simple Pandoc-based workflows to complex pipelines involving custom preprocessors, injections, and multiple output formats, while maintaining deterministic results.

2. Project Model Overview

TiefDown is first and formost a format of folders and files. A central manifest file defines the conversion processes, templates define the output, and markdown directories define the input/output operations.

3. Project Structure & Directories

First off, the folder structure is as follows:

3.1. TiefDown Folder Example

1

```
.
├── 2025-12-30_23-23-26
│   ├── docs.aux
│   ├── docs.log
│   ├── docs.out
│   ├── docs.pdf
│   ├── docs.synctex.gz
│   ├── docs.tex
│   ├── docs.toc
│   ├── docs.typ
│   ├── metadata.tex
│   ├── metadata.typ
│   ├── output.tex
│   ├── output.typ
│   ├── TeX Documentation_convdir
│   │   └── Chapter 1 - Introduction.md
│   ├── TeX Documentation.pdf
│   ├── Typst Documentation_convdir
│   │   └── Chapter 1 - Introduction.md
│   └── Typst Documentation.pdf
├── manifest.toml
├── Markdown
│   └── Chapter 1 - Introduction.md
├── template
│   ├── docs.tex
│   └── docs.typ
├── TeX Documentation.pdf
└── Typst Documentation.pdf
```

Basically, the files in `markdown/` are converted into the PDFs in the main directory using the templates in `template/`. `2025-12-31_04-00-45/` is the temporary conversion directory, containing the files needed for the conversion (Yes, I wrote this on new years eve at 4 a.m.).

3.2. TiefDown Manifest Example

Since the source of truth in any TiefDown project is the manifest, let us have a *very* quick look at that. This manifest example follows the folder structure above.

toml

```toml
version = 6

[custom_processors]
preprocessors = []
processors = []

[[markdown_projects]]
name = "Markdown"
output = "."
path = "Markdown"

```

```toml
[[templates]]
name = "Typst Documentation"
template_file = "docs.typ"
template_type = "Typst"

[[templates]]
name = "TeX Documentation"
template_file = "docs.tex"
template_type = "Tex"

[shared_metadata]
title = "Documentation"
```

As you can see, there's two templates defined in this project, as well as shared metadata that can get injected into the templates. No Custom Preprocessors or Processors are defined here.

3.3. Template Directory

The `template/` directory is mandatory for Epub, Typst and TeX templates, and optional for other template types. It contains files that get copied to the conversion directory, and is thus the source of truth for shared files like templates, resources and lua filters.

Since this directory gets copied to the conversion dir for every markdown project, resources are shared. For markdown project specific resources, see custom resources.

3.4. Markdown Project Directories

Markdown projects represent the input directories of a project. There can be multiple markdown projects. A markdown project has an input as well as output path and optionally resources. See markdown projects for more details.

4. Manifest

The manifest is the source of truth for TiefDown. It, and only it, defines the behavior of any TiefDown conversion. Note that the examples here are non-exhaustive. They simply serve to give you an idea of what to expect.

4.1. Versioning and compatibility/upgrades

The manifest version is at the top of the manifest, and similarly important. It consists of a single integer. Since TiefDown is under constant development, old manifests must be upgradeable in a consistent and expectable way. It also prevents an old version of TiefDownLib from messing up a manifest generated from a newer version.

The upgrade process consists of sequentially upgrading the manifest through the versions until it reaches the current version. That way, upgrades are reproducible. It also means that every prior version of the manifest is upgradeable.

Example:

toml

```toml
version = 6
```

4.2. Markdown Projects List

Markdown projects are the cornerstone of TiefDown. They define the input files and parameters. They can be used for a variety of use cases, but are especially useful when template sharing is a concern.

There can be multiple markdown projects per TiefDown project.

A markdown project contains, among others, an input and output directory, allowing clustering of output files. See markdown projects for more information.

Example:

toml

```toml
[[markdown_projects]]
default_profile = "Man"
name = "man"
output = "man"
path = "man_markdown"
```

4.3. Templates List

Templates are the basis for document conversion. Each template can be applied to multiple markdown projects using the queueing system.

Example:

toml

```toml
[[templates]]
filters = ["luafilters/mega_replacer_filter.lua"]
footer_injections = ["HTML footer"]
header_injections = ["HTML header"]
name = "GitHub Single File Documentation"
output = "index.html"
template_type = "CustomPreprocessors"

[templates.preprocessors]
combined_output = "index.html"
```

```toml
output_extension = "html"
preprocessors = ["HTML Conversion", "HTML Direct Copy"]
```

4.4. Custom Processors model

Custom processors and custom preprocessors are extensions on the usual conversion process, changing the arguments passed to the pandoc process, or even changing the executable of the preprocessing.

Example:

```toml
[[custom_processors.preprocessors]]
cli = "cat"
cli_args = []
extension_filter = "html"
name = "HTML Direct Copy"

[[custom_processors.processors]]
name = "HTML Standalone Conversion"
processor_args = ["--to", "html5", "-s", "--metadata", "title={{title}}", "--metadata",
"author={{author}}", "--css", "/TiefDownConverter/template/html_template/style.css", "--toc",
"-B", "html_template/header.html"]
```

4.5. Shared Metadata

Metadata in TiefDown is split in two parts: shared metadata that is accessible from all markdown projects and markdown project specific metadata.

Examples:

- Shared metadata:

```toml
[shared_metadata]
author = "Tiefseetauchner et al."
githubPagesUrl = "https://tiefseetauchner.github.io/TiefDownConverter/"
title = "TiefDownConverter Documentation"
```

- Markdown project specific metadata:

```toml
[markdown_projects.metadata_fields]
githubPagesDocsPath = "/"
```

4.6. Metadata Settings

Metadata settings define how metadata is injected into the conversion process.

Example:

```toml
[metadata_settings]
metadata_prefix = "projectMetadata"
```

4.7. Profiles

Profiles allow bundling of templates into a preset execution order, allowing the creation of subgroups as well as defining a default profile for a markdown project.

Example:

```toml
[[profiles]]
name = "Documentation"
templates = ["PDF Documentation LaTeX", "PDF Documentation", "Epub Documentation", "GitHub Multi Page Documentation"]
```

4.8. Injections

Injections are the intended way to create template specific conversion additions. There are header, body, and footer injections, allowing the user to insert template specific markup into all parts of the conversion process.

Example:

```toml
[[injections]]
files = ["head.html", "nav.md", "head2.html"]
name = "HTML header"
```

4.9. Smart Clean Settings

Smart Clean settings control how and whether smart clean should run during conversion.

Example:

```toml
smart_clean = true
smart_clean_threshold = 3
```

4.10. Full Example

Below is an example of a `manifest.toml` that illustrates the most important features of TiefDown-Converter, including injections, preprocessors, and multiple templates.

For a detailed explanation of every part of this manifest, see below. Note that this manifest has been rearranged and indented to improve clarity.

```toml
smart_clean = true
smart_clean_threshold = 3
version = 6

[shared_metadata]
author = "Lena Tauchner"

[[markdown_projects]]
name = "Dream"
output = "Dream"
path = "Dreams/1 - Dream"
resources = ["additional_meta.tex", "epub_meta.yaml", "cover.png"]

    [markdown_projects.metadata_fields]
    edition_date = "2025"
    edition_num = "1"
    title = "Dream"

[[markdown_projects]]
name = "Reality"
output = "Reality"
```

```toml
path = "Dreams/2 - Reality"
resources = ["additional_meta.tex", "epub_meta.yaml", "cover.png"]

    [markdown_projects.metadata_fields]
    edition_date = "2025"
    edition_num = "1"
    title = "Reality"

[[templates]]
filters = ["luafilters/"]
name = "lix_novel_a4.tex"
output = "a4_main.pdf"
template_type = "Tex"

[[templates]]
filters = ["luafilters/"]
name = "lix_novel_book_at.tex"
output = "140x205mm_main.pdf"
template_type = "Tex"

[[templates]]
name = "Original MD"
output = "original.md"
template_type = "CustomPreprocessors"

    [templates.preprocessors]
    combined_output = "original.md"
    preprocessors = ["Original MD"]

[[templates]]
filters = ["epub_lua_filters"]
header_injections = ["EPUB Copyright Block"]
name = "EPUB Conversion"
output = "main_ebook.epub"
processor = "Metadata for EPUB"
template_file = "main_epub/"
template_type = "Epub"

[[templates]]
filters = ["luafilters/mega_replacer_filter.lua"]
multi_file_output = true
name = "HTML Helper Export"
output = "html_raw/"
template_type = "CustomPreprocessors"

    [templates.preprocessors]
    output_extension = "html"
    preprocessors = ["HTML Multi Page", "HTML Raw"]

[[custom_processors.preprocessors]]
cli_args = ["-t", "markdown"]
name = "Original MD"

[[custom_processors.preprocessors]]
cli_args = ["-t", "html5"]
name = "HTML Multi Page"
```

```
78
79 [[custom_processors.preprocessors]]
80 cli = "cat"
81 cli_args = []
82 extension_filter = "html"
83 name = "HTML Raw"
84
85 [[custom_processors.processors]]
86 name = "Metadata for EPUB"
87 processor_args = ["--metadata-file", "epub_meta.yaml"]
88
89 [[injections]]
90 files = ["epub_copyright_block.html"]
91 name = "EPUB Copyright Block"
```

Now to explain the basic structure of the `manifest.toml`, I will start by addressing the basic project settings. Important are the `smart_clean` and `smart_clean_threshold` flags. These define the smart cleaning behavior of the project. See Smart Clean for more information. Also note the `version` field, which defines the compatibilty with TiefDown.o

After smart cleaning, we define the `shared_metadata` table, which contains metadata about the project that is shared between the items. In this case, the author.

Then, we define the markdown projects. The markdown projects in this case are Dream and Reality, each with a separate output and input folder, as well as metadata fields for filling in the LaTeX data and resources that are project specific but used in the template. In this example, we have a `cover.png`, which, during the epub conversion, is injected as the cover image.

After defining the markdown projects, we add templates. Here, we have five templates. The LiX novel templates (`lix_novel_a4.tex` and `lix_novel_book_at.tex`), a `CustomPreprocessors` conversion to a singular markdown file, an EPUB template and the HTML Helper export, which simply exports an HTML file per input file (in my case here, chapters).

Crucially, we define `[templates.preprocessors]` for the `CustomPreprocessors` conversions, which handle different output and input formats respectively.

The aforementioned preprocessors are links to the `custom_processors.preprocessors` array, which contains all preprocessors needed for conversion. Here, that is the Markdown conversion, Markdown to HTML5 and Raw copying of HTML documents. For the last case, we use the `cli` field to change the executable ran during conversion to `cat` from `pandoc`, to simply copy the file content to memory.

For EPUB conversion, we also need the processors argument, as there is metadata used. `epub_meta.yaml`, in this case, is a static template file which simply provides additional, common metadata to epub.

Lastly, the EPUB conversion requires a copyright block specifically. In this case, that is solved as an injection, which simply places a `epub_copyright_block.html` into the epub conversions header injections.

5. Conversion Pipeline

The conversion pipeline, at its heart, is a simple queue of template – markdown project pairs that get processed one by one. Inside the conversion process, there's the following steps:

- Creation of scratch directory
- Copying of templates and resources
- Merging of metadata
- Copying markdown files
- Running conversion
- Copying output files

5.1. Queueing System

Before a document can be converted, a queue must be created. The template – markdown project pair computation is done in TiefDownLib as a separate step to the conversion. It considers a given list of templates, profiles, and markdown projects, and based on that or, if not provided, the defaults, computes a queue to hand to the conversion pipeline.

There's a few defaults to keep in mind:

- Conversion is run against one manifest/project at a time.
- Markdown projects can be provided as a list.
  - If no markdown project is provided, all markdown projects are converted.
- Either templates or profile can be provided, never both.
- Templates can be provided as a list.
  - If one or more templates are provided, all are converted for all selected markdown projects, regardless of default profile.
  - If no templates are provided, the profile logic sets in.
- One profile can be provided.
  - If a profile is provided, all contained templates are converted for all selected markdown projects, regardless of default profile.
  - If no profile is provided, the default profile logic sets in.
- If neither templates nor profile is provided, the default profile for each markdown project is converted.
- If there is no default profile for a markdown project, all available templates are converted for this markdown project.

5.2. Scratch Directory

All conversion happens in a temporary scratch directory. This directory is created when conversion is first started, and is named after the current timestamp.

The scratch directory can be automatically removed using smart cleaning.

5.3. Template Directory

Next, the template directory is copied to the conversion directory. This is done for each markdown project seperately.

The items of the template directory are put in the markdown project specific folder, in which the primary conversion processes will run.

5.4. Resources

Each markdown project can include resources that may be used by templates. These are, similarly to the template directory, copied to the markdown project specific directory.

5.5. Shared Metadata and Project Metadata Merging Rules

Metadata gets merged after the copying of template and resources. Project specific metadata overrides shared metadata if conflicting.

5.6. Copying Markdown Files

The markdown files are then copied for each conversion task. They are copied to a directory in the markdown project specific directory named after the template.

5.7. Running Conversion

At the heart of the conversion pipeline is, unsurprisingly, the conversion.

First, a converter is decided from the available conversion engines:

- CustomPreprocessor
- CustomProcessor
- Tex
- Typst
- Epub

Each conversion engine has a converter associated with it.

After being decided, the converter is called. Shared in all converters is the retrieval of applicable preprocessors, injections, and input files. Input files are sorted according to the rules below.

Then, navigation metadata is generated, and it as well as the previously computed metadata is written to the markdown project specific directory in accordance with the metadata generation settings, after which the primary conversion in accordance with the template type is started.

5.7.1. Input File Sorting

Input file sorting relies on numbers in the to-be-converted input file names. These are parsed and sorted accordingly, with directories with the same number being recursively added after a file of said number. Take the following folder structure:

```
Markdown/
├── Take 1 - Introduction.md
├── Chapter 2 - Usage.md
├── 2 - Usage/
│   ├── Subchapter 1 - Usage detail 1.md
│   └── Subchapter 2 - Usage detail 2.md
└── Asdf 3 - Customisation.md
```

The resulting order would be:

1) Take 1 - Introduction.md
2) Chapter 2 - Usage.md
3) 2 - Usage/Subchapter 1 - Usage detail 1.md
4) 2 - Usage/Subchapter 2 - Usage detail 2.md
5) Asdf 3 - Customisation.md

Alphabetical order is ignored.

5.7.2. CustomPreprocessor Conversion Engine

The CustomPreprocessor conversion engine follows the following steps:

After the aforementioned steps, the selected preprocessors are run against the appropriate files. Each preprocessor can have a file extension filter, and the most specific filter is chosen for each file, falling back on a preprocessor without filter.

The results are either combined to a singular large file or written to individual files.

5.7.3. CustomProcessor Conversion Engine

The CustomProcessor conversion engine follows the following steps:

After the aforementioned steps, the input files are run against preprocessors that convert to pandoc AST.

The pandoc native is then combined and written to a file.

Then, it is converted via pandoc in accordance with the parameters from the custom processor.

5.7.4. TeX Conversion Engine

The TeX conversion engine follows the following steps:

After the aforementioned steps, the input files are converted to LaTeX using the preprocessors and combined.

The combined LaTeX is then written to the combined output file.

Then, XeLaTeX is run against the template file, converting it to PDF. This step is run twice, as XeLaTeX needs to first generate a bibliography.

5.7.5. Typst Conversion Engine

The Typst conversion engine follows the following steps:

After the aforementioned steps, the input files are converted to Typst using the preprocessors and combined.

The combined Typst is then written to the combined output file.

Then, Typst is run against the template file, converting it to PDF.

5.7.6. Epub Conversion Engine

The Epub conversion engine follows the following steps:

After the aforementioned steps, the input files are run against preprocessors that convert to pandoc AST.

The pandoc native is then combined and written to a file.

Then, pandoc is run against the AST using the processor arguments.

6. Templates

A pivotal point in any TiefDown user's life is once they truly understand templating. This section seeks to serve that purpose.

Templating in TiefDown is separated into two archetypes: template-based templates and logic-based templates.

TeX, Typst, and EPUB templates are template-based. They rely on a concrete template file located in the template directory, which defines the structure of the final output and into which converted content and metadata are injected.

CustomPreprocessors and CustomProcessor templates are logic-based. They define the conversion process purely through configured commands and processors, without relying on a single template file. They may still make use of files from the template directory, but the execution flow is driven by logic rather than a fixed document template.

For execution logic of each of the templates, see Running Conversion from the previous chapter.

6.1. TeX templates

TeX templates use a LaTeX file as the template for the primary execution point. This template file must be specified in the template, and should `\input` the combined output from the pandoc conversion.

Metadata is written seperately to a `metadata.tex` file, which creates macros for accessing metadata. Below is an example of such a `metadata.tex` file:

```tex
\newcommand{\meta}[1]{\csname meta@#1\endcsname}

\expandafter\def\csname meta@author\endcsname{Tiefseetauchner et al.}
\expandafter\def\csname meta@githubPagesDocsPath\endcsname{lib/}
\expandafter\def\csname meta@githubPagesUrl\endcsname{https://tiefseetauchner.github.io/TiefDownConverter/}
\expandafter\def\csname meta@title\endcsname{TiefDownConverter Documentation}
```

Metadata can then be accessed in the template file or in `.tex` files in the input like so:

```tex
\input{./metadata.tex}
\meta{title}
% Writes "TiefDownConverter Documentation"
```

Lua filters and preprocessors in LaTeX are fully supported. For example:

- TeX Raw

  A `cat` preprocessor that copies `.tex` files instead of running them through pandoc.
- –listings

  Pandoc supports outputting LaTeX with listings support. A preprocessor can enable that for your file.

Processor arguments are fully supported. Processor arguments can be used to adjust the XeTeX cli call.

6.2. Typst templates

Similarly to TeX templates, Typst templates also use a template file, in this case a `.typ` file, for conversion. This template file must be specified in the template, and should `#include` the combined output from the pandoc conversion.

Metadata is also written to a `metadata.typ` file, which creates a dictionary for accessing metadata. Below is the same example as above for Typst:

typst

```typst
#let meta = (
  author: "Tiefseetauchner et al.",
  githubPagesDocsPath: "lib/",
  githubPagesUrl: "https://tiefseetauchner.github.io/TiefDownConverter/",
  title: "TiefDownConverter Documentation",
)
```

Accessing metadata then becomes trivial:

typst

```typst
#import "metadata.typ": meta
#meta.title
// Writes "TiefDownConverter Documentation
```

Equally to the above template, lua filters and preprocessors are fully supported for Typst.

Processor arguments, as above, are fully supported. The processor arguments are added to the typst process on conversion.

6.3. EPUB templates

EPUB templates are a special kind of template, as they are less user-unfriendly CustomProcessor converters.

The primary simplification in EPUB templates is the addition of css and font search. For conversion, EPUB retrieves CSS as well as font files from the template directory and injects them into the output file. Fonts are searched within a `fonts/` subfolder in the template.

Lua filters are fully supported for epub conversion. Importantly, they are applied only to the last pandoc conversion process, and not to the AST conversion processes.

Preprocessors however are supported but adviced against, as the default preprocessor converts the input files to pandoc native.

Processor arguments are fully supported, and operate on the pandoc command.

6.4. CustomPreprocessors conversion

Custom preprocessors conversion uses preprocessors to convert to a common format, then concatenating the output to the final file. There is no template file for custom preprocessors conversion.

To convert via this method, you must define one or more custom preprocessor. There are no default custom preprocessors for this conversion, which means the custom preprocessors must cover each file type converted.

Lua filters are supported for this conversion, and only get added for pandoc processes.

Custom processors are **not** supported, as there is no unified processing step.

6.5. CustomProcessor conversion

Similarly to custom preprocessors conversion, custom processors conversion does not have a template file, but instead converts all input files to pandoc AST before combining them to one large AST mega file and converting that with a custom processor.

To convert via this method, you must set a custom processor with arguments for the target file type. The target file name is automatically appended to the process and must not be included in the arguments.

Lua filters behave identically to custom processor conversion, applying only to the final pandoc process.

Similarly, preprocessors are supported but adviced against unless entirely necessary.

Custom processor arguments are mandatory.

7. Lua filters

Note: for documentation on Lua filters in Pandoc, refer to the pandoc documentation

Lua filter integration in TiefDown is an important part of us manipulating documents. For example, to inject metadata, one can feasibly write a lua filter that uses the pandoc frontmatter to parse metadata and display it in documents.

As a matter of fact: I did that. See the `mega_replacer_filter.lua` and `navlib.lua` for more on that. It also integrates nicely with navigation metadata, see Navigation Metadata.

The basic operational principles of lua filters are as follows:

1. Lua filters are defined per template.
2. Discovery runs on Lua filters, recursively entering directories.
3. If the conversion allows lua filters (i.e. the cli is pandoc), they are added to the preprocessing step automatically.

Lua filters are also added to the processing step of EPUB and CustomProcessor conversions.

8. Markdown projects

Since TiefDown operates on structured, multifile inputs stored in directories, the directory has to be declared in the manifest. The concept behind this declaration is called "markdown projects", based on the outdated assumption that input files would always be markdown (they are not).

Regardless, markdown projects are defined and then ran through during conversion. Each markdown project has a unique identifier (`name`), an input path, and output path, optional resources and default profile, as well as optional metadata.

8.1. Input discovery & sorting rules

Files in the input directory defined by a markdown project are sorted by their file name order. This is calculated as the first whole number in a file name. E.g. if a file is named `Chapter 23.5 - the reckoning.md` , the extracted order is 23.

Folders similarly have an order number. Files in a folder are added recursively in accordance with the same sorting mechanism. Files in a folder (e.g. `Chapter 42 - More details/Detail 3 - The Chicken.md`) are inserted after the file with the same order number.

8.2. Custom resources copying

As markdown projects have input files that are converted using pandoc or similar during the preprocessing step, one can define resources that are only copied but not consumed by the preprocessors. Resources are copied to the conversion directory of the markdown project instead of to the `conv_dir` of the relevant markdown project, even though they initially reside in the markdown projects' input directory.

As a concrete example: take a cover image. The cover image would be injected into a PDF. But different markdown projects (e.g. books, papers, ...) need seperate cover images. The cover image is thus added as a resource. Take the following folder structure:

```
.
├── manifest.toml
├── Book 1 Markdown
│   ├── Chapter 1 - Introduction.md
│   └── cover.jpg
├── Book 2 Markdown
│   ├── Chapter 1 - Different Introduction.md
│   └── cover.jpg
├── Book 1
│   └── BookOut.pdf
├── Book 2
│   └── BookOut.pdf
└── template
    └── book.typ
```

where `manifest.toml` contains:

toml

```toml
[[markdown_projects]]
name = "My Book 1"
output = "Book 1"
path = "Book 1 Markdown"
resources = ["cover.jpg"]

```

```toml
[[markdown_projects]]
name = "My Book 2"
output = "Book 2"
path = "Book 2 Markdown"
resources = ["cover.jpg"]
```

The `cover.jpg` in this case is copied to the conversion directory, where docs.typ can consume it.

8.3. Project metadata fields

Projects can have specified metadata fields. These override the shared metadata. This can be helpful to adjust template behavior, e.g. changing the title of the book. See the manifest example.

9. Injections

Handling large projects with multiple templates can sometimes lead to a lot of template specific code. To enable template specific output without a template file (e.g. EPUB, HTML), injections enable you to insert an input file at any point of the conversion process.

9.1. Header and Footer Injections

Header and footer injections act as insertions before and after the content respectively. That means that, after the content is preprocessed, it gets pre-/appended with the preprocessed injection.

An example usecase for injections is adding an HTML scaffolding around the content. The header injection would hold the doctype declaration and head tags, while the footer could hold an HTML footer for displaying copyright. Injections are especially useful for multi-file output.

9.2. Body Injections

A body injection is treated significantly different to header or footer injections. In case of a body injection, it gets inserted into the body according to the sorting algorithm *before* the preprocessing step.

For example, if only a certain template should include something in the body, e.g. after the introduction, the body injection can be set up to be inserted there. Since the sorting rules are applied, we can use this example to illustrate:

A introduction is named `0 - Intro.md` and an injection is set up as `1 - Injection.md`. Let's also assume another input file named `2 - Usage.md`. The sorting rules mean, that the order is extracted as follows:

`0 - Intro.md` -> 0

`2 - Usage.md` -> 2

`1 - Injection.md` -> 1

Since the file gets injected into the body stream, it is then sorted in the same manner. Thus, the order of files is

1. `0 - Intro.md`
2. `1 - Injection.md`
3. `2 - Usage.md`

Importantly, the injection is grouped with the other files during preprocessing, enabling merging during the conversion process.

10. Multi-file Output Model

Templates can not only produce single files, but in case of CustomPreProcessors conversion, they are able to produce multi-file outputs. When enabled, multi-file output allows the conversion to export one output file for each input file (including body injections).

An example for a usecase of multi-file output is a wiki. You may want to provide a printable PDF of your wiki while also creating a statically rendered website from the same inputs. Multi-file output allows for the rendered website to not be a single page.

Injections for multi-file outputs have a slightly different mental model to other templates. Instead of rendering the injections once, they are rendered *for each output file separately*, meaning the injection can include code that is unique to each output file.

For example, a header injection could include accessing the navigation metadata (see below) to render the current position in a tree. The concrete implementation is left as an excercise to the reader.

Multi-file output is enabled on a per-template basis and is only available to CustomPreProcessors conversions.

11. Metadata Generation and Injection

Metadata being a first class citizen of TiefDown, it is imperative to understand how metadata can be accessed. There are four ways to access metadata generated by TiefDown:

1. using the TeX helper file generated during TeX conversion (see TeX templates for details),
2. using the Typst helper file generated during Typst conversion (see Typst templates for details),
3. by accessing metadata fields in the preprocessor or processor arguments using template strings (`{{metafield}}`),
4. by reading the generated metadata files (see below),
5. or by using metadata inside pandoc lua filters (if generated, see below).

There are two types of metadata, as described below: project metadata and navigation metadata. Only project metadata can be accessed inside the TeX and Typst helpers as well as the templating strings. If navigation metadata is generated, it can be used in lua filters as well as read as a file.

11.1. Metadata Generation Settings

Metadata, importantly, is only injected into the pandoc process if it is explicitly generated for a template. The metadata generation settings can take four values:

- `None` : no metadata is generated and written for this template. This also means that pandoc does not get metadata injected.
- `Full` : Both navigation and project metadata are generated for this template and injected into pandoc.
- `NavOnly` : Only navigation metadata as described below is generated and injected into pandoc.
- `MetadataOnly` : Only project metadata as described below is generated and injected into pandoc.

If metadata generation is enabled, there is always a yml file generated. The yml file is then injected into the pandoc process as a metadata file. There is also an option to enable JSON metadata file generation, which can be enabled via the MetaGenerationFormat.

11.2. Project Metadata

Project metadata is simply the metadata collected through the shared- and markdown project specific metadata. The metadata is merged and written to the corresponding YML/JSON file.

11.3. Navigation Metadata

Navigation Metadata is split into two parts: global nav metadata, being a list of nodes of all files in the input; and per-file metadata, containing a reference to the own file as well as the next and previous node.

The per file metadata is generated and injected into pandoc for each file separately. That means, one could access the metadata for the file using Lua filters.

Global metadata is generated once and injected into pandoc for every file. This enables a user to easily create a navigation with all nodes.

12. Smart Clean

Smart clean is a basic usability improvement. It automatically removes old conversion folders when converting, leaving only a set amount of folders.