TiefDownConverter Documentation

Tiefseetauchner et al.

December 30, 2025

Contents

1. Introduction

For the documenation of the library, see docs.rs°.

For the documentation of the CLI, see TiefDownConverter°.

This is the documentation for the TiefDown concepts. This won't explain the library or the CLI usage, but rather function as an introduction to the basics of TiefDown for users and contributors alike.

1.1. What is TiefDown?

TiefDown is a project format for managing markdown files and converting them to other formats. It's not a markdown parser, but rather a project format and management system.

Importantly, the project is split in a few parts:

- The `manifest.toml` file, which contains all the information needed to manage and convert the project.
- The `template` directory, which contains all the templates for the project.
- One or more markdown directories, corresponding to markdown projects.

2. Manifest File

The manifest file is the heart of the project. It contains all the information needed to manage and convert the project.

It consists of a few important parts (for the full documentation, check https://docs.rs/tiefdownlib/latest/tiefdownlib/manifest_model/index.html°):

- A version number
  - This is used to determine if the manifest is compatible with the current version of TiefDown-Converter. If it's not, the manifest will be automatically updated in the process of loading. Newer versions of the manifest file are rejected by the implementation.
- The automatic smart clean flag
  - This is a boolean flag that determines if the project should be cleaned automatically when a conversion is run. This is useful for projects that are constantly being updated, allowing a user to decide how many conversion folders they want to keep.
- The smart clean threshold
  - This is the number of conversion folders that are kept before the oldest ones are deleted.
- A list of markdown projects
- A list of templates
- A custom processors object
- A table of shared metadata for all markdown projects
- A metadata settings object
- A list of profiles available for the conversion
- A list of injections

2.1. Building your own manifest

A manifest can grow to a complex web relatively quickly. Thus, in this section, I want to give a little bit of an example on building and reading manifests. *Knowledge of TOML is presupposed*

Consider the following manifest:

toml

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

[[markdown_projects]]
name = "Documentation"
output = "."
path = "markdown"
resources = ["resources/"]

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

This is the most basic manifest, corresponding to a TeX project. It defines the project meta information, as well as a template and a markdown project. It corresponds to a basic folder structure:

```
.
├── markdown/
│   ├── resources/
│   │   └── image1.png
```

```
5 │   ├── 1 - Introduction.md
6 │   └── 2 - TiefDown.md
7 ├── template/
8 │   ├── docs.tex
9 │   └── lib.sty
10 ├── Documentation.pdf
11 └── manifest.toml
```

Let us dissect the manifest, starting with the meta information.

toml

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

# ...
```

`smart_clean = true` sets up the automatic smart clean feature to be enabled.

`smart_clean_threshold = 3` sets the threshold of automatic smart cleaning to three.

`version = 6` defines the supported featureset of this project, setting the required TiefDownLib version.

As far as meta information goes, TiefDown is relatively lean. Let's now look at the `[[markdown_projects]]` section, which is TiefDowns way of specifying an input directory, as described in markdown projects.

toml

```toml
# ...

[[markdown_projects]]
name = "Documentation"
output = "."
path = "markdown"
resources = ["resources/"]

# ...
```

First off, we define the markdown project with a `name` parameter. This name is primarily for logging purposes, as well as for converting just a single markdown project.

`output = "."` defines the folder that the result files will be copied to. Per default, the `.` directory defines the TiefDown projects root directory.

`path = "markdown"` is the input directory in which the input files will copied from before conversion, and thus is the primary source of truth of input files.

`resources = ["resources/"]` is important here: it specifies that the `resources/` folder does not include input files and should not be converted, but is still available during the conversion process. This allows for images and other resources to be added on a per-markdown-project basis without requiring logic in the template. See custom resources for more information.

Lastly, there's the template section:

toml

```toml
# ...

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

This defines the (in this case, singular) template.

`name = "Documentation"` sets the name of the template, which serves as basic meta information for conversion.

`template_file = "docs.tex"` tells TiefDown, which TeX file should be converted. Here, docs.tex in the `template/` folder gets converted.

`template_type = "Tex"` is the template type, in our case TeX.

That is a basic manifest toml. Following is a more complex manifest, but beware, this is likely an excessively complex usecase.

——————— **TODO** ———————

3. Conversion Pipeline

The conversion pipeline is relatively complex, but important to understand. It consists of a few important steps, as outlined in the (slightly out of date) workflow diagram:

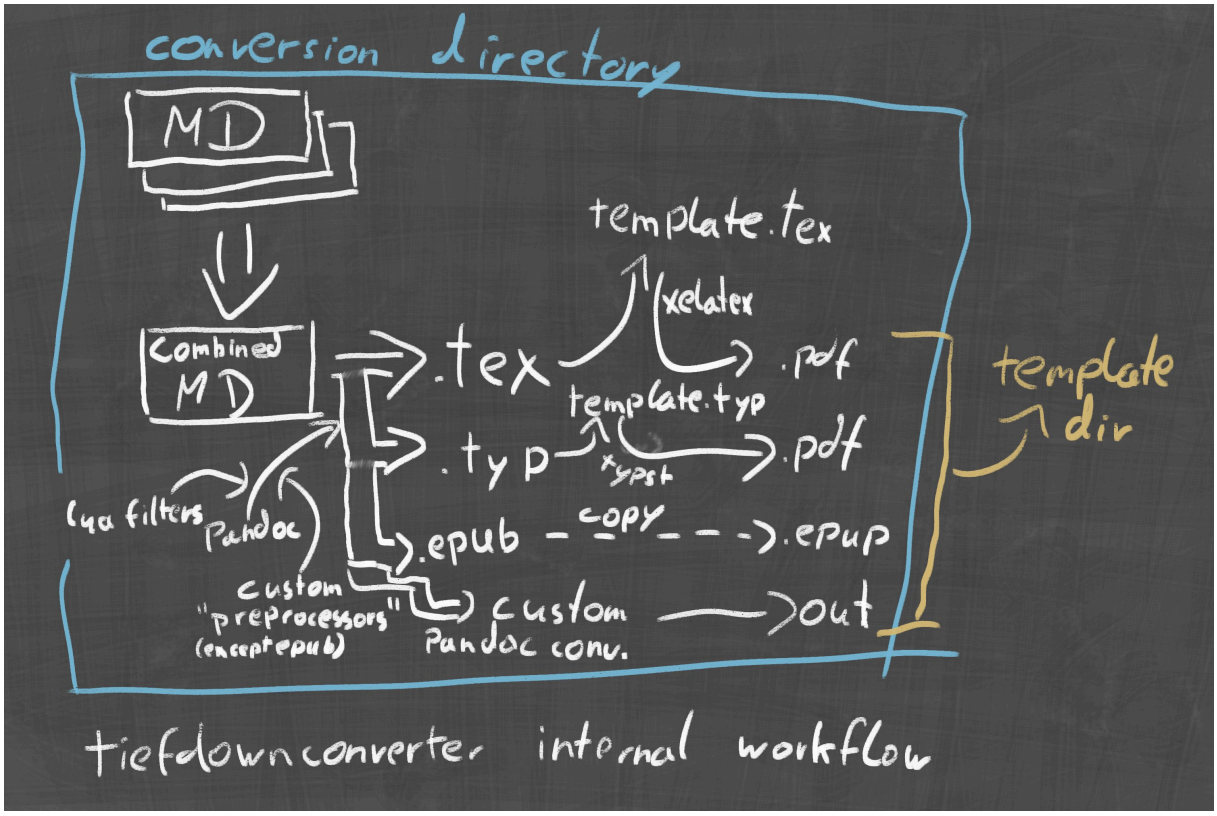

Figure 1: Workflow

Starting out, a new "compilation directory", more correctly named conversion directory, is created with the current timestamp. This is a scratch directory and should not be tracked via the VCS.

Then, the following steps are executed in order:

0) Conversion queue generation: First off, TiefDown takes the available Markdown projects and templates and computes a queue. This queue contains the markdown project as well as the templates to be converted. Per default, this is computed via the default profiles.
1) Input discovery and ordering: TiefDown copies and scans the markdown project directory, orders files by the first number in the filename (e.g. `Chapter 10 - ...`), and recurses into similarly numbered subfolders, preserving their order. Custom resources are not yet copied.
2) Preprocessing by extension: Inputs are grouped by file extension. For each group, TiefDown selects the matching preprocessor for the active template (either a default or a custom one filtered by extension) and runs it in the conversion folder. The stdout from each run is captured and stored in memory.
3) Combined output: The captured outputs are concatenated and written to the template's configured `preprocessors.combined_output` file (typically `output.tex` for LaTeX or `output.typ` for Typst). Your template includes this file (e.g. `#include "output.typ"`).
4) Metadata files: TiefDown generates `metadata.tex` or `metadata.typ` (only if they don't already exist) based on `[shared_metadata]`, any project-specific metadata, and your optional metadata settings.
5) Template processing: Depending on the template type, TiefDown runs XeLaTeX (twice) or Typst on the template file in the conversion folder, optionally passing arguments from a named processor. EPUB templates invoke Pandoc directly. `CustomPreprocessors` templates copy the

combined output as-is to the final destination. `CustomProcessor` templates run a final Pandoc invocation reading the combined Pandoc Native input and passing the configured processor arguments.

6) Finalization: The produced artifact (e.g. a PDF or EPUB) is then copied to the markdown project's configured output path.

4. Templates

Templating in TiefDown is done in several ways:

- LaTeX templates
  - The most basic form of templating, it generates a LaTeX document from the markdown files that can be included in a LaTeX document.
  - Supports Metadata file generation.
- Typst templates
  - Similar to LaTeX, it generates a Typst document from the markdown files that can be included in a Typst document.
  - Supports Metadata file generation.
- EPUB templates
  - A legacy templating system, it generates a EPUB document from the markdown files. Convoluted and very much custom to basic usage.
  - Adds Metadata directly to the EPUB file.
  - (!) This template type should be forgone in favour of CustomPreprocessors conversion.
- CustomPreprocessors conversion
  - A flexible pipeline where you define one or more preprocessors (Pandoc or any CLI) and write their combined output to a file your template or output references. No final processing step is performed.
  - Supports metadata insertion into CLI arguments.
- CustomProcessor conversion
  - Preprocesses inputs to a single Pandoc Native document and then runs a final Pandoc call with your processor arguments to produce the artifact (e.g., HTML, docx, reveal.js, etc.).

4.1. LaTeX Templates

LaTeX templates are the most intuitive form of templating in TiefDown, but also the most fleshed out. The basic usage generates a LaTeX document from markdown, usually output.tex, with lua-filters applied depending on the template, and then converts that template file to a PDF.

The LaTeX file must include the following:

```latex
\input{./output.tex}
```

This imports the converted markdown files into the LaTeX document. You may adjust the behaviour by using custom preprocessors and custom processors.

For Metadata, one can also import the metadata file, which is generated by TiefDown during the conversion process.

```latex
\input{./metadata.tex}
```

This file provides a macro to access metadata using the `\meta{}` keyword. This can be adjusted using the metadata settings.

There are preset templates available in the core library. These give a basic framework for extension and shouldn't be taken as the only way to use LaTeX templates.

4.2. Typst Templates

Typst templates are, in concept, identical to LaTeX templates. They generate a Typst document from markdown, usually output.typ, with lua-filters applied depending on the template, and then converts that template file to a PDF.

Importing the typst file works similar:

typst

```
#include "output.typ"
```

Again, see custom preprocessors and custom processors for more information on customization.

Metadata importing is easier as typst has an object system.

typst

```
#import "./metadata.typ": meta
```

One can then access metadata using the `meta` object.

There are also preset templates available in the core library. Again, these are just a basic suggestion.

4.3. EPUB Templates

Epub templates are a custom version of CustomProcessing templates. They add the ability to add CSS and embed fonts through the templating system without defining the files in the processor.

When adding a css file to the template directory of an epub template, it gets added to the pandoc conversion process with the `-c` flag.

Additionally, you can add a `fonts/` directory in the epub folder. Every file in this directory gets added to the conversion process using the `--epub-embed-font` flag.

NOTE: this template type is somewhat deprecated and will likely not be gaining features. It has some shortcuts to CustomProcessor conversion but for full control, use said template type instead.

4.4. CustomPreprocessors Conversion

CustomPreprocessors replaces the older "Custom Pandoc Conversion" naming. It lets you define exactly how inputs are preprocessed (with Pandoc or any CLI) and where the combined output is written. There is no additional processing step after the combined output is produced.

Key points

- Define one or more preprocessors under `[[custom_processors.preprocessors]]`.
- In the template, reference them via `preprocessors.preprocessors` and specify `preprocessors.combined_output`.
- The converter runs each preprocessor per extension group, captures stdout, and concatenates into the combined output file.
- Your template or output process must copy or reference this file to the final destination.

Example: single-file HTML without a LaTeX/Typst step

toml

```
[[templates]]
name = "HTML Article"
template_type = "CustomPreprocessors"
output = "article.html"

  [templates.preprocessors]
  preprocessors = ["md-to-html"]
  combined_output = "output.html"
```

```toml

[[custom_processors.preprocessors]]
name = "md-to-html"
cli = "pandoc"
cli_args = ["-f", "gfm", "-t", "html"]
```

4.5. CustomProcessor Conversion

CustomProcessor is a two-phase pipeline:

- Preprocess: Convert all inputs to Pandoc Native (defaults provided) and write a single `output.pandoc_native` file.
- Process: Run Pandoc once more, reading the native file and applying your processor's arguments to produce the final artifact.

Requirements and behavior

- The template's `processor` is required and must reference an entry under `[[custom_processors.processors]]`.
- `preprocessors.combined_output` should be a native file (defaults to `output.pandoc_native`).
- Lua filters configured on the template are applied to Pandoc preprocessing steps; the final Pandoc step uses `processor_args` only.

Example: produce a docx from the combined native document

toml

```toml
[[templates]]
name = "Docx"
template_type = "CustomProcessor"
output = "book.docx"
processor = "docx"

  [templates.preprocessors]
  preprocessors = ["native"]
  combined_output = "output.pandoc_native"

[[custom_processors.preprocessors]]
name = "native"
cli = "pandoc"
cli_args = ["-t", "native"]

[[custom_processors.processors]]
name = "docx"
processor_args = ["--reference-doc", "resources/ref.docx"]
```

5. Smart Clean

Smart clean automatically removes old conversion folders. When enabled via `smart_clean` in `manifest.toml`, TiefDown keeps only a given number of recent folders. The number is specified with `smart_clean_threshold` and defaults to `5`.

During conversion the library checks the amount of existing conversion folders and deletes the oldest ones once the threshold is exceeded.

6. Profiles

A profile is a named list of templates that can be converted together. Defining profiles avoids having to pass a long list of template names every time you run the converter.

Profiles are stored in the project's `manifest.toml`:

```toml
[[profiles]]
name = "Documentation"
templates = ["PDF Documentation LaTeX", "PDF Documentation"]
```

Use the `--profile` option with `tiefdownconverter convert` to select a profile. Markdown projects may also specify a `default_profile`; this profile is used if none is supplied on the command line.

7. Lua Filters

Lua filters allow you to modify the document structure during Pandoc's conversion step. They are attached to templates through the `filters` field. The value may be a single Lua file or a directory containing multiple filter scripts.

Pandoc executes the filters in the order they are listed. Filters can rename headers, insert custom blocks or otherwise transform the document before it reaches the template engine.

Example filter to adjust chapter headings:

lua

```lua
function Header(el)
  if el.level == 1 then
    return pandoc.RawBlock("latex", "\\chapter{" .. pandoc.utils.stringify(el.content) .. "}")
  end
end
```

For more details on writing filters see the Pandoc documentation.

Note: Lua filters apply to the Pandoc preprocessing step(s). If a template uses a custom preprocessor whose CLI is not Pandoc, those filters have no effect on that preprocessor's output.

8. Markdown Projects

A TiefDown project can contain multiple markdown projects. Each project defines where the source files live and where the converted results should be placed. The information is stored in `[[markdown_projects]]` entries in `manifest.toml`.

```toml
[[markdown_projects]]
name = "Book One"
path = "book_one/markdown"
output = "book_one/output"
```

A markdown project may define a `default_profile` used for conversion, a list of `resources` to copy into the conversion folder and its own metadata.

8.1. Custom Resources

Resources are additional files that are copied from the markdown project directory to the conversion folder before processing. Typical examples are images, CSS files or fonts needed by a template. Specify them in the `resources` array:

```toml
resources = ["resources/cover.png", "resources/styles.css"]
```

8.2. Markdown Project Metadata

Project specific metadata is stored under the `metadata_fields` table of a markdown project. These values are merged with the `[shared_metadata]` of the project during conversion. When keys collide, the markdown project metadata overrides the shared metadata.

9. Custom Processors

Custom processors let you change the commands used during conversion. They come in two forms:

- **Preprocessors** replace the default pandoc invocation that generates the intermediate file.
- **Processors** provide additional arguments to the program that handles the template itself (for example XeLaTeX or Typst).

A preprocessor is defined under `[[custom_processors.preprocessors]]`:

```toml
[[custom_processors.preprocessors]]
name = "Enable Listings"
cli_args = ["-t", "latex", "--listings"]
```

A preprocessor can also define a command using the `cli` field. This replaces the Pandoc preprocessing step with a custom cli command preprocessing step.

```toml
[[custom_processors.preprocessors]]
name = "Copy without modification"
cli = "cat"
cli_args = []
extension_filter = "typ"
```

Templates reference one or more preprocessors with their `preprocessors` field, which also has to define a `combined_output` field. The converter captures the stdout of each preprocessor run and writes it to this file, which your template then includes (`\input{./output.tex}` or `#include "./output.typ"`) or copies to the output location for `CustomPreprocessors` templates.

Processors are specified similarly and referenced via the `processor` field:

```toml
[[custom_processors.processors]]
name = "Typst Font Directory"
processor_args = ["--font-path", "fonts/"]
```

Usage notes:

- For `CustomPreprocessors` templates, there is no processor step. You are responsible for ensuring the `combined_output` is the final artifact you want to copy to the project's output.
- For `CustomProcessor` templates, a processor is required. TiefDown combines inputs to Pandoc Native (defaults provided) and then runs Pandoc with your `processor_args` to produce the final artifact.

These mechanisms allow fine-grained control over the conversion pipeline when the defaults are not sufficient.

9.1. Defaults

TiefDown provides reasonable defaults per template type:

- Tex templates preprocess inputs with Pandoc using `-t latex`, writing `output.tex`.
- Typst templates use two preprocessors: Pandoc with `-t typst` for non-`.typ` inputs, and a pass-through step for `.typ` inputs so existing Typst files are concatenated. The combined output is `output.typ`.

You can override a default for a particular extension by defining a preprocessor with a matching `extension_filter`; defaults for other extensions remain.

Preprocessors can be scoped by extension via `extension_filter`, which matches only the file extension (glob patterns such as `t*` are supported). If you omit the filter, the preprocessor acts as

a fallback when no more specific filter matches. Defaults exist per template type and are merged by extension; defining your own preprocessor for a particular extension replaces the default for that extension but leaves the others intact. Finally, `cli_args` support metadata substitution, so any occurrence of `{{key}}` is replaced with the corresponding metadata value at conversion time.

10. Shared Metadata

Shared metadata is defined once for the whole project and is available to every markdown project. It lives under `[shared_metadata]` in the manifest file and is merged with project specific metadata at conversion time. Values defined in a markdown project override entries from the shared metadata.

Use shared metadata for information that stays the same across multiple books or documents, like the publisher or an overarching author list.

11. Metadata Settings

Metadata settings influence how metadata files are generated. The `[metadata_settings]` table currently supports the `metadata_prefix` option. This prefix determines the name of the macro or object used to access metadata in templates.

For example, with

toml

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

the generated LaTeX file defines a `\book{}` command while Typst exposes a `book` object. In other words, the prefix fully replaces the default `meta` name. If no prefix is set the command and object are called `meta`.

12. Injections

Injections are a project-driven way to insert input files at either the top, inside, or bottom of a document, correspondingly named header, body and footer injections.

Injections serve as a template scoped way to add content to a conversion. An injection is defined once in the manifest and then assigned to a template as any of the aforementioned methods.

During conversion, the injected files are resolved and placed in the list of input files to be converted to the intermediary format.

12.1. Header and footer injections

Header injections are inserted at the top of the document, while footer injections are inserted at the bottom, both in the order as defined in the manifest. The first file defined in the first referenced injection is placed first, the last file of the last referenced injection last.

12.2. Body injections

Body injections are injected before the main sorting algorithm as any file in the input directory would be. That means they get sorted in accordance with the primary sorting algorithm.