wmk

static site generator

Baldur Kristinsson

version 1.18, 2024-09-01

Abstract wmk is a flexible and versatile static site generator written in Python. It supports themes, SCSS, shortcodes and site search, and although it is primarily designed for Markdown or HTML content files, it can handle multiple input formats via Pandoc. wmk can be extended by writing hooks in Python, making it possible to override or modify most aspects of its operation. To get the source code, install the program, or contribute to the project, please take a look at the GitHub repository.

Contents

Main features 2
Installation 3
    Method 1: git + pip 3
    Method 2: Homebrew 3
    Method 3: Docker 4
Usage 4
File organization 5
Input formats 7
A few gotchas 8
Context variables 8
Configuration file 10
A note on Pandoc 14
Available themes 14
Shortcodes 14
    Default shortcodes 16
Template library 18
    seo.mc 18
    atom_xml.mc 19
    sitemap_xml.mc 19
    Usage in Jinja templates 19
Site, page and nav variables 19
    The nav variable 20
        Manually configured 20

Automatically generated 21
The TOC variable 21
System variables 21
Templates 22
Taxonomy handling 22
Variables affecting rendering 23
Standard variables and their recommended meaning 24
Typical site variables 25
Classic meta tags 25
Dates 26
Media content 27
Taxonomy 27
Template filters 27
Working with lists of pages 28
General searching/filtering 28
Specialized searching/filtering 29
Searching/filtering using SQL 31
Sorting 32
Pagination 32
Render to an arbitrary file 32
Site search 33
Using Lunr 33
Limitations of Lunr 33
Overview of alternative solutions 34
Example: Pagefind 34
Overriding and extending wmk via hooks 35
Examples 36
Incorporating external sources 37
Example: Import from WordPress 37

Main features

The following features are present in several static site generators (SSGs); you might almost call them standard:

- Markdown or HTML content with YAML metadata in the frontmatter.
- Support for themes.
- Sass/SCSS support (via `libsass`).
- Can generate a search index for use by `lunr.js`.
- Shortcodes for more expressive and extensible content.

The following features are among the ones that set wmk apart:

- By default, the content is rendered using Mako, a template system which makes all the resources of Python easily available to you. However Jinja2 templates are also supported if that is what you prefer.
- "Stand-alone" templates – i.e. templates that are not used for presenting markdown-based content – are also rendered if present. This can e.g. be used for list pages or content based on external sources (such as a database).

- Additional data for the site may be loaded from separate YAML files or even (with a small amount of Python/Mako code) from other data sources such as CSV files, SQL databases or REST/graphql APIs.
- The shortcode system is quite powerful and flexible. For instance, among the default shortcodes are an image thumbnailer and a page list component. A shortcode is just a template, so you can easily build your own.
- Optional support for the powerful Pandoc document converter, for the entire site or on a page-by-page basis. This gives you access to such features as LaTeX math markup and academic citations, as well as to Pandoc's well-designed filter system for extending markdown. Pandoc also enables you to export your content to other formats (such as PDF) in addition to HTML, if you so wish.
- Also via Pandoc, support for several non-markdown input formats for content, namely LaTeX, Org, RST, Textile, Djot, Typst, man, JATS, TEI, Docbook, RTF, DOCX, ODT and EPUB.

The only major feature that wmk is missing compared to some other SSGs is tight integration with a Javascript assets pipeline and interaction layer. Although wmk allows you to configure virtually any assets processing you like, this nevertheless means that if your site is reliant upon React, Vue or similar, then other options are probably more convenient.

That exception aside, wmk is suitable for building any small or medium-sized static website (up to a couple of thousand pages, depending on the content).

Installation

Method 1: git + pip

Clone this repo into your chosen location (`$myrepo`) and install the necessary Python modules into a virtual environment:

```
cd $myrepo
python3 -m venv venv
. venv/bin/activate
pip install -r requirements.txt
```

After that, either put `$myrepo/bin` into your `$PATH` or create a symlink from somewhere in your `$PATH` to `$myrepo/bin/wmk`.

Required software (aside from Python, of course):

- `rsync` (for static file copying).
- For `wmk watch` functionality (as well as `watch-serve`), you need either `inotifywait` or `fswatch` to be installed and in your `$PATH`. If both are available, the former is preferred.

wmk requires a Unix-like environment. In particular, bash must be installed in `/bin/bash`, and the directory separator is assumed to be `/`.

Method 2: Homebrew

If you are on MacOS and already have Homebrew, this is the easiest installation method.

First add the tap to your repositories:

```
brew tap bk/wmk
```

Then install wmk from it:

```
brew install --build-from-source wmk
```

Method 3: Docker

If you are neither on a modern Linux system nor on MacOS with Homebrew, it may be a better option for you to run wmk via Docker. In that case, after cloning the repo (or simply copying the `Dockerfile` from it) you can give the command

```
docker build -t wmk .
```

in the directory containing the `Dockerfile`, in order to build an image called `wmk`. You can then run the various wmk subcommands via Docker, for instance

```
docker run --rm --volume $(pwd):/data --user $(id -u):$(id -g) wmk b .
```

to build the wmk project in the current directory, or

```
docker run --rm -i -t --volume $(pwd):/data --user $(id -u):$(id -g) -p 7007:7007
wmk ws . -i 0.0.0.0
```

to watch for changes in the current directory and run a webserver for the built files.

Obviously, such commands can be unwieldy, so if you run them regularly you may want to create aliases or wrappers for them.

Usage

The `wmk` command structure is `wmk <action> <base_directory>`. The base directory is of course the directory containing the source files for the site. (They are actually in subdirectories such as `templates`, `content`, etc. – see the "File organization" section below).

- `wmk info $basedir`: Shows the real path to the location of `wmk.py` and of the content base directory. E.g. `wmk info .`. Synonyms for `info` are `env` and `debug`.
- `wmk init $basedir`: In a folder which contains `content/` (with markdown or HTML files) but no `wmk_config.yaml`, creates some initial templates as well as a sample `wmk_config.yaml`, thus making it quicker for you to start a new project.
- `wmk build $basedir [-q|--quick]`: Compiles/copies files into `$basedir/htdocs`. If `-q` or `--quick` is specified as the third argument, only files considered to have changed, based on timestamp checking, are processed. Synonyms for `run` are `run`, `b` and `r`.
- `wmk watch $basedir`: Watches for changes in the source directories inside `$basedir` and recompiles if changes are detected. (Note that `build` is not performed automatically before setting up file wathcing, so you may want to run that first). A synonym for `watch` is `w`.
- `wmk serve $basedir [-p|--port <portnum>] [-i|--ip <ip-addr>]`: Serves the files in `$basedir/htdocs` on `http://127.0.0.1:7007/` by default. The IP and port can be modified with the `-p` and `-i` switches or be be configured via `wmk_config.yaml` – see the "Configuration file" section). Synonyms for `serve` are `srv` and `s`.
- `wmk watch-serve $basedir [-p|--port <portnum>] [-i|--ip <ip-addr>]`: Combines `watch` and `serve` in one command. Synonym: `ws`.

- `wmk clear-cache $basedir`: Remove the HTML rendering cache, which is a SQLite file in `$basedir/tmp/`. This should only be necessary in case of changed shortcodes or shortcode dependencies. Note that the cache can be disabled in `wmk_config.yaml` by setting `use_cache` to `false`, or on file-by-file basis via a frontmatter setting (`no_cache`). A synonym for `clear-cache` is `c`.
- `wmk preview $basedir $filename` where `$filename` is the name of a file relative to the `content` subdirectory of `$basedir`. This prints (to stdout) the HTML which the given file will be converted to (before it is passed to the template and before potential post-processing). Example: `wmk preview . index.md`.
- `wmk admin $basedir`: Build the site and then start wmkAdmin, which must have been installed beforehand into the `admin` subdirectory of the `$basedir` (or into the subdirectory specified with `wmk admin $basedir $subdir`). The subdirectory may be a symbolic link pointing to a central instance. wmkAdmin allows you to manage the content of the site via a web interface. It is not designed to allow you to install or modify themes or perform tasks that require more technical knowledge, and works best for a standard site based on Markdown or HTML files in the `content` directory.
- `wmk repl $basedir`: Launch a Python shell (ipython, bpython or python3, in order of preference) with the wmk environment loaded and with the `$basedir` as current working directory. Useful for examining wmk's view of the site content or debugging `MDContent` filtering methods. For these purposes, `from wmk import get_content_info`, followed by `content = get_content_info('.')` is often a good start.
- `wmk pip <pip-command>`: Run `pip` in the virtual environment used by wmk. Mainly useful for installing or upgrading Python modules that you want to use in Python files belonging to your projects.
- `wmk homedir`: Outputs the path to `wmk`'s installation directory. May be useful in shell scripts.

File organization

Inside a given working directory, `wmk` assumes the following subdirectories for content and output. They will be created if they do not exist:

- `htdocs`: The output directory. Rendered, processed or copied content is placed here, and `wmk serve` will serve files from this directory.
- `templates`: Mako templates (or Jinja2 templates if `jinja2_templates` is set to true in `wmk_config.yaml`). Templates with the extension `.mhtml` (`.html` if Jinja2 templates are being used) are rendered directly into `htdocs` as `.html` files (or another extension if the filename ends with `.$ext\.mhtml/$ext\.html`, where `$ext` is a string consisting of 2-4 alphanumeric characters), *unless* their filename starts with a dot or underscore or contains the string `base`, or if they are inside a subdirectory named `base`. For details on context variables received by such stand-alone templates, see the "Context variables" section below.
- `content`: typically markdown (`*.md`) and/or HTML (`*.html*`) content with YAML metadata, although other formats are also supported. For a full list, see the "Input formats" section above.

  - Markdown (or other supported content) will be converted into HTML and then "wrapped" in a layout using the `template` specified in the metadata or `md_base.mhtml` by default.
  - HTML files inside `content` are assumed to be fragments rather than complete documents. Accordingly, they will be wrapped in a layout just like the converted markdown. In general, such content is treated just like markdown files except that the markdown-to-html conversion step is skipped. For instance, shortcodes can be used normally, although they may not work as expected if they return markdown rather than HTML. (Complete HTML documents are best placed in `static` rather than `content`).
  - The YAML metadata may be (a) at the top of the md/html document itself, inside a frontmatter block delimited by `---`; (b) in a separate file with the same filename as the content file, but with an extra `.yaml` extension added; or (c) it may be in `index.yaml` files which are inherited by subdirectories and the files contained in them. For details, see the "Site, page and nav variables" section below.
  - The target filename will be `index.html` in a directory corresponding to the basename of the source file – unless `pretty_path` in the metadata is `false` or the name of the file itself is `index.md` or `index.html` (in which case the relative path is remains the same, except that the extension is of course changed to `.html` if the source is a markdown file).
  - The processed content will be passed to the template as a string in the context variable `CONTENT`, along with other metadata.
  - A YAML datasource can be specified in the metadata block as `LOAD`; the data in this file will be added to the context. For further details on the context variables, see the "Context variables" section.
  - Files that have other extensions than `.md`, `.html` or `.yaml` will be copied directly over to the (appropriate subdirectory of the) `htdocs` directory. This is so as to enable "bundling", i.e. keeping images and "attachments" together with related markdown files.
- `data`: YAML files for additional metadata. May be referenced in frontmatter data or used by templates. Other data files (CSV, SQLite, etc.) should typically also be placed here.
- `py`: Directory for Python files. This directory is automatically added to the front of `sys.path` before Mako or Jinja2 is initialized, meaning that templates can import modules placed here. Implicit imports (for Mako only) are possible by setting `mako_imports` in the config file (see the "Configuration file" section). There are also two special files that may be placed here: `wmk_autolaod.py` in your project, and `wmk_theme_autoload.py` in the theme's `py/` directory. If one or both of these is present, wmk imports a dict named `autoload` from them. This means that you can assign `PREPROCESS` and `POSTPROCESS` page actions by name (i.e. keys in the `autoload` dict) rather than as function references, which in turn makes it possible to specify them in the frontmatter directly rather than having to do it via a shortcode. (For more on `PRE-` and `POSTPROCESS`, see the "Site, page and nav variables" section).
- `assets`: Assets for an asset pipeline. The only default handling of assets involves compiling SCSS/Sass files in the subdirectory `scss`. They will be compiled to CSS which

is placed in the target directory `htdocs/css`. Other assets handling can be configured via settings in the configuration file, e.g. `assets_commands` and `assets_fingerprinting`. This will be described in more detail in the "Site, page and nav variables" section. Also take note of the `fingerprint` template filter, described in the "Template filters" section.
- `static`: Static files. Everything in here will be rsynced directly over to `htdocs`.

Input formats

The format of the files in the `content/` directory is determined on the basis if their file extension. The following extensions are recognized by default:

- `.md`, `.mdwn`, `.mdown`, `.markdown`, `.gfm`, `.mmd`: Markdown files. If Pandoc is being used, the input formats `.gfm` and `.mmd` will be assumed to be `gfm` (GitHub-flavored markdown) and `markdown_mmd` (MultiMarkdown), respectively. Note, however, that currently non-YAML metadata given in MultiMarkdown format is not picked up automatically in `.mmd` files).
- `.htm`, `.html`: HTML. These are typically not standalone HTML documents but will be "wrapped" by the configured layout template. Like other input files, they may have a YAML frontmatter block.
- `.tex`: LaTeX format. Currently ConTeXt is not supported.
- `.org`: Org-mode format.
- `.rst`: ReStructured Text format (RST).
- `.textile`: Textile markup format.
- `.dj`: The Djot lightweight markup format.
- `.man`: Roff man format.
- `.rtf`: Rich Text Format (RTF).
- `.typ`: Typst format.
- `.jats`, `.xml`: The XML-based JATS (Journal Article Tag Suite) format.
- `.docbook`: The XML-based DocBook format.
- `.tei`: The Simple variant of the XML-based TEI (Text Encoding Initiative) format.
- `.docx`: MS Word DOCX a.k.a. "Office Open XML" format.
- `.odt`: OpenDocument Text format.
- `.epub`: The EPUB e-book format.

Pandoc is turned on automatically for all non-markdown, non-HTML formats in the above list. In order to use such content, a sufficiently recent version of Pandoc therefore *must* be installed.

The list of input formats and how they are handled is configurable through the `content_extensions` setting in the config file. See the "Configuration file" section below for details.

Note: The three formats JATS, DocBook and TEI are all XML-based. Files in all three formats would therefore often use the generic `.xml` extension. However, `wmk` currently assumes that `.xml` implies that the JATS format is intended. If you want to force `wmk` to handle a file with that extension as DocBook or TEI, you would have to add an external YAML metadata file with `pandoc_input_format` set to the appropriate value.

In-file YAML frontmatter is supported for all of the above except for the three binary formats DOCX, ODT and EPUB. Of course, metadata from an associated external YAML file or inherited metadata applies in all cases. In addition, the "native" metadata seen by

Pandoc for most of the formats (more precisely all non-markdown, non-HTML formats other than Textile, which uses YAML frontmatter natively) will be used as a fallback source of in-file metadata, although this is limited to specific standard keys such as `title`, `author` and `date`.

Note that although other input formats are supported, the *canonical* format is still markdown. Unless there is a special reason to do otherwise it is the most sensible and efficient choice for websites generated using `wmk`.

A few gotchas

When creating a website with wmk, you might want to keep the following things in mind lest they surprise you:

- The order of operations is as follows: (1) Copy files from `static/`; (2) run asset pipeline; (3) render standalone templates from `templates`; (4) render markdown content from `content`. As a consequence, later steps **may overwrite** files placed by earlier steps. This is intentional but definitely something to keep in mind.
- For the `run` and `watch` actions when `-q` or `--quick` is specified as a modifier, `wmk.py` uses timestamps to prevent unnecessary re-rendering of templates, markdown files and SCSS sources. The check is rather primitive and does not take account of such things as shortcodes or changed dependencies in the template chain. As a rule, `--quick` is therefore **not recommended** unless you are working on a small, self-contained set of content files.
- If templates or shortcodes have been changed it may sometimes be necessary to clear out the page rendering cache with `wmc c`. During development you may want to add `use_cache: no` to the `wmk_config.yaml` file. Also, some pages should never be cached, in which case it is a good idea to add `no_cache: true` to their frontmatter.
- If files are removed from source directories the corresponding files in `htdocs/` **will not disappear** automatically. You have to clear them out manually – or simply remove the entire directory and regenerate.

Context variables

The Mako/Jinja2 templates, whether they are stand-alone or being used to render markdown (or other) content, receive the following context variables:

- `DATADIR`: The full path to the `data` directory.
- `WEBROOT`: The full path to the `htdocs` directory.
- `CONTENTDIR`: The full path to the `content` directory.
- `TEMPLATES`: A list of all templates which will potentially be rendered as stand-alone. Each item in the list contains the keys `src` (relative path to the source template), `src_path` (full path to the source template), `target` (full path of the file to be written), and `url` (relative url to the file to be written).
- `MDCONTENT`: An `MDContentList` representing all the content files which will potentially be rendered by a template. Each item in the list contains the keys `source_file`, `source_file_short` (truncated and full paths to the source), `target` (html file to be written), `template` (filename of the template which will be used for rendering), `data` (most of the context variables seen by this content), `doc` (the raw content document source), and `url` (the `SELF_URL` value for this content – see below). Note that `MDCONTENT`

is not available inside shortcodes. An `MDContentList` is a list object with some convenience methods for filtering and sorting. It will be described further later on.
- Whatever is defined under `template_context` in the `wmk_config.yaml` file (see the "Configuration file" section below).
- `SELF_URL`: The relative path to the HTML file which the output of the template will be written to.
- `SELF_TEMPLATE`: The path to the current template file (from the template root).
- `ASSETS_MAP`: A map of fingerprinted assets (such as javascript or css files), used by the `fingerprint` template filter.
- `LOADER`: The template loader/env. In the case of Mako, this is a `TemplateLookup` object; in the case of Jinja2 this is an `Environment` object with a `FileSystemLoader` loader.
- `site`: A dict-like object containing the variables specified under the `site` key in `wmk_config.yaml`.
- `CACHE`: An ordinary dictionary object, intended for use by templates as a simple shared in-memory cache.

In the case of Jinja2 templates, three extra context variables are available:
- `mako_lookup`: A Mako `TemplateLookup` instance which makes it possible to call Mako templates from a Jinja2 template.
- `get_context`: A function returning all context variables as a dict.
- `import`: An alias for `importlib.import_module` and can thus be used to import a Python module into a Jinja template as the value of a variable, e.g. `{% set utils = import('my_utils') %}`. The main intent is to make code inside the project `py/` subdirectory as easily available in Jinja templates as it is in Mako templates.

When templates are rendering markdown (or other) content, they additionally get the following context variables:
- `CONTENT`: The rendered HTML produced from the source document.
- `RAW_CONTENT`: The original source document.
- `SELF_FULL_PATH`: The full filesystem path to the source document file.
- `SELF_SHORT_PATH`: The path to the source document file relative to the content directory.
- `MTIME`: A datetime object representing the modification time for the source file.
- `DATE`: A datetime object representing the first found value of `date`, `pubdate`, `modified_date`, `expire_date`, or `created_date` found in the YAML front matter, or the `MTIME` value as a fallback. Since this is guaranteed to be present, it is natural to use it for sorting and generic display purposes.
- `RENDERER`: A callable which enables a template to render markdown in `wmk`'s own environment. This is mainly so that it is possible to support shortcodes which depend on other markdown content which itself may contain shortcodes. The callable receives a dict containing the keys `doc` (the markdown) and `data` (the context variables) and returns rendered HTML.
- `page`: A dict-like object containing the variables defined in the YAML meta section at the top of the markdown file, in `index.yaml` files in the markdown file directory and its parent directories inside `content`, and possibly in YAML files from the `data` directory loaded via the `LOAD` directive in the metadata.

For further details on context variables set in the document frontmatter and in `index.yaml` files, see the "Site, page and nav variables" section below.

Configuration file

A config file, `$basedir/wmk_config.yaml`, can be used to configure many aspects of how `wmk` operates. The name of the file may be changed by setting the environment variable `WMK_CONFIG` which should contain a filename without a leading directory path.

The configuration file **must** exist (but may be empty). If it specifies a theme and a file named `wmk_config.yaml` (*regardless* of the `WMK_CONFIG` environment variable setting) exists in the theme directory, then any settings in that file will be merged with the main config – unless `ignore_theme_conf` is true.

It is also possible to split the configuration file up into several smaller files. These are placed in the `wmk_config.d/` directory (inside the base directory). The filename of each yaml file in that directory (minus the `.yaml` extension) is treated as a key and the contents as its value. Subdirectories can be used to represent a nested structure. For instance, the file `wmk_config.d/site/colors/darkmode.yaml` would contain the settings that will be visible to templates as the `site.colors.darkmode` variable. Note that the `WMK_CONFIG` environment variable affects the name of the directory looked for; setting it to `myconf.yaml` would e.g. mean that `wmk` will inspect `myconf.d/` for extra configuration settings instead of `wmk_config.d/` (although this does not apply to themes, whose configuration file/directory name is fixed).

Currently there is support for the following settings:

- `template_context`: Default values for the context passed to templates. This should be a dict.
- `site`: Values for common information relating to the website. These are also added to the template context under the key `site`. They are often used by templates and themes to affect the look and feel of the website. For further discussion, see the "Site, page and nav variables" section below.
- `render_drafts`: Normally, content files with `draft` set to a true value in the metadata section will be skipped during rendering. This can be turned off (so that the `draft` status flag is ignored) by setting `render_drafts` to True in the config file.
- `markdown_extensions`: A list of extensions to enable for markdown processing by Python-Markdown. The default is `['extra', 'sane_lists']`. If you specify third-party extensions here, you have to install them into the Python virtual environment first. Obviously, this has no effect if `pandoc` is true. May be set or overridden through frontmatter variables.
- `markdown_extension_configs`: Settings for your markdown extensions. May be set in the config file or in the frontmatter. For convenience, there are special frontmatter settings for two extensions, namely for `toc` and `wikilinks`:
  - The `toc` boolean setting will turn the `toc` extension off if set to False and on if set to True, regardless of its presence in `markdown_extensions`.
  - If `toc` is in `markdown_extensions` (or has been turned on via the `toc` boolean), then the `toc_depth` frontmatter variable will affect the configuration of the extension regardless of the `markdown_extension_configs` setting.

    - If `wikilinks` is in `markdown_extensions` then the options specified in the `wikilinks` frontmatter setting will be passed on to the extension. Example: `wikilinks: {'base_url': '/somewhere'}`.
- `pandoc`: Normally Python-Markdown is used for markdown processing, but if this boolean setting is true, then Pandoc via Pypandoc is used by default instead. This can be turned off or on through frontmatter variables as well. Another config setting which affects whether Pandoc is used is `content_extensions`, for which see below.
- `pandoc_filters`, `pandoc_options`: Lists of filters and options for Pandoc. Has no effect unless `pandoc` is true. May be set or overridden through frontmatter variables.
- `pandoc_input_format`: Which input format to assume for Pandoc; has no effect unless `pandoc` is true. The default value is `markdown`. If set, the value should be a markdown subvariant for markdown-like content, i.e. one of `markdown` (pandoc-flavoured), `gfm` (github-flavoured), `markdown_mmd` (MultiMarkdown), `markdown_phpextra`, `markdown_strict`, `commonmark`, or `commonmark_x`. As for other supported input formats, there is little reason to set `pandoc_input_format` explicitly for them, since they have no variants in the relevant sense, and the right format is picked based on the file extension. May be set or overridden through frontmatter variables.
- `pandoc_output_format`: Output format for Pandoc; has no effect unless `pandoc` is true. This should be a HTML variant, i.e. either `html`, `html5` or `html4`, or alternatively one of the HTML-based slide formats, i.e. `s5`, `slideous`, `slidy`, `dzslides` or `revealjs`. Chunked HTML (new in Pandoc 3) is not supported. May be set or overridden through frontmatter variables.
- `pandoc_extra_formats`, `pandoc_extra_formats_settings`: If `pandoc` is True, then `pandoc_extra_formats` in the frontmatter can be used to convert to other formats than HTML, for instance PDF or MS Word (docx). `pandoc_extra_formats` is a dict where each key is a format name (e.g. `pdf`) and its value is the output filename relative to the web root (e.g. `subdir/myfile.pdf`). The special value `auto` indicates that the name of the output file should be based on that of the source file but with the file extension replaced by the name of the format. For instance, a source file named `subdir/index.md` (relative to the content directory) maps to an output file named `subdir/index.pdf` (relative to the web root directory) if the output format is `pdf`, and so on. `pandoc_extra_formats_settings`, if present, contains any special settings for the conversion in the form of a dict where each key is a format name and its value is either a dict with the keys `extra_args` and/or `filters`, or a list (which then is interpreted as the value of the `extra_args` setting).
- `slugify_dirs`: Affects the names of directories created in `htdocs` because of the `pretty_path` setting. If `true` (which is the default), the name will be identical to the `slug` of the source file. If explicitly set to `false`, then the directory name will be the same as the basename of the source file, almost regardless of the characters in the filename.
- `use_cache`: boolean, True by default. If you set this to False, the rendering cache will be disabled. This is useful for small and medium-sized projects where the final HTML output often depends on factors other than the content file alone. Note that caching for a specific file can be turned off by putting `no_cache: true` in the frontmatter.
- `cache_mtime_matters`: boolean, False by default. Normally only the body of the markdown file and a few selected processing settings make up the cache key. If, on the other

hand, this setting is True (either in the configuration file or in the frontmatter), then the modification time of the markdown file affects the cache key, so touching the file is sufficient for refreshing its cache entry.

- `use_sass`: A boolean indicating whether to handle Sass/SCSS files in `assets/scss` automatically. True by default.
- `sass_output_style`: The output style for Sass/SCSS rendering. This should be one of `compact`, `compressed`, `expanded` or `nested`. The default is `expanded`. Has no effect if `use_sass` is false.
- `assets_map`: An assets map is a mapping from filenames or aliases to names of files containing a hash identifier (under the webroot). A typical entry might thus map from `/css/style.css` to `/css/style.1234abcdef56.css`. The value of this setting is either a dict or the name of a JSON or YAML file (inside the data directory) containing the mapping. It will be available to templates as `ASSETS_MAP`.
- `assets_fingerprinting`: A boolean indicating whether to automatically fingerprint assets files (i.e. add hash indicators to their names). If true, any fingerprinted files will be added to the `ASSETS_MAP` template variable.
- `assets_fingerprinting_conf`: A dict where the keys are subdirectories of the webroot, e.g. `js` or `img/icons`, and the values are dicts containing the keys `pattern` and (optionally) `exclude`. These are regular expressions indicating which files to fingerprint under these directories. The filename is fingerprinted if it matches `pattern` but does not match `exclude`. (The default value of `exclude` looks for files that appear to have been fingerprinted already and thus does not normally need to be set). The default value of this setting is a simple setup for the `js` and `css` subdirectories of the webroot.
- `assets_commands`: A list of arbitrary commands to run at the assets compilation stage (just before Sass/SCSS files in `assets/scss` are processed, assuming `use_sass` is not false). The commands are run in order inside the base directory of the site. Example: `['bin/fetch_external_assets.sh', 'node esbuild.mjs']`.
- `lunr_index`: If this is True, a search index for `lunr.js` is written as a file named `idx.json` in the root of the `htdocs/` directory. Basic information about each page (title and summary) is additionally written to `idx.summaries.json`.
- `lunr_index_fields`: The default fields for generating the lunr search index are `title` and `body`. Additional fields and their weight can be configured through this variable. For instance `{"title": 10, "tags": 5, "body": 1}`. Aside from `body`, the fields are assumed to be attributes of `page`.
- `lunr_languages`: A two-letter language code or a list of such codes, indicating which language(s) to use for stemming when building a Lunr index. The default language is `en`. For more on this, see the "Site search" section below.
- `http`: This is is a dict for configuring the address used for `wmk serve`. It may contain either or both of two keys: `port` (default: 7007) and `ip` (default: 127.0.0.1). Can also be set directly via command line options.
- `output_directory`: Normally the output will be written to the directory `htdocs` inside the basedir, but this can be overridden by setting this configuration variable. The value should be a relative path that does not start with `/` or `.`, e.g. `site` or `public`.

- `mako_imports`: A list of Python statements to add to the top of each generated Mako template module file. Generally these are import statements.
- `theme`: This is the name of a subdirectory to the directory `$basedir/themes` (or a symlink placed there) in which to look for extra `static`, `assets`, `py` and `template` directories. Note that neither `content` nor `data` directories of a theme will be used by `wmk`. A theme-provided template may be rendered as stand-alone page, but only if no local template overrides it (i.e. has the same relative path). Mako's internal template lookup will similarly first look for referenced components in the normal `template` directory before looking in the theme directory. Configuration settings from `wmk_config.yaml` in the theme directory will be used as long as they do not conflict with those in the main config file.
- `ignore_theme_conf`: If set to true in the main configuration file, this tells wmk to ignore any settings in `wmk_config.yaml` in the theme directory.
- `extra_template_dirs`: A list of directories in which to look for template files. These are placed after both `$basedir/templates` and theme-provided templates in the template engine search path. This makes it possible to build up a library of components which can be easily used on multiple sites and across different themes.
- `jinja2_templates`: If this boolean setting is true, it indicates that the template files in the `template` directory (and supplied by the theme, or otherwise in the template engine search path) are to be interpreted by Jinja2 rather than Mako. Note that Jinja2 templates used standalone or as layout templates for Markdown content should have the extension `.html` rather than `.mhtml`.
- `redirects`: If this is True or a string pointing to a YAML file in the `data/` directory (whose default name is `redirects.yaml`), then wmk will write HTML stubs containing `<meta http-equiv="refresh" ...>` in the indicated locations. The contents of the YAML file is a list of entries with the keys `from` and `to`. The former is a path under `htdocs/` or a list of such paths, while `to` is an absolute or relative URL which you are to be redirected to.
- `content_extensions`: Customize which file extensions are handled inside the `content/` directory. May be a list (e.g. `['.md', '.html']`) or a dict. The value for each key in the dict should itself be a dict where the following keys have an effect: `pandoc` (boolean), `pandoc_input_format` (string), `is_binary` (boolean), `raw` (boolean), `pandoc_binary_format` (string). See the value of `DEFAULT_CONTENT_EXTENSIONS` in `wmk.py` for details.
- `mdcontent_json`: This option may specify the name of a JSON file to which to write the entire `MDCONTENT` object in serialized form, along with the environment variables for each page. The destination file may be either in `htdocs/`, `data/` or `tmp/`. If the file path does not start with one of these, `data` is assumed. The specified (or implied) directory must exist.
- `init_commands`: A list of arbitrary commands to run at the very beginning of processing, just after theme settings have been loaded and the Python search path configured. They are run in order inside the base directory of the site.
- `cleanup_commands`: A list of arbitrary commands to run at the very end of wmk processing. The commands are run in order inside the base directory of the site.

A note on Pandoc

Pandoc's variant of markdown is very featureful and sophisticated, but since its use in `wmk` involves spawning an external process for each content file being converted, it is quite a bit slower than Python-Markdown. Therefore, it is only recommended if you really do need it. Often, even if you do, it can be turned on for individual pages or site sections rather than for the entire site. (Of course, if you are working with non-markdown, non-HTML input content, using Pandoc is unavoidable.)

If you decide to use Pandoc for a medium or large site (or if you have a significant amount of non-markdown content), it is recommended to turn the `use_cache` setting on in the configuration file. When doing this, be aware that content that is sensitive to changes apart from the content file itself will need to be marked as non-cacheable by adding `no_cache: true` to the frontmatter. If you for instance call the `pagelist()` shortcode in the page, you would normally want to mark the file in this way.

The `markdown_extensions` setting will of course not affect `pandoc`, but there is one extension which is partially emulated in `wmk`'s Pandoc setup, namely toc.

If the `toc` frontmatter variable is true and the string `[TOC]` is present as a separate line in a document which is to be processed by pandoc, then it will be asked to generate a table of contents which will be placed in the indicated location, just like the `toc` extension for Python-Markdown does. The `toc_depth` setting (whose default value is 3) is respected as well, although only in its integer form and not as a range (such as `"2-4"`). This applies not only to markdown documents but also to the non-markdown formats handled by Pandoc.

Available themes

There are several wmk themes available:

- Lanyonesque, a blog-oriented theme based on the Jekyll theme Lanyon. Demo.
- Historia, a flexible single-page theme based on the Story template by HTML5 UP. Demo.
- Picompany, a general-purpose theme based on the Company template that accompanies the PicoCSS documentation. Demo.
- WDocs, a full-featured documentation theme. Demo.
- Walto, a more lightweight documentation theme. The wmk documentation site uses Walto.
- Birta, the theme used for bornogtonlist.net
- Cider, a simple and elegant theme for product or company sites.

Shortcodes

A shortcode consists of an opening tag, `{{<`, followed by any number of whitespace characters, followed by a string representing the "short version" of the content, followed by any number of whitespace characters and the closing tag `>}}`.

A typical use case is to easily embed content from external sites into your markdown (or other) content. More advanced possibilities include formatting a table containing data from a CSV file or generating a cropped and scaled thumbnail image.

Shortcodes are normally implemented as Mako components named `<shortcode>.mc` in the `shortcodes` subdirectory of `templates` (or of some other directory in your template search path, e.g. `themes/<my-theme>/templates/shortcodes`). If `jinja2_templates` is set to true, however, the shortcode templates are in Jinja2 format instead, and use the `.jc` extension rather than `.mc`.

The shortcode itself looks like a function call. Note that positional arguments can only be used if the component has an appropriate `<%page>` block declaring the expected arguments.

The shortcode component will have access to a context composed of (1) the parameters directly specified in the shortcode call; (2) the information from the metadata block of the markdown file in which it appears; (3) a counter variable, `nth`, indicating number of invocations for that kind of shortcode in that markdown document; and (4) the global template variables.

Shortcodes are applied **before** the content document is converted to HTML, so it is possible to replace a shortcode with markdown content which will then be processed normally. Note, however, that this may lead to undesirable results when you use such shortcodes in a non-markdown content document.

A consequence of this is that shortcodes do **not** have direct access to (1) the list of files to be processed, i.e. `MDCONTENT`, or (2) the rendered HTML (including the parts supplied by the Mako template). A shortcode which needs either of these must place a (potential) placeholder in the markdown source as well as a callback in `page.POSTPROCESS`. Each callback in this list will be called just before the generated HTML is written to `htdocs/` (or, in the case of a cached page, after document conversion but right before the Mako layout template is called), receiving the full HTML as a first argument followed by the rest of the context for the page. Examples of such shortcodes are `linkto` and `pagelist`, described below. (For more on `page.POSTPROCESS` and `page.PREPROCESS`, see the "Site, page and nav variables" section below).

Here is an example of a simple shortcode call in markdown content:

```
### Yearly expenses

{{< csv_table('expenses_2021.csv') >}}
```

Here is an example `csv_table.mc` Mako component that might handle the above shortcode call:

```
<%page args="csvfile, delimiter=',', caption=None"/>
<%! import os, csv %>
<%
info = []
with open(os.path.join(context.get('DATADIR'), csvfile.strip('/'))) as f:
    info = list(csv.DictReader(f, delimiter=delimiter))
if not info:
    return ''
keys = info[0].keys()
%>
<table class="csv-table">
  % if caption:
```

```
    <caption>${ caption }</caption>
  % endif
  <thead>
    <tr>
      % for k in keys:
        <th>${ k }</th>
      % endfor
    </tr>
  </thead>
  <tbody>
    % for row in info:
      <tr>
        % for k in keys:
          <td>${ row[k] }</td>
        % endfor
      </tr>
    % endfor
  </tbody>
</table>
```

Note that if Jinja2 templates are being used, positional arguments are not supported except for in built-in shortcodes, so the shortcode call in the Markdown in the above example would have to be changed to `cvs_table(csvfile='expenses_2021.csv')` or similar.

Shortcodes can take up more than one line if desired, for instance:

```
{{< figure(
      src="/img/2021/11/crocodile-or-alligator.jpg",
      caption="""
Although they appear similar, **crocodiles** and **alligators** differ in easy-
to-spot ways:

- crocodiles have narrower and longer heads;
- their snouts are more V-shaped;
- also, crocodiles have a protruding tooth, visible when their mouth is closed.
""") >}}
```

In this example, the caption contains markdown which would be converted to HTML by the shortcode component (assuming we're dealing with the default `figure` shortcode).

Note that shortcodes are not escaped inside code blocks, so if you need to show examples of shortcode usage in your content they must be escaped in some way in such contexts. One relatively painless way is to put a non-breaking space character after the opening tag `{{<` instead of a space.

Default shortcodes

The following default shortcodes are provided by the `wmk` installation:

- `figure`: An image wrapped in a `<figure>` tag. Accepts the following arguments: `src` (the image path or URL), `img_link`, `link_target`, `caption`, `figtitle`, `alt`, `credit` (image attribution), `credit_link`, `width`, `height`, `resize`. Except for `src`, all arguments are optional. The caption and credit will be treated as markdown. If `resize` is True and width and height have been provided, then a resized version of the image is used instead of the

original via the `resize_image` shortcode (the details can be controlled by specifying a dict representing `resize_image` arguments rather than a boolean; see below).

- `gist`: A Github gist. Two arguments, both required: `username` and `gist_id`.
- `include`: Insert the contents of the named file at this point. One required argument: `filename`. Optional argument: `fallback` (which defaults to the empty string), indicating what to show if the file is not found. The file must be inside the content directory (`CONTENTDIR`), otherwise it will not be read. The path is interpreted as relative to the directory in which the content file is placed. A path starting with `/` is taken to start at `CONTENTDIR`. Nested includes are possible but the paths of sub-includes are interpreted relative to the original directory (rather than the directory in which the included file has been placed). Note that `include()` is always handled before other shortcodes.
- `linkto`: Links to the first matching (markdown-based) page. The first parameter, `page`, specifies the page which is to be linked to. This is either
  (a) a simple string representing a slug, title, (partial) path/filename or (partial) URL; or (b) a `match_expr` in the form of a dict or list which will be passed to `page_match()` with a `limit` of 1. Optional arguments: `label` (the link text; the default is the title of the matching page); `ordering`, passed to `page_match()` if applicable; `fallback`, the text to be shown if no matching page is found: `(LINKTO: page not found)` by default; the boolean `unique`, which if set to True causes a fatal error to be raised if multiple pages are found to match; and `link_attr`, which is a string to insert into the `<a>` tag (by default `class="linkto"`). A query string or anchor ID fragment for the link can be added via `link_append`, e.g. `link_append='#section2'` or `link_append='?q=searchstring'`. If the boolean parameter `url_only` is True, then the output will not be a link but only the URL (including `link_append`, if any).
- `pagelist`: Runs a `page_match()` and lists the found pages. Required argument: `match_expr`. Optional arguments: `exclude_expr`, `ordering`, `limit`, `template`, `fallback`, `template_args`, `sql_match`. `exclude_expr` is a match expression which serves to *exclude* pages from the list found using the `match_expr`. For instance, `pagelist({'has_tag': True}, exclude_expr={'has_tag': 'private'})` finds all tagged pages except those that have the tag `private`. The default way of representing the found pages is a simple unordered list of links to them, using the page titles as the link text. If nothing is found, a string specified in the `fallback` parameter (by default an empty string) replaces the shortcode call. The formatting of the list can be changed by pointing to a Mako template using the `template` argument, which will receive the argument `pagelist` (a `MDContentList` of found pages), as well as `template_args`, if any. The template will only be called if something is found. If `sql_match` is True, the `match_expr` and `ordering` and `limit` will be passed to `page_match_sql()` (as `where_clause`, `order_by`, and `limit`, respectively) rather than to `page_match()`.
- `resize_image`: Scales and crops images to a specified size. Required arguments: `path`, `width`, `height`. Optional arguments: `op` ('fit_width', 'fit_height', 'fit', 'fill'; the last is the default), `format` ('jpg' or 'png'; default is 'jpg'), `quality` (default 0.75 and applies only to jpegs), `focal_point` (default `center`; only used for `op='fill'`). Returns a path under `/resized_images/` (possibly prefixed with the value of `site.leading_path`) pointing to the resized version of the image. The filename incorporates a SHA1 hash, so repeated

requests for the same resize operation are only performed once. The source `path` is taken to be relative to the `WEBROOT`, i.e. the project `htdocs` directory.

- `template`: The first argument (`template`) is either the filename of a template or literal template source code. The heuristic used to distinguish between these two cases is simply that filenames are assumed never to contain whitespace while source code always does. In either case, the template is called and its output inserted into the content document. The boolean argument `is_jinja` (default False) can be used to indicate that the given template source code is to be handled by Jinja2; otherwise Mako is assumed. For template files, however, the currently active engine as determined by the value of the `jinja2_templates` is always used, regardless of the `is_jinja` parameter. Any additional arguments are passed directly on to the template (which will also see the normal template context for the shortcode itself).
- `twitter`: A tweet. Takes a `tweet_id`, which may be a Twitter status URL or the last part (i.e. the actual ID) of the URL.
- `var`: The value of a variable, e.g. `"page.title"` or `"site.description"`. One required argument: `varname`. Optional argument: `default` (which defaults to the empty string), indicating what to show if the variable is not available.
- `vimeo`: A Vimeo video. One required argument: `id`. Optional arguments: `css_class`, `autoplay`, `dnt` (do not track), `muted`, `title`.
- `youtube`: A YouTube video. One required argument: `id`. Optional arguments: `css_class`, `autoplay`, `title`, `nowrap`, `nocookie`, `width`, `height`.
- `wp`: A link to Wikipedia. One required argument: `title`. Optional arguments: `label`, `lang`. Example: `{{< wp('L.L. Zamenhof', lang='eo') >}}`.

Template library

It is generally up to the site or theme author to define any needed Mako/Jinja templates. In rare cases, however, the templates are general enough that it may be natural to distribute them with wmk itself in the form of a Mako template library located under `/lib/`.

seo.mc

The template `/lib/seo.mc` makes it easier to format metadata for use in the `<head>` section of a base template. It is used in something like the following way:

```
<%namespace import="seo" file="/lib/seo.mc" />
% if page:
  ${ seo(site, page, url=SELF_URL, title=self.page_title) }
% else:
  ${ seo(site, page=None, url=SELF_URL, title=self.page_title,
         img=self.attr.main_image) }
% endif
```

This will add common meta tags (including basic OpenGraph and JSON-LD information). By default, it also adds a `<title>` tag. For further details regarding the functionality, see the template file itself.

atom_xml.mc

The template `/lib/atom_xml.mc` can be used to facilitate the creation of an Atom feed for the website. Set `site.base_url` to a valid URL and set `site.atom_feed` to a true value. Then create a file named `atom.xml.mhtml` in the template root, containing something like the following:

```
<%namespace name="atom" file="/lib/atom_xml.mc" />\
${ atom.feed(contentlist=MDCONTENT.sorted_by_date()) }\
```

There are several optional parameters (`with_img`, `get_img`, `with_summary`, `get_summary`, `pubdate_attr`, `updated_attr`, `with_full_text`, `limit`) for tweaking the output.

sitemap_xml.mc

Similarly, `/lib/sitemap_xml.mc` can be used to create a `siteamp.xml` file. Set `site.enable_sitemap` to a true value and ensure that `site.base_url` is present. Then create a file named `sitemap.xml.mhtml` in the template root, with the following content:

```
<%namespace import="sitemap" file="/lib/sitemap_xml.mc" />\
${ sitemap(contentlist=MDCONTENT) }\
```

Usage in Jinja templates

No Jinja version of these components has been created, but the Mako version can be called from a Jinja2 template using code such as the following:

```
{% set seo = mako_lookup.get_template("/lib/seo.mc").get_def("seo") %}
{{ seo.render(site, page, url=SELF_URL, title=page.title) |safe }}
```

Site, page and nav variables

When a markdown file (or other supported content) is rendered, the Mako template receives a number of context variables as partly described above. A few of these variables, such as `MDTEMPLATES` and `DATADIR` are set directly by `wmk` (see above). Others are user-configured either (1) in `wmk_config.yaml` (the contents of the `site` object and potentially additional "global" variables in `template_context`); or (2) the cascade of `index.yaml` files in the `content` directory and its subdirectories along with the YAML frontmatter of the markdown file itself, the result of which is placed in the `page` object.

When gathering the content of the `page` variable, `wmk` will start by looking for `index.yaml` files in each parent directory of the markdown file in question, starting at the root of the `content` directory and moving upwards, at each step extending and potentially overriding the data gathered at previous stages. Only then will the YAML in the front-matter of the file itself be parsed and added to the `page` data.

The file-specific frontmatter may be in the content file itself, or it may be in a separate YAML file with the same name as the content file but with an extra `.yaml` extension. For instance, if the content filename is `important.md`, then the YAML file would be named `important.md.yaml`. If both in-file and external frontmatter is present, the two will be merged, with the in-file values "winning" in case of conflict.

At any point, a data source in this cascade may specify an extra YAML file using the special `LOAD` variable. This file will then be loaded as well and subsequently treated as

if the data in it had been specified directly at the start of the file containing the `LOAD` directive.

Which variables are defined and used by templates is very much up the user, although a few of them have a predefined meaning to `wmk` itself. For making it easier to switch between different themes it is however suggested to stick to the following meaning of some of the variables:

The variables `site` and `page` are dicts with a thin convenience layer on top which makes it possible to reference subkeys belonging to them in templates using dot notation rather than subscripts. For instance, if `page` has a dict variable named `foo`, then a template could contain a fragment such as `${ page.foo.bar or 'splat' }` – even if the `foo` dict does not contain a key named `bar`. Without this syntactic sugar you would have to write something much more defensive and long-winded such as `${ page.foo.bar if page.foo and 'bar' in page.foo else 'splat' }`.

The `nav` variable

The `nav` variable is an easy way of configuring a navigation tree for websites with content that has a hierarchical structure, such as a typical documentation site. It is set via the `nav` key in the `wmk_config.yaml` file and is represented in templates as a `Nav` object.

A `Nav` instance is a list-like object with two types of entries: links and sections. A link is just a title and a URL. A section has a title and a list of links or sections (possibly nested). It may or may not have a url as well.

Each item has a `parent` (with the `nav` itself as the top level parent) and a `level` (starting from 0 for the immediate children of the `nav`). The `nav` has a `homepage` attribute which by default is the first local link in the nav. Each local link has `previous` and `next` attributes. Each section has `children`. There are other attributes but these are the basics.

Manually configured

A typical explicit nav setting looks something like this:

```
nav:
    - Home: /
    - 'User Guide [url=/guide/]':
        - Lorem:
            - Ipsum: /guide/ipsum/
            - Eu fuit: /guide/mageisse/
        - Dolor sit amet: /guide/concupescit/
    - Resources:
        - Community: 'https://example.com/'
        - Source code: 'https://github.com/example/com/'
    - About:
        - License: /about/license/
        - History: /about/history/
```

A manually configured `nav` setting of this kind is only necessary if you want to link to something outside of the site from the nav (as in the above example). Otherwise, it depends on the kind of content you have whether a manually defined or an automatically generated nav would be more appropriate to your use case.

Automatically generated

A `nav` object can also be generated by `wmk` from the frontmatter of the content files. In order for this to happen two conditions must be met:

1. The value of `nav` in `wmk_config.yaml` is set to `auto`.
2. Each item in the config that is to appear in the navigation tree must have at least the key `nav_section` in the frontmatter. To determine ordering, `nav_order` or (equivalently) `weight` may also be specified; and if necessary the page `title` may be overridden in the nav by setting the `nav_title` attribute.

The `nav_section` value `Root` is special. Pages assigned to that section are placed directly at the front of the `nav` structure. For many sites, you would simply place this in the `index.yaml` file at the root of your content directory.

Other sections are simply grouped by their `nav_section` values. Please note that these values are case-sensitive.

Within each section the link items are ordered by the their `nav_order`/`weight` value, which should be an integer. If two or more items have the same ordering number, they are ordered by `nav_title`/`title`.

The sections themselves are ordered within the nav by the lowest `nav_order`/`weight` value of the pages assigned to them. Sections with the same ordering number are sorted alphabetically.

A page may be excluded from the nav (even if it has a `nav_section`) by setting its `nav_exclude` to a true value.

The pages inside each section may be nested to an arbitrary depth by using the `nav_parent` (or `parent`) variable in the frontmatter of the subpages. The value of this is normally the `nav_title`/`title` (case-insensitive) of the parent page. However, if more than one page in the same section has the same title, then one may disambiguate by specifying the `slug` or (in extreme cases) the `id` of the target page instead.

The `TOC` variable

When a page is rendered, the generated HTML is examined and a simple table of contents object constructed, which will be available to templates as `TOC`. It contains a list of the top-level headings (i.e. H1 headings, or H2 headings if no H1 headings are present, etc.), with lower-level headings hierarchically arranged in its `children`. Other attributes are `url` and `title`. `TOC.item_count` contains the heading count (regardless of nesting).

The `TOC` variable can e.g. be used by the page template to show a table of contents elsewhere on the page.

The table of contents object is not constructed unless each heading has an `id` attribute. When using the default python-markdown, this means that the `toc` extension must be active.

System variables

The following frontmatter variables affect the operation of `wmk` itself, rather than being exclusively used by templates.

Templates

Note that a variable called something like `page.foo` below is referenced as such in templates but specified in YAML frontmatter simply as `foo: somevalue`.

- `page.template` specifies the template which will render the content.
- `page.layout` is used by several other static site generators. For compatibility with them, this variable is supported as a fallback synonym with `template`. It has no effect unless `template` has not been specified explicitly anywhere in the cascade of frontmatter data sources.

For both `template` and `layout`, the `.mhtml` (or `.html` in the case of Jinja2) extension of the template may be omitted. If the `template` value appears to have no extension, `.mhtml` or `.html` (depending on the template engine) is assumed; but if the intended template file has a different extension, then it must of course be specified.

Likewise, a leading `base/` directory may be omitted when specifying `template` or `layout`. For instance, a `layout` value of `post` would find the template file `base/post.mhtml` unless a `post.mhtml` file exists in the template root somewhere in the template search path.

If neither `template` nor `layout` has been specified and no `default_template` setting is found in `wmk_config.yaml`, the default template name for markdown files is `md_base.mhtml` (or `md_base.html` if Jinja2 templates have been selected).

The special `template`/`layout` value `__empty__` (case-insensitive) indicates that no base template should be applied to the given content file.

Taxonomy handling

A taxonomy is a classification of pieces of content for the purpose of grouping them together. Common taxonomy types are tags, categories, sections and article authors. However, the taxonomy that is appropriate to a particular website mainly depends on the content. On a site with book reviews you would have genres, book authors and publishers, on a movie site you would have genres and actors, and so on. Each set of frontmatter classifiers (e.g. the single classifier `tag` or the list `['tag', 'tags']`) used in a taxonomy may be called a *term*. Each term may have several *values*, and each value represents a list of content items associated with it.

Up to version 1.13 of wmk, taxonomies had to be handled by templates, and this is still the best way to do it if you want a form of presentation which is tailored to a particular term. However, as a consequence, themes had to be designed around specific taxonomies, typically tags, categories, or sections. In other words, the presentation of taxonomies was not primarily content-driven.

From version 1.13 it is therefore possible to specify the taxonomy criteria directly in the front matter of the main content page for the corresponding term. Here is an example based on a movie site, for the term *director*. The content file might be named `directors/index.md`:

```
---
title: Directors
date: 2024-11-01
template: base/taxonomy/list.mhtml
TAXONOMY:
```

```
  taxon: ['director', 'directors']
  order: name
  detail_template: base/taxonomy/detail.mhtml
  list_settings:
    pagination: true
    per_page: 24
  detail_settings:
    biographies: directors.yaml
    item_template: lib/movie_teaser.mc
---

Below is a list of the directors of the movies
that have been covered on this website.

Click on the name of a director to see a short biography
and an overview of their movies.
```

The frontmatter variable `page.TAXONOMY` triggers the special processing of the page, provided that it contains at least the subkeys `taxon` and `detail_template`. This special processing consists in the following:

1. wmk fetches a list of values for the term specified in `taxon` using the `taxonomy_info()` method of `MDCONTENT`. This will be added to the template context as `TAXONS`.
2. For each value in the list, wmk renders the template `detail_template` with the same context, except that the two keys `TAXON` (the value) and `TAXON_INDEX` (the 0-based index of the value in the list) are added. (If no `detailt_template` is specified, then the template for the page is used). Each `TAXON` has `items` which represent the pages tagged with that director, and the main job of thet detail page is to show a list of them to the user. The result is written to a destination file the name of which is based on the destination of the rendered Markdown content plus the slug of the string identifying the value (e.g. `directors/orson-welles/index.html` in this example). The target url will be available as `TAXON['url']` (and thus also under the key `'url'` for each item in `TAXONS`).
3. wmk resumes normal operation by calling the main template with the modified template context as well as the content from the markdown file, and writes the result to the target file.

Please note that the settings in `list_settings` and `detail_settings` in the example above are merely for the purposes of illustration. Whether any of them are actually supported is entirely up to the template or theme author. The only subvariables used by wmk itself are `taxon`, `order` (if present), and `detail_template` (if present).

Variables affecting rendering

- `page.slug`: If the value of `slug` is nonempty and consists exclusively of lowercase alphanumeric characters, underscores and hyphens (i.e. matches the regular expression `^[a-z0-9_-]+$`), then this will be used instead of the basename of the markdown file to determine where to write the output. If a `slug` variable is missing, one will be automatically added by `wmk` based on the basename of the current content file (as well as, in the case of `index.*` files, their proximate directory). Note that autogenerated

slugs do not affect the location of the destination file. Slugs are not necessarily unique, but `page.id` values are – see below.

- `page.pretty_path`: If this is true, the basename of the markdown filename (or the slug) will become a directory name and the HTML output will be written to `index.html` inside that directory. By default it is false for files named `index.md` or `index.html` and true for all other files. If the filename contains symbols that do not match the character class `[\w.,=-]`, then it will be "slugified" before final processing (although this only works for languages using the Latin alphabet).
- `page.do_not_render`: Tells `wmk` not to write the output of this template to a file in `htdocs`. All other processing will be done, so the gathered information can be used by templates for various purposes. (This is similar to the `headless` setting in Hugo).
- `page.draft`: If this is true, it prevents further processing of the markdown file unless `render_drafts` has been set to true in the config file.
- `page.no_cache`: If this is true, the rendering cache will not be used for this file. (See also the `use_cache` setting in the configuration file).
- `page.markdown_extensions`, `page.markdown_extension_configs`, `page.pandoc`, `page.pandoc_filters`, `page.pandoc_options`, `page.pandoc_input_format`, `page.pandoc_output_format`: See the description of these options in the section on the configuration file, above.
- `page.POSTPROCESS`: This contains a list of processing instructions which are called on the rendered HTML just before writing it to the output directory. Each instruction is either a function (placed into `POSTPROCESS` by a shortcode) or a string (possibly specified in the frontmatter). If the latter, it points to a function entry in the `autoload` dict imported from either the project's `py/wmk_autoload.py` file or the theme's `py/wmk_theme_autoload.py` file. In either case, the function receives the html as the first argument while the rest of the arguments constitute the template context. It should return the processed html.
- `page.PREPROCESS`: This is analogous to `page.POSTPROCESS`, except that the instructions in the list are applied to the markdown (or other content document) just before converting it to HTML. The function receives two arguments: the document text and the `page` object. It should return the altered document source. Note that this happens before shortcodes have been expanded, so (unlike `page.POSTPROCESS`) such actions cannot be added via shortcode.

Note that if two files in the same directory have the same slug, they may both be rendered to the same output file; it is unpredictable which of them will go last (and thus "win the race"). The same kind of conflict may arise between a slug and a filename or even between two filenames containing non-ascii characters. It is up to the content author to take care to avoid this; `wmk` does nothing to prevent it.

Standard variables and their recommended meaning

The following variables are not used directly by `wmk` but affect templates in different ways. It is a list of recommendations rather than something which must be necessarily followed.

Typical site variables

Site variables are the keys-value pairs under `site:` in `wmk_config.yaml`.

- `site.title`: Name or title of the site.
- `site.lang`: Language code, e.g. 'en' or 'en-us'. Used e.g. for translations by some themes.
- `site.locale`: Locale code, e.g. 'en_US.utf8'. Used when sorting `MDCONTENT` by name or title.
- `site.tagline`: Subtitle or slogan.
- `site.description`: Site description.
- `site.author`: Main author/proprietor of the site. Depending on the site templates (or the theme), may be a string or a dict with keys such as "name", "email", etc.
- `site.base_url`: The protocol and hostname of the site (perhaps followed by a directory path if `site.leading_path` is not being used). Normally without a trailing slash.
- `site.leading_path`: If the web pages built by `wmk` are not at the root of the website but in a subdirectory, this is the appropriate prefix path. Normally without a trailing slash.
- `site.build_time`: This is automatically added to the site variable by `wmk`. It is a datetime object indicating when the rendering phase of the current run started.
- `site.lunr_search`: A boolean automatically added to the site variable. It is true when `lunr_index` is true in the configuration file.

Templates or themes may be configurable through various site variables, e.g. `site.paginate` for number of items per page in listings or `site.mainfont` for configuring the font family.

Classic meta tags

These variables mostly relate to the text content and affect the metadata section of the `<head>` of the HTML page.

- `page.title`: The title of the page, typically placed in the `<title>` tag in the `<head>` and used as a heading on the page. Normally the title should not be repeated as a header in the body of the markdown file. Most markdown documents should have a title. If it is not explicitly specified, the title will be generated automatically from the filename.
- `page.slug`: See above. If it is missing, the slug is created from the title.
- `page.id`: This is guaranteed to be unique at rendering time. If it is present but not unique, then "–1", "–2", etc., will be appended as necessary. If it is not explicitly specified, then it is generated by slugifying the full path to the source markdown file (relative to the content directory). For instance, `blog/2022/09/The letter Þ in Old English.md` will become the ID `blog-2022-09-the-letter-th-in-old-english`.
- `page.description`: Affects the `<meta name="description" ...>` tag in the `<head>` of the page. The variable `summary` (see later) may also be used as fallback here.
- `page.keywords`: Affects the `<meta name="keywords" ...>` tag in the `<head>` of the page. This may be either a list or a string (where items are separated with commas).
- `page.robots`: Instructions for Google and other search engines relating to this content (e.g. `noindex, nofollow`) should be placed in this variable.
- `page.author`: The name of the author (if there is only one). May lead to `<meta name="keywords" ...>` tag in the `<head>` as well as appear in the body of the rendered

HTML file. Some themes may expect this to be a dict with keys such as `name`, `email`, `image`, etc.

- `page.authors`: If there are many authors they may be specified here as a list. It is up to the template how to handle it if both `author` and `authors` are specified, but one way is to add the `author` to the `authors` unless already present in the list.
- `page.summary`: This may affect the `<meta name="description" ...>` tag as a fallback if no `description` is provided, but its main purpose is for list pages with article teasers and similar content. If it is initially not present but `page.generate_summary` is True, then it will be generated from the body of the page, as follows: (1) if the HTML comment `<!--more-->` is present in the body, then any non-heading content before that will be used as the summary; (2) otherwise the first paragraph of the body will be used. In either case, if the autogenerated summary is longer than 300 characters, then it is truncated so as to be shorter than that (this maximum length is configurable with `page.summary_max_length`). Autogenerated summaries will contain neither HTML tags nor Markdown markup; if this is desired, the summary must be explicitly added to the frontmatter.

Note that this is by no means an exhaustive list of variables likely to affect the `<head>` part of the generated HTML. For instance, several other variables may affect meta tags used for sharing on social media. One of the more common ones is probably `page.image` (described below). In any case, the list of supported frontmatter attributes and how they are interpreted is for the most part up to the theme or template author.

Dates

Dates and datetimes should normally be in a format conformant with or similar to ISO 8601, e.g. `2021-09-19` and `2021-09-19T09:19:21+00:00`. The `T` may be replaced with a space and the time zone may be omitted (localtime is assumed). If the datetime string contains hours it should also contain minutes, but seconds may be omitted. If these rules are followed, the following variables are converted to date or datetime objects (depending on the length of the string) before they are passed on to templates.

- `page.date`: A generic date or datetime associated with the document.
- `page.pubdate`: The date/datetime when first published. Currently `wmk b` does not omit rendering files with `date` or `pubdate` in the future, but it may do so in a later version.
- `page.modified_date`: The last-modified date/datetime. Note that `wmk` will also add the variable `MTIME`, which is the modification time of the file containing the markdown source, so this information can be inferred from that if this variable is not explicitly specified.
- `page.created_date`: The date the document was first created.
- `page.expire_date`: The date from which the document should no longer be published. Similarly to `pubdate`, this currently has no direct effect on how `wmk` builds and renders the site but may do so in a later version.
- `page.auto_date`: If this is True and no `page.date` is present (or rather the field specified in `page.auto_date_field`, which deafults to `date`), then `wmk` tries to look for an ISO date in the source filename or its directory path. In this context, that means a group of 4+2+2 digits with a separator which may be either `-`, `_`, or `/`: e.g. `posts/2024-05-13-find-the-fish.md` or `diary/2024/02/19/spam.org`. If a date is found, then `page.date` is

set accordingly. (Obviously you would normally set `auto_date` in an `index.yaml` file so as to affect all content files in that directory and its subdirectories.)

See also the description of the `DATE` and `MTIME` context variables above.

Media content

- `page.image`: The main image associated with the document. Affects the `og:image` meta tag in HTML output and may be used for both teasers and content rendering.
- `page.images`: A list of images associated with the document. If `image` is not specified, the main image will be taken to be the first in the list.
- `page.audio`: A list of audio files/urls associated with this document.
- `page.videos`: A list of video files/urls associated with this document.
- `page.attachments`: A list of attachments (e.g. PDF files) associated with this document.

Taxonomy

See also the description of `page.TAXONOMY` above. The following are terms commonly used for taxonomy purposes:

- `page.section`: One of a quite small number of sections on the site, often corresponding to the leading subdirectory in `content`. E.g. “blog”, “docs”, “products”.
- `page.categories`: A list of broad categories the page belongs to. E.g. “Art”, “Science”, “Food”. The first-named category may be regarded as the primary one.
- `page.tags`: A list of tags relevant to the content of the page. E.g. “quantum physics”, “knitting”, “Italian food”.
- `page.weight`: A measure of importance attached to a page and used as an ordering key for a list of pages. This should be a positive integer. The list is normally ascending, i.e. with the lower numbers at the top. (Pages may of course be ordered by other criteria, e.g. by `pubdate`).

Template filters

In addition to the built-in template filters provided by Mako or Jinja2 respectively, the following filters are by default made available in templates:

- `date`: date formatting using strftime. By default, the format ‘%c’ is used. A different format is specified using the `fmt` parameter, e.g.: `${ page.pubdate | date(fmt=site.date_format) }`.
- `date_to_iso`: Format a datetime as ISO 8601 (or similarly, depending on parameters). The parameters are `sep` (the separator between the date part and the time part; by default ‘T’, but a space is sensible as well); `upto` (by default ‘sec’, but ‘day’, ‘hour’ and ‘frac’ are also acceptable values); and `with_tz` (by default False).
- `date_to_rfc822`: Format a datetime as RFC 822 (a common datetime format in email headers and some types of XML documents).
- `date_short`: E.g. “7 Nov 2022”.
- `date_short_us`: E.g. “Nov 7th, 2022”.
- `date_long`: E.g. “7 November 2022”.
- `date_long_us`: E.g. “November 7th, 2022”.
- `slugify`: Turns a string into a slug. Only works for strings in the Latin alphabet.

- `markdownify`: Convert markdown to HTML. It is possible to specify custom extensions using the `extensions` argument.
- `truncate`: Convert markdown/html to plaintext and return the first `length` characters (default: 200), with an `ellipsis` (default: "…") appended if any shortening has taken place.
- `truncate_words`: Convert markdown/html to plaintext and return the first `length` words (default: 25), with an `ellipsis` (default "…") appended if any shortening has taken place.
- `p_unwrap`: Remove a wrapping `<p>` tag if and only if there is only one paragraph of text. Suitable for short pieces of text to which a markdownify filter has previously been applied. Example: `<h1>${ page.title | markdownify,p_unwrap }</h1>`.
- `strip_html`: Remove any markdown/html markup from the text. Paragraphs will not be preserved.
- `cleanurl`: Remove trailing 'index.html' from URLs.
- `url`: Unless the given path already starts with '/', '.' or 'http', prefix it with the first defined leading path of `site.leading_path`, `site.base_url` or a literal `/`. Postfix a `/` unless the path already has one or seems to end with a file extension. Calls `cleanurl` on the result.
- `to_json`: converts the given data structure to JSON. Note that this should not normally be used as a string filter (i.e. `${ value | to_json }`) but directly as a function, like this: `${ to_json(value) }`.
- `fingerprint`: Replace an unadorned path to an assets file with its fingerprinted (i.e. versioned) equivalent. Example: `${ 'js/site.js' | url, fingerprint }`. Uses the corresponding entry from the `ASSETS_MAP` context variable if it is available but otherwise proceeds to do the fingerprinting itself.

If you wish to provide additional filters in Mako without having to explicitly define or import them in templates, the best way of doing this his to add them via the `mako_imports` setting in `wmk_config.yaml` (see above). There is currently no easy way to do this if Jinja2 templates are being used, however.

Please note that in order to avoid conflicts with the above filters you should not place a file named `wmk_mako_filters.py` or `wmk_jinja2_extras.py` in your `py/` directories.

Working with lists of pages

Templates which render a list of content files (e.g. a list of blog posts or pages belonging to a category) will need to filter or sort `MDCONTENT` accordingly. In order to make this easier, `MDCONTENT` is wrapped in a list-like object called `MDContentList`, which has the following methods:

General searching/filtering

Each of the following methods returns a new `MDContentList` containing those entries for which the predicate (`pred`) is True.

- `match_entry(self, pred)`: The `pred` (i.e. predicate) is a callable which receives the full information on each entry in the `MDContentList` and returns True or False.

- `match_ctx(self, pred)`: The `pred` receives the context for each entry and returns a boolean.
- `match_page(self, pred)`: The `pred` receives the `page` object for each entry and returns a boolean.
- `match_doc(self, pred)`: The `pred` receives the markdown body for each entry and returns a boolean.
- `url_match(self, url_pred)`: The `pred` receives the `url` (relative to `htdocs`) for each entry and returns a boolean.
- `path_match(self, src_pred)`: The `pred` receives the path to the source document for each entry and returns a boolean.

Specialized searching/filtering

All of these return a new `MDContentList` object (at least by default).

- `posts(self, ordered=True)`: Returns a new `MDContentList` with those entries which are blog posts. In practice this means those with markdown sources in the `posts/` or `blog/` subdirectories or those which have a `page.type` of "post", "blog", "blog-entry" or "blog_entry". Normally ordered by date (newest first), but this can be turned off by setting `ordered` to False.
- `not_posts(self)`: Returns a new `MDContentList` with "pages", i.e. those entries which are *not* blog posts.
- `has_slug(self, sluglist)`, `has_id(self, idlist)`: Entries with specific slugs/ids.
- `in_date_range(self, start, end, date_key='DATE')`: Posts/pages with a date between `start` and `end`. The key for the date field can be specifed using `date_key`. Unless the value for `date_key` is either `DATE` or `MTIME`, then the key is looked for in the `page` variables for the entry.
- `has_taxonomy(self, haystack_keys, needles)`: A general search for entries belonging to a taxonomy group, such as category, tag, section or type. They `haystack_keys` are the `page` variables to examine while `needles` is a list of the values to look for in the values of those variables. A string value for `needles` is treated as a one-item list. The search is case-insensitive.
- `in_category(self, catlist)`: A shortcut method for `self.has_taxonomy(['category', 'categories'], catlist)`
- `has_tag(self, taglist)`: A shortcut method for `self.has_taxonomy(['tag', 'tags'], taglist)`.
- `in_section(self, sectionlist)`: A shortcut method for `self.has_taxonomy(['section', 'sections'], sectionlist)`.
- `get_used_taxonomies(self)`: Get a list of all known taxonomies that are actually used by items in this MDContentList (i.e. content files). These may be of two types: (1) the standard taxonomies tags, sections, categories and authors; and (2) anything defined as a `TAXONOMY` in the frontmatter of a page. Returns a list of dicts with the keys `taxon`, `name`, `name_singular` and `name_plural`. If the taxonomy belongs to the latter group, then `order`, `list_url`, `item_url_pattern` and `page_id` will be present as well, and `name_singular`/`name_plural` may be empty. If a standard taxonomy (e.g. tags) has been handled as a content page `TAXONOMY`, then the latter type takes precedence (i.e. the standard one is omitted from the list).

- `group_by(self, pred, normalize=None, keep_empty=False)`: Group items in an MDContentList using a given criterion. Parameters: `pred` is a callable receiving a content item and returning a string or a list of strings. For convenience, `pred` may also be specified as a string and is then interpreted as the value of the named `page` variable, e.g. `category`; `normalize` is an optional callable that transforms the grouping values, e.g. by truncating and lowercasing them; `keep_empty` should be set to True when the content items whose predicate evaluates to the empty string are to be included in the result, since they otherwise will be omitted. Returns a dict whose keys are strings and whose values are `MDContentList` instances.
- `taxonomy_info(self, keys, order='count', tostring=None)`: Returns a list of dicts, where each dict corresponds to the slugified value of any of the keys in `keys`. The keys in the dict are `name`, `slug`, `forms` (different forms of `name` that appear in the result, e.g. upper/lowercase), `count`, and `items` (an MDContentList object). `tostring`, if present, is a callable that changes non-string and non-list values into strings for the purposes of grouping. Shorthand forms for common taxonomy types are available, namely `get_categories(self, order='name')`, `get_tags(self, order='name')`, `get_sections(self, order='name')`, and `get_authors(self, order='name', tostring=None)`. These look for both singular and plural forms of the given keys, e.g. `['tag', 'tags']` for `get_tags()`.
- `page_match(self, match_expr, ordering=None, limit=None)`: This is actually quite a general matching method but does not require the caller to pass a predicate callable to it, which means that it can be employed in more varied contexts than the general methods described in the last section. A `match_expr` contains the filtering specification. It will be described further below. The `ordering` parameter, if specified, should be either `title`, `slug`, `url` or `date`, with an optional `-` in front to indicate reverse ordering. The `date` option for `ordering` may be followed by the preferred frontmatter date field after a colon, e.g. `ordering='-date:modified_date'` for a list with the most recently changed files at the top. The `limit`, if specified, obviously indicates the maximum number of pages to return.
- `page_match_sql()`, `get_db()`, `get_db_columns()` – see "Searching/filtering using SQL" below.

A `match_expr` for `page_match()` is either a dict or a list of dicts. If it is a dict, each page in the result set must match each of the attributes specified in it. If it is a list of dicts, each page in the result set must match at least one of the dicts (i.e., the returned result set contains the union of all matches from all dicts in the list). When a string or regular expression match is being performed in this process, it will be case-insensitive. The supported attributes (i.e. dict keys) are as follows:

- `title`: A regular expression which will be applied to the page title.
- `slug`: A regular expression which will be applied to the slug.
- `id`: A string or list of strings (one of) which must match the page id exactly.
- `url`: A regular expression which will be applied to the target URL.
- `path`: A regular expression which will be applied to the path to the markdown source file (i.e. the `source_file_short` field).
- `doc`: A regular expression which will be applied to the body of the markdown source document.

- `date_range`: A list containing two ISO-formatted dates and optionally a date key (`DATE` by default) - see the description of `in_date_range()` above.
- `has_attrs`: A list of frontmatter variable names. Matching pages must have a non-empty value for each of them.
- `attrs`: A dict where each key is the name of a frontmatter variable and the value is the value of that attribute. If the value is a string, it will be matched case-insensitively. All key-value pairs must match.
- `has_tag`, `in_section`, `in_category`: The values are lists of tags, sections or categories, respectively, at least one of which must match (case-insensitively). See the methods with these names above.
- `is_post`: If set to True, will match if the page is a blog post; if set to False will match if the page is not a blog post.
- `exclude_url`: The page with this URL should be omitted from the results (normally the calling page).

Searching/filtering using SQL

An `MDContentList` has three methods for examining the content using an SQLite in-memory database:

- `get_db(self)`: Builds a SQLite database containing a single table, `content`, whose structure is described below. Returns a connection to this database which can then be worked with using normal sqlite3/DBAPI methods. The database has a locale-sensitive collation called `locale` (which applies `locale.strxfrm`) and a custom function `casefold` (which simply applies the Python `casefold` string method). The row factory is `sqlite3.Row`, so row fields can be read using either column names or integer indices.
- `get_db_columns(self)`: Returns a simple list of the columns in the `content` table.
- `page_match_sql(self, where_clause=None, bind=None, order_by=None, limit=None, offset=None, raw_sql=None, raw_result=False, first=False)`: Either `where_clause` or `raw_sql` must be specified. In either case, if `bind` is specified, the bind variables there will be applied to the SQL upon execution. If `order_by` (a string), `limit` or `offset` (integers) are specified, they will be appended to the SQL before executing it against the database connection. The result will be a `MDContentList` unless `raw_result` is True, in which case it is a cursor object. (If `raw_result` is False but `raw_sql` is supplied, the column list in the SQL select statement must include `source_file` so as to permit the construction of an appropriate `MDContentList`). If `first` is True, only the first item from the results is returned (or None, if the results are empty).

The `content` table constructed by `get_db()` always contains the columns `source_file`, `source_file_short`, `url target`, `template`, `MTIME`, `DATE`, `doc`, and `rendered`. In addition, it contains each `page` metadata field that appears in any of the entries in the `MDContentList` in question. These will be added as columns with the `page_` prefix; for instance, the `title` field will become `page_title`.

It should be noted that all `page` fields added to the table will have to match the regular expression `^[a-z]\w*$`. Thus, any metadata field with a key that is all uppercase, titlecased, or contains non-word characters (such as hyphens) will be omitted. Also, field names are case-sensitive in the raw metadata, but case-insensitive in the database table, so inconsistently capitalized field names may lead to unexpected results.

A field value that is not either string, integer, float, boolean, date, datetime, or None, will be serialized using `json.dumps()` with `ensure_ascii` set to False (for easier utf-8 matching). Dates and datetimes are stringified. Booleans will be represented as 1 or 0.

Sorting

All of these return a new `MDContentList` object with the entries in the specified order.

- `sorted_by(self, key, reverse=False, default_val=-1)`: A general sorting method. The `key` is the `page` variable to sort on, `default_val` is the value to assume if there is no such variable present in the entry, while `reverse` indicates whether the sort is to be descending (True) or ascending (False, the default).
- `sorted_by_date(self, newest_first=True, date_key='DATE')`: Sorting by date, newest first by default. The date key to sort on can be specified if desired.
- `sorted_by_title(self, reverse=False)`: Sorting by `page.title`, ascending by default.

Pagination

- `paginate(self, pagesize=5, context=None)`: Divides the `MDContentList` into chunks of size `pagesize` and returns a tuple consisting of the chunks and a list of `page_urls` (one for each page, in order). If an appropriate template context is provided, pages 2 and up will be written to the webroot output directory to destination files whose names are based upon the URL for the first page (and the page number, of course). Without the context, the `page_urls` will be None. It is the responsibility of the calling template to check the `_page` variable for the current page to be rendered (this defaults to 1). Each iteration will get all chunks and must use this variable to limit itself appropriately.

Typical usage of `paginate()`:

```
<%
  posts = MDCONTENT.posts()
  chunks, page_urls = posts.paginate(5, context)
  curpage = context.get('_page', 1)
%>

% for post in chunks[curpage-1]:
  ${ show_post(post) }
% endfor

% if len(chunks) > 1:
  ${ prevnext(len(chunks), curpage, page_urls) }
% endif
```

Render to an arbitrary file

- `def write_to(self, dest, context, extra_kwargs=None, template=None)`: Calls a template with the `MDContentList` in `self` as the value of `CHUNK` and write the result to the file named in `dest`. The file is of course relative to the webroot. Any directories are created if necessary. The `template` is by default the calling template while `extra_kwargs` may be added if desired.

Typical usage of `write_to()`:

```
<%
  if not CHUNK:
     for tag in tags:
         tagged = MDCONTENT.has_tag([tag])
         if not tagged:
             continue  # avoid potential infinite loop!
         outpath = '/tags/' + slugify(tag) + '/index.html'
         tagged.write_to(outpath, context, {'TAG': tag})
%>

% if CHUNK:
  ${ list_tagged_pages(TAG, CHUNK) }
% else:
  ${ list_tags() }
% endif
```

Site search

Using Lunr

Lunr is the only search solution "natively" supported by `wmk`. That being said, implementing site search is not a simple matter of turning lunr indexing on. It takes a bit of work by the author of the site or theme templates, so depending on your needs it may even be easier to base your search functionality on another solution.

With `lunr_index` (and optionally `lunr_index_fields`) in `wmk_config.yaml`, wmk will build a search index for Lunr.js and place it in `idx.json` in the webroot. In order to minimize its size, no metadata about each record is saved to the index. Instead, a simple list of pages (with title and summary) is placed in `idx.summaries.json`. The summary is taken either from one of the frontmatter fields `summary`, `intro` or `description` (in order of preference) or, failing that, from the start of the page body.

If `lunr_languages` is present in `wmk_config.yaml`, stemming rules for those languages will be applied when building the index. The value may be a two-letter lowercase country code (ISO-639-1) or a list of such codes. The currently accepted languages are `de`, `da`, `en`, `fi`, `fr`, `hu`, `it`, `nl`, `no`, `pt`, `ro`, and `ru` (this is the intersection of the languages supported by `lunr.js` and NLTK, respectively). The default language is `en`. Attempting to specify a non-supported language will raise an exception.

The index is built via the `lunr.py` module and the stemming support is provided by the Python Natural Language Toolkit.

For information about the supported syntax of the search expression, see the Lunr documentation.

Limitations of Lunr

- Building the index does not mean that the search functionality is complete. It remains to point to `lunr.js` in the templates and write some javascript to interface with it and display the results. However, since every website is different, this cannot be provided by wmk directly. It is up to the template (or theme) author to actually load the index and present a search interface to the user.

- Similarly, if a "fancy" preview of results is required which cannot be fulfilled using the information in `idx.summaries.json`, this must currently be solved independently by the template/theme author.
- Note that only the raw content document is indexed, not the HTML after the markdown (or other input content) has been processed. The only exception to this is that the binary input formats (DOCX, ODT, EPUB) are converted to markdown before being indexed. The output of templates (including even text resulting from shortcodes called from the content documents) is not indexed either.
- Because Lunr creates a single index file for the whole site, it may not be a practical option for large sites with lots of content – a realistic limit may be somewhere around 1,000 pages or so. Some other client-side search solutions break the index into smaller chunks and may therefore be a viable option for such sites.

Overview of alternative solutions

If you are looking for an alternative to lunr, the first thing to consider is whether a server-based solution is needed or whether a Javascript-based client-side solution would be enough.

If the site has a lot of text (more than 200,000 words or so) or if it needs to work even without Javascript, then a server-based solution is required. You then need to decide whether you want to self-host it or if you are ready to pay for a third-party hosted solution. Meilisearch is open source and allows for self-hosting (although a hosted solution called Meilisearch Cloud is also available), while the market leader in hosted site search is probably Algolia.

If, however, a client-side Javascript solution is sufficient, there are several alternatives to lunr that could come into consideration, e.g. Pagefind, Tinysearch, Elasticlunr or Stork.

Whichever solution is picked, you of course need to add the required HTML, CSS and Javascript to the templates for the search functionality to work. You also need to take care of updating the search index whenever the site is built.

Assuming you have opted not to use the built-in lunr support, the index creation/updating step can basically be implemented in two ways:

1. By running after the build step has finished via a `cleanup_commands` entry in `wmk_config.yaml`. This calls a script or another external program which can update the index based on either the HTML in the output folder or the JSON file specified using the `mdcontent_json` configuration option.
2. By implementing a hook function in `wmk_hooks.py` (or `wmk_theme_hooks.py`), most likely for `post_build_actions()` or `index_content()`; see the "Overriding and extending wmk via hooks" section below.

Example: Pagefind

Taking Pagefind as an example of the steps described above, you would, per their documentation, add something similar to this to your templates in an appropriate location:

```
<link href="/pagefind/pagefind-ui.css" rel="stylesheet">
<script src="/pagefind/pagefind-ui.js"></script>
<div id="search"></div>
<script>
    window.addEventListener('DOMContentLoaded', (event) => {
        new PagefindUI({ element: "#search", showSubResults: true });
    });
</script>
```

It would also be a good idea to make sure you modify all base templates so as to identify the main part of each page with the `data-pagefind-body` attribute and thus omit repeated elements such as navigation and footer from the index.

Finally, in order to actually create or update the search index whenever the site is built, you would need to add the following to the `wmk_config.yaml` file:

```
cleanup_commands:
  - "npx -y pagefind --site htdocs"
```

This obviously assumes that you have npm installed on your system.

Overriding and extending wmk via hooks

Much of the functionality of `wmk` can be changed by overriding or extending specific steps it performs. This is done by adding Python code to a file named `wmk_hooks.py` in the project `py/` directory. Themes can do the same thing via the `wmk_theme_hooks.py` file in the theme's `py/` directory. If both try to affect the same functionality, the project directory takes precedence.

Currently, the following defs from `wmk.py` can be extended by running hooks before or after them, or can be redefined entirely:

- `auto_nav_from_content`
- `binary_to_markdown`
- `build_lunr_index`
- `copy_static_files`
- `doc_with_yaml`
- `fingerprint_assets`
- `generate_summary`
- `get_assets_map`
- `get_content_extensions`
- `get_content`
- `get_extra_content`
- `get_index_yaml_data`
- `get_nav`
- `get_template_lookup`
- `get_template_vars`
- `get_templates`
- `handle_redirects`
- `handle_shortcode`
- `handle_taxonomy`

- `index_content`
- `locale_and_translation`
- `lunr_summary`
- `markdown_extensions_settings`
- `maybe_extra_meta`
- `maybe_save_mdcontent_as_json`
- `pandoc_extra_formats`
- `pandoc_metadata`
- `parse_dates`
- `post_build_actions`
- `postprocess_html`
- `preferred_date`
- `process_assets`
- `process_content_item`
- `process_markdown_content`
- `process_templates`
- `render_markdown`
- `run_init_commands`
- `run_cleanup_commands`
- `write_redir_file`

In order to override any of these entirely, define a function of the same name in the hooks file. One may also define a function that runs before or after:

- A function that runs before any of the above has the same name but with `__before` appended, e.g. `index_content__before`. It receives the arguments passed to the original function and can modify them and return new arguments in the form of either a two-tuple of a list and a dict (for `*args` and `**kwargs`) or a single dict (for `**kwargs` only). In either case, these will be passed to the affected function instead of the original arguments. If the before hook function returns nothing, the original arguments will be passed on unchanged.
- A function that runs after any of the above has the same name but with `__after` appended, e.g. `index_content__after`. It receives the return value of the original function and can return a new value that will be returned to the caller instead. (If it returns nothing, the original return value will be returned unchanged).

You should examine wmk's source code to make sure that any replacement function you may write is compatible with the original in terms of its parameters and possible return values. Updates to `wmk` may of course make it necessary to change your hook functions.

Examples

Here is a generic `get_extra_content()` def which adds HTML pages fetched from a database to the "normal" content from the `content/` directory:

```
def get_extra_content(content, ctdir, datadir, outputdir, template_vars, conf):
    known_ids = set([_['data']['page']['id'] for _ in content])
    content_extensions = { '.html': {'raw': True}, }
    extpat = re.compile(r'\.html$')
    result = _get_articles_from_database()
```

```
    for i, row in enumerate(result):
        meta, doc, pseudo = _munge_row(row, i, result, ctdir)
        wmk.process_content_item(
            meta, doc, content, conf, template_vars,
            ctdir, outputdir, datadir, content_extensions, known_ids,
            pseudo['root'], pseudo['fn'],
            pseudo['source_file'], pseudo['source_file_short'],
            extpat, False)
```

The functions `_get_articles_from_database()` and `_munge_row()` are left as an exercise for the reader.

Here is an `__after` hook for `maybe_extra_meta()` which fetches a conference schedule (e.g. from from an online calendar) if the `conference_id` key is present in the frontmatter. The retrieved information will then be available to the templates for that page as `page.schedule`.

```
def maybe_extra_meta__after(meta):
    if 'conference_id' in meta:
        meta['schedule'] = _get_conference_schedule(meta['conference_id'])
    return meta
```

A third example: Let's say you want to show information from a few RSS sources in a sidebar that will appear on several pages. In order to avoid refetching it for each page you can use something like this:

```
def get_template_vars__after(template_vars):
    if 'rss_sources' in template_vars:
        template_vars['rss_info'] = fetch_rss_feeds(template_vars['rss_sources'])
    return template_vars
```

This assumes that you set `rss_sources` in the `template_context` section of your `wmk_config.yaml` file.

Incorporating external sources

A wmk-maintained website may incorporate material that does not originate as content files in the site's `content/` directory. The source of the material may be a database or an external API, perhaps provided by a headless CMS system such as Sanity, Directus, or DatoCMS.

In either case, there are two main approaches as to how to integrate such content into a wmk site. The first is to use the hooks system described earlier, especially `get_extra_content()`. The second is to fetch the material independently of wmk (or perhaps from the `init_commands` that can be specified in the configuration file) and write it as a set of html or markdown files into `content/`, whereupon wmk can treat it as normal file-based content.

Example: Import from WordPress

As an example of the latter approach, a set of scripts is available in the `extras/` subdirectory to fetch and maintain content from a WordPress site.

The script `wordpress2content.py` uses the WordPress REST API to get posts and pages from a WordPress site and export them as content files in `content/`. Images and other media files from the origin's `wp-content/uploads/` folder go into `static/_fetched/`.

This may either be used to migrate from WordPress to a static site maintained by wmk, or to use a (possibly non-public) WordPress installation as a headless CMS for external authors or non-technical users.

When used in the latter way, the helper scripts `duplicate_wp_content.py` and `removed_wp_content.py` may help with the housekeeping involved in keeping the content properly synchronized.

For further details, see the readme in the `extras/` directory.