mdBook Documentation

Mathieu David Michael-F-Bryan

August 13, 2019

Contents

1 mdBook

mdBook is a command line tool and Rust crate to create books using Markdown files. It's very similar to Gitbook but written in Rust.

What you are reading serves as an example of the output of mdBook and at the same time as a high-level documentation.

mdBook is free and open source, you can find the source code on GitHub. Issues and feature requests can be posted on the GitHub issue tracker.

1.1 API docs

Alongside this book you can also read the API docs generated by Rustdoc if you would like to use mdBook as a crate or write a new renderer and need a more low-level overview.

1.2 License

2 Command Line Tool

mdBook can be used either as a command line tool or a Rust crate. Let's focus on the command line tool capabilities first.

2.1 Install From Binaries

Precompiled binaries are provided for major platforms on a best-effort basis. Visit the releases page to download the appropriate version for your platform.

2.2 Install From Source

mdBook can also be installed from source

2.2.1 Pre-requisite

mdBook is written in **Rust** and therefore needs to be compiled with **Cargo**. If you haven't already installed Rust, please go ahead and install it now.

2.2.2 Install Crates.io version

Installing mdBook is relatively easy if you already have Rust and Cargo installed. You just have to type this snippet in your terminal:

```
cargo install mdbook
```

This will fetch the source code for the latest release from Crates.io and compile it. You will have to add Cargo's `bin` directory to your `PATH`.

Run `mdbook help` in your terminal to verify if it works. Congratulations, you have installed mdBook!

2.2.3 Install Git version

The **git version** contains all the latest bug-fixes and features, that will be released in the next version on **Crates.io**, if you can't wait until the next release. You can build the git version yourself. Open your terminal and navigate to the directory of you choice. We need to clone the git repository and then build it with Cargo.

```
git clone --depth=1 https://github.com/rust-lang-nursery/mdBook.git
cd mdBook
cargo build --release
```

The executable `mdbook` will be in the `./target/release` folder, this should be added to the path.

3 The init command

There is some minimal boilerplate that is the same for every new book. It's for this purpose that mdBook includes an `init` command.

The `init` command is used like this:

```
mdbook init
```

When using the `init` command for the first time, a couple of files will be set up for you:

```
book-test/
├── book
└── src
    ├── chapter_1.md
    └── SUMMARY.md
```

- The `src` directory is were you write your book in markdown. It contains all the source files, configuration files, etc.

- The `book` directory is where your book is rendered. All the output is ready to be uploaded to a server to be seen by your audience.

- The `SUMMARY.md` file is the most important file, it's the skeleton of your book and is discussed in more detail in another chapter

Tip: Generate chapters from SUMMARY.md When a `SUMMARY.md` file already exists, the `init` command will first parse it and generate the missing files according to the paths used in the `SUMMARY.md`. This allows you to think and create the whole structure of your book and then let mdBook generate it for you.

Specify a directory The `init` command can take a directory as an argument to use as the book's root instead of the current working directory.

```
mdbook init path/to/book
```

–theme When you use the `--theme` flag, the default theme will be copied into a directory called `theme` in your source directory so that you can modify it.

The theme is selectively overwritten, this means that if you don't want to overwrite a specific file, just delete it and the default file will be used.

4 The build command

The build command is used to render your book:

```
mdbook build
```

It will try to parse your `SUMMARY.md` file to understand the structure of your book and fetch the corresponding files.

The rendered output will maintain the same directory structure as the source for convenience. Large books will therefore remain structured when rendered.

Specify a directory The `build` command can take a directory as an argument to use as the book's root instead of the current working directory.

```
mdbook build path/to/book
```

–open When you use the `--open` (`-o`) flag, mdbook will open the rendered book in your default web browser after building it.

–dest-dir The `--dest-dir` (`-d`) option allows you to change the output directory for the book. Relative paths are interpreted relative to the book's root directory. If not specified it will default to the value of the `build.build-dir` key in `book.toml`, or to `./book`.

Note: *Make sure to run the build command in the root directory and not in the source directory*

5 The watch command

The `watch` command is useful when you want your book to be rendered on every file change. You could repeatedly issue `mdbook build` every time a file is changed. But using `mdbook watch` once will watch your files and will trigger a build automatically whenever you modify a file.

Specify a directory The `watch` command can take a directory as an argument to use as the book's root instead of the current working directory.

```
mdbook watch path/to/book
```

–open When you use the `--open` (`-o`) option, mdbook will open the rendered book in your default web browser.

–dest-dir The `--dest-dir` (`-d`) option allows you to change the output directory for the book. Relative paths are interpreted relative to the book's root directory. If not specified it will default to the value of the `build.build-dir` key in `book.toml`, or to `./book`.

6 The serve command

The serve command is used to preview a book by serving it over HTTP at `localhost:3000` by default. Additionally it watches the book's directory for changes, rebuilding the book and refreshing clients for each change. A websocket connection is used to trigger the client-side refresh.

Note: *The `serve` command is for testing a book's HTML output, and is not intended to be a complete HTTP server for a website.*

Specify a directory The `serve` command can take a directory as an argument to use as the book's root instead of the current working directory.

```
mdbook serve path/to/book
```

Server options `serve` has four options: the HTTP port, the WebSocket port, the HTTP hostname to listen on, and the hostname for the browser to connect to for WebSockets.

For example: suppose you have an nginx server for SSL termination which has a public address of 192.168.1.100 on port 80 and proxied that to 127.0.0.1 on port 8000. To run use the nginx proxy do:

```
mdbook serve path/to/book -p 8000 -n 127.0.0.1 --websocket-hostname 192.168.1.100
```

If you were to want live reloading for this you would need to proxy the websocket calls through nginx as well from `192.168.1.100:<WS_PORT>` to `127.0.0.1:<WS_PORT>`. The `-w` flag allows for the websocket port to be configured.

–open When you use the `--open` (`-o`) flag, mdbook will open the book in your default web browser after starting the server.

–dest-dir The `--dest-dir` (`-d`) option allows you to change the output directory for the book. Relative paths are interpreted relative to the book's root directory. If not specified it will default to the value of the `build.build-dir` key in `book.toml`, or to `./book`.

7 The test command

When writing a book, you sometimes need to automate some tests. For example, The Rust Programming Book uses a lot of code examples that could get outdated. Therefore it is very important for them to be able to automatically test these code examples.

mdBook supports a `test` command that will run all available tests in a book. At the moment, only rustdoc tests are supported, but this may be expanded upon in the future.

Disable tests on a code block rustdoc doesn't test code blocks which contain the `ignore` attribute:

```
```rust,ignore
fn main() {}
```
```

rustdoc also doesn't test code blocks which specify a language other than Rust:

```
```markdown
**Foo**: _bar_
```
```

rustdoc *does* test code blocks which have no language specified:

```
```
This is going to cause an error!
```
```

Specify a directory The `test` command can take a directory as an argument to use as the book's root instead of the current working directory.

```
mdbook test path/to/book
```

–library-path The `--library-path` (`-L`) option allows you to add directories to the library search path used by `rustdoc` when it builds and tests the examples. Multiple directories can be specified with multiple options (`-L foo -L bar`) or with a comma-delimited list (`-L foo,bar`).

–dest-dir The `--dest-dir` (`-d`) option allows you to change the output directory for the book. Relative paths are interpreted relative to the book's root directory. If not specified it will default to the value of the `build.build-dir` key in `book.toml`, or to `./book`.

8 The clean command

The clean command is used to delete the generated book and any other build artifacts.

```
mdbook clean
```

Specify a directory The `clean` command can take a directory as an argument to use as the book's root instead of the current working directory.

```
mdbook clean path/to/book
```

–dest-dir The `--dest-dir` (`-d`) option allows you to override the book's output directory, which will be deleted by this command. Relative paths are interpreted relative to the book's root directory. If not specified it will default to the value of the `build.build-dir` key in `book.toml`, or to `./book`.

```
mdbook clean --dest-dir=path/to/book
```

`path/to/book` could be absolute or relative.

9 Format

In this section you will learn how to:

- Structure your book correctly
- Format your `SUMMARY.md` file
- Configure your book using `book.toml`
- Customize your theme

10 SUMMARY.md

The summary file is used by mdBook to know what chapters to include, in what order they should appear, what their hierarchy is and where the source files are. Without this file, there is no book.

Even though `SUMMARY.md` is a markdown file, the formatting is very strict to allow for easy parsing. Let's see how you should format your `SUMMARY.md` file.

Allowed elements

1. ***Title*** It's common practice to begin with a title, generally <code ">#
   Summary. But it is not mandatory, the parser just ignores it. So you can too if you feel like it.

2. ***Prefix Chapter*** Before the main numbered chapters you can add a couple of elements that will not be numbered. This is useful for forewords, introductions, etc. There are however some constraints. You can not nest prefix chapters, they should all be on the root level. And you can not add prefix chapters once you have added numbered chapters.

   ```
   [Title of prefix element](relative/path/to/markdown.md)
   ```

3. ***Numbered Chapter*** Numbered chapters are the main content of the book, they will be numbered and can be nested, resulting in a nice hierarchy (chapters, sub-chapters, etc.)

```
- [Title of the Chapter](relative/path/to/markdown.md)
```

You can either use - or * to indicate a numbered chapter.

4. ***Suffix Chapter*** After the numbered chapters you can add a couple of non-numbered chapters. They are the same as prefix chapters but come after the numbered chapters instead of before.

All other elements are unsupported and will be ignored at best or result in an error.

11 Configuration

You can configure the parameters for your book in the ***book.toml*** file.

Here is an example of what a ***book.toml*** file might look like:

```toml
[book]
title = "Example book"
author = "John Doe"
description = "The example book covers examples."

[build]
build-dir = "my-example-book"
create-missing = false

[preprocessor.index]

[preprocessor.links]

[output.html]
additional-css = ["custom.css"]

[output.html.search]
limit-results = 15
```

11.1 Supported configuration options

It is important to note that **any** relative path specified in the configuration will always be taken relative from the root of the book where the configuration file is located.

11.1.1 General metadata

This is general information about your book.

- **title:** The title of the book
- **authors:** The author(s) of the book
- **description:** A description for the book, which is added as meta information in the html `<head>` of each page
- **src:** By default, the source directory is found in the directory named `src` directly under the root folder. But this is configurable with the `src` key in the configuration file.
- **language:** The main language of the book, which is used as a language attribute `<html lang="en">` for example.

book.toml

```
[book]
title = "Example book"
authors = ["John Doe", "Jane Doe"]
description = "The example book covers examples."
src = "my-src"  # the source files will be found in `root/my-src` instead of `root/src`
language = "en"
```

11.1.2 Build options

This controls the build process of your book.

- **build-dir:** The directory to put the rendered book in. By default this is `book/` in the book's root directory.

- **create-missing:** By default, any missing files specified in `SUMMARY.md` will be created when the book is built (i.e. `create-missing = true`). If this is `false` then the build process will instead exit with an error if any files do not exist.

- **use-default-preprocessors:** Disable the default preprocessors of (`links` & `index`) by setting this option to `false`.

  If you have the same, and/or other preprocessors declared via their table of configuration, they will run instead.

- For clarity, with no preprocessor configuration, the default `links` and `index` will run.
- Setting `use-default-preprocessors = false` will disable these default preprocessors from running.
- Adding `[preprocessor.links]`, for example, will ensure, regardless of `use-default-preprocessors` that `links` it will run.

11.2 Configuring Preprocessors

The following preprocessors are available and included by default:

- `links`: Expand the `{{ #playpen }}` and `{{ #include }}` handlebars helpers in a chapter to include the contents of a file.
- `index`: Convert all chapter files named `README.md` into `index.md`. That is to say, all `README.md` would be rendered to an index file `index.html` in the rendered book.

book.toml

```
[build]
build-dir = "build"
create-missing = false

[preprocessor.links]

[preprocessor.index]
```

11.2.1 Custom Preprocessor Configuration

Like renderers, preprocessor will need to be given its own table (e.g. `[preprocessor.mathjax]`). In the section, you may then pass extra configuration to the preprocessor by adding key-value pairs to the table.

For example

```
[preprocessor.links]
# set the renderers this preprocessor will run for
renderers = ["html"]
some_extra_feature = true
```

Locking a Preprocessor dependency to a renderer You can explicitly specify that a preprocessor should run for a renderer by binding the two together.

```
[preprocessor.mathjax]
renderers = ["html"]  # mathjax only makes sense with the HTML renderer
```

11.2.2 Provide Your Own Command

By default when you add a `[preprocessor.foo]` table to your `book.toml` file, `mdbook` will try to invoke the `mdbook-foo` executable. If you want to use a different program name or pass in command-line arguments, this behaviour can be overridden by adding a `command` field.

```
[preprocessor.random]
command = "python random.py"
```

11.3 Configuring Renderers

11.3.1 HTML renderer options

The HTML renderer has a couple of options as well. All the options for the renderer need to be specified under the TOML table `[output.html]`.

The following configuration options are available:

- **theme:** mdBook comes with a default theme and all the resource files needed for it. But if this option is set, mdBook will selectively overwrite the theme files with the ones found in the specified folder.
- **default-theme:** The theme color scheme to select by default in the 'Change Theme' dropdown. Defaults to `light`.
- **curly-quotes:** Convert straight quotes to curly quotes, except for those that occur in code blocks and code spans. Defaults to `false`.
- **mathjax-support:** Adds support for MathJax. Defaults to `false`.
- **google-analytics:** If you use Google Analytics, this option lets you enable it by simply specifying your ID in the configuration file.

- **additional-css:** If you need to slightly change the appearance of your book without overwriting the whole style, you can specify a set of stylesheets that will be loaded after the default ones where you can surgically change the style.
- **additional-js:** If you need to add some behaviour to your book without removing the current behaviour, you can specify a set of JavaScript files that will be loaded alongside the default one.
- **no-section-label:** mdBook by defaults adds section label in table of contents column. For example, "1.", "2.1". Set this option to true to disable those labels. Defaults to `false`.
- **playpen:** A subtable for configuring various playpen settings.
- **search:** A subtable for configuring the in-browser search functionality. mdBook must be compiled with the `search` feature enabled (on by default).
- **git-repository-url:** A url to the git repository for the book. If provided an icon link will be output in the menu bar of the book.
- **git-repository-icon:** The FontAwesome icon class to use for the git repository link. Defaults to `fa-github`.

Available configuration options for the `[output.html.playpen]` table:

- **editable:** Allow editing the source code. Defaults to `false`.
- **copy-js:** Copy JavaScript files for the editor to the output directory. Defaults to `true`.

Available configuration options for the `[output.html.search]` table:

- **enable:** Enables the search feature. Defaults to `true`.
- **limit-results:** The maximum number of search results. Defaults to `30`.
- **teaser-word-count:** The number of words used for a search result teaser. Defaults to `30`.
- **use-boolean-and:** Define the logical link between multiple search words. If true, all search words must appear in each result. Defaults to `true`.
- **boost-title:** Boost factor for the search result score if a search word appears in the header. Defaults to `2`.
- **boost-hierarchy:** Boost factor for the search result score if a search word appears in the hierarchy. The hierarchy contains all titles of the parent documents and all parent headings. Defaults to `1`.

- **boost-paragraph:** Boost factor for the search result score if a search word appears in the text. Defaults to `1`.
- **expand:** True if search should match longer results e.g. search `micro` should match `microwave`. Defaults to `true`.
- **heading-split-level:** Search results will link to a section of the document which contains the result. Documents are split into sections by headings this level or less. Defaults to `3`. (*`### This is a level 3 heading`*)
- **copy-js:** Copy JavaScript files for the search implementation to the output directory. Defaults to `true`.

This shows all available HTML output options in the **book.toml**:

```
[book]
title = "Example book"
authors = ["John Doe", "Jane Doe"]
description = "The example book covers examples."

[output.html]
theme = "my-theme"
default-theme = "light"
curly-quotes = true
mathjax-support = false
google-analytics = "123456"
additional-css = ["custom.css", "custom2.css"]
additional-js = ["custom.js"]
no-section-label = false
git-repository-url = "https://github.com/rust-lang-nursery/mdBook"
git-repository-icon = "fa-github"

[output.html.playpen]
editable = false
copy-js = true

[output.html.search]
enable = true
limit-results = 30
teaser-word-count = 30
use-boolean-and = true
boost-title = 2
boost-hierarchy = 1
boost-paragraph = 1
expand = true
heading-split-level = 3
copy-js = true
```

11.3.2 Custom Renderers

A custom renderer can be enabled by adding a `[output.foo]` table to your `book.toml`. Similar to preprocessors this will instruct `mdbook` to pass a representation

of the book to `mdbook-foo` for rendering.

Custom renderers will have access to all configuration within their table (i.e. anything under `[output.foo]`), and the command to be invoked can be manually specified with the `command` field.

11.4 Environment Variables

All configuration values can be overridden from the command line by setting the corresponding environment variable. Because many operating systems restrict environment variables to be alphanumeric characters or `_`, the configuration key needs to be formatted slightly differently to the normal `foo.bar.baz` form.

Variables starting with `MDBOOK_` are used for configuration. The key is created by removing the `MDBOOK_` prefix and turning the resulting string into `kebab-case`. Double underscores (`__`) separate nested keys, while a single underscore (`_`) is replaced with a dash (`-`).

For example:

- `MDBOOK_foo` -> `foo`
- `MDBOOK_FOO` -> `foo`
- `MDBOOK_FOO__BAR` -> `foo.bar`
- `MDBOOK_FOO_BAR` -> `foo-bar`
- `MDBOOK_FOO_bar__baz` -> `foo-bar.baz`

So by setting the `MDBOOK_BOOK__TITLE` environment variable you can override the book's title without needing to touch your `book.toml`.

Note: To facilitate setting more complex config items, the value of an environment variable is first parsed as JSON, falling back to a string if the parse fails.

This means, if you so desired, you could override all book metadata when building the book with something like

```
$ export MDBOOK_BOOK="{'title': 'My Awesome Book', authors: ['↪
    Michael-F-Bryan']}"
$ mdbook build
```

The latter case may be useful in situations where `mdbook` is invoked from a script or CI, where it sometimes isn't possible to update the `book.toml` before building.

12 Theme

The default renderer uses a handlebars template to render your markdown files and comes with a default theme included in the mdBook binary.

The theme is totally customizable, you can selectively replace every file from the theme by your own by adding a `theme` directory next to `src` folder in your project root. Create a new file with the name of the file you want to override and now that file will be used instead of the default file.

Here are the files you can override:

- ***index.hbs*** is the handlebars template.
- ***book.css*** is the style used in the output. If you want to change the design of your book, this is probably the file you want to modify. Sometimes in conjunction with `index.hbs` when you want to radically change the layout.
- ***book.js*** is mostly used to add client side functionality, like hiding / unhiding the sidebar, changing the theme, ...
- ***highlight.js*** is the JavaScript that is used to highlight code snippets, you should not need to modify this.
- ***highlight.css*** is the theme used for the code highlighting
- ***favicon.png*** the favicon that will be used

Generally, when you want to tweak the theme, you don't need to override all the files. If you only need changes in the stylesheet, there is no point in overriding all the other files. Because custom files take precedence over built-in ones, they will not get updated with new fixes / features.

Note: When you override a file, it is possible that you break some functionality. Therefore I recommend to use the file from the default theme as template and only add / modify what you need. You can copy the default theme into your source directory automatically by using `mdbook init --theme` just remove the files you don't want to override.

13 index.hbs

`index.hbs` is the handlebars template that is used to render the book. The markdown files are processed to html and then injected in that template.

If you want to change the layout or style of your book, chances are that you will have to modify this template a little bit. Here is what you need to know.

13.1 Data

A lot of data is exposed to the handlebars template with the "context". In the handlebars template you can access this information by using

```
{{name_of_property}}
```

Here is a list of the properties that are exposed:

- ***language*** Language of the book in the form `en`, as specified in `book.toml` (if not specified, defaults to `en`). To use in <code "><html lang=" language "> for example.
- ***title*** Title of the book, as specified in `book.toml`
- ***chapter_title*** Title of the current chapter, as listed in `SUMMARY.md`
- ***path*** Relative path to the original markdown file from the source directory
- ***content*** This is the rendered markdown.
- ***path_to_root*** This is a path containing exclusively `../`'s that points to the root of the book from the current file. Since the original directory structure is maintained, it is useful to prepend relative links with this `path_to_root`.
- ***chapters*** Is an array of dictionaries of the form

```
{"section": "1.2.1", "name": "name of this chapter", "path":↪
    "dir/markdown.md"}
```

containing all the chapters of the book. It is used for example to construct the table of contents (sidebar).

13.2 Handlebars Helpers

In addition to the properties you can access, there are some handlebars helpers at your disposal.

13.2.1 1. toc

The toc helper is used like this

```
{{#toc}}{{/toc}}
```

and outputs something that looks like this, depending on the structure of your book

```
<ul class="chapter">
    <li><a href="link/to/file.html">Some chapter</a></li>
    <li>
        <ul class="section">
            <li><a href="link/to/other_file.html">Some other Chapter</a></li>
        </ul>
    </li>
</ul>
```

If you would like to make a toc with another structure, you have access to the chapters property containing all the data. The only limitation at the moment is that you would have to do it with JavaScript instead of with a handlebars helper.

```
<script>
var chapters = {{chapters}};
// Processing here
</script>
```

13.2.2 2. previous / next

The previous and next helpers expose a `link` and `name` property to the previous and next chapters.

They are used like this

```
{{#previous}}
    <a href="{{link}}" class="nav-chapters previous">
        <i class="fa fa-angle-left"></i>
    </a>
{{/previous}}
```

The inner html will only be rendered if the previous / next chapter exists. Of course the inner html can be changed to your liking.

If you would like other properties or helpers exposed, please create a new issue

14 Syntax Highlighting

For syntax highlighting I use Highlight.js with a custom theme.

Automatic language detection has been turned off, so you will probably want to specify the programming language you use like this

```
fn main() {
    // Some code
}
```
‘‘‘

‘

14.1 Custom theme

Like the rest of the theme, the files used for syntax highlighting can be overridden with your own.

- ***highlight.js*** normally you shouldn't have to overwrite this file, unless you want to use a more recent version.
- ***highlight.css*** theme used by highlight.js for syntax highlighting.

If you want to use another theme for `highlight.js` download it from their website, or make it yourself, rename it to `highlight.css` and put it in `src/theme` (or the equivalent if you changed your source folder)

Now your theme will be used instead of the default theme.

14.2 Hiding code lines

There is a feature in mdBook that lets you hide code lines by prepending them with a `#`.

```
# fn main() {
    let x = 5;
    let y = 6;

    println!("{}", x + y);
# }
```

Will render as

```
# fn main() {
    let x = 5;
    let y = 7;

    println!("{}", x + y);
# }
```

At the moment, this only works for code examples that are annotated with `rust`. Because it would collide with semantics of some programming languages. In the future, we want to make this configurable through the `book.toml` so that everyone can benefit from it.

14.3 Improve default theme

If you think the default theme doesn't look quite right for a specific language, or could be improved. Feel free to submit a new issue explaining what you have in mind and I will take a look at it.

You could also create a pull-request with the proposed improvements.

Overall the theme should be light and sober, without to many flashy colors.

15 Editor

In addition to providing runnable code playpens, mdBook optionally allows them to be editable. In order to enable editable code blocks, the following needs to be added to the ***book.toml***:

```
[output.html.playpen]
editable = true
```

To make a specific block available for editing, the attribute `editable` needs to be added to it:

````
```
fn main() {
    let number = 5;
    print!("{}", number);
}
```
````

The above will result in this editable playpen:

```
fn main() {
    let number = 5;
    print!("{}", number);
}
```

Note the new `Undo Changes` button in the editable playpens.

15.1 Customizing the Editor

By default, the editor is the Ace editor, but, if desired, the functionality may be overriden by providing a different folder:

```
[output.html.playpen]
editable = true
editor = "/path/to/editor"
```

Note that for the editor changes to function correctly, the `book.js` inside of the `theme` folder will need to be overriden as it has some couplings with the default Ace editor.

16 MathJax Support

mdBook has optional support for math equations through MathJax.

To enable MathJax, you need to add the `mathjax-support` key to your `book.toml` under the `output.html` section.

```
[output.html]
mathjax-support = true
```

Note: The usual delimiters MathJax uses are not yet supported. You can't currently use `$$ ... $$` as delimiters and the `\[ ... \]` delimiters need an extra backslash to work. Hopefully this limitation will be lifted soon.

Note: When you use double backslashes in MathJax blocks (for example in commands such as `\begin{cases} \frac 1 2 \\ \frac 3 4 \end{cases}`) you need to add *two extra* backslashes (e.g., `\begin{cases} \frac 1 2 \\\\ \frac 3 4 \end{cases}`).

16.0.1 Inline equations

Inline equations are delimited by `\\(` and `\\)`. So for example, to render the following inline equation $\int xdx = \frac{x^2}{2} + C$ you would write the following:

```
\\( \int x dx = \frac{x^2}{2} + C \\)
```

16.0.2 Block equations

Block equations are delimited by `\\[` and `\\]`. To render the following equation

$$\mu = \frac{1}{N}\sum_i = 0x_i$$

you would write:

```
\\[ \mu = \frac{1}{N} \sum_{i=0} x_i \\]
```

17 mdBook-specific markdown

17.1 Hiding code lines

There is a feature in mdBook that lets you hide code lines by prepending them with a `#`.

```
# fn main() {
    let x = 5;
    let y = 6;

    println!("{}", x + y);
# }
```

Will render as

```
# fn main() {
    let x = 5;
    let y = 7;

    println!("{}", x + y);
# }
```

17.2 Including files

With the following syntax, you can include files into your book:

```
{{#include file.rs}}
```

The path to the file has to be relative from the current source file.

mdBook will interpret included files as markdown. Since the include command is usually used for inserting code snippets and examples, you will often wrap the command with ‘‘‘ to display the file contents without interpretting them.

```
‘‘‘
{{#include file.rs}}
‘‘‘
```

17.3 Including portions of a file

Often you only need a specific part of the file e.g. relevant lines for an example. We support four different modes of partial includes:

```
{{#include file.rs:2}}
{{#include file.rs::10}}
{{#include file.rs:2:}}
{{#include file.rs:2:10}}
```

The first command only includes the second line from file `file.rs`. The second command includes all lines up to line 10, i.e. the lines from 11 till the end of the file are omitted. The third command includes all lines from line 2, i.e. the first line is omitted. The last command includes the excerpt of `file.rs` consisting of lines 2 to 10.

To avoid breaking your book when modifying included files, you can also include a specific section using anchors instead of line numbers. An anchor is a pair of matching lines. The line beginning an anchor must match the regex "ANCHOR:\s*[\w_-]+" and similarly the ending line must match the regex "ANCHOR_END:\s*[\w_-]+". This allows you to put anchors in any kind of commented line.

Consider the following file to include:

```
/* ANCHOR: all */

// ANCHOR: component
struct Paddle {
    hello: f32,
}
// ANCHOR_END: component

////////// ANCHOR: system
impl System for MySystem { ... }
////////// ANCHOR_END: system

/* ANCHOR_END: all */
```

Then in the book, all you have to do is:

```
Here is a component:
```rust,no_run,noplaypen
{{#include file.rs:component}}
```

Here is a system:
```

```
```rust,no_run,noplaypen
{{#include file.rs:system}}
```

This is the full file.
```rust,no_run,noplaypen
{{#include file.rs:all}}
```
```

Lines containing anchor patterns inside the included anchor are ignored.

17.4 Inserting runnable Rust files

With the following syntax, you can insert runnable Rust files into your book:

```
{{#playpen file.rs}}
```

The path to the Rust file has to be relative from the current source file.

When play is clicked, the code snippet will be sent to the Rust Playpen to be compiled and run. The result is sent back and displayed directly underneath the code.

Here is what a rendered code snippet looks like:

```
fn main() {
    println!("Hello World!");
#
#    // You can even hide lines! :D
#    println!("I am hidden! Expand the code snippet to see me");
}
```

18 Running `mdbook` in Continuous Integration

While the following examples use Travis CI, their principles should straightforwardly transfer to other continuous integration providers as well.

18.1 Ensuring Your Book Builds and Tests Pass

Here is a sample Travis CI `.travis.yml` configuration that ensures `mdbook build` and `mdbook test` run successfully. The key to fast CI turnaround times is caching

`mdbook` installs, so that you aren't compiling `mdbook` on every CI run.

```
language: rust
sudo: false

cache:
  - cargo

rust:
  - stable

before_script:
  - (test -x $HOME/.cargo/bin/cargo-install-update || cargo install cargo-update)
  - (test -x $HOME/.cargo/bin/mdbook || cargo install --vers "^0.3" mdbook)
  - cargo install-update -a

script:
  - mdbook build path/to/mybook && mdbook test path/to/mybook
```

18.2 Deploying Your Book to GitHub Pages

Following these instructions will result in your book being published to GitHub pages after a successful CI run on your repository's `master` branch.

First, create a new GitHub "Personal Access Token" with the "public_repo" permissions (or "repo" for private repositories). Go to your repository's Travis CI settings page and add an environment variable named `GITHUB_TOKEN` that is marked secure and *not* shown in the logs.

Then, append this snippet to your `.travis.yml` and update the path to the `book` directory:

```
deploy:
  provider: pages
  skip-cleanup: true
  github-token: $GITHUB_TOKEN
  local-dir: path/to/mybook/book
  keep-history: false
  on:
    branch: master
```

That's it!

18.2.1 Deploying to GitHub Pages manually

If your CI doesn't support GitHub pages, or you're deploying somewhere else with integrations such as Github Pages: *note: you may want to use different tmp dirs*:

```
$> git worktree add /tmp/book gh-pages
$> mdbook build
$> rm -rf /tmp/book/* # this won't delete the .git directory
$> cp -rp book/* /tmp/book/
$> cd /tmp/book
$> git add -A
$> git commit 'new book message'
$> git push origin gh-pages
$> cd -
```

Or put this into a Makefile rule:

```
.PHONY: deploy
deploy: book
  @echo "====> deploying to github"
  git worktree add /tmp/book gh-pages
  rm -rf /tmp/book/*
  cp -rp book/* /tmp/book/
  cd /tmp/book && \
    git add -A && \
    git commit -m "deployed on $(shell date) by ${USER}" && \
    git push origin gh-pages
```

19 For Developers

While `mdbook` is mainly used as a command line tool, you can also import the underlying library directly and use that to manage a book. It also has a fairly flexible plugin mechanism, allowing you to create your own custom tooling and consumers (often referred to as *backends*) if you need to do some analysis of the book or render it in a different format.

The *For Developers* chapters are here to show you the more advanced usage of `mdbook`.

The two main ways a developer can hook into the book's build process is via,

- Preprocessors
- Alternative Backends

19.1 The Build Process

The process of rendering a book project goes through several steps.

1. Load the book
    - Parse the `book.toml`, falling back to the default `Config` if it doesn't exist
    - Load the book chapters into memory
    - Discover which preprocessors/backends should be used
2. Run the preprocessors
3. Call each backend in turn

19.2 Using `mdbook` as a Library

The `mdbook` binary is just a wrapper around the `mdbook` crate, exposing its functionality as a command-line program. As such it is quite easy to create your own programs which use `mdbook` internally, adding your own functionality (e.g. a custom preprocessor) or tweaking the build process.

The easiest way to find out how to use the `mdbook` crate is by looking at the API Docs. The top level documentation explains how one would use the `MDBook` type to load and build a book, while the config module gives a good explanation on the configuration system.

20 Preprocessors

A *preprocessor* is simply a bit of code which gets run immediately after the book is loaded and before it gets rendered, allowing you to update and mutate the book. Possible use cases are:

- Creating custom helpers like `{{#include /path/to/file.md}}`
- Updating links so `[some chapter](some_chapter.md)` is automatically changed to `[some chapter](some_chapter.html)` for the HTML renderer
- Substituting in latex-style expressions (`$$ \frac{1}{3} $$`) with their mathjax equivalents

20.1 Hooking Into MDBook

MDBook uses a fairly simple mechanism for discovering third party plugins. A new table is added to `book.toml` (e.g. `preprocessor.foo` for the `foo` preprocessor) and then `mdbook` will try to invoke the `mdbook-foo` program as part of the build process.

While preprocessors can be hard-coded to specify which backend it should be run for (e.g. it doesn't make sense for MathJax to be used for non-HTML renderers) with the `preprocessor.foo.renderer` key.

```
[book]
title = "My Book"
authors = ["Michael-F-Bryan"]

[preprocessor.foo]
# The command can also be specified manually
command = "python3 /path/to/foo.py"
# Only run the `foo` preprocessor for the HTML and EPUB renderer
renderer = ["html", "epub"]
```

In typical unix style, all inputs to the plugin will be written to `stdin` as JSON and `mdbook` will read from `stdout` if it is expecting output.

The easiest way to get started is by creating your own implementation of the `Preprocessor` trait (e.g. in `lib.rs`) and then creating a shell binary which translates inputs to the correct `Preprocessor` method. For convenience, there is an example no-op preprocessor in the `examples/` directory which can easily be adapted for other preprocessors.
<details> <summary>Example no-op preprocessor</summary>

```
// nop-preprocessors.rs

use crate::nop_lib::Nop;
use clap::{App, Arg, ArgMatches, SubCommand};
use mdbook::book::Book;
use mdbook::errors::Error;
use mdbook::preprocess::{CmdPreprocessor, Preprocessor, ↪
    PreprocessorContext};
use std::io;
use std::process;

pub fn make_app() -> App<'static, 'static> {
    App::new("nop-preprocessor")
        .about("A mdbook preprocessor which does precisely ↪
    nothing")
        .subcommand(
            SubCommand::with_name("supports")
                .arg(Arg::with_name("renderer").required(true))
                .about("Check whether a renderer is supported by ↪
    this preprocessor"),
```

```
        )
}

fn main() {
    let matches = make_app().get_matches();

    // Users will want to construct their own preprocessor here
    let preprocessor = Nop::new();

    if let Some(sub_args) = matches.subcommand_matches("supports"↪
    ) {
        handle_supports(&preprocessor, sub_args);
    } else if let Err(e) = handle_preprocessing(&preprocessor) {
        eprintln!("{}", e);
        process::exit(1);
    }
}

fn handle_preprocessing(pre: &dyn Preprocessor) -> Result<(), ↪
    Error> {
    let (ctx, book) = CmdPreprocessor::parse_input(io::stdin())?;

    if ctx.mdbook_version != mdbook::MDBOOK_VERSION {
        // We should probably use the `semver` crate to check ↪
    compatibility
        // here...
        eprintln!(
            "Warning: The {} plugin was built against version {} ↪
    of mdbook, \
             but we're being called from version {}",
            pre.name(),
            mdbook::MDBOOK_VERSION,
            ctx.mdbook_version
        );
    }

    let processed_book = pre.run(&ctx, book)?;
    serde_json::to_writer(io::stdout(), &processed_book)?;

    Ok(())
}

fn handle_supports(pre: &dyn Preprocessor, sub_args: &ArgMatches)↪
     -> ! {
    let renderer = sub_args.value_of("renderer").expect("Required↪
     argument");
    let supported = pre.supports_renderer(&renderer);

    // Signal whether the renderer is supported by exiting with 1↪
     or 0.
    if supported {
        process::exit(0);
    } else {
        process::exit(1);
    }
}

```

```rust
/// The actual implementation of the `Nop` preprocessor. This would usually go
/// in your main `lib.rs` file.
mod nop_lib {
    use super::*;

    /// A no-op preprocessor.
    pub struct Nop;

    impl Nop {
        pub fn new() -> Nop {
            Nop
        }
    }

    impl Preprocessor for Nop {
        fn name(&self) -> &str {
            "nop-preprocessor"
        }

        fn run(&self, ctx: &PreprocessorContext, book: Book) -> Result<Book, Error> {
            // In testing we want to tell the preprocessor to blow up by setting a
            // particular config value
            if let Some(nop_cfg) = ctx.config.get_preprocessor(self.name()) {
                if nop_cfg.contains_key("blow-up") {
                    return Err("Boom!!1!".into());
                }
            }

            // we *are* a no-op preprocessor after all
            Ok(book)
        }

        fn supports_renderer(&self, renderer: &str) -> bool {
            renderer != "not-supported"
        }
    }
}
```

</details>

20.2 Hints For Implementing A Preprocessor

By pulling in `mdbook` as a library, preprocessors can have access to the existing infrastructure for dealing with books.

For example, a custom preprocessor could use the `CmdPreprocessor::parse_input()` function to deserialize the JSON written to `stdin`. Then each chapter of the `Book` can be mutated in-place via `Book::for_each_mut()`, and then written to

`stdout` with the `serde_json` crate.

Chapters can be accessed either directly (by recursively iterating over chapters) or via the `Book::for_each_mut()` convenience method.

The `chapter.content` is just a string which happens to be markdown. While it's entirely possible to use regular expressions or do a manual find & replace, you'll probably want to process the input into something more computer-friendly. The `pulldown-cmark` crate implements a production-quality event-based Markdown parser, with the `pulldown-cmark-to-cmark` allowing you to translate events back into markdown text.

The following code block shows how to remove all emphasis from markdown, without accidentally breaking the document.

```
fn remove_emphasis(
    num_removed_items: &mut usize,
    chapter: &mut Chapter,
) -> Result<String> {
    let mut buf = String::with_capacity(chapter.content.len());

    let events = Parser::new(&chapter.content).filter(|e| {
        let should_keep = match *e {
            Event::Start(Tag::Emphasis)
            | Event::Start(Tag::Strong)
            | Event::End(Tag::Emphasis)
            | Event::End(Tag::Strong) => false,
            _ => true,
        };
        if !should_keep {
            *num_removed_items += 1;
        }
        should_keep
    });

    cmark(events, &mut buf, None).map(|_| buf).map_err(|err| {
        Error::from(format!("Markdown serialization failed: {}", err))
    })
}
```

For everything else, have a look at the complete example.

21 Alternative Backends

A "backend" is simply a program which `mdbook` will invoke during the book rendering process. This program is passed a JSON representation of the book and configuration information via `stdin`. Once the backend receives this information

it is free to do whatever it wants.

There are already several alternative backends on GitHub which can be used as a rough example of how this is accomplished in practice.

- mdbook-linkcheck - a simple program for verifying the book doesn't contain any broken links
- mdbook-epub - an EPUB renderer
- mdbook-test - a program to run the book's contents through rust-skeptic to verify everything compiles and runs correctly (similar to `rustdoc --test`)

This page will step you through creating your own alternative backend in the form of a simple word counting program. Although it will be written in Rust, there's no reason why it couldn't be accomplished using something like Python or Ruby.

21.1 Setting Up

First you'll want to create a new binary program and add `mdbook` as a dependency.

```
$ cargo new --bin mdbook-wordcount
$ cd mdbook-wordcount
$ cargo add mdbook
```

When our `mdbook-wordcount` plugin is invoked, `mdbook` will send it a JSON version of `RenderContext` via our plugin's `stdin`. For convenience, there's a `RenderContext::from_json()` constructor which will load a `RenderContext`.

This is all the boilerplate necessary for our backend to load the book.

```
// src/main.rs
extern crate mdbook;

use std::io;
use mdbook::renderer::RenderContext;

fn main() {
    let mut stdin = io::stdin();
    let ctx = RenderContext::from_json(&mut stdin).unwrap();
}
```

Note: The `RenderContext` contains a `version` field. This lets backends figure out whether they are compatible with the version of `mdbook` it's being called by. This `version` comes directly from the corresponding field in `mdbook`'s `Cargo.toml`.

It is recommended that backends use the `semver` crate to inspect this field and emit a warning if there may be a compatibility issue.

21.2 Inspecting the Book

Now our backend has a copy of the book, lets count how many words are in each chapter!

Because the `RenderContext` contains a `Book` field (`book`), and a `Book` has the `Book::iter()` method for iterating over all items in a `Book`, this step turns out to be just as easy as the first.

```
1
2  fn main() {
3      let mut stdin = io::stdin();
4      let ctx = RenderContext::from_json(&mut stdin).unwrap();
5
6      for item in ctx.book.iter() {
7          if let BookItem::Chapter(ref ch) = *item {
8              let num_words = count_words(ch);
9              println!("{}: {}", ch.name, num_words);
10         }
11     }
12 }
13
14 fn count_words(ch: &Chapter) -> usize {
15     ch.content.split_whitespace().count()
16 }
```

21.3 Enabling the Backend

Now we've got the basics running, we want to actually use it. First, install the program.

```
1 $ cargo install --path .
```

Then `cd` to the particular book you'd like to count the words of and update its `book.toml` file.

```
  [book]
  title = "mdBook Documentation"
  description = "Create book from markdown files. Like Gitbook
    but implemented in Rust"
  authors = ["Mathieu David", "Michael-F-Bryan"]

+ [output.html]

+ [output.wordcount]
```

When it loads a book into memory, `mdbook` will inspect your `book.toml` file to try and figure out which backends to use by looking for all `output.*` tables. If none are provided it'll fall back to using the default HTML renderer.

Notably, this means if you want to add your own custom backend you'll also need to make sure to add the HTML backend, even if its table just stays empty.

Now you just need to build your book like normal, and everything should *Just Work*.

```
$ mdbook build
...
2018-01-16 07:31:15 [INFO] (mdbook::renderer): Invoking the "
    mdbook-wordcount" renderer
mdBook: 126
Command Line Tool: 224
init: 283
build: 145
watch: 146
serve: 292
test: 139
Format: 30
SUMMARY.md: 259
Configuration: 784
Theme: 304
index.hbs: 447
Syntax highlighting: 314
MathJax Support: 153
Rust code specific features: 148
For Developers: 788
Alternative Backends: 710
Contributors: 85
```

The reason we didn't need to specify the full name/path of our `wordcount` backend is because `mdbook` will try to *infer* the program's name via convention. The executable for the `foo` backend is typically called `mdbook-foo`, with an associated `[output.foo]` entry in the `book.toml`. To explicitly tell `mdbook` what command to invoke (it may require command-line arguments or be an interpreted script),

you can use the `command` field.

```
  [book]
  title = "mdBook Documentation"
  description = "Create book from markdown files. Like Gitbook but implemented in Rust"
  authors = ["Mathieu David", "Michael-F-Bryan"]

  [output.html]

  [output.wordcount]
+ command = "python /path/to/wordcount.py"
```

21.4 Configuration

Now imagine you don't want to count the number of words on a particular chapter (it might be generated text/code, etc). The canonical way to do this is via the usual `book.toml` configuration file by adding items to your `[output.foo]` table.

The `Config` can be treated roughly as a nested hashmap which lets you call methods like `get()` to access the config's contents, with a `get_deserialized()` convenience method for retrieving a value and automatically deserializing to some arbitrary type `T`.

To implement this, we'll create our own serializable `WordcountConfig` struct which will encapsulate all configuration for this backend.

First add `serde` and `serde_derive` to your `Cargo.toml`,

```
$ cargo add serde serde_derive
```

And then you can create the config struct,

```
extern crate serde;
#[macro_use]
extern crate serde_derive;

...

#[derive(Debug, Default, Serialize, Deserialize)]
#[serde(default, rename_all = "kebab-case")]
pub struct WordcountConfig {
  pub ignores: Vec<String>,
}
```

Now we just need to deserialize the `WordcountConfig` from our `RenderContext` and then add a check to make sure we skip ignored chapters.

```
  fn main() {
      let mut stdin = io::stdin();
      let ctx = RenderContext::from_json(&mut stdin).unwrap();
+     let cfg: WordcountConfig = ctx.config
+         .get_deserialized("output.wordcount")
+         .unwrap_or_default();

      for item in ctx.book.iter() {
          if let BookItem::Chapter(ref ch) = *item {
+             if cfg.ignores.contains(&ch.name) {
+                 continue;
+             }
+
              let num_words = count_words(ch);
              println!("{}: {}", ch.name, num_words);
          }
      }
  }
```

21.5 Output and Signalling Failure

While it's nice to print word counts to the terminal when a book is built, it might also be a good idea to output them to a file somewhere. `mdbook` tells a backend where it should place any generated output via the `destination` field in `RenderContext`.

```
+ use std::fs::{self, File};
+ use std::io::{self, Write};
- use std::io;
  use mdbook::renderer::RenderContext;
  use mdbook::book::{BookItem, Chapter};

  fn main() {
    ...

+     let _ = fs::create_dir_all(&ctx.destination);
+     let mut f = File::create(ctx.destination.join("wordcounts.txt")).unwrap();
+
      for item in ctx.book.iter() {
          if let BookItem::Chapter(ref ch) = *item {
              ...

              let num_words = count_words(ch);
```

```
18                 println!("{}: {}", ch.name, num_words);
19 +               writeln!(f, "{}: {}", ch.name, num_words).unwrap();
20             }
21         }
22     }
```

Note: There is no guarantee that the destination directory exists or is empty (`mdbook` may leave the previous contents to let backends do caching), so it's always a good idea to create it with `fs::create_dir_all()`.

If the destination directory already exists, don't assume it will be empty. To allow backends to cache the results from previous runs, `mdbook` may leave old content in the directory.

There's always the possibility that an error will occur while processing a book (just look at all the `unwrap()`'s we've written already), so `mdbook` will interpret a non-zero exit code as a rendering failure.

For example, if we wanted to make sure all chapters have an *even* number of words, erroring out if an odd number is encountered, then you may do something like this:

```
1  + use std::process;
2    ...
3
4    fn main() {
5        ...
6
7        for item in ctx.book.iter() {
8            if let BookItem::Chapter(ref ch) = *item {
9                ...
10
11               let num_words = count_words(ch);
12               println!("{}: {}", ch.name, num_words);
13               writeln!(f, "{}: {}", ch.name, num_words).unwrap();
14
15 +             if cfg.deny_odds && num_words % 2 == 1 {
16 +               eprintln!("{} has an odd number of words!", ch.↪
       name);
17 +               process::exit(1);
18               }
19           }
20       }
21   }
22
23   #[derive(Debug, Default, Serialize, Deserialize)]
24   #[serde(default, rename_all = "kebab-case")]
25   pub struct WordcountConfig {
26       pub ignores: Vec<String>,
27 +     pub deny_odds: bool,
28   }
```

Now, if we reinstall the backend and build a book,

```
$ cargo install --path . --force
$ mdbook build /path/to/book
...
2018-01-16 21:21:39 [INFO] (mdbook::renderer): Invoking the "wordcount" renderer
mdBook: 126
Command Line Tool: 224
init: 283
init has an odd number of words!
2018-01-16 21:21:39 [ERROR] (mdbook::renderer): Renderer exited with non-zero return code.
2018-01-16 21:21:39 [ERROR] (mdbook::utils): Error: Rendering failed
2018-01-16 21:21:39 [ERROR] (mdbook::utils):    Caused By: The "mdbook-wordcount" renderer failed
```

As you've probably already noticed, output from the plugin's subprocess is immediately passed through to the user. It is encouraged for plugins to follow the "rule of silence" and only generate output when necessary (e.g. an error in generation or a warning).

All environment variables are passed through to the backend, allowing you to use the usual `RUST_LOG` to control logging verbosity.

21.6 Wrapping Up

Although contrived, hopefully this example was enough to show how you'd create an alternative backend for `mdbook`. If you feel it's missing something, don't hesitate to create an issue in the issue tracker so we can improve the user guide.

The existing backends mentioned towards the start of this chapter should serve as a good example of how it's done in real life, so feel free to skim through the source code or ask questions.

22 Contributors

Here is a list of the contributors who have helped improving mdBook. Big shout-out to them!

- mdinger
- Kevin (kbknapp)
- Steve Klabnik (steveklabnik)
- Adam Solove (asolove)
- Wayne Nilsen (waynenilsen)
- funnkill
- Fu Gangqiang (FuGangqiang)
- Michael-F-Bryan
- Chris Spiegel (cspiegel)
- projektir
- Phaiax
- Matt Ickstadt (mattico)
- Weihang Lo (@weihanglo)

If you feel you're missing from this list, feel free to add yourself in a PR.