Contents

Introduction

mdBook is a command line tool to create books with Markdown. It is ideal for creating product or API documentation, tutorials, course materials or anything that requires a clean, easily navigable and customizable presentation.

• Lightweight Markdown syntax helps you focus more on your content
• Integrated search support
• Color syntax highlighting for code blocks for many different languages
• Theme files allow customizing the formatting of the output
• Preprocessors can provide extensions for custom syntax and modifying content
• Backends can render the output to multiple formats
• Written in Rust for speed, safety, and simplicity
• Automated testing of Rust code samples

This guide is an example of what mdBook produces. mdBook is used by the Rust programming language project, and The Rust Programming Language book is another fine example of mdBook in action.

Contributing

mdBook is free and open source. You can find the source code on GitHub and issues and feature requests can be posted on the GitHub issue tracker. mdBook relies on the community to fix bugs and add features: if you'd like to contribute, please read the CONTRIBUTING guide and consider opening a pull request.

License

The mdBook source and documentation are released under the Mozilla Public License v2.0.

User Guide

Installation

There are multiple ways to install the mdBook CLI tool. Choose any one of the methods below that best suit your needs. If you are installing mdBook for automatic deployment, check out the continuous integration chapter for more examples on how to install.

Pre-compiled binaries

Executable binaries are available for download on the GitHub Releases page. Download the binary for your platform (Windows, macOS, or Linux) and extract the archive. The archive contains an `mdbook` executable which you can run to build your books.

To make it easier to run, put the path to the binary into your `PATH`.

Build from source using Rust

To build the `mdbook` executable from source, you will first need to install Rust and Cargo. Follow the instructions on the Rust installation page. mdBook currently requires at least Rust version 1.66.

Once you have installed Rust, the following command can be used to build and install mdBook:

```
cargo install mdbook
```

This will automatically download mdBook from crates.io, build it, and install it in Cargo's global binary directory (`~/.cargo/bin/` by default).

To uninstall, run the command `cargo uninstall mdbook`.

Installing the latest master version

The version published to crates.io will ever so slightly be behind the version hosted on GitHub. If you need the latest version you can build the git version of mdBook yourself. Cargo makes this **super easy**!

```
cargo install --git https://github.com/rust-lang/mdBook.git mdbook
```

Again, make sure to add the Cargo bin directory to your `PATH`.

If you are interested in making modifications to mdBook itself, check out the Contributing Guide for more information.

Reading Books

This chapter gives an introduction on how to interact with a book produced by mdBook. This assumes you are reading an HTML book. The options and formatting will be different for other output formats such as PDF.

A book is organized into *chapters*. Each chapter is a separate page. Chapters can be nested into a hierarchy of sub-chapters. Typically, each chapter will be organized into a series of *headings* to subdivide a chapter.

Navigation

There are several methods for navigating through the chapters of a book.

The **sidebar** on the left provides a list of all chapters. Clicking on any of the chapter titles will load that page.

The sidebar may not automatically appear if the window is too narrow, particularly on mobile displays. In that situation, the menu icon (three horizontal bars) at the top-left of the page can be pressed to open and close the sidebar.

The **arrow buttons** at the bottom of the page can be used to navigate to the previous or the next chapter.

The **left and right arrow keys** on the keyboard can be used to navigate to the previous or the next chapter.

Top menu bar

The menu bar at the top of the page provides some icons for interacting with the book. The icons displayed will depend on the settings of how the book was generated.

Icon	Description
——	————-
	Opens and closes the chapter listing sidebar.
	Opens a picker to choose a different color theme.
	Opens a search bar for searching within the book.
	Instructs the web browser to print the entire book.
	Opens a link to the website that hosts the source code of the book.
	Opens a page to directly edit the source of the page you are currently reading.

Tapping the menu bar will scroll the page to the top.

Search

Each book has a built-in search system. Pressing the search icon () in the menu bar, or pressing the S key on the keyboard will open an input box for entering search terms. Typing some terms will show matching chapters and sections in real time.

Clicking any of the results will jump to that section. The up and down arrow keys can be used to navigate the results, and enter will open the highlighted section.

After loading a search result, the matching search terms will be highlighted in the text. Clicking a highlighted word or pressing the Esc key will remove the highlighting.

Code blocks

mdBook books are often used for programming projects, and thus support highlighting code blocks and samples. Code blocks may contain several different icons for interacting with them:

| Icon | Description | |——|————-| | | Copies the code block into your local clipboard, to allow pasting into another application. | | | For Rust code examples, this will execute the sample code and display the compiler output just below the example (see playground). | | | For Rust code examples, this will toggle visibility of "hidden" lines. Sometimes, larger examples will hide lines which are not particularly relevant to what is being illustrated (see hiding code lines). | | | For editable code examples, this will undo any changes you have made. |

Here's an example:

```rust
println!("Hello, World!");
```

Creating a Book

Once you have the `mdbook` CLI tool installed, you can use it to create and render a book.

Initializing a book

The `mdbook init` command will create a new directory containing an empty book for you to get started. Give it the name of the directory that you want to create:

```
mdbook init my-first-book
```

It will ask a few questions before generating the book. After answering the questions, you can change the current directory into the new book:

```
cd my-first-book
```

There are several ways to render a book, but one of the easiest methods is to use the `serve` command, which will build your book and start a local webserver:

```
mdbook serve --open
```

The `--open` option will open your default web browser to view your new book. You can leave the server running even while you edit the content of the book, and `mdbook` will automatically rebuild the output *and* automatically refresh your web browser.

Check out the [CLI Guide](#) for more information about other `mdbook` commands and CLI options.

Anatomy of a book

A book is built from several files which define the settings and layout of the book.

`book.toml`

In the root of your book, there is a `book.toml` file which contains settings for describing how to build your book. This is written in the [TOML markup language](#). The default settings are usually good enough to get you started. When you are interested in exploring more features and options that mdBook provides, check out the [Configuration chapter](#) for more details.

A very basic `book.toml` can be as simple as this:

```
[book]
title = "My First Book"
```

SUMMARY.md

The next major part of a book is the summary file located at `src/SUMMARY.md`. This file contains a list of all the chapters in the book. Before a chapter can be viewed, it must be added to this list.

Here's a basic summary file with a few chapters:

```
# Summary

[Introduction](README.md)

- [My First Chapter](my-first-chapter.md)
- [Nested example](nested/README.md)
    - [Sub-chapter](nested/sub-chapter.md)
```

Try opening up `src/SUMMARY.md` in your editor and adding a few chapters. If any of the chapter files do not exist, `mdbook` will automatically create them for you.

For more details on other formatting options for the summary file, check out the [Summary chapter](#).

Source files

The content of your book is all contained in the `src` directory. Each chapter is a separate Markdown file. Typically, each chapter starts with a level 1 heading with the title of the chapter.

```
# My First Chapter

Fill out your content here.
```

The precise layout of the files is up to you. The organization of the files will correspond to the HTML files generated, so keep in mind that the file layout is part of the URL of each chapter.

While the `mdbook serve` command is running, you can open any of the chapter files and start editing them. Each time you save the file, `mdbook` will rebuild the book and refresh your web browser.

Check out the [Markdown chapter](#) for more information on formatting the content of your chapters.

All other files in the `src` directory will be included in the output. So if you have images or other static files, just include them somewhere in the `src` directory.

Publishing a book

Once you've written your book, you may want to host it somewhere for others to view. The first step is to build the output of the book. This can be done with the `mdbook build` command in the same directory where the `book.toml` file is located:

```
mdbook build
```

This will generate a directory named `book` which contains the HTML content of your book. You can then place this directory on any web server to host it.

For more information about publishing and deploying, check out the [Continuous Integration chapter](#) for more.

Reference Guide

Command Line Tool

The `mdbook` command-line tool is used to create and build books. After you have [installed](#) `mdbook`, you can run the `mdbook help` command in your terminal to view the available commands.

This following sections provide in-depth information on the different commands available.

- `mdbook init <directory>` — Creates a new book with minimal boilerplate to start with.
- `mdbook build` — Renders the book.
- `mdbook watch` — Rebuilds the book any time a source file changes.
- `mdbook serve` — Runs a web server to view the book, and rebuilds on changes.
- `mdbook test` — Tests Rust code samples.
- `mdbook clean` — Deletes the rendered output.
- `mdbook completions` — Support for shell auto-completion.

init

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 where 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` is 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.

–title

Specify a title for the book. If not supplied, an interactive prompt will ask for a title.

`mdbook` init --title=`"my amazing book"`

–ignore

Create a `.gitignore` file configured to ignore the `book` directory created when building a book. If not supplied, an interactive prompt will ask whether it should be created.

`mdbook` init --ignore=none
`mdbook` init --ignore=git

–force

Skip the prompts to create a `.gitignore` and for the title for the book.

build

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. Note that files mentioned in `SUMMARY.md` but not present will be created.

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: The build command copies all files (excluding files with `.md` extension) from the source directory into the build directory.

watch

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; this includes re-creating deleted files still mentioned in `SUMMARY.md`!

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`.

Specify exclude patterns

The `watch` command will not automatically trigger a build for files listed in the `.gitignore` file in the book root directory. The `.gitignore` file may contain file patterns described in the [gitignore documentation](). This can be useful for ignoring temporary files created by some editors.

Note: Only `.gitignore` from book root directory is used. Global `$HOME/.gitignore` or `.gitignore` files in parent directories are not used.

serve

The serve command is used to preview a book by serving it via HTTP at `localhost:3000` by default:

`mdbook serve`

The `serve` command watches the book's `src` directory for changes, rebuilding the book and refreshing clients for each change; this includes re-creating deleted files still mentioned in `SUMMARY.md`! 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

The `serve` hostname defaults to `localhost`, and the port defaults to `3000`. Either option can be specified on the command line:

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

–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`.

Specify exclude patterns

The `serve` command will not automatically trigger a build for files listed in the `.gitignore` file in the book root directory. The `.gitignore` file may contain file patterns described in the [gitignore documentation](). This can be useful for ignoring temporary files created by some editors.

__Note:__ Only the `.gitignore` from the book root directory is used. Global `$HOME/.gitignore` or `.gitignore` files in parent directories are not used.

test

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 Rust tests are supported.

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`). The path should point to the Cargo [build cache](#) `deps` directory that contains the build output of your project. For

example, if your Rust project's book is in a directory named `my-book`, the following command would include the crate's dependencies when running `test`:

```
mdbook test my-book -L target/debug/deps/
```

See the `rustdoc` command-line [documentation](#) for more information.

–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`.

–chapter

The `--chapter` (`-c`) option allows you to test a specific chapter of the book using the chapter name or the relative path to the chapter.

clean

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.

completions

The completions command is used to generate auto-completions for some common shells. This means when you type `mdbook` in your shell, you can then press your shell's auto-complete key (usually the Tab key) and it may display what the valid options are, or finish partial input.

The completions first need to be installed for your shell:

```
# bash
mdbook completions bash > ~/.local/share/bash-completion/completions/mdbook
# oh-my-zsh
mdbook completions zsh > ~/.oh-my-zsh/completions/_mdbook
autoload -U compinit && compinit
```

The command prints a completion script for the given shell. Run `mdbook completions --help` for a list of supported shells.

Where to place the completions depend on which shell you are using and your operating system. Consult your shell's documentation for more information one where to place the script.

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

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.

This markdown file must be named `SUMMARY.md`. Its formatting is very strict and must follow the structure outlined below to allow for easy parsing. Any element not specified below, be it formatting or textual, is likely to be ignored at best, or may cause an error when attempting to build the book.

Structure

1. ***Title*** - While optional, it's common practice to begin with a title, generally # Summary. This is ignored by the parser however, and can be omitted.

```
# Summary
```

1. ***Prefix Chapter*** - Before the main numbered chapters, prefix chapters can be added that will not be numbered. This is useful for forewords, introductions, etc. There are, however, some constraints. Prefix chapters cannot be nested; they should all be on the root level. And you cannot add prefix chapters once you have added numbered chapters.

```
[A Prefix Chapter](relative/path/to/markdown.md)

- [First Chapter](relative/path/to/markdown2.md)
```

1. ***Part Title*** - Headers can be used as a title for the following numbered chapters. This can be used to logically separate different sections of the book. The title is rendered as unclickable text. Titles are optional, and the numbered chapters can be broken into as many parts as desired.

```
# My Part Title

- [First Chapter](relative/path/to/markdown.md)
```

1. ***Numbered Chapter*** - Numbered chapters outline the main content of the book and can be nested, resulting in a nice hierarchy (chapters, sub-chapters, etc.).

Title of Part

- [First Chapter](relative/path/to/markdown.md)
- [Second Chapter](relative/path/to/markdown2.md)
   - [Sub Chapter](relative/path/to/markdown3.md)

Title of Another Part

- [Another Chapter](relative/path/to/markdown4.md)

Numbered chapters can be denoted with either - or * (do not mix delimiters).

1. ***Suffix Chapter*** - Like prefix chapters, suffix chapters are unnumbered, but they come after numbered chapters.

- [Last Chapter](relative/path/to/markdown.md)

[Title of Suffix Chapter](relative/path/to/markdown2.md)

1. ***Draft chapters*** - Draft chapters are chapters without a file and thus content. The purpose of a draft chapter is to signal future chapters still to be written. Or when still laying out the structure of the book to avoid creating the files while you are still changing the structure of the book a lot. Draft chapters will be rendered in the HTML renderer as disabled links in the table of contents, as you can see for the next chapter in the table of contents on the left. Draft chapters are written like normal chapters but without writing the path to the file.

- [Draft Chapter]()

1. ***Separators*** - Separators can be added before, in between, and after any other element. They result in an HTML rendered line in the built table of contents. A separator is a line containing exclusively dashes and at least three of them: ---.

My Part Title

[A Prefix Chapter](relative/path/to/markdown.md)

- [First Chapter](relative/path/to/markdown2.md)

Example

Below is the markdown source for the SUMMARY.md for this guide, with the resulting table of contents as rendered to the left.

Summary

[Introduction](README.md)

User Guide

- [Installation](guide/installation.md)
- [Reading Books](guide/reading.md)
- [Creating a Book](guide/creating.md)

Reference Guide

- [Command Line Tool](cli/README.md)
    - [init](cli/init.md)
    - [build](cli/build.md)
    - [watch](cli/watch.md)
    - [serve](cli/serve.md)
    - [test](cli/test.md)
    - [clean](cli/clean.md)
    - [completions](cli/completions.md)
- [Format](format/README.md)
    - [SUMMARY.md](format/summary.md)
        - [Draft chapter]()
    - [Configuration](format/configuration/README.md)
        - [General](format/configuration/general.md)
        - [Preprocessors](format/configuration/preprocessors.md)
        - [Renderers](format/configuration/renderers.md)
        - [Environment Variables](format/configuration/environment-variables.md)
    - [Theme](format/theme/README.md)
        - [index.hbs](format/theme/index-hbs.md)
        - [Syntax highlighting](format/theme/syntax-highlighting.md)
        - [Editor](format/theme/editor.md)
    - [MathJax Support](format/mathjax.md)
    - [mdBook-specific features](format/mdbook.md)
    - [Markdown](format/markdown.md)
- [Continuous Integration](continuous-integration.md)
- [For Developers](for_developers/README.md)
    - [Preprocessors](for_developers/preprocessors.md)
    - [Alternative Backends](for_developers/backends.md)

[Contributors](misc/contributors.md)

Draft chapter

Configuration

This section details the configuration options available in the **book.toml**:

- **General** configuration including the `book, rust, build` sections
- **Preprocessor** configuration for default and custom book preprocessors
- **Renderer** configuration for the HTML, Markdown and custom renderers
- **Environment Variable** configuration for overriding configuration options in your environment

General

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"
authors = ["John Doe"]
description = "The example book covers examples."

[rust]
edition = "2018"

[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
```

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.

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. This is also used to derive the direction of text (RTL, LTR) within the book.
- **text-direction**: The direction of text in the book: Left-to-right (LTR) or Right-to-left (RTL). Possible values: `ltr`, `rtl`. When not specified, the text direction is derived from the book's `language` attribute.

book.toml

```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"
text-direction = "ltr"
```

Rust options

Options for the Rust language, relevant to running tests and playground integration.

```toml
[rust]
edition = "2015"   # the default edition for code blocks
```

- **edition**: Rust edition to use by default for the code snippets. Default is "2015". Individual code blocks can be controlled with the `edition2015`, `edition2018` or `edition2021` annotations, such as:

```rust,edition2015
// This only works in 2015.
let try = true;
```

Build options

This controls the build process of your book.

```
[build]
build-dir = "book"                 # the directory where the output is placed
create-missing = true              # whether or not to create missing pages
use-default-preprocessors = true   # use the default preprocessors
extra-watch-dirs = []              # directories to watch for triggering builds
```

- **build-dir:** The directory to put the rendered book in. By default this is `book/` in the book's root directory. This can overridden with the `--dest-dir` CLI option.

- **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.

- **extra-watch-dirs**: A list of paths to directories that will be watched in the `watch` and `serve` commands. Changes to files under these directories will trigger rebuilds. Useful if your book depends on files outside its `src` directory.

Preprocessors

Preprocessors are extensions that can modify the raw Markdown source before it gets sent to the renderer.

The following preprocessors are built-in and included by default:

- `links`: Expands the `{{ #playground }}`, `{{ #include }}`, and `{{ #rustdoc_include }}` handlebars helpers in a chapter to include the contents of a file. See Including files for more.
- `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.

The built-in preprocessors can be disabled with the `build.use-default-preprocessors` config option.

The community has developed several preprocessors. See the Third Party Plugins wiki page for a list of available preprocessors.

For information on how to create a new preprocessor, see the Preprocessors for Developers chapter.

Custom Preprocessor Configuration

Preprocessors can be added by including a `preprocessor` table in `book.toml` with the name of the preprocessor. For example, if you have a preprocessor called `mdbook-example`, then you can include it with:

```
[preprocessor.example]
```

With this table, mdBook will execute the `mdbook-example` preprocessor.

This table can include additional key-value pairs that are specific to the preprocessor. For example, if our example preprocessor needed some extra configuration options:

```
[preprocessor.example]
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.example]
renderers = ["html"]  # example preprocessor only runs with the HTML renderer
```

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"
```

Require A Certain Order

The order in which preprocessors are run can be controlled with the `before` and `after` fields. For example, suppose you want your `linenos` preprocessor to process lines that may have been `{{#include}}`d; then you want it to run after the built-in `links` preprocessor, which you can require using either the `before` or `after` field:

```
[preprocessor.linenos]
after = [ "links" ]
```

or

```
[preprocessor.links]
before = [ "linenos" ]
```

It would also be possible, though redundant, to specify both of the above in the same config file.

Preprocessors having the same priority specified through `before` and `after` are sorted by name. Any infinite loops will be detected and produce an error.

Renderers

Renderers (also called "backends") are responsible for creating the output of the book.

The following backends are built-in:

- `html` — This renders the book to HTML. This is enabled by default if no other `[output]` tables are defined in `book.toml`.
- `markdown` — This outputs the book as markdown after running the preprocessors. This is useful for debugging preprocessors.

The community has developed several backends. See the Third Party Plugins wiki page for a list of available backends.

For information on how to create a new backend, see the Backends for Developers chapter.

Output tables

Backends can be added by including a `output` table in `book.toml` with the name of the backend. For example, if you have a backend called `mdbook-wordcount`, then you can include it with:

```
[output.wordcount]
```

With this table, mdBook will execute the `mdbook-wordcount` backend.

This table can include additional key-value pairs that are specific to the backend. For example, if our example backend needed some extra configuration options:

```
[output.wordcount]
ignores = ["Example Chapter"]
```

If you define any `[output]` tables, then the `html` backend is not enabled by default. If you want to keep the `html` backend running, then just include it in the `book.toml` file. For example:

```
[book]
title = "My Awesome Book"

[output.wordcount]

[output.html]
```

If more than one `output` table is included, this changes the behavior for the layout of the output directory. If there is only one backend, then it places its output directly in the `book` directory (see

`build.build-dir` to override this location). If there is more than one backend, then each backend is placed in a separate directory underneath `book`. For example, the above would have directories `book/html` and `book/wordcount`.

Custom backend commands

By default when you add an `[output.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.

```toml
[output.random]
command = "python random.py"
```

Optional backends

If you enable a backend that isn't installed, the default behavior is to throw an error. This behavior can be changed by marking the backend as optional:

```toml
[output.wordcount]
optional = true
```

This demotes the error to a warning.

HTML renderer options

The HTML renderer has a variety of options detailed below. They should be specified in the `[output.html]` table of the `book.toml` file.

```toml
# Example book.toml file with all output options.
[book]
title = "Example book"
authors = ["John Doe", "Jane Doe"]
description = "The example book covers examples."

[output.html]
theme = "my-theme"
default-theme = "light"
preferred-dark-theme = "navy"
curly-quotes = true
mathjax-support = false
copy-fonts = true
additional-css = ["custom.css", "custom2.css"]
additional-js = ["custom.js"]
no-section-label = false
```

```
git-repository-url = "https://github.com/rust-lang/mdBook"
git-repository-icon = "fa-github"
edit-url-template = "https://github.com/rust-lang/mdBook/edit/master/guide/{path}"
site-url = "/example-book/"
cname = "myproject.rs"
input-404 = "not-found.md"
```

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`.
- **preferred-dark-theme:** The default dark theme. This theme will be used if the browser requests the dark version of the site via the ['prefers-color-scheme'](prefers-color-scheme) CSS media query. Defaults to `navy`.
- **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`.
- **copy-fonts:** (**Deprecated**) If `true` (the default), mdBook uses its built-in fonts which are copied to the output directory. If `false`, the built-in fonts will not be used. This option is deprecated. If you want to define your own custom fonts, create a `theme/fonts/fonts.css` file and store the fonts in the `theme/fonts/` directory.
- **google-analytics:** This field has been deprecated and will be removed in a future release. Use the `theme/head.hbs` file to add the appropriate Google Analytics code instead.
- **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 numeric section labels in the table of contents column. For example, "1.", "2.1". Set this option to true to disable those labels. Defaults to `false`.
- **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` which looks like . If you are not using GitHub, another option to consider is `fa-code-fork` which looks like .

- **edit-url-template:** Edit url template, when provided shows a "Suggest an edit" button (which looks like) for directly jumping to editing the currently viewed page. For e.g. GitHub projects set this to `https://github.com/<owner>/<repo>/edit/<branch>/{path}` or for Bitbucket projects set it to `https://bitbucket.org/<owner>/<repo>/src/<branch>/{path}?mode=edit` where {path} will be replaced with the full path of the file in the repository.
- **input-404:** The name of the markdown file used for missing files. The corresponding output file will be the same, with the extension replaced with `html`. Defaults to `404.md`.
- **site-url:** The url where the book will be hosted. This is required to ensure navigation links and script/css imports in the 404 file work correctly, even when accessing urls in subdirectories. Defaults to `/`. If `site-url` is set, make sure to use document relative links for your assets, meaning they should not start with `/`.
- **cname:** The DNS subdomain or apex domain at which your book will be hosted. This string will be written to a file named CNAME in the root of your site, as required by GitHub Pages (see *Managing a custom domain for your GitHub Pages site*).

[output.html.print]

The `[output.html.print]` table provides options for controlling the printable output. By default, mdBook will include an icon on the top right of the book (which looks like) that will print the book as a single page.

```
[output.html.print]
enable = true    # include support for printable output
page-break = true # insert page-break after each chapter
```

- **enable:** Enable print support. When `false`, all print support will not be rendered. Defaults to `true`.
- **page-break:** Insert page breaks between chapters. Defaults to `true`.

[output.html.fold]

The `[output.html.fold]` table provides options for controlling folding of the chapter listing in the navigation sidebar.

```
[output.html.fold]
enable = false   # whether or not to enable section folding
level = 0        # the depth to start folding
```

- **enable:** Enable section-folding. When off, all folds are open. Defaults to `false`.
- **level:** The higher the more folded regions are open. When level is 0, all folds are closed. Defaults to `0`.

[output.html.playground]

The `[output.html.playground]` table provides options for controlling Rust sample code blocks, and their integration with the Rust Playground.

```toml
[output.html.playground]
editable = false        # allows editing the source code
copyable = true         # include the copy button for copying code snippets
copy-js = true          # includes the JavaScript for the code editor
line-numbers = false    # displays line numbers for editable code
runnable = true         # displays a run button for rust code
```

- **editable:** Allow editing the source code. Defaults to `false`.
- **copyable:** Display the copy button on code snippets. Defaults to `true`.
- **copy-js:** Copy JavaScript files for the editor to the output directory. Defaults to `true`.
- **line-numbers:** Display line numbers on editable sections of code. Requires both `editable` and `copy-js` to be `true`. Defaults to `false`.
- **runnable:** Displays a run button for rust code snippets. Changing this to `false` will disable the run in playground feature globally. Defaults to `true`.

[output.html.code]

The `[output.html.code]` table provides options for controlling code blocks.

```toml
[output.html.code]
# A prefix string per language (one or more chars).
# Any line starting with whitespace+prefix is hidden.
hidelines = { python = "~" }
```

- **hidelines:** A table that defines how [hidden code lines](#) work for each language. The key is the language and the value is a string that will cause code lines starting with that prefix to be hidden.

[output.html.search]

The `[output.html.search]` table provides options for controlling the built-in text search. mdBook must be compiled with the `search` feature enabled (on by default).

```toml
[output.html.search]
enable = true            # enables the search feature
limit-results = 30       # maximum number of search results
teaser-word-count = 30   # number of words used for a search result teaser
use-boolean-and = true   # multiple search terms must all match
boost-title = 2          # ranking boost factor for matches in headers
boost-hierarchy = 1      # ranking boost factor for matches in page names
boost-paragraph = 1      # ranking boost factor for matches in text
```

```
expand = true          # partial words will match longer terms
heading-split-level = 3 # link results to heading levels
copy-js = true          # include Javascript code for search
```

- **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 `false`.
- **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`.

`[output.html.redirect]`

The `[output.html.redirect]` table provides a way to add redirects. This is useful when you move, rename, or remove a page to ensure that links to the old URL will go to the new location.

```
[output.html.redirect]
"/appendices/bibliography.html" = "https://rustc-dev-guide.rust-lang.org/appendix/
bibliography.html"
"/other-installation-methods.html" = "../infra/other-installation-methods.html"
```

The table contains key-value pairs where the key is where the redirect file needs to be created, as an absolute path from the build directory, (e.g. `/appendices/bibliography.html`). The value can be any valid URI the browser should navigate to (e.g. `https://rust-lang.org/`, `/overview.html`, or `../bibliography.html`).

This will generate an HTML page which will automatically redirect to the given location. Note that the source location does not support # anchor redirects.

Markdown Renderer

The Markdown renderer will run preprocessors and then output the resulting Markdown. This is mostly useful for debugging preprocessors, especially in conjunction with `mdbook test` to see the Markdown that `mdbook` is passing to `rustdoc`.

The Markdown renderer is included with `mdbook` but disabled by default. Enable it by adding an empty table to your `book.toml` as follows:

```
[output.markdown]
```

There are no configuration options for the Markdown renderer at this time; only whether it is enabled or disabled.

See [the preprocessors documentation](#) for how to specify which preprocessors should run before the Markdown renderer.

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.

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.
- **head.hbs** is appended to the HTML `<head>` section.
- **header.hbs** content is appended on top of every book page.
- **css/** contains the CSS files for styling the book.- **css/chrome.css** is for UI elements.
- **css/general.css** is the base styles.
- **css/print.css** is the style for printer output.
- **css/variables.css** contains variables used in other CSS files.
- **book.js** is mostly used to add client side functionality, like hiding / un-hiding 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.svg** and **favicon.png** the favicon that will be used. The SVG version is used by newer browsers.
- **fonts/fonts.css** contains the definition of which fonts to load. Custom fonts can be included in the `fonts` directory.

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` and just remove the files you don't want to override.

`mdbook init --theme` will not create every file listed above. Some files, such as `head.hbs`, do not have built-in equivalents. Just create the file if you need it.

If you completely replace all built-in themes, be sure to also set `output.html.preferred-dark-theme` in the config, which defaults to the built-in `navy` theme.

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.

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 <html lang="{{ language }}"> for example.
- **title** Title used for the current page. This is identical to `{{ chapter_title }}` - `{{ book_title }}` unless `book_title` is not set in which case it just defaults to the `chapter_title`.
- **book_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).

Handlebars Helpers

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

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

```html
<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.

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

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](#)

Syntax highlighting

mdBook uses [Highlight.js](#) with a custom theme for syntax highlighting.

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

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

Supported languages

These languages are supported by default, but you can add more by supplying your own `highlight.js` file:

- apache
- armasm
- bash
- c
- coffeescript
- cpp
- csharp
- css
- d
- diff
- go
- handlebars
- haskell
- http
- ini
- java
- javascript
- json
- julia
- kotlin
- less
- lua
- makefile

- markdown
- nginx
- objectivec
- perl
- php
- plaintext
- properties
- python
- r
- ruby
- rust
- scala
- scss
- shell
- sql
- swift
- typescript
- x86asm
- xml
- yaml

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 the `theme` folder of your book.

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

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 too many flashy colors.

Editor

In addition to providing runnable code playgrounds, 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.playground]
editable = true
```

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

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

The above will result in this editable playground:

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

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

Customizing the Editor

By default, the editor is the [Ace](#) editor, but, if desired, the functionality may be overridden by providing a different folder:

```
[output.html.playground]
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 overridden as it has some couplings with the default Ace editor.

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}`).

Inline equations

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

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

Block equations

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

you would write:

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

mdBook-specific features

Hiding code lines

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

For the Rust language, you can use the # character as a prefix which will hide lines like you would with Rustdoc.

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

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

Will render as

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

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

When you tap or hover the mouse over the code block, there will be an eyeball icon () which will toggle the visibility of the hidden lines.

By default, this only works for code examples that are annotated with `rust`. However, you can define custom prefixes for other languages by adding a new line-hiding prefix in your `book.toml` with the language name and prefix character(s):

```
[output.html.code.hidelines]
python = "~"
```

The prefix will hide any lines that begin with the given prefix. With the python prefix shown above, this:

```
~hidden()
nothidden():
~   hidden()
    ~hidden()
    nothidden()
```

will render as

```
~hidden()
nothidden():
~    hidden()
    ~hidden()
    nothidden()
```

This behavior can be overridden locally with a different prefix. This has the same effect as above:

```
```python,hidelines=!!!
!!!hidden()
nothidden():
!!!    hidden()
    !!!hidden()
    nothidden()
```
```

Rust Playground

Rust language code blocks will automatically get a play button () which will execute the code and display the output just below the code block. This works by sending the code to the Rust Playground.

```
println!("Hello, World!");
```

If there is no `main` function, then the code is automatically wrapped inside one.

If you wish to disable the play button for a code block, you can include the `noplayground` option on the code block like this:

```
```rust,noplayground
let mut name = String::new();
std::io::stdin().read_line(&mut name).expect("failed to read line");
println!("Hello {}!", name);
```
```

Or, if you wish to disable the play button for all code blocks in your book, you can write the config to the `book.toml` like this.

```
[output.html.playground]
runnable = false
```

Rust code block attributes

Additional attributes can be included in Rust code blocks with comma, space, or tab-separated terms just after the language term. For example:

```rust,ignore
# This example won't be tested.
panic!("oops!");
```

These are particularly important when using `mdbook test` to test Rust examples. These use the same attributes as rustdoc attributes, with a few additions:

- `editable` — Enables the editor.
- `noplayground` — Removes the play button, but will still be tested.
- `mdbook-runnable` — Forces the play button to be displayed. This is intended to be combined with the `ignore` attribute for examples that should not be tested, but you want to allow the reader to run.
- `ignore` — Will not be tested and no play button is shown, but it is still highlighted as Rust syntax.
- `should_panic` — When executed, it should produce a panic.
- `no_run` — The code is compiled when tested, but it is not run. The play button is also not shown.
- `compile_fail` — The code should fail to compile.
- `edition2015`, `edition2018`, `edition2021` — Forces the use of a specific Rust edition. See `rust.edition` to set this globally.

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 interpreting them.

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

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,noplayground
{{#include file.rs:component}}
```
```

```
Here is a system:
```rust,no_run,noplayground
{{#include file.rs:system}}
```
```

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

Lines containing anchor patterns inside the included anchor are ignored.

Including a file but initially hiding all except specified lines

The `rustdoc_include` helper is for including code from external Rust files that contain complete examples, but only initially showing particular lines specified with line numbers or anchors in the same way as with `include`.

The lines not in the line number range or between the anchors will still be included, but they will be prefaced with #. This way, a reader can expand the snippet to see the complete example, and Rustdoc will use the complete example when you run `mdbook test`.

For example, consider a file named `file.rs` that contains this Rust program:

```rust
fn main() {
    let x = add_one(2);
    assert_eq!(x, 3);
}

fn add_one(num: i32) -> i32 {
    num + 1
}
```

We can include a snippet that initially shows only line 2 by using this syntax:

```
To call the `add_one` function, we pass it an `i32` and bind the returned value to `x`:
```

```
```rust
{{#rustdoc_include file.rs:2}}
```
```

This would have the same effect as if we had manually inserted the code and hidden all but line 2 using #:

```
To call the `add_one` function, we pass it an `i32` and bind the returned value to `x`:
```

````
```rust
# fn main() {
    let x = add_one(2);
#     assert_eq!(x, 3);
# }
#
# fn add_one(num: i32) -> i32 {
#     num + 1
# }
```
````

That is, it looks like this (click the "expand" icon to see the rest of the file):

```
# fn main() {
    let x = add_one(2);
#     assert_eq!(x, 3);
# }
#
# fn add_one(num: i32) -> i32 {
#     num + 1
# }
```

Inserting runnable Rust files

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

```
{{#playground 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 Playground 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");
}
```

Any additional values passed after the filename will be included as attributes of the code block. For example `{{#playground example.rs editable}}` will create the code block like the following:

```rust,editable
# Contents of example.rs here.
```

And the `editable` attribute will enable the editor as described at [Rust code block attributes](#).

Controlling page <title>

A chapter can set a <title> that is different from its entry in the table of contents (sidebar) by including a `{{#title ...}}` near the top of the page.

```
{{#title My Title}}
```

HTML classes provided by mdBook

`class="left"` **and** `"right"`

These classes are provided by default, for inline HTML to float images.

```html
<img class="right" src="images/rust-logo-blk.svg" alt="The Rust logo">
```

`class="hidden"`

HTML tags with class `hidden` will not be shown.

```html
<div class="hidden">This will not be seen.</div>
```

`class="warning"`

To make a warning or similar note stand out, wrap it in a warning div.

```html
<div class="warning">

This is a bad thing that you should pay attention to.

Warning blocks should be used sparingly in documentation, to avoid "warning
fatique," where people are trained to ignore them because they usually don't
matter for what they're doing.

</div>
```

This is a bad thing that you should pay attention to.

Warning blocks should be used sparingly in documentation, to avoid "warning fatique," where people are trained to ignore them because they usually don't matter for what they're doing.

Markdown

mdBook's [parser](#) adheres to the [CommonMark](#) specification with some extensions described below. You can take a quick [tutorial](#), or [try out](#) CommonMark in real time. A complete Markdown overview is out of scope for this documentation, but below is a high level overview of some of the basics. For a more in-depth experience, check out the [Markdown Guide](#).

Text and Paragraphs

Text is rendered relatively predictably:

```
Here is a line of text.

This is a new line.
```

Will look like you might expect:

Here is a line of text.

This is a new line.

Headings

Headings use the # marker and should be on a line by themselves. More # mean smaller headings:

```
### A heading

Some text.

#### A smaller heading

More text.
```

A heading

Some text.

A smaller heading

More text.

Lists

Lists can be unordered or ordered. Ordered lists will order automatically:

```
* milk
* eggs
* butter

1. carrots
1. celery
1. radishes
```

• milk
• eggs
• butter

1. carrots
2. celery
3. radishes

Links

Linking to a URL or local file is easy:

```
Use [mdBook](https://github.com/rust-lang/mdBook).

Read about [mdBook](mdbook.md).

A bare url: <https://www.rust-lang.org>.
```

Use [mdBook](https://github.com/rust-lang/mdBook).

Read about [mdBook](mdbook.md).

A bare url: https://www.rust-lang.orghttps://www.rust-lang.org.

Relative links that end with `.md` will be converted to the `.html` extension. It is recommended to use `.md` links when possible. This is useful when viewing the Markdown file outside of mdBook, for example on GitHub or GitLab which render Markdown automatically.

Links to `README.md` will be converted to `index.html`. This is done since some services like GitHub render README files automatically, but web servers typically expect the root file to be called `index.html`.

You can link to individual headings with # fragments. For example, `mdbook.md#text-and-paragraphs` would link to the [Text and Paragraphs](#) section above. The ID is created by transforming the heading such as converting to lowercase and replacing spaces with dashes. You can click on any heading and look at the URL in your browser to see what the fragment looks like.

Images

Including images is simply a matter of including a link to them, much like in the *Links* section above. The following markdown includes the Rust logo SVG image found in the `images` directory at the same level as this file:

`![The Rust Logo](images/rust-logo-blk.svg)`

Produces the following HTML when built with mdBook:

`<p><img src="images/rust-logo-blk.svg" alt="The Rust Logo" /></p>`

Which, of course displays the image like so:

The Rust Logo

Extensions

mdBook has several extensions beyond the standard CommonMark specification.

Strikethrough

Text may be rendered with a horizontal line through the center by wrapping the text with one or two tilde characters on each side:

`An example of ~~strikethrough text~~.`

This example will render as:

An example of ~~strikethrough text~~.

This follows the GitHub Strikethrough extension.

Footnotes

A footnote generates a small numbered link in the text which when clicked takes the reader to the footnote text at the bottom of the item. The footnote label is written similarly to a link

reference with a caret at the front. The footnote text is written like a link reference definition, with the text following the label. Example:

```
This is an example of a footnote[^note].

[^note]: This text is the contents of the footnote, which will be rendered
    towards the bottom.
```

This example will render as:

This is an example of a footnote[^note].

[^note]: This text is the contents of the footnote, which will be rendered towards the bottom.

The footnotes are automatically numbered based on the order the footnotes are written.

Tables

Tables can be written using pipes and dashes to draw the rows and columns of the table. These will be translated to HTML table matching the shape. Example:

```
| Header1 | Header2 |
|---------|---------|
| abc     | def     |
```

This example will render similarly to this:

| Header1 | Header2 | | ——|———| | abc | def |

See the specification for the GitHub Tables extension for more details on the exact syntax supported.

Task lists

Task lists can be used as a checklist of items that have been completed. Example:

```
- [x] Complete task
- [ ] Incomplete task
```

This will render as:

- [x] Complete task
- [ ] Incomplete task

See the specification for the task list extension for more details.

Smart punctuation

Some ASCII punctuation sequences will be automatically turned into fancy Unicode characters:

| ASCII sequence | Unicode | | | ——————-|———| | -- | – | | --- | — | | ... | … | | " | " or ", depending on context | | ' | ' or ', depending on context |

So, no need to manually enter those Unicode characters!

This feature is disabled by default. To enable it, see the `output.html.curly-quotes` config option.

Heading attributes

Headings can have a custom HTML ID and classes. This lets you maintain the same ID even if you change the heading's text, it also lets you add multiple classes in the heading.

Example:

```
# Example heading { #first .class1 .class2 }
```

This makes the level 1 heading with the content `Example heading`, ID `first`, and classes `class1` and `class2`. Note that the attributes should be space-separated.

More information can be found in the [heading attrs spec page](#).

Continuous Integration

There are a variety of services such as GitHub Actions or GitLab CI/CD which can be used to test and deploy your book automatically.

The following provides some general guidelines on how to configure your service to run mdBook. Specific recipes can be found at the Automated Deployment wiki page.

Installing mdBook

There are several different strategies for installing mdBook. The particular method depends on your needs and preferences.

Pre-compiled binaries

Perhaps the easiest method is to use the pre-compiled binaries found on the GitHub Releases page. A simple approach would be to use the popular `curl` CLI tool to download the executable:

```
mkdir bin
curl -sSL https://github.com/rust-lang/mdBook/releases/download/v0.4.35/mdbook-v0.4.35-
x86_64-unknown-linux-gnu.tar.gz | tar -xz --directory=bin
bin/mdbook build
```

Some considerations for this approach:

- This is relatively fast, and does not necessarily require dealing with caching.
- This does not require installing Rust.
- Specifying a specific URL means you have to manually update your script to get a new version. This may be a benefit if you want to lock to a specific version. However, some users prefer to automatically get a newer version when they are published.
- You are reliant on the GitHub CDN being available.

Building from source

Building from source will require having Rust installed. Some services have Rust pre-installed, but if your service does not, you will need to add a step to install it.

After Rust is installed, `cargo install` can be used to build and install mdBook. We recommend using a SemVer version specifier so that you get the latest **non-breaking** version of mdBook. For example:

```
cargo install mdbook --no-default-features --features search --vers "^0.4" --locked
```

This includes several recommended options:

- `--no-default-features` — Disables features like the HTTP server used by `mdbook serve` that is likely not needed on CI. This will speed up the build time significantly.
- `--features search` — Disabling default features means you should then manually enable features that you want, such as the built-in search capability.
- `--vers "^0.4"` — This will install the most recent version of the `0.4` series. However, versions after like `0.5.0` won't be installed, as they may break your build. Cargo will automatically upgrade mdBook if you have an older version already installed.
- `--locked` — This will use the dependencies that were used when mdBook was released. Without `--locked`, it will use the latest version of all dependencies, which may include some fixes since the last release, but may also (rarely) cause build problems.

You will likely want to investigate caching options, as building mdBook can be somewhat slow.

Running tests

You may want to run tests using `mdbook test` every time you push a change or create a pull request. This can be used to validate Rust code examples in the book.

This will require having Rust installed. Some services have Rust pre-installed, but if your service does not, you will need to add a step to install it.

Other than making sure the appropriate version of Rust is installed, there's not much more than just running `mdbook test` from the book directory.

You may also want to consider running other kinds of tests, like mdbook-linkcheck which will check for broken links. Or if you have your own style checks, spell checker, or any other tests it might be good to run them in CI.

Deploying

You may want to automatically deploy your book. Some may want to do this every time a change is pushed, and others may want to only deploy when a specific release is tagged.

You'll also need to understand the specifics on how to push a change to your web service. For example, GitHub Pages just requires committing the output onto a specific git branch. Other services may require using something like SSH to connect to a remote server.

The basic outline is that you need to run `mdbook build` to generate the output, and then transfer the files (which are in the `book` directory) to the correct location.

You may then want to consider if you need to invalidate any caches on your web service.

See the Automated Deployment wiki page for examples of various different services.

404 handling

mdBook automatically generates a 404 page to be used for broken links. The default output is a file named `404.html` at the root of the book. Some services like GitHub Pages will automatically use this page for broken links. For other services, you may want to consider configuring the web server to use this page as it will provide the reader navigation to get back to the book.

If your book is not deployed at the root of the domain, then you should set the `output.html.site-url` setting so that the 404 page works correctly. It needs to know where the book is deployed in order to load the static files (like CSS) correctly. For example, this guide is deployed at [https://rust-lang.github.io/mdBook/https://rust-lang.github.io/mdBook/](https://rust-lang.github.io/mdBook/), and the `site-url` setting is configured like this:

```toml
# book.toml
[output.html]
site-url = "/mdBook/"
```

You can customize the look of the 404 page by creating a file named `src/404.md` in your book. If you want to use a different filename, you can set `output.html.input-404` to a different filename.

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](#)

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

1. For each backend:+ Run all the preprocessors.
2. Call the backend to render the processed result.

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.

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}}`
- Substituting in latex-style expressions (`$$ \frac{1}{3} $$`) with their mathjax equivalents

See [Configuring Preprocessors](#) for more information about using preprocessors.

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.

Once the preprocessor has been defined and the build process starts, mdBook executes the command defined in the `preprocessor.foo.command` key twice. The first time it runs the preprocessor to determine if it supports the given renderer. mdBook passes two arguments to the process: the first argument is the string `supports` and the second argument is the renderer name. The preprocessor should exit with a status code 0 if it supports the given renderer, or return a non-zero exit code if it does not.

If the preprocessor supports the renderer, then mdbook runs it a second time, passing JSON data into stdin. The JSON consists of an array of `[context, book]` where `context` is the serialized object `PreprocessorContext` and `book` is a `Book` object containing the content of the book.

The preprocessor should return the JSON format of the `Book` object to stdout, with any modifications it wishes to perform.

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.

```rust
// nop-preprocessors.rs

use crate::nop_lib::Nop;
use clap::{Arg, ArgMatches, Command};
use mdbook::book::Book;
use mdbook::errors::Error;
use mdbook::preprocess::{CmdPreprocessor, Preprocessor, PreprocessorContext};
use semver::{Version, VersionReq};
use std::io;
use std::process;
```

```rust
pub fn make_app() -> Command {
    Command::new("nop-preprocessor")
        .about("A mdbook preprocessor which does precisely nothing")
        .subcommand(
            Command::new("supports")
                .arg(Arg::new("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())?;

    let book_version = Version::parse(&ctx.mdbook_version)?;
    let version_req = VersionReq::parse(mdbook::MDBOOK_VERSION)?;

    if !version_req.matches(&book_version) {
        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) -> ! {
```

```rust
    let renderer = sub_args
        .get_one::<String>("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);
    }
}

/// 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") {
                    anyhow::bail!("Boom!!1!");
                }
            }

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

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

```rust
        }
    }

    #[cfg(test)]
    mod test {
        use super::*;

        #[test]
        fn nop_preprocessor_run() {
            let input_json = r##"[
                {
                    "root": "/path/to/book",
                    "config": {
                        "book": {
                            "authors": ["AUTHOR"],
                            "language": "en",
                            "multilingual": false,
                            "src": "src",
                            "title": "TITLE"
                        },
                        "preprocessor": {
                            "nop": {}
                        }
                    },
                    "renderer": "html",
                    "mdbook_version": "0.4.21"
                },
                {
                    "sections": [
                        {
                            "Chapter": {
                                "name": "Chapter 1",
                                "content": "# Chapter 1\n",
                                "number": [1],
                                "sub_items": [],
                                "path": "chapter_1.md",
                                "source_path": "chapter_1.md",
                                "parent_names": []
                            }
                        }
                    ],
                    "__non_exhaustive": null
                }
            ]"##;
            let input_json = input_json.as_bytes();

            let (ctx, book) =
```

```
mdbook::preprocess::CmdPreprocessor::parse_input(input_json).unwrap();
        let expected_book = book.clone();
        let result = Nop::new().run(&ctx, book);
        assert!(result.is_ok());

        // The nop-preprocessor should not have made any changes to the book
content.
        let actual_book = result.unwrap();
        assert_eq!(actual_book, expected_book);
    }
  }
}
```

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` crate 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)
```

```rust
            | 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.

Implementing a preprocessor with a different language

The fact that mdBook utilizes stdin and stdout to communicate with the preprocessors makes it easy to implement them in a language other than Rust. The following code shows how to implement a simple preprocessor in Python, which will modify the content of the first chapter. The example below follows the configuration shown above with `preprocessor.foo.command` actually pointing to a Python script.

```python
import json
import sys


if __name__ == '__main__':
    if len(sys.argv) > 1: # we check if we received any argument
        if sys.argv[1] == "supports":
            # then we are good to return an exit status code of 0, since the other
            # argument will just be the renderer's name
            sys.exit(0)

    # load both the context and the book representations from stdin
    context, book = json.load(sys.stdin)
    # and now, we can just modify the content of the first chapter
    book['sections'][0]['Chapter']['content'] = '# Hello'
    # we are done with the book's modification, we can just print it to stdout,
    print(json.dumps(book))
```

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.

See [Configuring Renderers](#) for more information about using backends.

The community has developed several backends. See the Third Party Plugins wiki page for a list of available backends.

Setting Up

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.

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.

```rust
// 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.

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.

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

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

fn count_words(ch: &Chapter) -> usize {
    ch.content.split_whitespace().count()
}
```

Enabling the Backend

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

```
$ 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"
```

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);
          }
      }
  }
```

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);
              println!("{}: {}", ch.name, num_words);
+             writeln!(f, "{}: {}", ch.name, num_words).unwrap();
          }
      }
  }
```

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:

```
+ use std::process;
  ...

  fn main() {
      ...

      for item in ctx.book.iter() {
          if let BookItem::Chapter(ref ch) = *item {
              ...

              let num_words = count_words(ch);
              println!("{}: {}", ch.name, num_words);
              writeln!(f, "{}: {}", ch.name, num_words).unwrap();

+             if cfg.deny_odds && num_words % 2 == 1 {
+                 eprintln!("{} has an odd number of words!", ch.name);
+                 process::exit(1);
              }
          }
      }
  }

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

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.

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.

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](#))
- Avision Ho ([avisionh](#))
- Vivek Akupatni ([apatniv](#))
- Eric Huss ([ehuss](#))
- Josh Rotenberg ([joshrotenberg](#))

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