mdBook

mdBook is a command line tool and Rust crate to create books using Markdow
similar to Gitbook but written in Rust.

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

mdBook is free and open source, you can find the source code on Github. Issue
requests can be posted on the Github Issue tracker.

API docs

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

License

Command Line Tool

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

Install

Pre-requisite

mdBook is written in **Rust** and therefore needs to be compiled with **Cargo**, bec
offer ready-to-go binaries. If you haven't already installed Rust, please go ahea

Install Crates.io version

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

```
cargo install mdbook
```

This will fetch the source code from Crates.io and compile it. You will have to a
directory to your `PATH`.

Run `mdbook help` in your terminal to verify if it works. Congratulations, you ha

Install Git version

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

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

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

The init command

There is some minimal boilerplate that is the same for every new book. It's for 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 u

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

- The `src` directory is were you write your book in markdown. It contains a configuration files, etc.
- The `book` directory is where your book is rendered. All the output is read server to be seen by your audience.
- The `SUMMARY.md` file is the most important file, it's the skeleton of your bo in more detail in another chapter.

Tip & Trick: Hidden Feature

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

Specify a directory

When using the `init` command, you can also specify a directory, instead of us working directory, by appending a path to the command:

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

--theme

When you use the `--theme` argument, the default theme will be copied into a `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 overw delete it and the default file will be used.

The build command

The build command is used to render your book:

```
mdbook build
```

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

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

Specify a directory

Like `init`, the `build` command can take a directory as an argument to use ins working directory.

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

--open

When you use the `--open` (`-o`) option, mdbook will open the rendered book i
browser after building it.

--dest-dir

The `--dest-dir` (`-d`) option allows you to change the output directory for you

---

note: *make sure to run the build command in the root directory and not in the sou*

The watch command

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

Specify a directory

Like `init` and `build`, `watch` can take a directory as an argument to use inste
working directory.

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

--open

When you use the `--open` (`-o`) option, mdbook will open the rendered book i
browser.

--dest-dir

The `--dest-dir` (`-d`) option allows you to change the output directory for you

---

note: *the `watch` command has not gotten a lot of testing yet, there could be some
discover a problem, please report it on Github*

The serve command

The `serve` command is useful when you want to preview your book. It also do
the webpage whenever a file changes. It achieves this by serving the books con
`localhost:3000` (unless otherwise configured, see below) and runs a websock
`localhost:3001` which triggers the reloads. This preferred by many for writing
because it allows for you to see the result of your work instantly after every file

Specify a directory

Like `watch`, `serve` can take a directory as an argument to use instead of the c
directory.

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

Server options

`serve` has four options: the http port, the websocket port, the interface to ser
address of the server so that the browser may reach the websocket server.

For example: suppose you had an nginx server for SSL termination which has a
192.168.1.100 on port 80 and proxied that to 127.0.0.1 on port 8000. To run us

```
mdbook serve path/to/book -p 8000 -i 127.0.0.1 -a 192.168.1.100
```

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

--open

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

--dest-dir

The `--dest-dir` (`-d`) option allows you to change the output directory for you

---

note: *the `serve` command has not gotten a lot of testing yet, there could be some
discover a problem, please report it on Github*

The test command

When writing a book, you sometimes need to automate some tests. For examp
Programming Book uses a lot of code examples that could get outdated. There
important for them to be able to automatically test these code examples.

mdBook supports a `test` command that will run all available tests in mdBook.
one test is available: *"Test Rust code examples using Rustdoc"*, but I hope this will
future to include more tests like:

- checking for broken links
- checking for unused files
- ...

In the future I would like the user to be able to enable / disable test from the `b`
configuration file and support custom tests.

How to use it:

```
$ mdbook test
[*]: Testing file: "/mdBook/book-example/src/README.md"
```

The clean command

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

```
mdbook clean
```

It will try to delete the built book. If a path is provided, it will be used.

Specify a directory

Like `init`, the `clean` command can take a directory as an argument to use in
build directory.

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

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

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 wha
appear, what their hierarchy is and where the source files are. Without this file,

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

Allowed elements

1. ***Title*** It's common practice to begin with a title, generally `# Summary`. But
   the parser just ignores it. So you can too if you feel like it.

2. ***Prefix Chapter*** Before the main numbered chapters you can add a couple
   not be numbered. This is useful for forewords, introductions, etc. There a
   constraints. You can not nest prefix chapters, they should all be on the ro
   not add prefix chapters once you have added numbered chapters.

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

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

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

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

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

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

Configuration

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

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

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

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

[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 in the configuratio
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 informat `<head>` of each page
- **src:** By default, the source directory is found in the directory named `src` root folder. But this is configurable with the `src` key in the configuration

book.toml

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

Build options

This controls the build process of your book.

- **build-dir:** The directory to put the rendered book in. By default this is `bo` directory.
- **create-missing:** By default, any missing files specified in `SUMMARY.md` will book is built (i.e. `create-missing = true`). If this is `false` then the build exit with an error if any files do not exist.

book.toml

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

HTML renderer options

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

The following configuration options are available:

- **theme:** mdBook comes with a default theme and all the resource files ne option is set, mdBook will selectively overwrite the theme files with the or specified folder.
- **curly-quotes:** Convert straight quotes to curly quotes, except for those th blocks and code spans. Defaults to `false`.
- **google-analytics:** If you use Google Analytics, this option lets you enable your ID in the configuration file.
- **additional-css:** If you need to slightly change the appearance of your boc the whole style, you can specify a set of stylesheets that will be loaded aft where you can surgically change the style.
- **additional-js:** If you need to add some behaviour to your book without re behaviour, you can specify a set of JavaScript files that will be loaded alon
- **no-section-label:** mdBook by defaults adds section label in table of conte example, "1.", "2.1". Set this option to true to disable those labels. Default
- **playpen:** A subtable for configuring various playpen settings.
- **search:** A subtable for configuring the in-browser search functionality. m compiled with the `search` feature enabled (on by default).

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

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

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

- **limit-results:** The maximum number of search results. Defaults to `30`.
- **teaser-word-count:** The number of words used for a search result teaser
- **use-boolean-and:** Define the logical link between multiple search words. words must appear in each result. Defaults to `true`.

- **boost-title:** Boost factor for the search result score if a search word appe
  Defaults to `2`.
- **boost-hierarchy:** Boost factor for the search result score if a search word
  hierarchy. The hierarchy contains all titles of the parent documents and a
  Defaults to `1`.
- **boost-paragraph:** Boost factor for the search result score if a search wor
  Defaults to `1`.
- **expand:** True if search should match longer results e.g. search `micro` sh
  `microwave`. Defaults to `true`.
- **heading-split-level:** Search results will link to a section of the document
  result. Documents are split into sections by headings this level or less. De
  `### This is a level 3 heading`)
- **copy-js:** Copy JavaScript files for the search implementation to the output
  `true`.

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

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

[output.html]
theme = "my-theme"
curly-quotes = true
google-analytics = "123456"
additional-css = ["custom.css", "custom2.css"]
additional-js = ["custom.js"]

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

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

Environment Variables

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

Variables starting with `MDBOOK_` are used for configuration. The key is created l
`MDBOOK_` prefix and turning the resulting string into `kebab-case`. Double unde
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
without needing to touch your `book.toml`.

> **Note:** To facilitate setting more complex config items, the value of an enviro
> 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 bu
with something like

```
$ export MDBOOK_BOOK="{'title': 'My Awesome Book', authors: ['Micha
$ mdbook build
```

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

Theme

The default renderer uses a handlebars template to render your markdown file
default theme included in the mdBook binary.

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

Here are the files you can override:

- ***index.hbs*** is the handlebars template.
- ***book.css*** is the style used in the output. If you want to change the design
  probably the file you want to modify. Sometimes in conjunction with `ind`
  want to radically change the layout.
- ***book.js*** is mostly used to add client side functionality, like hiding / un-hidi
  changing the theme, ...
- ***highlight.js*** is the JavaScript that is used to highlight code snippets, you sh
  modify this.
- ***highlight.css*** is the theme used for the code highlighting
- ***favicon.png*** the favicon that will be used

Generally, when you want to tweak the theme, you don't need to override all th
need changes in the stylesheet, there is no point in overriding all the other files
files take precedence over built-in ones, they will not get updated with new fixe

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

index.hbs

`index.hbs` is the handlebars template that is used to render the book. The ma
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 wil
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 h
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`. To use in `<html lang="{`
  example. At the moment it is hardcoded.
- ***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 from the current file. Since the original directory structure is maintained, 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": "di
  ```

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

Handlebars Helpers

In addition to the properties you can access, there are some handlebars helper

1. **toc**

   The toc helper is used like this

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

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

   ```
   <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 Cha
           </ul>
       </li>
   </ul>
   ```

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

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

2. **previous / next**

   The previous and next helpers expose a `link` and `name` property to the 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. html can be changed to your liking.

If you would like me to expose other properties or helpers, please create a new issu

Syntax Highlighting

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

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

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

Custom theme

Like the rest of the theme, the files used for syntax highlighting can be overrid

- ***highlight.js*** normally you shouldn't have to overwrite this file, unless you
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 we
yourself, rename it to `highlight.css` and put it in `src/theme` (or the equivale
your source folder)

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

Hiding code lines

There is a feature in mdBook that let's you hide code lines by prepending them

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

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

Will render as

```
    let x = 5;
    let y = 7;

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

**At the moment, this only works for code examples that are annotated wi
would collide with semantics of some programming languages. In the futu
make this configurable through the `book.toml` so that everyone can bene**

Improve default theme

If you think the default theme doesn't look quite right for a specific language, o
Feel free to submit a new issue explaining what you have in mind and I will tak

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

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

Editor

In addition to providing runnable code playpens, mdBook optionally allows the
order to enable editable code blocks, the following needs to be added to the ***b***

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

To make a specific block available for editing, the attribute `editable` needs to

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

The above will result in this editable playpen:

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

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

Customizing the Editor

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

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

Note that for the editor changes to function correctly, the `book.js` inside of th need to be overriden 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.t` `output.html` section.

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

> **Note:** The usual delimiters MathJax uses are not yet supported. You can't cu `$$ ... $$` as delimiters and the `\[ ... \]` delimiters need an extra backs Hopefully this limitation will be lifted soon.

> **Note:** When you use double backslashes in MathJax blocks (for example in c `\begin{cases} \frac 1 2 \\ \frac 3 4 \end{cases}`) you need to add *tw* (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 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 equati

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

you would write:

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

mdBook-specific markdown

Hiding code lines

There is a feature in mdBook that lets you hide code lines by prepending them

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

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

Will render as

```
    let x = 5;
    let y = 7;

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

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.

Usually, this command is used for including code snippets and examples. In thi
would include a specific part of the file e.g. which only contains the relevant lin
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 secon
all lines up to line 10, i.e. the lines from 11 till the end of the file are omitted. Th
includes all lines from line 2, i.e. the first line is omitted. The last command incl
`file.rs` consisting of lines 2 to 10.

Inserting runnable Rust files

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

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

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

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

Here is what a rendered code snippet looks like:

```
fn main() {
    println!("Hello World!");
}
```

For Developers

While `mdbook` is mainly used as a command line tool, you can also import the u
directly and use that to manage a book. It also has a fairly flexible plugin mecha
create your own custom tooling and consumers (often referred to as *backends*)
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

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

- Preprocessors
- Alternate 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
   - Load the book chapters into memory
   - Discover which preprocessors/backends should be used
2. Run the preprocessors
3. Call each backend in turn

Using `mdbook` as a Library

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

The easiest way to find out how to use the `mdbook` crate is by looking at the AP
documentation explains how one would use the `MDBook` type to load and build
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 boo
it gets rendered, allowing you to update and mutate the book. Possible use cas

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

Implementing a Preprocessor

A preprocessor is represented by the `Preprocessor` trait.

```
pub trait Preprocessor {
    fn name(&self) -> &str;
    fn run(&self, ctx: &PreprocessorContext, book: &mut Book) -> Resu
}
```

Where the `PreprocessorContext` is defined as

```
pub struct PreprocessorContext {
    pub root: PathBuf,
    pub config: Config,
}
```

A complete Example

The magic happens within the `run(...)` method of the `Preprocessor` trait im

As direct access to the chapters is not possible, you will probably end up iterati `for_each_mut(...)`:

```
book.for_each_mut(|item: &mut BookItem| {
    if let BookItem::Chapter(ref mut chapter) = *item {
      eprintln!("{}: processing chapter '{}'", self.name(), chapter.n
      res = Some(
          match Deemphasize::remove_emphasis(&mut num_removed_items,
              Ok(md) => {
                  chapter.content = md;
                  Ok(())
              }
              Err(err) => Err(err),
          },
      );
  }
});
```

The `chapter.content` is just a markdown formatted string, and you will have t way. Even though it's entirely possible to implement some sort of manual find that feels too unsafe you can use `pulldown-cmark` to parse the string into ever instead.

Finally you can use `pulldown-cmark-to-cmark` to transform these events back

The following code block shows how to remove all emphasis from markdown,

```
fn remove_emphasis(num_removed_items: &mut i32, chapter: &mut Chapter
Result<String> {
    let mut buf = String::with_capacity(chapter.content.len());
    let events = Parser::new(&chapter.content).filter(|e| {
        let should_keep = match *e {
            Event::Start(Tag::Emphasis)
            | Event::Start(Tag::Strong)
            | Event::End(Tag::Emphasis)
            | Event::End(Tag::Strong) => false,
            _ => true,
        };
        if !should_keep {
            *num_removed_items += 1;
        }
        should_keep
    });
    cmark(events, &mut buf, None)
        .map(|_| buf)
        .map_err(|err| Error::from(format!("Markdown serialization fa
}
```

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

Alternate Backends

A "backend" is simply a program which `mdbook` will invoke during the book ren program is passed a JSON representation of the book and configuration inform Once the backend receives this information it is free to do whatever it wants.

There are already several alternate backends on GitHub which can be used as how this is accomplished in practice.

- mdbook-linkcheck - a simple program for verifying the book doesn't conta

- mdbook-epub - an EPUB renderer
- mdbook-test - a program to run the book's contents through rust-skeptic compiles and runs correctly (similar to `rustdoc --test`)

This page will step you through creating your own alternate backend in the for counting program. Although it will be written in Rust, there's no reason why it c accomplished using something like Python or Ruby.

Setting Up

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

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

When our `mdbook-wordcount` plugin is invoked, `mdbook` will send it a JSON ver `RenderContext` via our plugin's `stdin`. For convenience, there's a `RenderCont` constructor which will load a `RenderContext`.

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

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

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

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

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

It is recommended that backends use the `semver` crate to inspect this field an 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 ea

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

```
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 pr

```
$ cargo install
```

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

```
  [book]
  title = "mdBook Documentation"
  description = "Create book from markdown files. Like Gitbook but im
  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 which backends to use by looking for all `output.*` tables. If none are provided the default HTML renderer.

Notably, this means if you want to add your own custom backend you'll also ne 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 W*

```
$ mdbook build
...
2018-01-16 07:31:15 [INFO] (mdbook::renderer): Invoking the "mdbook-w
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
Alternate Backends: 710
Contributors: 85
```

The reason we didn't need to specify the full name/path of our `wordcount` bac `mdbook` will try to *infer* the program's name via convention. The executable for typically called `mdbook-foo`, with an associated `[output.foo]` entry in the `boo` tell `mdbook` what command to invoke (it may require command-line arguments script), you can use the `command` field.

```
  [book]
  title = "mdBook Documentation"
  description = "Create book from markdown files. Like Gitbook but im
  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 cha generated text/code, etc). The canonical way to do this is via the usual `book.to` by adding items to your `[output.foo]` table.

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

To implement this, we'll create our own serializable `WordcountConfig` struct w 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 `RenderContex` 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 migh to output them to a file somewhere. `mdbook` tells a backend where it should pl 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")
+
      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 leave the previous contents to let backends do caching), so it's always a good with `fs::create_dir_all()`.

There's always the possibility that an error will occur while processing a book (j `unwrap()`'s we've written already), so `mdbook` will interpret a non-zero exit cod failure.

For example, if we wanted to make sure all chapters have an *even* number of w 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 --force
$ mdbook build /path/to/book
...
2018-01-16 21:21:39 [INFO] (mdbook::renderer): Invoking the "wordcour
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
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 "mdboo
renderer failed
```

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

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

Wrapping Up

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

The existing backends mentioned towards the start of this chapter should serv
of how it's done in real life, so feel free to skim through the source code or ask

Contributors

Here is a list of the contributors who have helped improving mdBook. Big shou

If you have contributed to mdBook and I forgot to add you, don't hesitate to ad
If you are in the list, feel free to add your real name & contact information if yo

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