Table of Contents

Getting Started

Your Content

Customization

GitBook Toolchain Documentation

This document aims to be a comprehensive guide to the GitBook command line tool, version **3.2.3**. This is the same toolchain used by the legacy GitBook platform, living at legacy.gitbook.com. Help for using the platform can be found at help.legacy.gitbook.com. If you are looking for the new GitBook service, head to gitbook.com.

What is `gitbook`?

`gitbook` is a command line tool (and Node.js library) for building beautiful books using GitHub/Git and Markdown (or AsciiDoc). This documentation has been generated using `gitbook.

`gitbook` can output your content as a website (customizable and extensibles) or as an ebook (PDF, ePub or Mobi).

legacy.gitbook.com is the online platform to create and host books built using the GitBook format. It offers hosting, collaboration features and an easy-to-use editor.

Help and Support

If you have problems with the toolchain, you can search for or open a discussion on GitHub.

Check out the GitBook Community Slack Channel for help, and stay updated by following @GitBookIO on Twitter or GitBook on Facebook.

FAQ

Some questions are frequently asked. If you have a problem you should check this out first.

Contribute to this documentation

You can contribute to improve this documentation on GitHub by signaling issues or proposing changes.

Setup and Installation of GitBook

Getting GitBook installed and ready-to-go should only take a few minutes.

legacy.gitbook.com

legacy.gitbook.com is an easy to use solution to write, publish and host books. It is the easiest solution for publishing your content and collaborating on it.

It integrates well with the GitBook Editor.

Local Installation

Requirements

Installing GitBook is easy and straightforward. Your system just needs to meet these two requirements:

- NodeJS (v4.0.0 and above is recommended)
- Windows, Linux, Unix, or Mac OS X

Install with NPM

The best way to install GitBook is via **NPM**. At the terminal prompt, simply run the following command to install GitBook:

```
$ npm install gitbook-cli -g
```

`gitbook-cli` is an utility to install and use multiple versions of GitBook on the same system. It will automatically install the required version of GitBook to build a book.

Create a book

GitBook can setup a boilerplate book:

```
$ gitbook init
```

If you wish to create the book into a new directory, you can do so by running `gitbook init ./directory`

Preview and serve your book using:

```
$ gitbook serve
```

Or build the static website using:

```
$ gitbook build
```

Install pre-releases

`gitbook-cli` makes it easy to download and install other versions of GitBook to test with your book:

```
$ gitbook fetch beta
```

Use `gitbook ls-remote` to list remote versions available for install.

Debugging

You can use the options `--log=debug` and `--debug` to get better error messages (with stack trace). For example:

```
$ gitbook build ./ --log=debug --debug
```

Directory Structure

GitBook uses a simple directory structure. All Markdown/Asciidoc files listed in the SUMMARY will be transformed as HTML. Multi-Lingual books have a slightly different structure.

A basic GitBook usually looks something like this:

```
.
├── book.json
├── README.md
├── SUMMARY.md
├── chapter-1/
|   ├── README.md
|   └── something.md
└── chapter-2/
    ├── README.md
    └── something.md
```

An overview of what each of these does:

File	Description
`book.json`	Stores configuration data (**optional**)
`README.md`	Preface / Introduction for your book (**required**)
`SUMMARY.md`	Table of Contents (See Pages) (**optional**)
`GLOSSARY.md`	Lexicon / List of terms to annotate (See Glossary) (**optional**)

Static files and Images

A static file is a file that is not listed in the `SUMMARY.md`. All static files, unless ignored, are copied to the output.

Ignoring files & folders

GitBook will read the `.gitignore`, `.bookignore` and `.ignore` files to get a list of files and folders to skip. The format inside those files, follows the same convention as `.gitignore`:

```
# This is a comment

# Ignore the file test.md
test.md

# Ignore everything in the directory "bin"
bin/*
```

Project integration with subdirectory

For software projects, you can use a subdirectory (like `docs/`) to store the book for the project's documentation. You can configure the `root` option to indicate the folder where GitBook can find the book's files:

```
.
├── book.json
└── docs/
    ├── README.md
    └── SUMMARY.md
```

With `book.json` containing:

```
{
    "root": "./docs"
}
```

Pages and Summary

Summary

GitBook uses a `SUMMARY.md` file to define the structure of chapters and subchapters of the book. The `SUMMARY.md` file is used to generate the book's table of contents.

The format of `SUMMARY.md` is just a list of links. The link's title is used as the chapter's title, and the link's target is a path to that chapter's file.

Adding a nested list to a parent chapter will create subchapters.

Simple example

```
# Summary

* [Part I](part1/README.md)
    * [Writing is nice](part1/writing.md)
    * [GitBook is nice](part1/gitbook.md)
* [Part II](part2/README.md)
    * [We love feedback](part2/feedback_please.md)
    * [Better tools for authors](part2/better_tools.md)
```

Each chapter has a dedicated page (`part#/README.md`) and is split into subchapters.

Anchors

Chapters in the Table of Contents can be pointing to specific part of a file using anchor.

```
# Summary

### Part I

* [Part I](part1/README.md)
    * [Writing is nice](part1/README.md#writing)
    * [GitBook is nice](part1/README.md#gitbook)
* [Part II](part2/README.md)
    * [We love feedback](part2/README.md#feedback)
    * [Better tools for authors](part2/README.md#tools)
```

Parts

The Table of Contents can be divided into parts separated by headings or horizontal lines:

```
# Summary

### Part I

* [Writing is nice](part1/writing.md)
* [GitBook is nice](part1/gitbook.md)

### Part II

* [We love feedback](part2/feedback_please.md)
* [Better tools for authors](part2/better_tools.md)

----

* [Last part without title](part3/title.md)
```

Parts are just groups of chapters and do not have dedicated pages, but according to the theme, it will show in the navigation.

Pages

Markdown syntax

Most of the files for GitBook use the Markdown syntax by default. GitBook infers your pages's structure from it. The syntax used is similar to the GitHub Flavored Markdown syntax. One can also opt for the AsciiDoc syntax.

Example of a chapter file

```
# Title of the chapter

This is a great introduction.

## Section 1

Markdown will dictates _most_ of your **book's structure**

## Section 2

...
```

Front Matter

Pages can contain an optional front matter. It can be used to define the page's description. The front matter must be the first thing in the file and must take the form of valid YAML set between triple-dashed lines. Here is a basic example:

```
---
description: This is a short description of my page
---

# The content of my page
...
```

The front matter can define variables of your own, they will be added to the page variable so you can use them in your templating.

Configuration

GitBook allows you to customize your book using a flexible configuration. These options are specified in a `book.json` file. For authors unfamiliar with the JSON syntax, you can validate the syntax using tools such as JSONlint.

General Settings

Variable	Description
`root`	Path to the root folder containing all the book's files, except `book.json`
`structure`	To specify paths for Readme, Summary, Glossary etc. See Structure paragraph.
`title`	Title of your book, default value is extracted from the README. On legacy.gitbook.com this field is pre-filled.
`description`	Description of your book, default value is extracted from the README. On legacy.gitbook.com this field is pre-filled.
`author`	Name of the author. On legacy.gitbook.com this field is pre-filled.
`isbn`	ISBN of the book
`language`	ISO code of the book's language, default value is `en`
`direction`	Text's direction. Can be `rtl` or `ltr`, the default value depends on the value of `language`
`gitbook`	Version of GitBook that should be used. Uses the SemVer specification and accepts conditions like `">= 3.0.0"`

Plugins

Plugins and their configurations are specified in the `book.json`. See the plugins section for more details.

Since version 3.0.0, GitBook can use themes. See the theming section for more details.

Variable	Description
`plugins`	List of plugins to load
`pluginsConfig`	Configuration for plugins

Structure

In addition to the `root` variable, you can tell Gitbook the name of the files for Readme, Summary, Glossary, Languages (instead of using the default names such as `README.md`). These files must be at the root of your book (or the root of every language book). Paths such as `dir/MY_README.md` are not accepted.

Variable	Description
`structure.readme`	Readme file name (defaults to `README.md`)
`structure.summary`	Summary file name (defaults to `SUMMARY.md`)
`structure.glossary`	Glossary file name (defaults to `GLOSSARY.md`)
`structure.languages`	Languages file name (defaults to `LANGS.md`)

PDF Options

PDF Output can be customized using a set of options in the `book.json`:

Variable	Description

`pdf.pageNumbers`	Add page numbers to the bottom of every page (default is `true`)
`pdf.fontSize`	Base font size (default is `12`)
`pdf.fontFamily`	Base font family (default is `Arial`)
`pdf.paperSize`	Paper size, options are `'a0', 'a1', 'a2', 'a3', 'a4', 'a5', 'a6', 'b0', 'b1', 'b2', 'b3', 'b4', 'b5', 'b6', 'legal', 'letter'` (default is `a4`)
`pdf.margin.top`	Top margin (default is `56`)
`pdf.margin.bottom`	Bottom margin (default is `56`)
`pdf.margin.right`	Right margin (default is `62`)
`pdf.margin.left`	Left margin (default is `62`)

Glossary

Allows you to specify terms and their respective definitions to be displayed as annotations. Based on those terms, GitBook will automatically build an index and highlight those terms in pages.

The `GLOSSARY.md` format is a list of `h2` headings, along with a description paragraph:

```
## Term
Definition for this term

## Another term
With it's definition, this can contain bold text
and all other kinds of inline markup ...
```

Multi-Languages

GitBook supports building books written in multiple languages. Each language should be a sub-directory following the normal GitBook format, and a file named `LANGS.md` should be present at the root of the repository with the following format:

```
# Languages

* [English](en/)
* [French](fr/)
* [Español](es/)
```

Configuration for each language

When a language book (ex: `en`) has a `book.json`, its configuration will extend the main configuration.

The only exception is plugins, plugins are specified globally, and language specific plugins cannot be specified.

Markdown

Most of the examples from this documentation are in Markdown. Markdown is default parser for GitBook, but one can also opt for the AsciiDoc syntax.

Here's an overview of Markdown syntax that you can use with GitBook (same as GitHub with some additions).

Headings

To create a heading, add one to six `#` symbols before your heading text. The number of # you use will determine the size of the heading.

```
# This is an <h1> tag
## This is an <h2> tag
###### This is an <h6> tag
```

GitBook supports a nice way for explicitly setting the header ID. If you follow the header text with an opening curly bracket (separated from the text with a least one space), a hash, the ID and a closing curly bracket, the ID is set on the header. If you use the trailing hash feature of atx style headers, the header ID has to go after the trailing hashes. For example:

```
Hello {#id}
-----

# Hello {#id}

# Hello # {#id}
```

Paragraphs and Line Breaks

A paragraph is simply one or more consecutive lines of text, separated by one or more blank lines. (A blank line is any line that looks like a blank line — a line containing nothing but spaces or tabs is considered blank.) Normal paragraphs should not be indented with spaces or tabs.

```
Here's a line for us to start with.

This line is separated from the one above by two newlines, so it will be a *separate paragraph*.
```

Emphasis

```
*This text will be italic*
_This will also be italic_

**This text will be bold**
__This will also be bold__

~~This text will be crossed out.~~

_You **can** combine them_
```

Lists

Markdown supports ordered (numbered) and unordered (bulleted) lists.

Unordered

Unordered lists use asterisks, pluses, and hyphens — interchangably — as list markers:

```
* Item 1
* Item 2
  * Item 2a
  * Item 2b
```

Ordered

Ordered lists use numbers followed by periods:

```
1. Item 1
2. Item 2
3. Item 3
   * Item 3a
   * Item 3b
```

Links

Markdown supports two style of links: inline and reference.

A simple link can be created by surrounding the text with square brackets and the link URL with parentheses:

```
This is [an example](http://example.com/ "Title") inline link with a title.

[This link](http://example.net/) has no title attribute.
```

Links can point to relative paths, anchors or absolute urls.

References

There is another way to create links which does not interrupt the text flow. The URL and title are defined using a reference name and this reference name is then used in square brackets instead of the link URL:

```
This is [an example][id] reference-style link.
```

Then, anywhere in the document, you define your link label like this, on a line by itself:

```
[id]: http://example.com/  "Optional Title Here"
```

Images

Images can be created in a similar way than links: just use an exclamation mark before the square brackets. The link text will become the alternative text of the image and the link URL specifies the image source:

```
An image: ![gras](img/image.jpg)
```

Blockquotes

A blockquote is started using the `>` marker followed by an optional space; all following lines that are also started with the blockquote marker belong to the blockquote. You can use any block-level elements inside a blockquote:

```
As Kanye West said:

> We're living the future so
> the present is our past.
```

Tables

You can create tables by assembling a list of words and dividing them with hyphens `-` (for the first row), and then separating each column with a pipe `|`:

```
| First Header  | Second Header |
| ------------- | ------------- |
| Content Cell  | Content Cell  |
| Content Cell  | Content Cell  |
```

The pipes on either end of the table are optional. Cells can vary in width and do not need to be perfectly aligned within columns. There must be at least three hyphens in each column of the header row.

Code

Markdown supports two different code block styles. One uses lines indented with either four spaces or one tab whereas the other uses lines with tilde characters as delimiters – therefore the content does not need to be indented:

```
This is a sample code block.

    Continued here.
```

Fenced code blocks

You can create fenced code blocks by placing triple backticks ```` ``` ```` before and after the code block. We recommend placing a blank line before and after code blocks to make the raw formatting easier to read.

````
```
function test() {
  console.log("notice the blank line before this function?");
}
```
````

Syntax highlighting

You can add an optional language identifier to enable syntax highlighting in your fenced code block.

For example, to syntax highlight Ruby code:

````
```ruby
require 'redcarpet'
markdown = Redcarpet.new("Hello World!")
puts markdown.to_html
```
````

Inline code

Text phrases can be marked up as code by surrounding them with backticks:

```
Use `gitbook` to convert the `text` in markdown
syntax to HTML.
```

Footnotes

GitBook supports a simple syntax for such footnotes. Footnotes are relative to each pages.

```
Text prior to footnote reference.[^2]

[^2]: Comment to include in footnote.
```

HTML

GitBook supports use of raw HTML in your text, Markdown syntax in HTML is not processed:

```
<div>
Markdown here will not be **parsed**
</div>
```

Horizontal Rule

Horizontal Rules can be inserted using three or more asterisks, dashes or underscores, optionally separated by spaces or tabs, on an otherwise blank line:

```
Three or more...

---

Hyphens

***

Asterisks
```

Ignoring Markdown formatting

You can tell GitBook to ignore (or escape) Markdown formatting by using `\` before the Markdown character.

```
Let's rename \*our-new-project\* to \*our-old-project\*.
```

AsciiDoc

Since version `2.0.0`, GitBook can also accept AsciiDoc as an input format.

Please refer to the AsciiDoc Syntax Quick Reference for more informations about the format.

Just like for markdown, GitBook is using some special files to extract structures: `README.adoc`, `SUMMARY.adoc`, `LANGS.adoc` and `GLOSSARY.adoc`.

README.adoc

This is the main entry of your book: the introduction. This file is **required**.

SUMMARY.adoc

This file defines the list of chapters and subchapters. Just like in Markdown, the `SUMMARY.adoc`'s format is simply a list of links, the name of the link is used as the chapter's name, and the target is a path to that chapter's file.

Subchapters are defined simply by adding a nested list to a parent chapter.

```
= Summary

. link:chapter-1/README.adoc[Chapter 1]
.. link:chapter-1/ARTICLE1.adoc[Article 1]
.. link:chapter-1/ARTICLE2.adoc[Article 2]
... link:chapter-1/ARTICLE-1-2-1.adoc[Article 1.2.1]
. link:chapter-2/README.adoc[Chapter 2]
. link:chapter-3/README.adoc[Chapter 3]
. link:chapter-4/README.adoc[Chapter 4]
.. Unfinished article
. Unfinished Chapter
```

LANGS.adoc

For Multi-Languages books, this file is used to define the different supported languages and translations.

This file is following the same syntax as the `SUMMARY.adoc`:

```
= Languages

. link:en/[English]
. link:fr/[French]
```

GLOSSARY.adoc

This file is used to define terms. See the glossary section.

```
= Glossary

== Magic

Sufficiently advanced technology, beyond the understanding of the
observer producing a sense of wonder.

== PHP

A popular web programming language, used by many large websites such
as Facebook. Rasmus Lerdorf originally created PHP in 1994 to power
his personal homepage (PHP originally stood for "Personal Home Page"
but now stands for "PHP: Hypertext Preprocessor").
```

Generating eBooks and PDFs

GitBook can generates a website, but can also output content as ebook (ePub, Mobi, PDF).

```
# Generate a PDF file
$ gitbook pdf ./ ./mybook.pdf

# Generate an ePub file
$ gitbook epub ./ ./mybook.epub

# Generate a Mobi file
$ gitbook mobi ./ ./mybook.mobi
```

Installing ebook-convert

`ebook-convert` is required to generate ebooks (epub, mobi, pdf).

OS X

Download the Calibre application. After moving the `calibre.app` to your Applications folder create a symbolic link to the ebook-convert tool:

```
$ sudo ln -s ~/Applications/calibre.app/Contents/MacOS/ebook-convert /usr/bin
```

You can replace `/usr/bin` with any directory that is in your $PATH.

Cover

Covers are used for all the ebook formats. You can either provide one yourself, or generate one using the autocover plugin.

To provide a cover, place a **`cover.jpg`** file at the root directory of your book. Adding a **`cover_small.jpg`** will specify a smaller version of the cover. The cover should be a **JPEG** file.

A good cover should respect the following guidelines:

- Size of 1800x2360 pixels for `cover.jpg`, 200x262 for `cover_small.jpg`
- No border
- Clearly visible book title
- Any important text should be visible in the small version

Templating

GitBook uses the Nunjucks templating language to process pages and theme's templates.

The Nunjucks syntax is very similar to **Jinja2** or **Liquid**. Its syntax uses surrounding braces `{ }` to mark content that needs to be processed.

Variables

A variable looks up a value from the template context. If you wanted to simply display a variable, you would use the `{{ variable }}` syntax. For example :

```
My name is {{ name }}, nice to meet you
```

This looks up username from the context and displays it. Variable names can have dots in them which lookup properties, just like JavaScript. You can also use the square bracket syntax.

```
{{ foo.bar }}
{{ foo["bar"] }}
```

If a value is undefined, nothing is displayed. The following all output nothing if foo is undefined: `{{ foo }}`, `{{ foo.bar }}`, `{{ foo.bar.baz }}`.

GitBook provides a set of predefined variables from the context.

Filters

Filters are essentially functions that can be applied to variables. They are called with a pipe operator (`|`) and can take arguments.

```
{{ foo | title }}
{{ foo | join(",") }}
{{ foo | replace("foo", "bar") | capitalize }}
```

The third example shows how you can chain filters. It would display "Bar", by first replacing "foo" with "bar" and then capitalizing it.

Tags

if

`if` tests a condition and lets you selectively display content. It behaves exactly as JavaScript's `if` behaves.

```
{% if variable %}
  It is true
{% endif %}
```

If variable is defined and evaluates to true, "It is true" will be displayed. Otherwise, nothing will be.

You can specify alternate conditions with `elif` and `else`:

```
{% if hungry %}
  I am hungry
{% elif tired %}
  I am tired
{% else %}
  I am good!
{% endif %}
```

for

`for` iterates over arrays and dictionaries.

```
# Chapters about GitBook

{% for article in glossary.terms['gitbook'].articles %}
* [{{ article.title }}]({{ article.path }})
{% endfor %}
```

set

`set` lets you create/modify a variable.

```
{% set softwareVersion = "1.0.0" %}

Current version is {{ softwareVersion }}.
[Download it](website.com/download/{{ softwareVersion }})
```

include and block

Inclusion and inheritance is detailled in the Content References section.

Escaping

If you want GitBook to ignore any of the special templating tags, you can use raw and anything inside of it will be output as plain text.

```
{% raw %}
  this will {{ not be processed }}
{% endraw %}
```

Content References

Content referencing (conref) is a convenient mechanism to reuse content from other files or books.

Importing local files

Importing an other file's content is easy using the `include` tag:

```
{% include "./test.md" %}
```

Importing file from another book

GitBook can also resolve the include path by using git:

```
{% include "git+https://github.com/GitbookIO/documentation.git/README.md#0.0.1" %}
```

The format of git url is:

```
git+https://user@hostname/owner/project.git/file#commit-ish
```

The real git url part should finish with `.git`, the filename to import is extracted after the `.git` till the fragment of the url.

The `commit-ish` can be any tag, sha, or branch which can be supplied as an argument to `git checkout`. The default is `master`.

Inheritance

Template inheritance is a way to make it easy to reuse templates. When writing a template, you can define "blocks" that child templates can override. The inheritance chain can be as long as you like.

`block` defines a section on the template and identifies it with a name. Base templates can specify blocks and child templates can override them with new content.

```
{% extends "./mypage.md" %}

{% block pageContent %}
# This is my page content
{% endblock %}
```

In the file `mypage.md`, you should specify the blocks that can be extended:

```
{% block pageContent %}
This is the default content
{% endblock %}

# License

{% include "./LICENSE" %}
```

Variables

The following is a reference of the available data during book's parsing and theme generation.

Global Variables

Variable	Description
`book`	Book-wide information + configuration settings from `book.json`. See below for details.
`gitbook`	GitBook specific information
`page`	Current page specific information
`file`	File associated with the current page specific information
`readme`	Information about the Readme
`glossary`	Information about the Glossary
`summary`	Information about the table of contents
`languages`	List of languages for multi-lingual books
`output`	Information about the output generator
`config`	Dump of the `book.json`

Book Variables

Variable	Description
`book.[CONFIGURATION_DATA]`	All the `variables` set via the `book.json` are available through the book variable.
`book.language`	Current language for a multilingual book

GitBook Variables

Variable	Description
`gitbook.time`	The current time (when you run the `gitbook` command) .
`gitbook.version`	Version of GitBook used to generate the book

File Variables

Variable	Description
`file.path`	The path to the raw page
`file.mtime`	Modified Time. Last time the file was modified
`file.type`	The name of the parser used to compile this file (ex: `markdown` , `asciidoc` , etc)

Page Variables

Variable	Description
`page.title`	Title of the page
`page.previous`	Previous page in the Table of Contents (can be `null`)

`page.next`	Next page in the Table of Contents (can be `null`)
`page.dir`	Text direction, based on configuration or detected from content (`rtl` or `ltr`)

Table of Contents Variables

Variable	Description
`summary.parts`	List of sections in the Table of Contents

The whole table of contents (`SUMMARY.md`) can be accessed:

`summary.parts[0].articles[0].title` will return the title of the first article.

Multi-lingual book Variable

Variable	Description
`languages.list`	List of languages for this book

Languages are defined by `{ id: 'en', title: 'English' }`.

Output Variables

Variable	Description
`output.name`	Name of the output generator, possible values are `website`, `json`, `ebook`
`output.format`	When `output.name == "ebook"`, `format` defines the ebook format that will be generated, possible values are `pdf`, `epub` or `mobi`

Readme Variables

Variable	Description
`readme.path`	Path to the Readme in the book

Glossary Variables

Variable	Description
`glossary.path`	Path to the Glossary in the book

Builtin Templating Helpers

GitBook provides a serie of builtin filters and blocks to help you write templates.

Filters

`value|default(default, [boolean])` If value is strictly undefined, return default, otherwise value. If boolean is true, any JavaScript falsy value will return default (false, "", etc)

`arr|sort(reverse, caseSens, attr)` Sort arr with JavaScript's arr.sort function. If reverse is true, result will be reversed. Sort is case-insensitive by default, but setting caseSens to true makes it case-sensitive. If attr is passed, will compare attr from each item.

Blocks

`{% markdown %}Markdown string{% endmarkdown %}` Render inline markdown

`{% asciidoc %}AsciiDoc string{% endasciidoc %}` Render inline asciidoc

Plugins

Plugins are the best way to extend GitBook functionalities (ebook and website). There exist plugins to do a lot of things: bring math formulas display support, track visits using Google Analytic, etc.

How to find plugins?

Plugins can be easily searched on plugins.gitbook.com.

How to install a plugin?

Once you find a plugin that you want to install, you need to add it to your `book.json` :

```
{
    "plugins": ["myPlugin", "anotherPlugin"]
}
```

You can also specify a specific version using: `"myPlugin@0.3.1"`. By default GitBook will resolve the latest version of the plugin compatbile with the current GitBook version.

legacy.gitbook.com

Plugins are automatically installed on legacy.gitbook.com. Locally, run `gitbook install` to install and prepare all plugins for your books.

Configuring plugins

Plugins specific configurations are stored in `pluginsConfig`. You have to refer to the documentation of the plugin itself for details about the available options.

Create and publish a plugin

A GitBook plugin is a node package published on NPM that follow a defined convention.

Structure

package.json

The `package.json` is a manifest format for describing **Node.js modules**. GitBook plugins are built on top of Node modules. It declares dependencies, version, ownership, and other information required to run a plugin in GitBook. This document describes the schema in detail.

A plugin manifest `package.json` can also contain details about the required configuration. The configuration schema is defined in the `gitbook` field of the `package.json` (This field follow the JSON-Schema guidelines):

```
{
    "name": "gitbook-plugin-mytest",
    "version": "0.0.1",
    "description": "This is my first GitBook plugin",
    "engines": {
        "gitbook": ">1.x.x"
    },
    "gitbook": {
        "properties": {
            "myConfigKey": {
                "type": "string",
                "default": "it's the default value",
                "description": "It defines my awesome config!"
            }
        }
    }
}
```

You can learn more about `package.json` from the NPM documentation.

The **package name** must begin with `gitbook-plugin-` and the **package engines** should contains `gitbook`.

index.js

The `index.js` is main entry point of your plugin runtime:

```
module.exports = {
    // Map of hooks
    hooks: {},

    // Map of new blocks
    blocks: {},

    // Map of new filters
    filters: {}
};
```

Publish your plugin

GitBook plugins can be published on NPM.

To publish a new plugin, you need to create an account on npmjs.com then publish it from the command line:

```
$ npm publish
```

Private plugins

Private plugins can be hosted on GitHub and included using `git` urls:

```
{
    "plugins": [
        "myplugin@git+https://github.com/MyCompany/mygitbookplugin.git#1.0.0"
    ]
}
```

Hooks

Hooks is a method of augmenting or altering the behavior of the process, with custom callbacks.

List of hooks

Relative to the global pipeline

Name	Description	Arguments
`init`	Called after parsing the book, before generating output and pages.	None
`finish:before`	Called after generating the pages, before copying assets, cover, ...	None
`finish`	Called after everything else.	None

Relative to the page pipeline

> It is recommended using templating to extend page parsing.

Name	Description	Arguments
`page:before`	Called before running the templating engine on the page	Page Object
`page`	Called before outputting and indexing the page.	Page Object

Page Object

```
{
    // Parser named
    "type": "markdown",

    // File Path relative to book root
    "path": "page.md",

    // Absolute file path
    "rawpath": "/usr/...",

    // Title of the page in the SUMMARY
    "title": "",

    // Content of the page
    // Markdown/Asciidoc in "page:before"
    // HTML in "page"
    "content": "# Hello"
}
```

Example to add a title

In the `page:before` hook, `page.content` is the markdown/asciidoc content.

```
{
    "page:before": function(page) {
        page.content = "# Title\n" +page.content;
        return page;
    }
}
```

Example to replace some html

In the `page` hook, `page.content` is the HTML generated from the markdown/asciidoc conversion.

```
{
    "page": function(page) {
        page.content = page.content.replace("<b>", "<strong>")
            .replace("</b>", "</strong>");
        return page;
    }
}
```

Asynchronous Operations

Hooks callbacks can be asynchronous and return promises.

Example:

```
{
    "init": function() {
        return writeSomeFile()
        .then(function() {
            return writeAnotherFile();
        });
    }
}
```

Extend Blocks

Extending templating blocks is the best way to provide extra functionalities to authors.

The most common usage is to process the content within some tags at runtime. It's like filters, but on steroids because you aren't confined to a single expression.

Defining a new block

Blocks are defined by the plugin, blocks is a map of name associated with a block descriptor. The block descriptor needs to contain at least a `process` method.

```
module.exports = {
    blocks: {
        tag1: {
            process: function(block) {
                return "Hello "+block.body+", How are you?";
            }
        }
    }
};
```

The `process` should return the html content that will replace the tag. Refer to Context and APIs to learn more about `this` and GitBook API.

Handling block arguments

Arguments can be passed to blocks:

```
{% tag1 "argument 1", "argument 2", name="Test" %}
This is the body of the block.
{% endtag1 %}
```

And arguments are easily accessible in the `process` method:

```
module.exports = {
    blocks: {
        tag1: {
            process: function(block) {
                // block.args equals ["argument 1", "argument 2"]
                // block.kwargs equals { "name": "Test" }
            }
        }
    }
};
```

Handling sub-blocks

A defined block can be parsed into different sub-blocks, for example let's consider the source:

```
{% myTag %}
    Main body
    {% subblock1 %}
    Body of sub-block 1
    {% subblock 2 %}
    Body of sub-block 1
{% endmyTag %}
```

Extend Filters

Filters are essentially functions that can be applied to variables. They are called with a pipe operator (`|`) and can take arguments.

```
{{ foo | title }}
{{ foo | join(",") }}
{{ foo | replace("foo", "bar") | capitalize }}
```

Defining a new filter

Plugins can extend filters by defining custom functions in their entry point under the `filters` scope.

A filter function takes as first argument the content to filter, and should return the new content. Refer to Context and APIs to learn more about `this` and GitBook API.

```
module.exports = {
    filters: {
        hello: function(name) {
            return 'Hello '+name;
        }
    }
};
```

The filter `hello` can then be used in the book:

```
{{ "Aaron"|hello }}, how are you?
```

Handling block arguments

Arguments can be passed to filters:

```
Hello {{ "Samy"|fullName("Pesse", man=true}} }}
```

Arguments are passed to the function, named-arguments are passed as a last argument (object).

```
module.exports = {
    filters: {
        fullName: function(firstName, lastName, kwargs) {
            var name = firstName + ' ' + lastName;

            if (kwargs.man) name = "Mr" + name;
            else name = "Mrs" + name;

            return name;
        }
    }
};
```

Context and APIs

GitBooks provides different APIs and contexts to plugins. These APIs can vary according to the GitBook version being used, your plugin should specify the `engines.gitbook` field in `package.json` accordingly.

Book instance

The `Book` class is the central point of GitBook, it centralize all access read methods. This class is defined in book.js.

```
// Read configuration from book.json
var value = book.config.get('title', 'Default Value');

// Resolve a filename to an absolute path
var filepath = book.resolve('README.md');

// Render an inline markup string
book.renderInline('markdown', 'This is **Markdown**')
    .then(function(str) { ... })

// Render a markup string (block mode)
book.renderBlock('markdown', '* This is **Markdown**')
    .then(function(str) { ... })
```

Output instance

The `Output` class represent the output/write process.

```
// Return root folder for the output
var root = output.root();

// Resolve a file in the output folder
var filepath = output.resolve('myimage.png');

// Convert a filename to an URL (returns a path to an html file)
var fileurl = output.toURL('mychapter/README.md');

// Write a file in the output folder
output.writeFile('hello.txt', 'Hello World')
    .then(function() { ... });

// Copy a file to the output folder
output.copyFile('./myfile.jpg', 'cover.jpg')
    .then(function() { ... });

// Verify that a file exists
output.hasFile('hello.txt')
    .then(function(exists) { ... });
```

Page instance

A page instance represent the current parsed page.

```
// Title of the page (from SUMMARY)
page.title

// Content of the page (Markdown/Asciidoc/HTML according to the stage)
page.content

// Relative path in the book
page.path

// Absolute path to the file
```

```
page.rawPath

// Type of parser used for this file
page.type ('markdown' or 'asciidoc')
```

Context for Blocks and Filters

Blocks and filters have access to the same context, this context is bind to the template engine execution:

```
{
    // Current templating syntax
    "ctx": {
        // For example, after a {% set message = "hello" %}
        "message": "hello"
    },

    // Book instance
    "book" <Book>,

    // Output instance
    "output": <Output>
}
```

For example a filter or block function can access the current book using: `this.book`.

Context for Hooks

Hooks only have access to the `<Book>` instance using `this.book`.

Testing your plugin

Testing your plugin locally

Testing your plugin on your book before publishing it is possible using npm link.

In the plugin's folder, run:

```
$ npm link
```

Then in your book's folder:

```
$ npm link gitbook-plugin-<plugin's name>
```

Unit testing on Travis

gitbook-tester makes it easy to write **Node.js/Mocha** unit tests for your plugins. Using Travis.org, tests can be run on each commits/tags.

Theming

Since version 3.0.0, GitBook can be easily themed. Books use the theme-default theme by default.

> **Caution**: Custom theming can block some plugins from working correctly.

Structure of a theme

A theme is a plugin containing templates and assets. Overriding any individual template is optional, since themes always extend the default theme.

Folder	Description
`_layouts`	Main folder containing all the templates
`_layouts/website/page.html`	Template for a normal page
`_layouts/ebook/page.html`	Template for a normal page during ebook generation (PDF< ePub, Mobi)

Extend/Customize theme in a book

Authors can extend the templates of a theme directly from their book's source (without creating an external theme). Templates will be resolved in the `_layouts` folder of the book first, then in the installed plugins/themes.

Extend instead of Forking

When you want to make your theme changes available to multiple books, instead of forking the default theme, you can extend it using the templating syntax:

```
{% extends template.self %}

{% block body %}
    {{ super() }}
    ... This will be added to the "body" block
{% endblock %}
```

Take a look at the API theme for a more complete example.

Publish a theme

Themes are published as plugins (see related docs) with a `theme-` prefix. For example the theme `awesome` will be loaded from the `theme-awesome` plugin, and then from the `gitbook-plugin-theme-awesome` NPM package.

GitBook FAQ

This page gathers common questions and answers concerning the GitBook format and toolchain.

Questions about the legacy platform at legacy.gitbook.com and the Editor are gather into the help.legacy.gitbook.com's FAQ.

How can I host/publish my book?

Books can easily be published and hosted on legacy.gitbook.com. But GitBook output can be hosted on any static file hosting solution.

What can I use to edit my content?

Any text editor should work! But we advise using the GitBook Editor. legacy.gitbook.com also provides a web version of this editor.

Does GitBook supports RTL/bi-directional text ?

The GitBook format supports right to left, and bi-directional writing. To enable it, you either need to specify a language (ex: `ar`), or force GitBook to use RTL in your `book.json`:

```
{
    "language": "ar",
    "direction": "rtl"
}
```

With version 3.0 of GitBook, it's automatically detected according to the content. *Note that, while the output book will indeed respect RTL, the Editor doesn't support RTL writing yet.*

Should I use an `.html` or `.md` extensions in my links?

You should always use paths and the `.md` extensions when linking to your files, GitBook will automatically replace these paths by the appropriate link when the pointing file is referenced in the Table of Contents.

Can I create a GitBook in a sub-directory of my repository?

Yes, GitBooks can be created in sub-directories. legacy.gitbook.com and the CLI also looks by default in a serie of folders.

Does GitBook supports RTL languages?

Yes, GitBook automatically detect the direction in your pages (`rtl` or `ltr`) and adjust the layout accordingly. The direction can also be specified globally in the book.json.

Does GitBook support Math equations?

GitBook supports math equations and TeX thanks to plugins. There are currently 2 official plugins to display math: mathjax and katex.

Can I customize/theme the output?

Yes, both the website and ebook outputs can be customized using themes.

Can I add interactive content (videos, etc)?

GitBook is very extensible. You can use existing plugins or create your own!

Examples

More than 50,000 books have been published on legacy.gitbook.com.

Books

- Front-end Handbook by Cody Lindley
- How to make an Operating System by @SamyPesse
- Building Web Apps with Go by @codegangsta
- Django Girls Tutorial by Django Girls
- Linux Inside by 0xAX
- Learn Javascript by GitBook

Research Papers

- TowCenter Collection by Columbia Journalism School
- Block Relaxation Algorithms in Statistics by Jan de Leeuw

Documentation

- DuckDuckHack Documentation by DuckDuckGo
- Loomio Handbook by Loomio
- Enspiral Handbook by Enspiral
- This documentation