Books.jl

Create books with Julia

Rik Huijzer

Contents

1 About

Basically, this package is a wrapper around Pandoc; similar to Bookdown. Note that Pandoc does the heavy lifting and this package adds features on top. For websites, this package allows for:

- Building a website spanning multiple pages.
- Live reloading the website to see changes quickly; thanks to Pandoc and LiveServer.jl.
- Cross-references from one web page to a section on another page.
- Embedding dynamic output, while still allowing normal Julia package utilities, such as unit testing and live reloading (Revise.jl).
- Showing code blocks as well as output.

If you don't need PDFs or EPUBs, then Franklin.jl is probably a better choice. To create single pages and PDFs containing code blocks, see Weave.jl.

One of the main differences with Franklin.jl, Weave.jl and knitr (Bookdown) is that this package completely decouples the computations from the building of the output. The benefit of this is that you can spawn two separate processes, namely the one to serve your webpages:

```
$ julia --project -e 'using Books; serve()'
Watching ./pandoc/favicon.png
Watching ./src/plots.jl
[...]✓
  LiveServer listening on http://localhost:8001/ ...
  (use CTRL+C to shut down)
```

and the one where you do the computations for your package `Foo`:

```
$ julia --project -e  'using Books; using Foo; M = Foo'

julia> Books.generate_content(; M)
Running example() for _generated/example.md
Running julia_version() for _generated/julia_version.md
Running example_plot() for _generated/example_plot.md
Writing plot images for example_plot
[...]
```

This way, the website remains responsive when the computations are running. Thanks to LiveServer.jl and Pandoc, updating the page after changing text or code takes less than a second. Also, because the `serve` process does relatively few things,

it doesn’t often crash. A drawback of this decoupling is that you need to link your text to the correct computation in the Markdown file, whereas in other packages you would insert the code as a string.

The decoupling also allows the output, which you want to include, to be evaluated inside your package, see Section 3.1. This means that you don’t have to define all your dependencies in a `@setup` (Documenter.jl) or `# hideall` (Franklin.jl / Literate.jl) code block. (Granted, you could work your way around it by only calling methods inside a package.) The dependencies, such as `using DataFrames`, are available from your package. This provides all the benefits which Julia packages normally have, such as unit testing and live reloading via Revise.jl.

2 Getting started

The easiest way to get started is to

1. copy over the files in docs/
2. step inside that directory and
3. serve your book via:

```
$ julia --project -e 'using Books; serve()'
Watching ./pandoc/favicon.png
Watching ./src/plots.jl
[...]✓
  LiveServer listening on http://localhost:8001/ ...
  (use CTRL+C to shut down)
```

To generate all the Julia output (see Section 3.1 for more information) use

```
$ julia --project -e  'using Books; using Foo; M = Foo'

julia> Books.generate_content(; M)
Running example() for _generated/example.md
Running julia_version() for _generated/julia_version.md
Running example_plot() for _generated/example_plot.md
Writing plot images for example_plot
[...]
```

As the number of outputs increases, you might want to only update one output:

```
julia> module Foo
       version() = "This book is built with Julia $VERSION"
       end;

julia> generate_content(Foo.version)
Running version() for _generated/version.md
```

To avoid code duplication between projects, this package tries to have good defaults for many settings. For your project, you can override the default settings by creating `config.toml` and `metadata.yml` files. In summary, the `metadata.yml` file is read by Pandoc while generating the outputs. This file contains settings for the output appearance, author and more, see Section 2.1. The `config.toml` file is read by Books.jl before calling Pandoc, so contains settings which are essentially passed to Pandoc, see Section 2.2. Still, these defaults can be overwritten. If you also want to override the templates, then see Section 2.3.

2.1 metadata.yml

The `metadata.yml` file is read by Pandoc. Settings in this file affect the behaviour of Pandoc and get inserted in the templates. For more info on templates, see Section 2.3. The following default settings are used by Books.jl. You can override settings by placing a `metadata.yml` file at the root directory of your project.

```
---
title: My book
subtitle: My book subtitle
author: John Doe
title-prefix: Book

# This affects things as whether to include a white page before each chapter in the
    PDF.
# See the `.tex` template for more information.
# For reports, set `book: false`.
book: true

# Licenses; can be empty.
html-license: <a href="http://creativecommons.org/licenses/by-sa/4.0/">CC BY-SA
    4.0</a>
tex-license: Creative Commons Attribution-ShareAlike 4.0 International (CC BY-SA
    4.0)

# Link to repository.
repo: https://github.com/johndoe/Book.jl

# Make margins a bit smaller. LaTeX has huge default margins.
geometry:
  - top=20mm
  - left=24mm
  - right=24mm
  - bottom=28mm

# A setting for the PDF. I don't know whether it is important.
lang: en-US

tags: [pandoc, Books.jl, JuliaLang]
number-sections: true
mainfont: DejaVu Sans

# This font contains unicode characters.
monofont: Source Code Pro

code-block-font-size: \scriptsize

titlepage: true
linkReferences: true
bibliography: bibliography.bib
link-citations: true
```

```
# These table of contents settings only affect the PDF.
toc: true
toc-depth: 2

# Cross-reference prefixes.
eqnPrefix: Equation
figPrefix: Figure
tblPrefix: Table
secPrefix: Section
---
```

2.2 config.toml

The `config.toml` file is used by Books.jl. Settings in this file affect how Pandoc is called. In `config.toml`, you can define multiple projects; at least define `projects.default`. The settings of `projects.default` are used when you call `pdf()` or `serve()`. To use other settings, for example the settings for `dev`, use `pdf(project="dev")` or `serve(project="dev")`.

Below, the default configuration is shown. When not defining a `config.toml` file or omitting any of the settings, such as `port`, these defaults will be used. The benefit of multiple projects is, for example, that you can run a `dev` project locally which contains more information than the `default` project. One example could be where you write a paper, book or report and have a page with some notes.

The meaning of `contents` is discussed in Section 2.2.1. The `pdf_filename` is used by `pdf()` and the `port` setting is used by `serve()`.

```
[projects]

  # Default project, used when calling serve() or pdf().
  [projects.default]
  contents = [
    "introduction",
    "analysis",
    "references"
  ]

  # Output pdf or docx filename.
  output_filename = "analysis"

  # Port used by serve().
  port = 8010

  # Alternative project, used when calling, for example, serve(project="develop").
  [projects.dev]
  contents = [
    "introduction",
    "analysis",
```

```
  "notes",
  "references"
]

output_filename = "analysis-with-notes"

port = 8011
```

2.2.1 About contents

The files listed in `contents` are read from the `contents/` directory and passed to Pandoc in the order specified by this list. It doesn't matter whether the files contain headings or at what levels the heading are. Pandoc will just place the texts behind each other.

This list doesn't mention `index.md` located at the root directory of your project. `index.md` is added automatically when generating html output and will be the homepage for the website. It typically contains the link to the generated PDF.

2.3 Templates

Unlike `metadata.yml` and `config.toml`, the default templates should be good for most users. To override these, create one or more of the files listed in Table 2.1.

Table 2.1: Default templates.

File	Description	Affects
`pandoc/style.csl`	citation style	all outputs
`pandoc/style.css`	style sheet	website
`pandoc/template.html`	HTML template	website
`pandoc/template.tex`	PDF template	PDF

Here, the citation style defaults to APA, because it is the only style that I could find that correctly supports parenthetical and in-text citations. For example,

- in-text: Orwell (1945)
- parenthetical: (Orwell, 1945)

For other citation styles from the citation-style-language, users have to manually specify the author in the in-text citations.

3 Demo

We can refer to a section with the normal pandoc-crossref syntax. For example,

> See Section 2.

We can refer to citations such as Orwell (1945) and (Orwell, 1945) or to equations like Equation 3.1.

$$y = sin(x) \tag{3.1}$$

3.1 Embedding code

For embedding code, you can use the `include-files` Lua filter. This package can automatically run methods based on the included filenames. For example, generate a Markdown file `sum.md` with Julia and include it with

Then, in your package, define the method `julia_version`:

```
julia_version() = "This book is built with Julia $VERSION."
```

Next, ensure that you call `Books.generate_content(; M = Foo)`, where `Foo` is the name of your module. This will place the text

```
This book is built with Julia 1.6.0-rc1.
```

at the aforementioned path so that it can be included by Pandoc. Note that it doesn't matter where you define the function `julia_version`, as long as it is in your module.

Of these evaluated methods, the output is passed through `convert_output(path, out::T)` where `T` can, for example, be a DataFrame. To show this, we define a method

```
df_example() = DataFrame(X = [1, 2], Y = ["a", "b"])
```

and add its output to the Markdown file with

Then, it will show

X	Y
1	a
2	b

Use `outputs` to show multiple objects:

```
multiple_df_example() =
    outputs([DataFrame(Z = [3]), DataFrame(U = [4, 5], V = [6, 7])])
```

which will appear as

Z
3

U	V
4	6
5	7

See Section 3.3 for showing multiple plots.

3.2 Showing code blocks

Like in Section 3.1, first define a method like

```
sum_example() = code("""
    a = 3
    b = 4

    a + b
    """)
```

Then, add this method via

which gives as output

```
a = 3
b = 4

a + b
```

```
7
```

Here, how the output should be handled is based on the output type of the function. In this case, the output type is of type `Code`. Methods for other outputs exist too:

```
example_table() = DataFrame(A = [1, 2], B = [3, 4])
```

shows

A	B
1	3
2	4

Alternatively, we can show the same by creating something of type `Code`.

```
code_example_table() = code("""
    using DataFrames

    DataFrame(A = [1, 2], B = [3, 4])
    """)
```

which shows as

```
using DataFrames

DataFrame(A = [1, 2], B = [3, 4])
```

A	B
1	3
2	4

because the output of the code block is of type DataFrame.

In essence, this package doesn't hide the implementation behind synctactic sugar. Instead, this package calls functions and gives you the freedom to decide what to do from there. As an example, we can pass `Module` objects to `code` to evaluate the code block in a specific module.

```
module U end
function module_example()
    code("x = 3"; mod=U)
end
```

When calling `module_example`, it shows as

module: BooksDocs.U

```
x = 3
```

```
3
```

Similarily, we can get the value of x:

module: BooksDocs.U

```
x
```

```
3
```

Unsuprisingly, creating a DataFrame will now fail because we haven't loaded DataFrames

module: BooksDocs.U

```
DataFrame(A = [1])
```

```
UndefVarError(:DataFrame)
```

Which is easy to fix

module: BooksDocs.U

```
using DataFrames

DataFrame(A = [1])
```

A
1

3.3 Plots

Conversions for Gadfly are also included, see Figure 3.1. This is actually a bit tricky, because we want to show vector graphics (SVG) on the web, but these are not supported (well) by LaTeX. Therefore, portable network graphics (PNG) images are passed to LaTeX via cairosvg; I found that this tool does the best conversions without relying on Cairo.jl. (Cairo.jl doesn't work for me on NixOS.)

```
using Gadfly
```

```
X = 1:30
plot(x = X, y = X.^2)
```

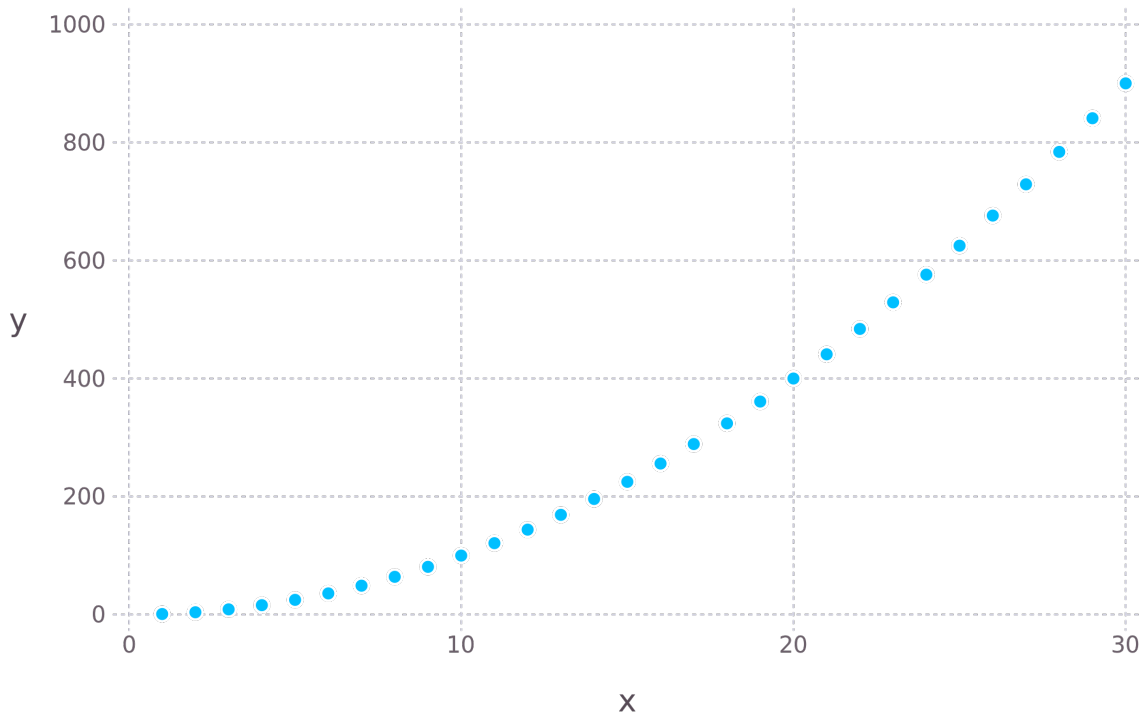

Figure 3.1: Example plot.

If the output is a string instead of the output you expected, then check whether you load the related packages in time. For example, for this Gadfly plot, you need to load Gadfly.jl together with Books.jl for Requires.jl to work.

For multiple images, use `outputs(paths, objects)`:

```
function multiple_example_plots()
    paths = ["example_plot_$i" for i in 2:3]
    X = 1:30
    objects = [
        plot(x = X, y = X),
        plot(x = X, y = X.^3)
    ]
    outputs(paths, objects)
end
```

Resulting in Figure 3.2 and Figure 3.3:

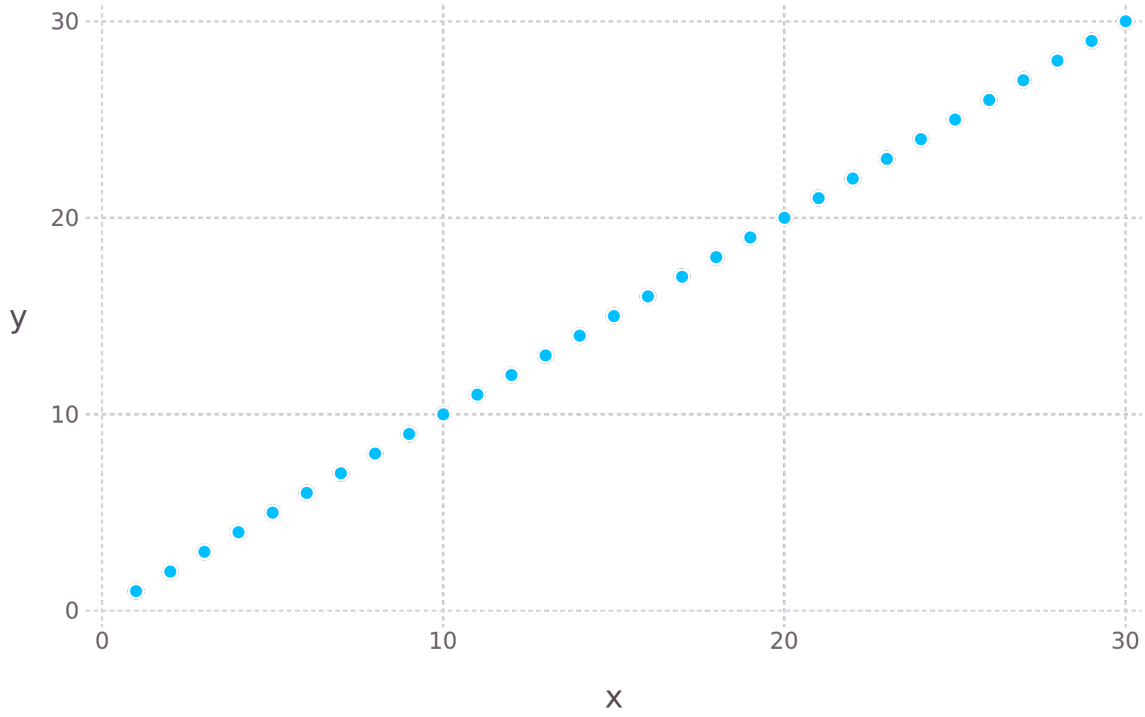

Figure 3.2: Example plot 2.

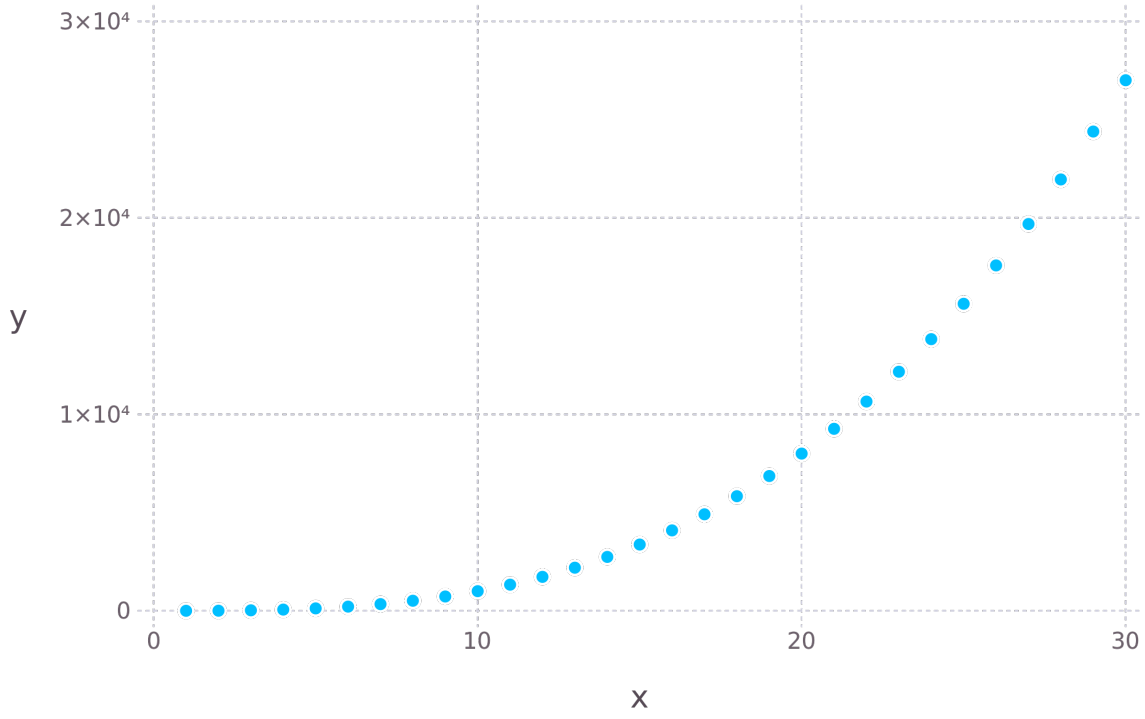

Figure 3.3: Example plot 3.

3.4 Other notes

3.4.1 Level 3 headings

These are hidden from the website menu.

References

Orwell, G. (1945). *Animal Farm: A Fairy Story*. Houghton Mifflin Harcourt.