Books.jl

Create books with Julia

Rik Huijzer and contributors

https://huijzer.xyz/Books.jl/

Version: 2025-06-14

Contents

About

Similar to Bookdown[1], this package wraps around Pandoc[2]. 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[3].
- 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.
- Interacting with code from within the REPL.

If you don't need to generate PDFs, then Franklin.jl[4]is probably a better choice. To create single pages and PDFs containing code blocks, see Weave.jl[5].

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:

```
$ julia --project -ie 'using Books'
```

[1]https://bookdown.org
[2]https://pandoc.org/
[3]https://github.com/tlienart/LiveServer.jl
[4]https://github.com/tlienart/Franklin.jl
[5]https://github.com/JunoLab/Weave.jl

````
julia> gen()
[...]
Updating html
````

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 almost never crashes.

As another benefit, the decoupling allows you to have more flexibility in when you want to run what code. In combination with Revise.jl, you can quickly update your code and see the updated output.

Another reason why this package looks different than other packages is because this package has been aimed at a REPL workflow. Via the REPL, the package evaluates the code blocks inside `Main` by default. This provides easy access to the variables.

Finally, a big difference with this package and other packages is that you decide yourself what you want to show for a code block. For example, in R

~~~
```{r, results='hide'}
print("Hello, world!")
```
~~~

shows the code and not the output. Instead, in Books, you would write

~~~
```jl
s = """print("Hello, world!")"""
sc(s)
```
~~~

which is displayed as

````
print("Hello, world!")
````

Here, `sc` is one of the convenience methods exported by Books.jl. Although this approach is more verbose in some cases, it is also much more flexible. In essence, you can come up with your own pre- or post-processing logic. For example, lets write

~~~
```jl
code = """
    df = DataFrame(a=[1, 2], b=[3, 4])
    Options(df, caption="A table.", label=nothing)
    """
repeat(sco(code), 4)
```
~~~

which shows the code and output (`sco`) 4 times:

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table.", label=nothing)
```

a	b
1	3
2	4

Table 1: A table.

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table.", label=nothing)
```

a	b
1	3
2	4

Table 2: A table.

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table.", label=nothing)
```

a	b
1	3
2	4

Table 3: A table.

```
df = DataFrame(a=[1, 2], b=[3, 4])
Options(df, caption="A table.", label=nothing)
```

a	b
1	3
2	4

Table 4: A table.

Getting started

The easiest way to get started is to use the template repository at https://github.com/JuliaBooks/BookTemplate[6].

From this repository, you can 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 -ie  'using Books; using MyPackage'

julia> gen(; )
[...]
Updating html
```

where `MyPackage` is the name of your package. This evaluates all the code from the code blocks inside your `Main` module. The benefit of this is that you can easy interact with and access variables defined inside code blocks.

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 .

[6]https://github.com/JuliaBooks/BookTemplate

To generate the PDF, use

```
julia> pdf()
```

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 . You can override settings by placing a `metadata.yml` file at the root directory of your project. For example, the metadata for this project contains:

```
---
title: Books.jl
subtitle: Create books with Julia
author:
  - Rik Huijzer
  - and contributors

# An example additional header include for html.
# Note that the url will be updated by \`Books.fix_links\`.
header-includes:
- |
  \`\`\`{=html}
  <link rel="stylesheet" href="/files/style.css"/>
  \`\`\`
mousetrap: true

#
# PDF only settings.
#
pdf-footer: ""

# For example, to add extra packages.
extra-pdf-header: |
  \usepackage{cancel}

# Avoid adding a blank page before each chapter.
disable-cleardoublepage: true

bibliography: bibliography.bib

titlepage-top: ""

titlepage-bottom: |
  \`\`\`{=typst}
  #link("https://huijzer.xyz/Books.jl/")
  \`\`\`
---
```

And, the following defaults are set by Books.jl.

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

# Licenses; can be empty.
html-license: <a href="http://creativecommons.org/licenses/by-nc-sa/4.
0/">CC BY-NC-SA 4.0</a>
pdf-license: Creative Commons Attribution-NonCommercial-ShareAlike 4.0
International

pdf-footer: "\\url{https://github.com/johndoe/Book.jl}"

links-as-notes: true

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

code-block-font-size: \scriptsize

titlepage: true
linkReferences: true
link-citations: true

# These table of contents settings only affect the PDF.
toc: true
tocdepth: 1

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

# Keyboard shortcuts.
mousetrap: true
---
```

Note that Pandoc has a great templating system. In the template `default/template.typ` of this project, I tried to get the basics right and haven't spend hours on making it configurable. However, if you want to add options to the templates, feel free to open a pull request. If you want to configure the template completely by yourself, you can place “template.typ” in “pandoc/” at the root of your project. This same holds for the HTML and CSS template.

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. You don’t have to copy all these defaults, only *override* the settings that you want to change. 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()`. For this documentation, the following config is used

```
[projects]

  [projects.default]
  contents = [
    "about",
    "getting-started",
    "demo",
    "references",
  ]
  output_filename = "books"

  # Full URL, required for the sitemap and robots.txt.
  online_url = "https://huijzer.xyz"
  online_url_prefix = ""
  # Extra directories to be copied.
  extra_directories = [
    "images",
    "files"
  ]

  port = 8012

  [projects.notes]
  contents = [
    "demo",
    "notes",
    "references"
  ]

  # This project is only used when testing Books.jl.
  [projects.test]
  contents = [
    "test"
  ]

  online_url = "https://example.com"
  online_url_prefix = "/Example.jl"
```

Which overrides some settings from the following default settings

```
[projects]

  # Default project, used when calling serve() or pdf().
  [projects.default]
  homepage_contents = "index"

  metadata_path = "metadata.yml"

  contents = [
    "introduction",
    "analysis",
    "references"
  ]

  # Output filename for the PDF.
  output_filename = "analysis"

  # Full URL, required for the sitemap.
  online_url = "https://example.com"

  # Prefix for GitHub or GitLab Pages.
  online_url_prefix = ""
  # Port used by serve().
  port = 8010

  # Extra directories to be copied from the project root into `_build/`.
  extra_directories = []

  # For large books, it can be nice to show some information on the
homepage
  # which is only visible to online visitors and hidden from offline
users (PDF).
  include_homepage_outside_html = false

  # Syntax highlighting.
  highlight = true

  # Alternative project, used when calling, for example,
serve(project="dev").
  [projects.dev]
  homepage_contents = "index"

  metadata_path = "metadata.yml"

  contents = [
    "introduction",
    "analysis",
    "notes",
    "references"
  ]

  output_filename = "analysis-with-notes"

  port = 8011

  extra_directories = []

  include_homepage_outside_html = false
```

Here, the `extra_directories` allows you to specify directories which need to be moved into `_build`, which makes them available for the local server and online. This is, for instance, useful for images like Figure 1 :

```
![Book store.](images/book-store.jpg){#fig:book_store}
```

shows as

Figure 1: Figure 1: Book store.

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 the homepage for the website. That one is specified on a per project basis with `homepage_contents`, which defaults to `index`. The homepage typically contains the link to the generated PDF. Note that the homepage is only added to the html output and not to pdf or other outputs.

Website landing page

By default, Books.jl assumes that you will want a separate landing page for your book when you host it. This page is not added to the generated outputs, like PDF, so it's a nice place to put links to the generated outputs. You will need to create a `index.md` file in the `contents` directory. Then, using an top-level header from Markdown (e.g. "# Title"), give the file a title. Immediately after the title, you need to write `{-}` to avoid this chapter showing up in your HTML menu.

Here is an example of how an example `index.md` file looks like:

```
# My Book's Awesome Title! {-}

Welcome to the landing page for my awesome book!
```

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 1 .

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

Table 5: Table 1: Default templates.

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: G. Orwell [1]
- parenthetical: [1]

For other citation styles from the citation-style-language[7], users have to manually specify the author in the in-text citations.

[7]https://github.com/citation-style-language/styles

Demo

We can refer to a section with the normal pandoc-crossref[8] syntax. For example,

```
See @sec:getting-started.
```

See Section 2 .

```
We can refer to citations such as @orwell1945animal and
[@orwell1949nineteen] or to equations such as @eq:example.
```

We can refer to citations such as G. Orwell [1] and [2] or to equations such as Equation 1 .

```
$$ y = \frac{\sin{x}}{\cos{x}} $$ {#eq:example}
```

$$y = \frac{\sin x}{\cos x} \qquad (1)$$

```
Use single dollar symbols for inline math: $x = 3$.
```

Use single dollar symbols for inline math: $x = 3$.

Embedding output

For embedding code, you can use the `jl` inline code or code block. For example, to show the Julia version, define a code block like

~~~
```jl
YourModule.julia_version()
```
~~~

in a Markdown file. Then, in your package, define the method `julia_version()`:

```
```

[8]https://lierdakil.github.io/pandoc-crossref/

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

Next, call `using Books, MyPackage` and `gen()` to run all the defined in the Markdown files. If you prefer to be less explicit, you can call `gen(; M=YourModule)` to allow for:

~~~
```jl
julia_version()
```
~~~

instead of `YourModule.julia_version()`. When passing your module `M` as keyword argument, `Books.jl` will evaluate all code blocks inside that module.

Alternatively, if you work on a large project and want to only generate the output for one or more Markdown files in `contents/`, such as `index.md`, use

```
gen("index")
```

Calling `gen` will place the text

```
This book is built with Julia 1.11.5.
```

at the right path so that it can be included by Pandoc. You can also embed output inline with single backticks like

```
`jl YourModule.julia_version()`
```

or just call Julia's constant `VERSION` directly from within the Markdown file. For example,

```
This book is built with Julia `jl VERSION`.
```

This book is built with Julia 1.11.5.

While doing this, it is expected that you also have the browser open and a server running, see Section 2 . That way, the page is immediately updated when you run `gen`.

Note that it doesn't matter where you define the function `julia_version`, as long as it is in your module. To save yourself some typing, and to allow yourself to get some coffee while Julia gets up to speed, you can start Julia for your package with

```
$ julia --project -ie 'using MyPackage'
```

which allows you to re-generate all the content by calling

```
julia> gen()
```

To run this method automatically when you make a change in your package, ensure that you loaded `Revise.jl`[9]before loading your package and run

```
entr(gen, ["contents"], [MyPackage])
```

Which will automatically run `gen()` whenever one of the files in `contents/` changes or any code in the `MyPackage` module. To only run `gen` for one file, such as `contents/my_text.md`, use:

```
entr(() -> gen("my_text"), ["contents"], [MyPackage])
```

Or, the equivalent helper function exported by `Books.jl`:

```
entr_gen("my_text"; M=[MyPackage])
```

With this, `gen("my_text")` will be called every time something changes in one of the files in the contents folder or when something changes in `YourModule`. Note that you can run this while `serve` is running in another terminal in the background. Then, your Julia code is executed and the website is automatically updated every time you

[9]https://github.com/timholy/Revise.jl

change something in content or MyPackage. Also note that gen is a drop-in replacement for entr_gen, so you can always add or remove entr_ to run a block one time or multiple times.

In the background, gen passes the methods through convert_output(expr::String, path, out::T) where T can, for example, be a DataFrame or a plot. To show that a DataFrame is converted to a Markdown table, we define a method

```
my_table() = DataFrame(U = [1, 2], V = [:a, :b], W = [3, 4])
```

and add its output to the Markdown file with

~~~
```jl
BooksDocs.my_table()
```
~~~

Then, it will show as

U	V	W
1	a	3
2	b	4

Table 6: Table 2: My table.

where the caption and the label are inferred from the path. Refer to Table 2 with

```
@tbl:my_table
```

To show multiple objects, pass a Vector:

```
function multiple_df_vector()
    [DataFrame(Z = [3]), DataFrame(U = [4, 5], V = [6, 7])]
end
BooksDocs.multiple_df_vector()
```

Z
3

U	V
4	6
5	7

When you want to control where the various objects are saved, use `Options`. This way, you can pass a informative path with plots for which informative captions, cross-reference labels and image names can be determined.

```
function multiple_df_example()
    objects = [
        DataFrame(X = [3, 4], Y = [5, 6]),
        DataFrame(U = [7, 8], V = [9, 10])
    ]
    filenames = ["a", "b"]
    Options.(objects, filenames)
end
BooksDocs.multiple_df_example()
```

X	Y
3	5
4	6

Table 9: Table 3: A.

U	V
7	9
8	10

Table 10: Table 4: B.

To define the labels and/or captions manually, see Section 3.2 . For showing multiple plots, see Section 3.4 .

Most things can be done via functions. However, defining a struct is not possible, because `@sco` cannot locate the struct definition inside the module. Therefore, it is also possible to pass code and specify that you want to evaluate and show code (sc) without showing the output:

~~~
```jl
s = """
    struct Point
        x
        y
    end
    """
sc(s)
```
~~~

which shows as

```
struct Point
    x
    y
end
```

and show code and output (sco). For example,

~~~
```jl
sco("p = Point(1, 2)")
```
~~~

shows as

```
p = Point(1, 2)
```

```
Point(1, 2)
```

Note that this is starting to look a lot like R Markdown where the syntax would be something like

~~~
```{r, results='hide'}
x = rnorm(100)
```
~~~

I guess that there is no perfect way here. The benefit of evaluating the user input directly, as Books.jl is doing, seems to be that it is more extensible if I'm not mistaken. Possibly, the reasoning is that R Markdown needs to convert the output directly, whereas Julia's better type system allows for converting in much later stages, but I'm not sure.

> **Tip**: When using `sco`, the code is evaluated in the `Main` module. This means that the objects, such as the `Point` struct defined above, are available in your REPL after running `gen()`.

Labels and captions

To set labels and captions, wrap your object in `Options`:

```
function options_example()
    df = DataFrame(A=[1], B=[2], C=[3])
    caption = "My DataFrame."
    label = "foo"
    return Options(df; caption, label)
end
BooksDocs.options_example()
```

A	B	C
1	2	3

Table 11: Table 5: My DataFrame.

which can be referred to with

```
@tbl:foo
```

> Table 5

It is also possible to pass only a caption or a label. This package will attempt to infer missing information from the `path`, `caption` or `label` when possible:

```
julia> Books.caption_label("foo_bar()", missing, missing)
(caption = "Foo bar.", label = "foo_bar")

julia> Books.caption_label("foo_bar()", "My caption.", missing)
(caption = "My caption.", label = "foo_bar")
```

```
julia> Books.caption_label("foo_bar()", "My caption.", nothing)
(caption = "My caption.", label = nothing)

julia> Books.caption_label(missing, "My caption.", missing)
(caption = "My caption.", label = nothing)

julia> Books.caption_label(missing, missing, "my_label")
(caption = "My label.", label = "my_label")

julia> Books.caption_label(missing, missing, missing)
(caption = nothing, label = nothing)
```

Obtaining function definitions

So, instead of passing a string which `Books.jl` will evaluate, `Books.jl` can also obtain the code for a method directly. (Thanks to `CodeTracking.@code_string`.) For example, inside our package, we can define the following method:

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
```

To show code and output (sco) for this method, use the `@sco` macro. This macro is exported by Books, so ensure that you have `using Books` in your package.

~~~
```jl
@sco BooksDocs.my_data()
```
~~~

This gives

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
BooksDocs.my_data()
```

A	B	C	D
1	3	5	7
2	4	6	8

Table 12: Table 6: My data.

To only show the source code, use `@sc`:

~~~
```jl
@sc BooksDocs.my_data()
```
~~~

resulting in

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
```

To override options for your output, use the `pre` keyword argument of `@sco`:

~~~
```jl
let
    caption = "This caption is set via the pre keyword."
    pre(out) = Options(out; caption)
    @sco pre=pre my_data()
end
```
~~~

which appears to the reader as:

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
my_data()
```

A	B	C	D
1	3	5	7
2	4	6	8

Table 13: This caption is set via the pre keyword.

See `?sco` for more information. Since we're using methods as code blocks, we can use the code shown in one code block in another. For example, to determine the mean of column A:

~~~
```jl
@sco BooksDocs.my_data_mean(my_data())
```
~~~

shows as

```
function my_data_mean(df::DataFrame)
    Statistics.mean(df.A)
end
BooksDocs.my_data_mean(my_data())
```

1.5

Or, we can show the output inline, namely 1.5, by using

```
`jl BooksDocs.my_data_mean(my_data())`
```

It is also possible to show methods with parameters. For example,

~~~
```jl
@sc BooksDocs.hello("" )
```
~~~

shows

```
hello(name) = "Hello, $name"
```

Now, we can show

```
BooksDocs.hello("World")
```

```


Hello, World
```

Here, the `M` can be a bit confusing for readers. If this is a problem, you can export the method `hello` to avoid it. If you are really sure, you can export all symbols in your module with something like this[10].

Plots

For image types from libraries that `Books.jl` doesn't know about such as plotting types from `Plots.jl` and `Makie.jl`, it is required to extend two methods. First of all, extend `Books.is_image` so that it returns true for the figure type of the respective plotting library. For example for `Plots.jl` set

```
import Books

Books.is_image(plot::Plots.Plot) = true
```

and extend `Books.svg` and `Books.png` too. For example, for `Plots.jl`:

```
Books.svg(svg_path::String, p::Plot) = savefig(p, svg_path)
```

Adding plots to books 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 also created and passed to LaTeX, so set `Books.png` too:

```
Books.png(png_path::String, p::Plot) = savefig(p, png_path)
```

Then, plotting works:

```
function example_plot()
    I = 1:30
    plot(I, I.^2)
end
BooksDocs.example_plot()
```

[10]https://discourse.julialang.org/t/exportall/4970/16

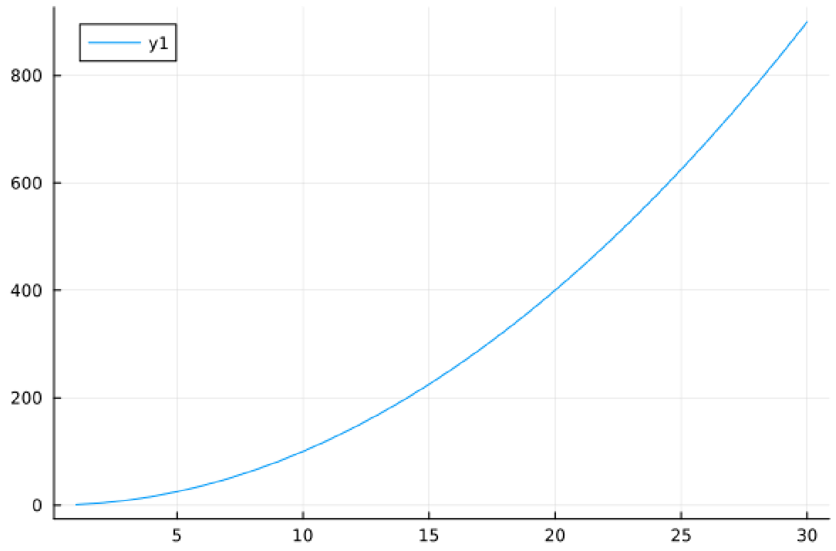

Figure 2: Figure 2: Example plot.

For multiple images, use `Options.(objects, paths)`:

```
function multiple_example_plots()
    filenames = ["example_plot_$i" for i in 2:3]
    I = 1:30
    objects = [
        plot(I, I.^2),
        scatter(I, I.^3)
    ]
    return Options.(objects, filenames)
end
```

Resulting in one `Plots.jl` (Figure 3) and one `CairoMakie.jl` (Figure 4) plot:

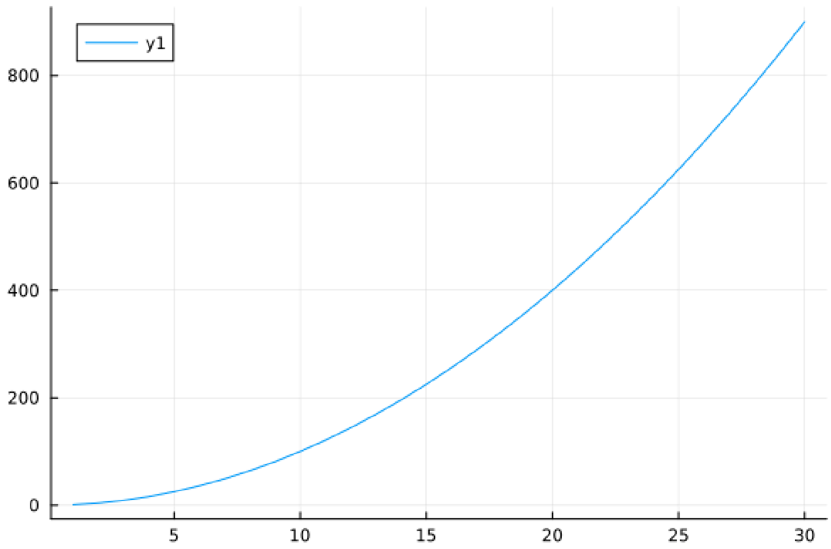

Figure 3: Figure 3: Example plot 2.

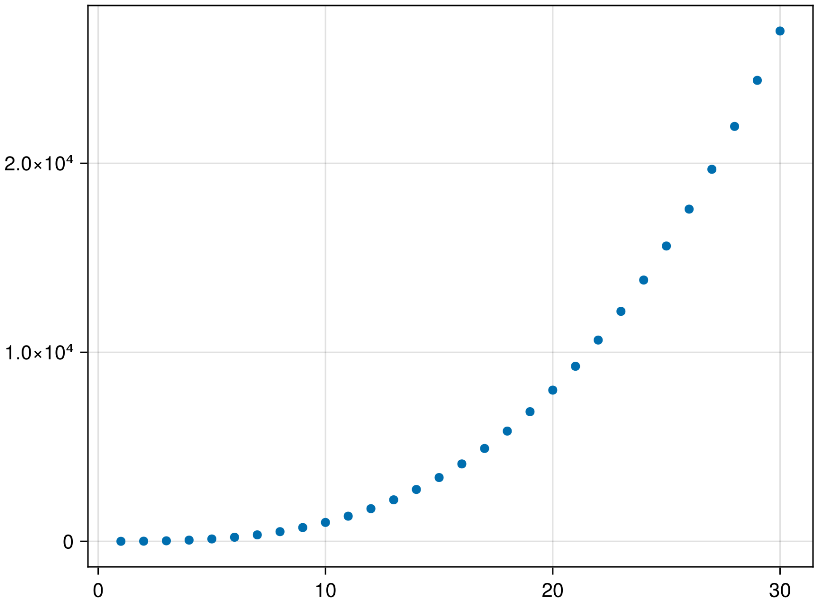

Figure 4: Figure 4: Example plot 3.

To change the size, change the resolution of the image:

```
function image_options_plot()
    I = 1:30
    fig = Figure(; size=(600, 140))
    ax = Axis(fig[1, 1]; xlabel="x", ylabel="y")
    scatterlines!(ax, I, 3 .* sin.(I))
    return fig
end
BooksDocs.image_options_plot()
```

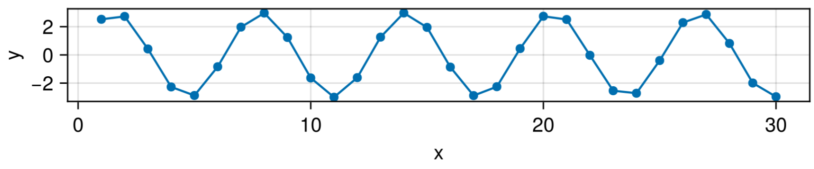

Figure 5: Figure 5: Image options plot.

And, for adjusting the caption, use `Options`:

```
function combined_options_plot()
    fg = image_options_plot()
    Options(fg; caption="Sine function.")
end
BooksDocs.combined_options_plot()
```

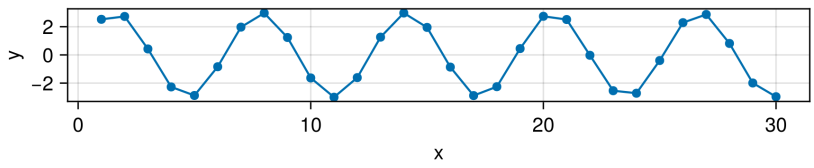

Figure 6: Sine function.

or the caption can be specified in the Markdown file:

~~~
```jl
p = BooksDocs.image_options_plot()
Options(p; caption="Label specified in Markdown.")
```
~~~

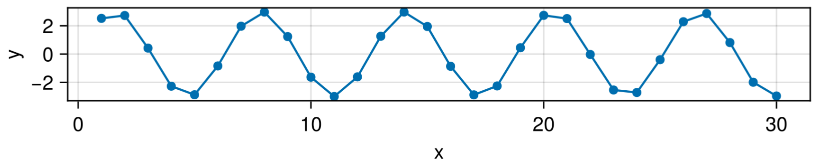

Figure 7: Label specified in Markdown.

```
function plotsjl()
    p = plot(1:10, 1:2:20)
    caption = "An example plot with Plots.jl."
    # Label defaults to `nothing`, which will not create a cross-
reference.
    label = missing
    Options(p; caption, label)
end
BooksDocs.plotsjl()
```

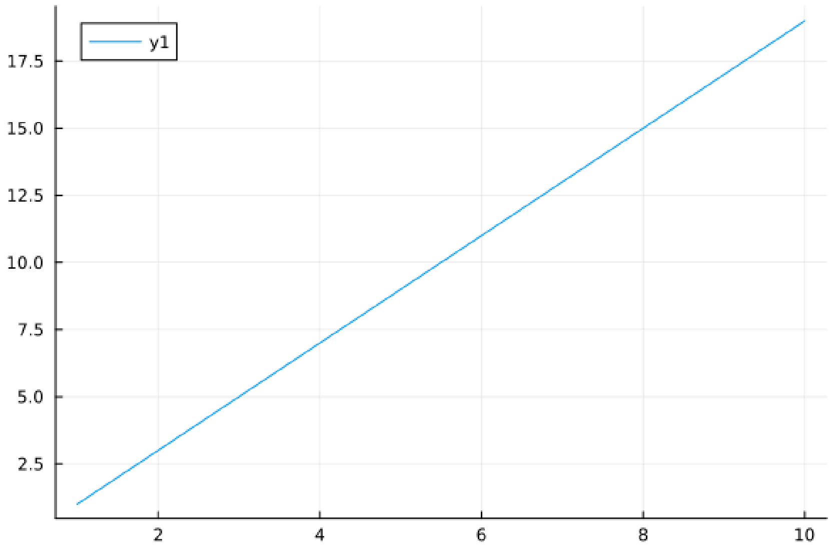

Figure 8: An example plot with Plots.jl.

This time, we also pass `link_attributes` to Pandoc (Figure 6) to shrink the image width on the page:

```
function makiejl()
    x = range(0, 10, length=100)
    y = sin.(x)
    p = lines(x, y)
    caption = "An example plot with Makie.jl."
    label = "makie"
    link_attributes = "width=70%"
    Options(p; caption, label, link_attributes)
end
BooksDocs.makiejl()
```

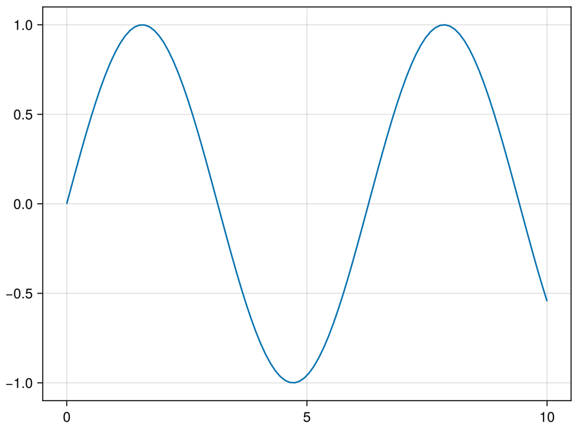

Figure 9: Figure 6: An example plot with Makie.jl.

Other notes

Multilingual books

For an example of a multilingual book setup, say English and Chinese, see https://juliadatascience.io[11].

Footnotes

Footnotes can be added via regular Markdown syntax:

```
Some sentence[^foot].

[^foot]: Footnote text.
```

[11]https://juliadatascience.io

Some sentence[12].

Show

When your method returns an output type `T` which is unknown to Books.jl, it will be passed through `show(io::IO, ::MIME"text/plain", object::T)`. So, if the package that you're using has defined a new `show` method, this will be used. For example, for a grouped DataFrame:

```
groupby(DataFrame(; A=[1]), :A)
```

```
GroupedDataFrame with 1 group based on key: A
Group 1 (1 row): A = 1
 Row │ A
     │ Int64
─────┼───────
   1 │     1
```

Note box

To write note boxes, you can use

```
> **_NOTE:_**  The note content.
```

> ***NOTE:*** The note content.

This way is fully supported by Pandoc, so it will be correctly converted to outputs such as PDF.

Advanced `sco` options

To enforce output to be embedded inside a code block, use `scob`. For example,

```
scob("
df = DataFrame(A = [1], B = [Date(2018)])
string(df)
")
```

[12]Footnote text.

```
df = DataFrame(A = [1], B = [Date(2018)])
string(df)
```

```
1×2 DataFrame
 Row │ A      B
     │ Int64  Date
─────┼───────────────────
   1 │     1  2018-01-01
```

or, with a string

```
s = "Hello"
```

```
Hello
```

Another way to change the output is via the keyword arguments `pre`, `process` and `post` for `sco`. The idea of these arguments is that they allow you to pass a function to alter the processing that Books.jl does. `pre` is applied **before** `Books.convert_output`, `process` is applied **instead** of `Books.convert_output` and `post` is applied **after** `Books.convert_output`. For example, to force books to convert a DataFrame to a string instead of a Markdown table, use:

~~~
```jl
s = "df = DataFrame(A = [1], B = [Date(2018)])"
sco(s; process=string, post=output_block)
```
~~~

which shows the following to the reader:

```
df = DataFrame(A = [1], B = [Date(2018)])
```

```
1×2 DataFrame
 Row │ A      B
     │ Int64  Date
─────┼───────────────────
   1 │     1  2018-01-01
```

Without `process=string`, the output would automatically be converted to a Markdown table by Books.jl and then wrapped inside a code block, which will cause Pandoc to show the raw output instead of a table.

```
df = DataFrame(A = [1], B = [Date(2018)])
```

```
|   A |          B |
| ---:| ----------:|
|   1 | 2018-01-01 |
```

Without `post=output_block`, the DataFrame would be converted to a string, but not wrapped inside a code block so that Pandoc will treat is as normal Markdown:

```
df = DataFrame(A = [2], B = [Date(2018)])
```

Options(1×2 DataFrame Row │ A B │ Int64 Date
──────┼──────────────────────── 1 │ 2 2018-01-01,
missing, nothing, nothing, missing)

This also works for `@sco`. For example, for `my_data` we can use:

~~~
```jl
@sco process=string post=output_block my_data()
```
~~~

which will show as:

```
function my_data()
    DataFrame(A = [1, 2], B = [3, 4], C = [5, 6], D = [7, 8])
end
my_data()
```

```
2×4 DataFrame
 Row │ A      B      C      D
     │ Int64  Int64  Int64  Int64
─────┼────────────────────────────
   1 │     1      3      5      7
   2 │     2      4      6      8
```

Fonts

The code blocks default to JuliaMono in HTML and PDF. For the HTML, this package automatically handles JuliaMono. However, for the PDF, this just doesn't work out (see, e.g., PR #257[13]). To get JuliaMono to work with the PDF build, install it globally. See the instructions at the JuliaMono site[14]. On Linux, you can use `Books.install_extra_fonts()`, but beware that it might override user settings.

Ligatures from JuliaMono are disabled. For example, none of these symbols are combined into a single glyph.

```
|> => and <=
```

Long lines in code blocks

```
When code or output is getting too long, a horizontal scrollbar is
visible on the website to scroll horizontally and a red arrow is visible
in the PDF.
```

Code blocks in lists

To embed code blocks inside lists, indent by 3 spaces and place an empty line before and after the code block. For example, this will show as:

1. This is a list item with some code and output:

   ```
   x = 2 + 1
   ```

   ```

   3
   ```

2. And the list continues

   - with an example on the third level:

[13]https://github.com/JuliaBooks/Books.jl/pull/257
[14]https://juliamono.netlify.app/download/#installation

```
x = 3 + 1
```

```

4
```

- another third level item
- and another

Orwell, George. 1945. *Animal farm: a fairy story*. Houghton Mifflin Harcourt.

———. 1949. *Nineteen eighty-four: a novel*. Secker & Warburg.

Bibliography

[1] G. Orwell, *Animal farm: a fairy story*. Houghton Mifflin Harcourt, 1945.

[2] G. Orwell, *Nineteen eighty-four: a novel.* Secker & Warburg, 1949.