Documenter.jl

Michael Hatherly, Morten Piibeleht, and contributors.

November 7, 2019

Contents

Contents i

I Home 1

1 Documenter.jl 3
- 1.1 Package Features . . . 3
- 1.2 Manual Outline . . . 4
- 1.3 Library Outline . . . 4
- Index . . . 4

II Manual 5

2 Guide 7
- 2.1 Package Guide . . . 7
- Installation . . . 7
- Setting up the Folder Structure . . . 7
- Building an Empty Document . . . 8
- Adding Some Docstrings . . . 9
- Cross Referencing . . . 11
- Navigation . . . 11
- Pages in the Sidebar . . . 12

3 Examples 13
- 3.1 Registered . . . 13
- 3.2 Documentation repositories . . . 14

4 Syntax **15**
4.1 @docs block . . . 15
4.2 @autodocs block . . . 16
4.3 @ref link . . . 17
Duplicate Headers . . . 18
Named doc @refs . . . 18
4.4 @meta block . . . 18
4.5 @index block . . . 19
4.6 @contents block . . . 19
4.7 @example block . . . 20
4.8 @repl block . . . 22
4.9 @setup <name> block . . . 23
4.10 @eval block . . . 23
4.11 @raw <format> block . . . 24

5 Doctests **25**
5.1 "Script" Examples . . . 25
5.2 REPL Examples . . . 26
5.3 Exceptions . . . 26
5.4 Preserving Definitions Between Blocks . . . 27
5.5 Setup Code . . . 27
DocTestSetup in @meta blocks . . . 28
Module-level metadata . . . 28
Block-level setup code . . . 28
5.6 Filtering Doctests . . . 29
5.7 Doctesting as Part of Testing . . . 30
5.8 Fixing Outdated Doctests . . . 30
5.9 Skipping Doctests . . . 30

6 LaTeX Syntax **31**
6.1 Escaping Characters in Docstrings . . . 31
6.2 Inline Equations . . . 32
6.3 Display Equations . . . 32

7 Hosting Documentation **33**
7.1 Overview . . . 33
7.2 Travis CI . . . 34
Authentication: SSH Deploy Keys . . . 34
GitHub Actions . . . 35
7.3 docs/Project.toml . . . 36
7.4 The deploydocs Function . . . 36
7.5 .gitignore . . . 37
7.6 gh-pages Branch . . . 37
7.7 Documentation Versions . . . 37
7.8 Deployment systems . . . 38
7.9 SSH Deploy Keys Walkthrough . . . 40
Generating an SSH Key . . . 40
Adding the Public Key to GitHub . . . 45
Adding the Private Key . . . 45

8 Other Output Formats **47**
8.1 Markdown & MkDocs . . . 47

The MkDocs `mkdocs.yml` file . . . 47
Deployment with MkDocs . . . 48
LaTeX: MkDocs and MathJax . . . 48
8.2 PDF Output via LaTeX . . . 49
Compiling using natively installed latex . . . 49
Compiling using docker image . . . 50

III Showcase **51**

9 Table of contents **55**

10 Basic Markdown **57**

IV Heading 1 **59**

11 Heading 2 **61**
11.1 Heading 3 . . . 61
Heading 4 . . . 61
11.2 Lists . . . 62
11.3 Tables . . . 64
11.4 Footnotes . . . 64
11.5 Headings . . . 64
11.6 Heading level 3 . . . 64
Heading level 4 . . . 64

12 Docstrings **67**
12.1 An index of docstrings . . . 68

13 Doctesting example **69**

14 Running interactive code **71**
14.1 REPL-type . . . 72

15 Doctest showcase **73**

V Library **75**

16 Public **77**
16.1 Public Documentation . . . 77
Contents . . . 77
Index . . . 77
Public Interface . . . 78
DocumenterTools . . . 86

17 Internals **89**
17.1 Anchors . . . 89
17.2 Builder . . . 90
17.3 CrossReferences . . . 91
17.4 DocChecks . . . 92
17.5 DocMeta . . . 92

17.6 DocSystem . . . 93
17.7 DocTests . . . 95
17.8 Documenter . . . 95
17.9 DocumenterTools . . . 95
Generator . . . 96
17.10 Documents . . . 96
17.11 DOM . . . 98
17.12 Expanders . . . 100
17.13 Markdown2 . . . 103
Index . . . 103
Docstrings . . . 103
17.14 MDFlatten . . . 105
17.15 Selectors . . . 105
17.16 TextDiff . . . 108
17.17 Utilities . . . 108
17.18 Writers . . . 111

VI Contributing **119**

18 Branches **123**
18.1 Backports . . . 123
18.2 `release-*` branches . . . 123

19 Style Guide **125**
19.1 Julia . . . 125
19.2 Markdown . . . 126

Part I

Home

Chapter 1

Documenter.jl

A documentation generator for Julia.

A package for building documentation from docstrings and markdown files.

> **Note**
>
> Please read through the Documentation section of the main Julia manual if this is your first time using Julia's documentation system. Once you've read through how to write documentation for your code then come back here.

1.1 Package Features

- Write all your documentation in Markdown.
- Minimal configuration.
- Supports Julia `0.7` and `1.0`.
- Doctests Julia code blocks.
- Cross references for docs and section headers.
- LaTeX syntax support.
- Checks for missing docstrings and incorrect cross references.
- Generates tables of contents and docstring indexes.
- Automatically builds and deploys docs from Travis to GitHub Pages.

The Package Guide provides a tutorial explaining how to get started using Documenter.

Some examples of packages using Documenter can be found on the Examples page.

See the Index for the complete list of documented functions and types.

1.2 Manual Outline

- Package Guide
- Examples
- Syntax
- Doctests
- Hosting Documentation
- LaTeXSyntax

1.3 Library Outline

- Public Documentation
  - Contents
  - Index
  - Public Interface
  - DocumenterTools

Index

- `Documenter`
- `Documenter.Deps`
- `Documenter.DocMeta`
- `Documenter.Deps.pip`
- `Documenter.DocMeta.getdocmeta`
- `Documenter.DocMeta.setdocmeta!`
- `Documenter.Writers.HTMLWriter.asset`
- `Documenter.deploydocs`
- `Documenter.doctest`
- `Documenter.hide`
- `Documenter.makedocs`
- `DocumenterTools.generate`
- `DocumenterTools.genkeys`

Part II

Manual

Chapter 2

Guide

2.1 Package Guide

Documenter is designed to do one thing – combine markdown files and inline docstrings from Julia's docsystem into a single inter-linked document. What follows is a step-by-step guide to creating a simple document.

Installation

Documenter can be installed using the Julia package manager. From the Julia REPL, type `]` to enter the Pkg REPL mode and run

```
pkg> add Documenter
```

Setting up the Folder Structure

> **Note**
>
> The function `DocumenterTools.generate` from the `DocumenterTools` package can generate the basic structure that Documenter expects.

Firstly, we need a Julia module to document. This could be a package generated via `PkgDev.generate` or a single `.jl` script accessible via Julia's `LOAD_PATH`. For this guide we'll be using a package called `Example.jl` that has the following directory layout:

```
Example/├──
 src/│
   └── Example.jl
...
```

Note that the `...` just represent unimportant files and folders.

We must decide on a location where we'd like to store the documentation for this package. It's recommended to use a folder named `docs/` in the toplevel of the package, like so

```
Example/├──
 docs/│
   └── ...├──
 src/│
   └── Example.jl
...
```

Inside the `docs/` folder we need to add two things. A source folder which will contain the markdown files that will be used to build the finished document and a Julia script that will be used to control the build process. The following names are recommended

```
docs/├──
 src/└──
 make.jl
```

Building an Empty Document

With our `docs/` directory now setup we're going to build our first document. It'll just be a single empty file at the moment, but we'll be adding to it later on.

Add the following to your `make.jl` file

```
using Documenter, Example
```

This assumes you've installed Documenter as discussed in Installation and that your `Example.jl` package can be found by Julia.

> **Note**
>
> If your source directory is not accessible through Julia's LOAD_PATH, you might wish to add the following line at the top of make.jl
>
> ```
> 
> ```

Now add an `index.md` file to the `src/` directory.

> **Note**
>
> If you use Documenter's default HTML output the name `index.md` is mandatory. This file will be the main page of the rendered HTML documentation.

Leave the newly added file empty and then run the following command from the `docs/` directory

```
$ julia make.jl
```

Note that `$` just represents the prompt character. You don't need to type that.

If you'd like to see the output from this command in color use

```
$ julia --color=yes make.jl
```

When you run that you should see the following output

```
Documenter: setting up build directory.
Documenter: expanding markdown templates.
Documenter: building cross-references.
Documenter: running document checks.
 > checking for missing docstrings.
 > running doctests.
 > checking footnote links.
Documenter: populating indices.
Documenter: rendering document.
```

The docs/ folder should contain a new directory – called build/. It's structure should look like the following

```
build/├──
 assets/│
   ├── arrow.svg│
   ├── documenter.css│
   ├── documenter.js│
   └── search.js├──
 index.html├──
 search/index.html└──
 search_index.js
```

Note

By default, Documenter has pretty URLs enabled, which means that src/foo.md is turned into src/foo/index.html, instead of simply src/foo.html, which is the preferred way when creating a set of HTML to be hosted on a web server.

However, this can be a hindrance when browsing the documentation locally as browsers do not resolve directory URLs like foo/ to foo/index.html for local files. You have two options:

1. You can run a local web server out of the docs/build directory. If you have Python installed, you can simple start one with python3 -m http.server --bind localhost (or python -m SimpleHTTPServer with Python 2).
2. You can disable the pretty URLs feature by passing prettyurls = false with the Documenter.HTML plugin:

   Alternatively, if your goal is to eventually set up automatic documentation deployment with Travis CI (see Hosting Documentation), you can also use their environment variables to determine Documenter's behavior in make.jl on the fly:

   ```
   makedocs(...,
       format = Documenter.HTML(
           prettyurls = get(ENV, "CI", nothing) == "true"
       )
   ```

Warning

Never git commit the contents of build (or any other content generated by Documenter) to your repository's master branch. Always commit generated files to the gh-pages branch of your repository. This helps to avoid including unnecessary changes for anyone reviewing commits that happen to include documentation changes.

See the Hosting Documentation section for details regarding how you should go about setting this up correctly.

At this point build/index.html should be an empty page since src/index.md is empty. You can try adding some text to src/index.md and re-running the make.jl file to see the changes.

Adding Some Docstrings

Next we'll splice a docstring defined in the Example module into the index.md file. To do this first document a function in that module:

```
module Example

export func

"""
    func(x)

Returns double the number `x` plus `1`.
"""
func(x) = 2x + 1
```

Then in the `src/index.md` file add the following

```
# Example.jl Documentation

```@docs
func(x)
```
```

When we next run `make.jl` the docstring for `Example.func(x)` should appear in place of the `@docs` block in `build/index.md`. Note that more than one object can be referenced inside a `@docs` block – just place each one on a separate line.

Note that a `@docs` block is evaluated in the `Main` module. This means that each object listed in the block must be visible there. The module can be changed to something else on a per-page basis with a `@meta` block as in the following

```
# Example.jl Documentation

```@meta
CurrentModule = Example
```

```@docs
func(x)
```
```

Filtering included docstrings

In some cases you may want to include a docstring for a `Method` that extends a `Function` from a different module – such as `Base`. In the following example we extend `Base.length` with a new definition for the struct `T` and also add a docstring:

```
struct T
    # ...
end

"""
Custom `length` docs for `T`.
"""
```

When trying to include this docstring with

```
```@docs
length
```
```

all the docs for `length` will be included – even those from other modules. There are two ways to solve this problem. Either include the type in the signature with

````
```@docs
length(::T)
```
````

or declare the specific modules that `makedocs` should include with

```
makedocs(
    # options
    modules = [MyModule]
```

Cross Referencing

It may be necessary to refer to a particular docstring or section of your document from elsewhere in the document. To do this we can make use of Documenter's cross-referencing syntax which looks pretty similar to normal markdown link syntax. Replace the contents of `src/index.md` with the following

````
# Example.jl Documentation

```@docs
func(x)
```

- link to [Example.jl Documentation](@ref)
- link to [`func(x)`](@ref)
````

So we just have to replace each link's url with `@ref` and write the name of the thing we'd link to cross-reference. For document headers it's just plain text that matches the name of the header and for docstrings enclose the object in backticks.

This also works across different pages in the same way. Note that these sections and docstrings must be unique within a document.

Navigation

Documenter can auto-generate tables of contents and docstring indexes for your document with the following syntax. We'll illustrate these features using our `index.md` file from the previous sections. Add the following to that file

````
# Example.jl Documentation

```@contents
```

## Functions

```@docs
func(x)
```

## Index

```@index
```
````

The `@contents` block will generate a nested list of links to all the section headers in the document. By default it will gather all the level 1 and 2 headers from every page in the document, but this can be adjusted using `Pages` and `Depth` settings as in the following

````
```@contents
Pages = ["foo.md", "bar.md"]
Depth = 3
```
````

The `@index` block will generate a flat list of links to all the docs that that have been spliced into the document using `@docs` blocks. As with the `@contents` block the pages to be included can be set with a `Pages = [...]` line. Since the list is not nested `Depth` is not supported for `@index`.

Pages in the Sidebar

By default all the pages (`.md` files) in your source directory get added to the sidebar, sorted by their filenames. However, in most cases you want to use the `pages` argument to `makedocs` to control how the sidebar looks like. The basic usage is as follows:

```
makedocs(
    ...,
    pages = [
        "page.md",
        "Page title" => "page2.md",
        "Subsection" => [
            ...
        ]
    ]
```

Using the `pages` argument you can organize your pages into subsections and hide some pages from the sidebar with the help of the `hide` functions.

Chapter 3

Examples

Sometimes the best way to learn how to use a new package is to look for examples of what others have already built with it.

The following packages use Documenter to build their documentation and so should give a good overview of what this package is currently able to do.

> **Note**
>
> Packages are listed alphabetically. If you have a package that uses Documenter then please open a PR that adds it to the appropriate list below; a simple way to do so is to navigate to https://github.com/JuliaDocs/Documenter.jl/edit/master/docs/src/man/examples.md.
>
> The `make.jl` file for all listed packages will be tested to check for potential regressions prior to tagging new Documenter releases whenever possible.

3.1 Registered

Packages that have tagged versions available in `METADATA.jl`.

- Augmentor.jl
- BanditOpt.jl
- BeaData.jl
- Bio.jl
- ControlSystems.jl
- DiscretePredictors.jl
- Documenter.jl
- EvolvingGraphs.jl
- ExtractMacro.jl
- EzXML.jl
- FourierFlows.jl
- Gadfly.jl

- GeoStats.jl
- Highlights.jl
- IntervalConstraintProgramming.jl
- Luxor.jl
- MergedMethods.jl
- Mimi.jl
- NumericSuffixes.jl
- NLOptControl.jl
- OhMyREPL.jl
- OnlineStats.jl
- POMDPs.jl
- PhyloNetworks.jl
- PrivateModules.jl
- Query.jl
- TaylorSeries.jl
- Weave.jl

3.2 Documentation repositories

Some projects or organizations maintain dedicated documentation repositories that are separate from specific packages.

- DifferentialEquations.jl
- JuliaDocs landing page
- JuliaMusic
- Plots.jl

Chapter 4

Syntax

This section of the manual describes the syntax used by Documenter to build documentation. For supported Markdown syntax, see the documentation for the Markdown standard library in the Julia manual.

- Syntax
  - `@docs` block
  - `@autodocs` block
  - `@ref` link
  - `@meta` block
  - `@index` block
  - `@contents` block
  - `@example` block
  - `@repl` block
  - `@setup <name>` block
  - `@eval` block
  - `@raw <format>` block

4.1 `@docs` block

Splice one or more docstrings into a document in place of the code block, i.e.

````
```@docs
Documenter
makedocs
deploydocs
```
````

This block type is evaluated within the `CurrentModule` module if defined, otherwise within `Main`, and so each object listed in the block should be visible from that module. Undefined objects will raise warnings during documentation generation and cause the code block to be rendered in the final document unchanged.

Objects may not be listed more than once within the document. When duplicate objects are detected an error will be raised and the build process will be terminated.

To ensure that all docstrings from a module are included in the final document the `modules` keyword for `makedocs` can be set to the desired module or modules, i.e.

```
makedocs(
    modules = [Documenter],
```

which will cause any unlisted docstrings to raise warnings when `makedocs` is called. If `modules` is not defined then no warnings are printed, even if a document has missing docstrings.

4.2 `@autodocs` block

Automatically splices all docstrings from the provided modules in place of the code block. This is equivalent to manually adding all the docstrings in a `@docs` block.

````
```@autodocs
Modules = [Foo, Bar]
Order   = [:function, :type]
```
````

The above `@autodocs` block adds all the docstrings found in modules `Foo` and `Bar` that refer to functions or types to the document.

Each module is added in order and so all docs from `Foo` will appear before those of `Bar`. Possible values for the `Order` vector are

- `:module`
- `:constant`
- `:type`
- `:function`
- `:macro`

If no `Order` is provided then the order listed above is used.

When a potential docstring is found in one of the listed modules, but does not match any value from `Order` then it will be omitted from the document. Hence `Order` acts as a basic filter as well as sorter.

In addition to `Order`, a `Pages` vector may be included in `@autodocs` to filter docstrings based on the source file in which they are defined:

````
```@autodocs
Modules = [Foo]
Pages   = ["a.jl", "b.jl"]
```
````

In the above example docstrings from module `Foo` found in source files that end in `a.jl` and `b.jl` are included. The page order provided by `Pages` is also used to sort the docstrings. Note that page matching is done using the end of the provided strings and so `a.jl` will be matched by any source file that ends in `a.jl`, i.e. `src/a.jl` or `src/foo/a.jl`.

To filter out certain docstrings by your own criteria, you can provide function with the `Filter` keyword:

````
```@autodocs
Modules = [Foo]
Filter = t -> typeof(t) === DataType && t <: Foo.C
```
````

In the given example, only the docstrings of the subtypes of `Foo.C` are shown. Instead of an anonymous function you can give the name of a function you defined beforehand, too:

````
```@autodocs
Modules = [Foo]
Filter =  myCustomFilterFunction
```
````

To include only the exported names from the modules listed in `Modules` use `Private = false`. In a similar way `Public = false` can be used to only show the unexported names. By default both of these are set to `true` so that all names will be shown.

````
Functions exported from `Foo`:

```@autodocs
Modules = [Foo]
Private = false
Order = [:function]
```

Private types in module `Foo`:

```@autodocs
Modules = [Foo]
Public = false
Order = [:type]
```
````

Note

When more complex sorting is needed then use `@docs` to define it explicitly.

4.3 @ref link

Used in markdown links as the URL to tell Documenter to generate a cross-reference automatically. The text part of the link can be a docstring, header name, or GitHub PR/Issue number.

````
# Syntax

... [`makedocs`](@ref) ...

# Functions

```@docs
makedocs
```

... [Syntax](@ref) ...

... [#42](@ref) ...
````

Plain text in the "text" part of a link will either cross-reference a header, or, when it is a number preceded by a #, a GitHub issue/pull request. Text wrapped in backticks will cross-reference a docstring from a `@docs` block.

`@ref`s may refer to docstrings or headers on different pages as well as the current page using the same syntax.

Note that depending on what the `CurrentModule` is set to, a docstring `@ref` may need to be prefixed by the module which defines it.

Duplicate Headers

In some cases a document may contain multiple headers with the same name, but on different pages or of different levels. To allow `@ref` to cross-reference a duplicate header it must be given a name as in the following example

```
# [Header](@id my_custom_header_name)

...

## Header

... [Custom Header](@ref my_custom_header_name) ...
```

The link that wraps the named header is removed in the final document. The text for a named `@ref ...` does not need to match the header that it references. Named `@ref ...`s may refer to headers on different pages in the same way as unnamed ones do.

Duplicate docstring references do not occur since splicing the same docstring into a document more than once is disallowed.

Named doc `@refs`

Docstring `@refs` can also be "named" in a similar way to headers as shown in the Duplicate Headers section above. For example

```
module Mod

"""
Both of the following references point to `g` found in module `Main.Other`:

  * [`Main.Other.g`](@ref)
  * [`g`](@ref Main.Other.g)

"""
f(args...) = # ...
```

This can be useful to avoid having to write fully qualified names for references that are not imported into the current module, or when the text displayed in the link is used to add additional meaning to the surrounding text, such as

```
Use [`for i = 1:10 ...`](@ref for) to loop over all the numbers from 1 to 10.
```

> **Note**
>
> Named doc `@refs` should be used sparingly since writing unqualified names may, in some cases, make it difficult to tell which function is being referred to in a particular docstring if there happen to be several modules that provide definitions with the same name.

4.4 `@meta` block

This block type is used to define metadata key/value pairs that can be used elsewhere in the page. Currently recognised keys:

- `CurrentModule`: module where Documenter evaluates, for example, `@docs`-block and `@ref`-links.

- `DocTestSetup`: code to be evaluated before a doctest, see the Setup Code section under Doctests.
- `DocTestFilters`: filters to deal with, for example, unpredictable output from doctests, see the Filtering Doctests section under Doctests.
- `EditURL`: link to where the page can be edited. This defaults to the `.md` page itself, but if the source is something else (for example if the `.md` page is generated as part of the doc build) this can be set, either as a local link, or an absolute url.

Example:

````
```@meta
CurrentModule = FooBar
DocTestSetup  = quote
    using MyPackage
end
DocTestFilters = [r"Stacktrace:[\s\S]+"]
EditURL = "link/to/source/file"
```
````

Note that `@meta` blocks are always evaluated in `Main`.

4.5 `@index` block

Generates a list of links to docstrings that have been spliced into a document. Valid settings are `Pages`, `Modules`, and `Order`. For example:

````
```@index
Pages   = ["foo.md"]
Modules = [Foo, Bar]
Order   = [:function, :type]
```
````

When `Pages` or `Modules` are not provided then all pages or modules are included. `Order` defaults to

if not specified. `Order` and `Modules` behave the same way as in `@autodocs` blocks and filter out docstrings that do not match one of the modules or categories specified.

Note that the values assigned to `Pages`, `Modules`, and `Order` may be any valid Julia code and thus can be something more complex that an array literal if required, i.e.

````
```@index
Pages = map(file -> joinpath("man", file), readdir("man"))
```
````

It should be noted though that in this case `Pages` may not be sorted in the order that is expected by the user. Try to stick to array literals as much as possible.

4.6 `@contents` block

Generates a nested list of links to document sections. Valid settings are `Pages` and `Depth`.

````
```@contents
Pages = ["foo.md"]
Depth = 5
```
````

As with `@index` if `Pages` is not provided then all pages are included. The default `Depth` value is 2.

4.7 `@example` block

Evaluates the code block and inserts the result into the final document along with the original source code.

````
```@example
a = 1
b = 2
a + b
```
````

The above `@example` block will splice the following into the final document

````
```julia
a = 1
b = 2
a + b
```

```
3
```
````

Leading and trailing newlines are removed from the rendered code blocks. Trailing whitespace on each line is also removed.

> **Note**
>
> The working directory, `pwd`, is set to the directory in `build` where the file will be written to, and the paths in `include` calls are interpreted to be relative to `pwd`. This can be customized with the `workdir` keyword of `makedocs`.

Hiding Source Code

Code blocks may have some content that does not need to be displayed in the final document. `# hide` comments can be appended to lines that should not be rendered, i.e.

````
```@example
import Random # hide
Random.seed!(1) # hide
A = rand(3, 3)
b = [1, 2, 3]
A \ b
```
````

Note that appending `# hide` to every line in an `@example` block will result in the block being hidden in the rendered document. The results block will still be rendered though. `@setup` blocks are a convenient shorthand for hiding an entire block, including the output.

`stdout` and `stderr`

The Julia output streams are redirected to the results block when evaluating `@example` blocks in the same way as when running doctest code blocks.

`nothing` Results

When the `@example` block evaluates to `nothing` then the second block is not displayed. Only the source code block will be shown in the rendered document. Note that if any output from either `stdout` or `stderr` is captured then the results block will be displayed even if `nothing` is returned.

Named `@example` Blocks

By default `@example` blocks are run in their own anonymous `Module`s to avoid side-effects between blocks. To share the same module between different blocks on a page the `@example` can be named with the following syntax

````
```@example 1
a = 1
```

```@example 1
println(a)
```
````

The name can be any text, not just integers as in the example above, i.e. `@example foo`.

Named `@example` blocks can be useful when generating documentation that requires intermediate explanation or multimedia such as plots as illustrated in the following example

````
First we define some functions

```@example 1
using PyPlot # hide
f(x) = sin(2x) + 1
g(x) = cos(x) - x
```

and then we plot `f` over the interval from π``-`` to π````

```@example 1
x = linspaceπ(-, π)
plot(x, f(x), color = "red")
savefig("f-plot.svg"); nothing # hide
```

![](f-plot.svg)

and then we do the same with `g`

```@example 1
plot(x, g(x), color = "blue")
savefig("g-plot.svg"); nothing # hide
```

![](g-plot.svg)
````

Note that `@example` blocks are evaluated within the directory of `build` where the file will be rendered . This means than in the above example `savefig` will output the `.svg` files into that directory. This allows the images to be easily referenced without needing to worry about relative paths.

`@example` blocks automatically define `ans` which, as in the Julia REPL, is bound to the value of the last evaluated expression. This can be useful in situations such as the following one where where binding the object returned by `plot` to a named variable would look out of place in the final rendered documentation:

````
```@example
using Gadfly # hide
plot([sin, x -> 2sin(x) + x], π-2, π2)
draw(SVG("plot.svg", 6inch, 4inch), ans); nothing # hide
```

![](plot.svg)
````

Delayed Execution of `@example` Blocks

`@example` blocks accept a keyword argument `continued` which can be set to `true` or `false` (defaults to `false`). When `continued = true` the execution of the code is delayed until the next `continued = false` `@example`-block. This is needed for example when the expression in a block is not complete. Example:

````
```@example half-loop; continued = true
for i in 1:3
    j = i^2
```
Some text explaining what we should do with `j`
```@example half-loop
    println(j)
end
```
````

Here the first block is not complete – the loop is missing the `end`. Thus, by setting `continued = true` here we delay the evaluation of the first block, until we reach the second block. A block with `continued = true` does not have any output.

4.8 `@repl` block

These are similar to `@example` blocks, but adds a `julia>` prompt before each toplevel expression. `;` and `# hide` syntax may be used in `@repl` blocks in the same way as in the Julia REPL and `@example` blocks.

````
```@repl
a = 1
b = 2
a + b
```
````

will generate

````
```julia
julia> a = 1
1

julia> b = 2
2

julia> a + b
3
```
````

Named `@repl <name>` blocks behave in the same way as named `@example <name>` blocks.

Note

The working directory, `pwd`, is set to the directory in `build` where the file will be written to, and the paths in `include` calls are interpreted to be relative to `pwd`. This can be customized with the `workdir` keyword of `makedocs`.

4.9 `@setup <name>` block

These are similar to `@example` blocks, but both the input and output are hidden from the final document. This can be convenient if there are several lines of setup code that need to be hidden.

Note

Unlike `@example` and `@repl` blocks, `@setup` requires a `<name>` attribute to associate it with downstream `@example <name>` and `@repl <name>` blocks.

````
```@setup abc
using RDatasets
using DataFrames
iris = dataset("datasets", "iris")
```

```@example abc
println(iris)
```
````

4.10 `@eval` block

Evaluates the contents of the block and inserts the resulting value into the final document.

In the following example we use the PyPlot package to generate a plot and display it in the final document.

````
```@eval
using PyPlot

x = linspace(-π, π)
y = sin(x)

plot(x, y, color = "red")
savefig("plot.svg")

nothing
```


![](plot.svg)
````

Another example is to generate markdown tables from machine readable data formats such as CSV or JSON.

````
```@eval
using CSV
using Latexify
df = CSV.read("table.csv")
mdtable(df,latex=false)
```
````

Which will generate a markdown version of the CSV file table.csv and render it in the output format.

Note that each `@eval` block evaluates its contents within a separate module. When evaluating each block the present working directory, `pwd`, is set to the directory in `build` where the file will be written to, and the paths in `include` calls are interpreted to be relative to `pwd`.

Also, instead of returning `nothing` in the example above we could have returned a new `Markdown.MD` object through `Markdown.parse`. This can be more appropriate when the filename is not known until evaluation of the block itself.

> **Note**
>
> In most cases `@example` is preferred over `@eval`. Just like in normal Julia code where `eval` should be only be considered as a last resort, `@eval` should be treated in the same way.

4.11 `@raw <format>` block

Allows code to be inserted into the final document verbatim. E.g. to insert custom HTML or LaTeX code into the output.

The `format` argument is mandatory and Documenter uses it to determine whether a particular block should be copied over to the output or not. Currently supported formats are `html` and `latex`, used by the respective writers. A `@raw` block whose `format` is not recognized is usually ignored, so it is possible to have a raw block for each output format without the blocks being duplicated in the output.

The following example shows how SVG code with custom styling can be included into documents using the `@raw` block.

````
```@raw html
<svg style="display: block; margin: 0 auto;" width="5em" heigth="5em">
        <circle cx="2.5em" cy="2.5em" r="2em" stroke="black" stroke-width=".1em" fill="red" />
</svg>
```
````

It will show up as follows, with code having been copied over verbatim to the HTML file.

Chapter 5

Doctests

Documenter will, by default, run `jldoctest` code blocks that it finds and makes sure that the actual output matches what's in the doctest. This can help to avoid documentation examples from becoming outdated, incorrect, or misleading. It's recommended that as many of a package's examples be runnable by Documenter's doctest.

This section of the manual outlines how to go about enabling doctests for code blocks in your package's documentation.

5.1 "Script" Examples

The first, of two, types of doctests is the "script" code block. To make Documenter detect this kind of code block the following format must be used:

````
```jldoctest
a = 1
b = 2
a + b

# output

3
```
````

The code block's "language" must be `jldoctest` and must include a line containing the text `# output`. The text before this line is the contents of the script which is run. The text that appears after `# output` is the textual representation that would be shown in the Julia REPL if the script had been `included`.

The actual output produced by running the "script" is compared to the expected result and any difference will result in `makedocs` throwing an error and terminating.

Note that the amount of whitespace appearing above and below the `# output` line is not significant and can be increased or decreased if desired.

It is possible to suppress the output from the doctest by setting the `output` keyword argument to `false`, for example

````
```jldoctest; output = false
a = 1
b = 2
a + b
````
````

````
# output

3
```
````

Note that the output of the script will still be compared to the expected result, i.e. what is `# output` section, but the `# output` section will be suppressed in the rendered documentation.

5.2 REPL Examples

The other kind of doctest is a simulated Julia REPL session. The following format is detected by Documenter as a REPL doctest:

````
```jldoctest
julia> a = 1
1

julia> b = 2;

julia> c = 3;  # comment

julia> a + b + c
6
```
````

As with script doctests, the code block must have it's language set to `jldoctest`. When a code block contains one or more `julia>` at the start of a line then it is assumed to be a REPL doctest. Semi-colons, `;`, at the end of a line works in the same way as in the Julia REPL and will suppress the output, although the line is still evaluated.

Note that not all features of the REPL are supported such as shell and help modes.

5.3 Exceptions

Doctests can also test for thrown exceptions and their stacktraces. Comparing of the actual and expected results is done by checking whether the expected result matches the start of the actual result. Hence, both of the following errors will match the actual result.

````
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
 in div(::Int64, ::Int64) at ./int.jl:115

julia> div(1, 0)
ERROR: DivideError: integer division error
```
````

If instead the first `div(1, 0)` error was written as

````
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
 in div(::Int64, ::Int64) at ./int.jl:114
```
````

where line 115 is replaced with 114 then the doctest will fail.
````

In the second `div(1, 0)`, where no stacktrace is shown, it may appear to the reader that it is expected that no stacktrace will actually be displayed when they attempt to try to recreate the error themselves. To indicate to readers that the output result is truncated and does not display the entire (or any of) the stacktrace you may write `[...]` at the line where checking should stop, i.e.

````
```jldoctest
julia> div(1, 0)
ERROR: DivideError: integer division error
[...]
```
````

5.4 Preserving Definitions Between Blocks

Every doctest block is evaluated inside its own `module`. This means that definitions (types, variables, functions etc.) from a block can not be used in the next block. For example:

````
```jldoctest
julia> foo = 42
42
```
````

The variable `foo` will not be defined in the next block:

````
```jldoctest
julia> println(foo)
ERROR: UndefVarError: foo not defined
```
````

To preserve definitions it is possible to label blocks in order to collect several blocks into the same module. All blocks with the same label (in the same file) will be evaluated in the same module, and hence share scope. This can be useful if the same definitions are used in more than one block, with for example text, or other doctest blocks, in between. Example:

````
```jldoctest mylabel
julia> foo = 42
42
```
````

Now, since the block below has the same label as the block above, the variable `foo` can be used:

````
```jldoctest mylabel
julia> println(foo)
42
```
````

> **Note**
>
> Labeled doctest blocks do not need to be consecutive (as in the example above) to be included in the same module. They can be interspaced with unlabeled blocks or blocks with another label.

5.5 Setup Code

Doctests may require some setup code that must be evaluated prior to that of the actual example, but that should not be displayed in the final documentation. There are three ways to specify the setup code, each appropriate in a different situation.

`DocTestSetup` in `@meta` blocks

For doctests in the Markdown source files, an `@meta` block containing a `DocTestSetup = ...` value can be used. In the example below, the function `foo` is defined inside a `@meta` block. This block will be evaluated at the start of the following doctest blocks:

````
```@meta
DocTestSetup = quote
    function foo(x)
        return x^2
    end
end
```

```jldoctest
julia> foo(2)
4
```

```@meta
DocTestSetup = nothing
```
````

The `DocTestSetup = nothing` is not strictly necessary, but good practice nonetheless to help avoid unintentional definitions in following doctest blocks.

While technically the `@meta` blocks also work within docstrings, their use there is discouraged since the `@meta` blocks will show up when querying docstrings in the REPL.

> **Historic note**
>
> It used to be that `DocTestSetup`s in `@meta` blocks in Markdown files that included docstrings also affected the doctests in the docstrings. Since Documenter 0.23 that is no longer the case. You should use Module-level metadata or Block-level setup code instead.

Module-level metadata

For doctests that are in docstrings, the exported `DocMeta` module provides an API to attach metadata that applies to all the docstrings in a particular module. Setting up the `DocTestSetup` metadata should be done before the `makedocs` or `doctest` call:

```
using MyPackage, Documenter
DocMeta.setdocmeta!(MyPackage, :DocTestSetup, :(using MyPackage); recursive=true)
```

Block-level setup code

Yet another option is to use the `setup` keyword argument to the `jldoctest` block, which is convenient for short definitions, and for setups needed in inline docstrings.

````
```jldoctest; setup = :(foo(x) = x^2)
julia> foo(2)
4
```
````

Note

The `DocTestSetup` and the `setup` values are **re-evaluated** at the start of each doctest block and no state is shared between any code blocks. To preserve definitions see Preserving Definitions Between Blocks.

5.6 Filtering Doctests

A part of the output of a doctest might be non-deterministic, e.g. pointer addresses and timings. It is therefore possible to filter a doctest so that the deterministic part can still be tested.

A filter takes the form of a regular expression. In a doctest, each match in the expected output and the actual output is removed before the two outputs are compared. Filters are added globally, i.e. applied to all doctests in the documentation, by passing a list of regular expressions to `makedocs` with the keyword `doctestfilters`.

For more fine grained control it is possible to define filters in `@meta` blocks by assigning them to the `DocTestFilters` variable, either as a single regular expression (`DocTestFilters = [r"foo"]`) or as a vector of several regex (`DocTestFilters = [r"foo", r"bar"]`).

An example is given below where some of the non-deterministic output from `@time` is filtered.

````
```@meta
DocTestFilters = r"[0-9\.]+ seconds \(.*\)"
```

```jldoctest
julia> @time [1,2,3,4]
  0.000003 seconds (5 allocations: 272 bytes)
4-element Array{Int64,1}:
 1
 2
 3
 4
```

```@meta
DocTestFilters = nothing
```
````

The `DocTestFilters = nothing` is not strictly necessary, but good practice nonetheless to help avoid unintentional filtering in following doctest blocks.

Another option is to use the `filter` keyword argument. This defines a doctest-local filter which is only active for the specific doctest. Note that such filters are not shared between named doctests either. It is possible to define a filter by a single regex (filter = r"foo") or as a list of regex (filter = [r"foo", r"bar"]). Example:

````
```jldoctest; filter = r"[0-9\.]+ seconds \(.*\)"
julia> @time [1,2,3,4]
  0.000003 seconds (5 allocations: 272 bytes)
4-element Array{Int64,1}:
 1
 2
 3
 4
```
````

Note

The global filters, filters defined in `@meta` blocks, and filters defined with the `filter` keyword argument are all applied to each doctest.

5.7 Doctesting as Part of Testing

Documenter provides the `doctest` function which can be used to verify all doctests independently of manual builds. It behaves like a `@testset`, so it will return a testset if all the tests pass or throw a `TestSetException` if it does not.

For example, it can be used to verify doctests as part of the normal test suite by having e.g. the following in `runtests.jl`:

```
using Test, Documenter, MyPackage
```

By default, it will also attempt to verify all the doctests on manual `.md` files, which it assumes are located under `docs/src`. This can be configured or disabled with the `manual` keyword (see `doctest` for more information).

It can also be included in another testset, in which case it gets incorporated into the parent testset. So, as another example, to test a package that does have separate manual pages, just docstrings, and also collects all the tests into a single testset, the `runtests.jl` might look as follows:

```
using Test, Documenter, MyPackage
@testset "MyPackage" begin
    ... # other tests & testsets
    doctest(MyPackage; manual = false)
    ... # other tests & testsets
```

Note that you still need to make sure that all the necessary Module-level metadata for the doctests is set up before `doctest` is called. Also, you need to add Documenter and all the other packages you are loading in the doctests as test dependencies.

5.8 Fixing Outdated Doctests

To fix outdated doctests, the `doctest` function can be called with `fix = true`. This will run the doctests, and overwrite the old results with the new output. This can be done just in the REPL:

```
julia> using Documenter, MyPackage
```

Alternatively, you can also pass the `doctest = :fix` keyword to `makedocs`.

Note

- The `:fix` option currently only works for LF line endings (`'\n'`)
- It is recommended to `git commit` any code changes before running the doctest fixing. That way it is simple to restore to the previous state if the fixing goes wrong.
- There are some corner cases where the fixing algorithm may replace the wrong code snippet. It is therefore recommended to manually inspect the result of the fixing before committing.

5.9 Skipping Doctests

Doctesting can be disabled by setting the `makedocs` keyword `doctest = false`. This should only be done when initially laying out the structure of a package's documentation, after which it's encouraged to always run doctests when building docs.

Chapter 6

LaTeX Syntax

The following section describes how to add equations written using LaTeX to your documentation.

6.1 Escaping Characters in Docstrings

Since some characters used in LaTeX syntax, such as `$` and `\`, are treated differently in docstrings. They need to be escaped using a `\` character as in the following example:

```
"""
Here's some inline maths: ``\\sqrt[n]{1 + x + x^2 + \\ldots}``.

Here's an equation:

``\\frac{n!}{k!(n - k)!} = \\binom{n}{k}``

This is the binomial coefficient.
"""
```

Note that for equations on the manual pages (in `.md` files) the escaping is not necessary. So, when moving equations between the manual and docstrings, the escaping `\` characters have to the appropriately added or removed.

To avoid needing to escape the special characters in docstrings the `raw""` string macro can be used, combined with `@doc`:

```
@doc raw"""
Here's some inline maths: ``\sqrt[n]{1 + x + x^2 + \ldots}``.

Here's an equation:

``\frac{n!}{k!(n - k)!} = \binom{n}{k}``

This is the binomial coefficient.
"""
```

A related issue is how to add dollar signs to a docstring. They need to be double-escaped as follows:

```
"""
The cost was \\$1.
```

6.2 Inline Equations

````
Here's some inline maths: ``\sqrt[n]{1 + x + x^2 + \ldots}``.
````

which will be displayed as

Here's some inline maths: $\sqrt[n]{1 + x + x^2 + \ldots}$.

6.3 Display Equations

````
Here's an equation:

```math
\frac{n!}{k!(n - k)!} = \binom{n}{k}
```

This is the binomial coefficient.
````

which will be displayed as

Here's an equation:

$$\frac{n!}{k!(n - k)!} = \binom{n}{k}$$

This is the binomial coefficient.

Chapter 7

Hosting Documentation

After going through the Package Guide and Doctests page you will need to host the generated documentation somewhere for potential users to read. This guide will describe how to setup automatic updates for your package docs using either the Travis CI build service or GitHub Actions together with and GitHub Pages for hosting the generated HTML files. This is the same approach used by this package to host its own docs – the docs you're currently reading.

> **Note**
>
> Following this guide should be the final step you take after you are comfortable with the syntax and build process used by `Documenter.jl`. It is recommended that you only proceed with the steps outlined here once you have successfully managed to build your documentation locally with Documenter.
>
> This guide assumes that you already have GitHub and Travis accounts setup. If not then go set those up first and then return here.
>
> It is possible to deploy from other systems than Travis CI or GitHub Actions, see the section on Deployment systems.

7.1 Overview

Once set up correctly, the following will happen each time you push new updates to your package repository:

- Buildbots will start up and run your package tests in a "Test" stage.
- After the Test stage completes, a single bot will run a new "Documentation" stage, which will build the documentation.
- If the documentation is built successfully, the bot will attempt to push the generated HTML pages back to GitHub.

Note that the hosted documentation does not update when you make pull requests; you see updates only when you merge to `master` or push new tags.

In the upcoming sections we describe how to configure the build service to run the documentation build stage. In general it is easiest to choose the same service as the one testing your package. If you don't explicitly select the service with the `deploy_config` keyword argument to `deploydocs` Documenter will try to automatically detect which system is running and use that.

7.2 Travis CI

To tell Travis that we want a new build stage we can add the following to the `.travis.yml` file:

```
jobs:
  include:
    - stage: "Documentation"
      julia: 1.0
      os: linux
      script:
        - julia --project=docs/ -e 'using Pkg; Pkg.develop(PackageSpec(path=pwd()));
                                                Pkg.instantiate()'
        - julia --project=docs/ docs/make.jl
      after_success: skip
```

where the `julia:` and `os:` entries decide the worker from which the docs are built and deployed. In the example above we will thus build and deploy the documentation from a linux worker running Julia 1.0. For more information on how to setup a build stage, see the Travis manual for Build Stages.

The three lines in the `script:` section do the following:

1. Instantiate the doc-building environment (i.e. `docs/Project.toml`, see below).
2. Install your package in the doc-build environment.
3. Run the docs/make.jl script, which builds and deploys the documentation.

> **Note**
>
> If your package has a build script you should call `Pkg.build("PackageName")` after the call to `Pkg.develop` to make sure the package is built properly.

Authentication: SSH Deploy Keys

In order to push the generated documentation from Travis you need to add deploy keys. Deploy keys provide push access to a single repository, to allow secure deployment of generated documentation from the builder to GitHub. The SSH keys can be generated with `DocumenterTools.genkeys` from the DocumenterTools package.

> **Note**
>
> You will need several command line programs (`which`, `git` and `ssh-keygen`) to be installed for the following steps to work. If DocumenterTools fails, please see the the SSH Deploy Keys Walkthrough section for instruction on how to generate the keys manually (including in Windows).

Install and load DocumenterTools with

```
pkg> add DocumenterTools
```

Then call the `DocumenterTools.genkeys` function as follows:

```
julia> using MyPackage
```

where `MyPackage` is the name of the package you would like to create deploy keys for and `MyUser` is your GitHub username. Note that the keyword arguments are optional and can be omitted.

If the package is checked out in development mode with `] dev MyPackage`, you can also use `DocumenterTools.genkeys` as follows:

```
julia> using MyPackage
```

where `MyPackage` is the package you would like to create deploy keys for. The output will look similar to the text below:

```
[ Info: add the public key below to https://github.com/USER/REPO/settings/keys
      with read/write access:

[SSH PUBLIC KEY HERE]

[ Info: add a secure environment variable named 'DOCUMENTER_KEY' to
  https://travis-ci.com/USER/REPO/settings with value:

[LONG BASE64 ENCODED PRIVATE KEY]
```

Follow the instructions that are printed out, namely:

1. Add the public ssh key to your settings page for the GitHub repository that you are setting up by following the `.../settings/key` link provided. Click on **`Add deploy key`**, enter the name **`documenter`** as the title, and copy the public key into the **`Key`** field. Check **`Allow write access`** to allow Documenter to commit the generated documentation to the repo.

2. Next add the long private key to the Travis settings page using the provided link. Again note that you should include **no whitespace** when copying the key. In the **`Environment Variables`** section add a key with the name `DOCUMENTER_KEY` and the value that was printed out. **Do not** set the variable to be displayed in the build log. Then click **`Add`**.

   Security warning

   To reiterate: make sure that this key is hidden. In particular, in the Travis CI settings the "Display value in build log" option should be **OFF** for the variable, so that it does not get printed when the tests run. This base64-encoded string contains the unencrypted private key that gives full write access to your repository, so it must be kept safe. Also, make sure that you never expose this variable in your tests, nor merge any code that does. You can read more about Travis environment variables in Travis User Documentation.

Note

There are more explicit instructions for adding the keys to Travis in the SSH Deploy Keys Walk-through section of the manual.

GitHub Actions

To run the documentation build from GitHub Actions you should add the following to your workflow configuration file:

```
name: Documentation

on: [push]
```

```
jobs:
  build:
    runs-on: ${{ matrix.os }}
    strategy:
      matrix:
        julia-version: [1.2.0]
        julia-arch: [x86]
        os: [ubuntu-latest]
    steps:
      - uses: actions/checkout@v1.0.0
      - uses: julia-actions/setup-julia@latest
        with:
          version: ${{ matrix.julia-version }}
      - name: Install dependencies
        run: julia --project=docs/ -e 'using Pkg; Pkg.develop(PackageSpec(path=pwd())); Pkg.
     instantiate()'
      - name: Build and deploy
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
        run: julia --project=docs/ docs/make.jl
```

which will install Julia, checkout the correct commit of your repository, and run the build of the documentation.

where the `julia-version:`, `julia-arch:` and `os:` entries decide the worker from which the docs are built and deployed. In the example above we will thus build and deploy the documentation from a ubuntu worker running Julia 1.2. For more information on how to setup a GitHub workflow see the manual for Configuring a workflow.

The commands in the lines in the `run:` section do the same as for Travis, see the previous section.

7.3 docs/Project.toml

The doc-build environment `docs/Project.toml` includes Documenter and other doc-build dependencies your package might have. If Documenter is the only dependency, then the `Project.toml` should include the following:

```
[deps]
Documenter = "e30172f5-a6a5-5a46-863b-614d45cd2de4"

[compat]
Documenter = "0.24"
```

Note that it is recommended that you have a `[compat]` section, like the one above, in your `Project.toml` file, which would restrict Documenter's version that gets installed when the build runs. This is to make sure that your builds do not start failing suddenly due to a new major release of Documenter, which may include breaking changes. However, it also means that you will not get updates to Documenter automatically, and hence need to upgrade Documenter's major version yourself.

7.4 The deploydocs Function

At the moment your `docs/make.jl` file probably only contains

```
using Documenter, PACKAGE_NAME
```

We'll need to add an additional function call to this file after `makedocs` which would perform the deployment of the docs to the `gh-pages` branch. Add the following at the end of the file:

```
deploydocs(
    repo = "github.com/USER_NAME/PACKAGE_NAME.jl.git",
```

where `USER_NAME` and `PACKAGE_NAME` must be set to the appropriate names. Note that `repo` should not specify any protocol, i.e. it should not begin with `https://` or `git@`.

See the `deploydocs` function documentation for more details.

7.5 .gitignore

Add the following to your package's `.gitignore` file

```
docs/build/
```

These are needed to avoid committing generated content to your repository.

7.6 gh-pages Branch

By default, Documenter pushes documentation to the `gh-pages` branch. If the branch does not exist it will be created automatically by `deploydocs`. If it does exist then Documenter simply adds an additional commit with the built documentation. You should be aware that Documenter may overwrite existing content without warning.

If you wish to create the `gh-pages` branch manually that can be done following these instructions.

7.7 Documentation Versions

The documentation is deployed as follows:

- Documentation built for a tag `vX.Y.Z` will be stored in a folder `vX.Y.Z`.
- Documentation built from the `devbranch` branch (`master` by default) is stored a folder determined by the `devurl` keyword to `deploydocs` (dev by default).

Which versions that will show up in the version selector is determined by the `versions` argument to `deploydocs`.

Unless a custom domain is being used, the pages are found at:

```
https://USER_NAME.github.io/PACKAGE_NAME.jl/vX.Y.Z
https://USER_NAME.github.io/PACKAGE_NAME.jl/dev
```

By default Documenter will create a link called `stable` that points to the latest release

```
https://USER_NAME.github.io/PACKAGE_NAME.jl/stable
```

It is recommended to use this link, rather then the versioned links, since it will be updated with new releases.

Once your documentation has been pushed to the `gh-pages` branch you should add links to your `README.md` pointing to the `stable` (and perhaps dev) documentation URLs. It is common practice to make use of "badges" similar to those used for Travis and AppVeyor build statuses or code coverage. Adding the following to your package `README.md` should be all that is necessary:

```
[![](https://img.shields.io/badge/docs-stable-blue.svg)](https://USER_NAME.github.io/PACKAGE_NAME.jl
    /stable)
[![](https://img.shields.io/badge/docs-dev-blue.svg)](https://USER_NAME.github.io/PACKAGE_NAME.jl/
    dev)
```

PACKAGE_NAME and USER_NAME should be replaced with their appropriate values. The colour and text of the image can be changed by altering `docs-stable-blue` as described on shields.io, though it is recommended that package authors follow this standard to make it easier for potential users to find documentation links across multiple package README files.

Final Remarks

That should be all that is needed to enable automatic documentation building. Pushing new commits to your `master` branch should trigger doc builds. **Note that other branches do not trigger these builds and neither do pull requests by potential contributors.**

If you would like to see a more complete example of how this process is setup then take a look at this package's repository for some inspiration.

7.8 Deployment systems

It is possible to customize Documenter to use other systems then the ones described in the sections above. This is done by passing a configuration (a `DeployConfig`) to `deploydocs` by the `deploy_config` keyword argument. Documenter natively supports `Travis` and `GitHubActions` natively, but it is easy to define your own by following the simple interface described below.

`Documenter.DeployConfig` – Type.

Abstract type which new deployment configs should be subtypes of.

source

`Documenter.should_deploy` – Function.

Return `true` if the current build should deploy, and `false` otherwise. This function is called with the `repo` and `devbranch` arguments from `deploydocs`.

source

`Documenter.git_tag` – Function.

Return the git tag of the build. If the build is not for a tag, return `nothing`.

This function determines the subfolder where the built docs are deployed. Either a folder named as the tag (e.g. `vX.Y.Z`) or `devurl` (see `deploydocs`) for non-tag builds.

source

`Documenter.authentication_method` – Function.

Return enum instance SSH or HTTPS depending on push method to be used.

Configs returning SSH should support Documenter.documenter_key. Configs returning HTTPS should support Documenter.authenticated_repo_url.

source

Documenter.authenticated_repo_url – Function.

Return an authenticated URL to the upstream repository.

This method must be supported by configs that push with HTTPS, see Documenter.authentication_method.

source

Documenter.documenter_key – Function.

Return the Base64-encoded SSH private key for the repository. Defaults to reading the DOCUMENTER_KEY environment variable.

This method must be supported by configs that push with SSH, see Documenter.authentication_method.

source

Documenter.Travis – Type.

Default implementation of DeployConfig.

The following environment variables influences the build when using the Travis configuration:

- DOCUMENTER_KEY: must contain the Base64-encoded SSH private key for the repository. This variable should be set in the Travis settings for the repository. Make sure this variable is marked **NOT** to be displayed in the build log.
- TRAVIS_PULL_REQUEST: must be set to false. This avoids deployment on pull request builds.
- TRAVIS_REPO_SLUG: must match the value of the repo keyword to deploydocs.
- TRAVIS_EVENT_TYPE: may not be set to cron. This avoids re-deployment of existing docs on builds that were triggered by a Travis cron job.
- TRAVIS_BRANCH: unless TRAVIS_TAG is non-empty, this must have the same value as the devbranch keyword to deploydocs. This makes sure that only the development branch (commonly, the master branch) will deploy the "dev" documentation (deployed into a directory specified by the devurl keyword to deploydocs).
- TRAVIS_TAG: if set, a tagged version deployment is performed instead; the value must be a valid version number (i.e. match Base.VERSION_REGEX). The documentation for a package version tag gets deployed to a directory named after the version number in TRAVIS_TAG instead.

The TRAVIS_* variables are set automatically on Travis. More information on how Travis sets the TRAVIS_* variables can be found in the Travis documentation.

source

Documenter.GitHubActions – Type.

Implementation of `DeployConfig` for deploying from GitHub Actions.

The following environment variables influences the build when using the `GitHubActions` configuration:

- `DOCUMENTER_KEY`: must contain the Base64-encoded SSH private key for the repository. This variable should be set in the GitHub Actions configuration file using a repository secret, see the documentation for secret environment variables.
- `GITHUB_EVENT_NAME`: must be set to `push`. This avoids deployment on pull request builds.
- `GITHUB_REPOSITORY`: must match the value of the `repo` keyword to `deploydocs`.
- `GITHUB_REF`: must match the `devbranch` keyword to `deploydocs`, alternatively correspond to a git tag.

The `GITHUB_*` variables are set automatically on GitHub Actions, see the documentation.

source

7.9 SSH Deploy Keys Walkthrough

If the instructions in Authentication: SSH Deploy Keys did not work for you (for example, `ssh-keygen` is not installed), don't worry! This walkthrough will guide you through the process. There are three main steps:

1. Generating an SSH Key
2. Adding the Public Key to GitHub
3. Adding the Private Key

Generating an SSH Key

The first step is to generate an SSH key. An SSH key is made up of two components: a public key, which can be shared publicly, and a private key, which you should ensure is **never** shared publicly.

The public key usually looks something like this

```
ssh-rsa [base64-encoded-key] [optional-comment]
```

And the private key usually look something like this

```
-----BEGIN RSA PRIVATE KEY-----
 ... base64-encoded key over several lines ...
-----END RSA PRIVATE KEY-----
```

If you have `ssh-keygen` installed

If you have `ssh-keygen` installed, but `DocumenterTools.genkeys()` didn't work, you can generate an SSH key as follows. First, generate a key using `ssh-keygen` and save it to the file `privatekey`:

Next, we need to encode the private key in Base64. Run the following command:

```
julia> using Base64
```

Copy and paste the output somewhere. This is your private key and is required for the last step.

Now we need to get the public key. Run the following command:

```
```

Copy and paste the output somewhere. This is your public key and is required for the step Adding the Public Key to GitHub.

If you do not have `ssh-keygen`

If you're using Windows, you probably don't have `ssh-keygen` installed. Instead, we're going to use a program called PuTTY. The first step in the process to generate a new SSH key is to download PuTTY:

- Download and install PuTTY

PuTTY is actually a collection of a few different programs. We need to use PuTTYgen. Open it, and you should get a window that looks like:

Now we need to generate a key.

- Click the "Generate" button, then follow the instructions and move the mouse around to create randomness.

Once you've moved the mouse enough, the window should look like:

Now we need to save the public key somewhere.

- Copy the text in the box titled "Public key for pasting into OpenSSH authorized_keys file" and paste it somewhere for later. This is your public key and is required for the step Adding the Public Key to GitHub

Finally, we need to save the private key somewhere.

- Click the "Conversions" tab, and then click "Export OpenSSH key". Save that file somewhere. That file is your private key and is required for the last step.

  Note

  Don't save your key via the "Save private key" button as this will save the key in the wrong format.

If you made it this far, congratulations! You now have the private and public keys needed to set up automatic deployment of your documentation. The next steps are to add the keys to GitHub and Travis.

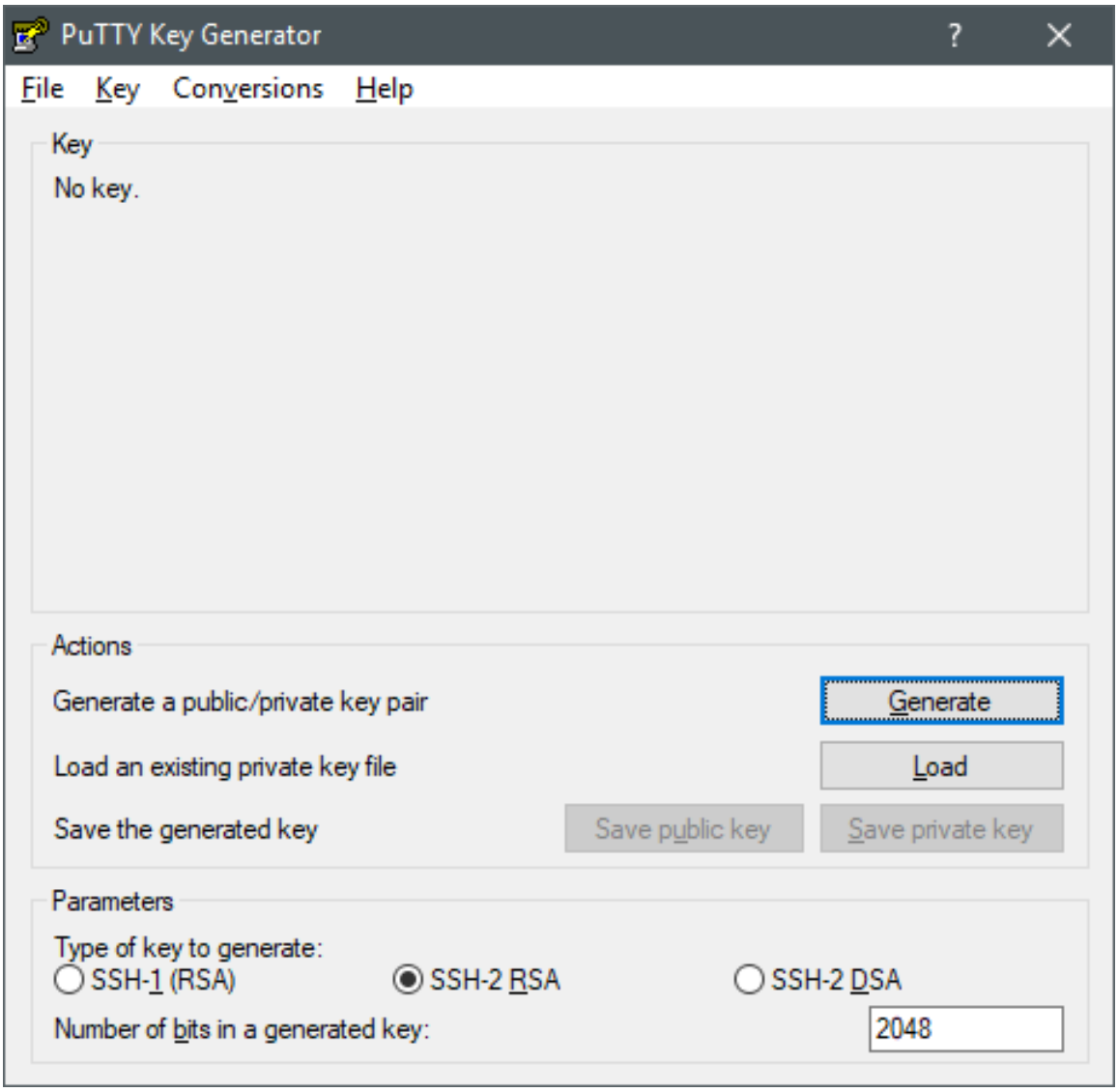
PuTTY Key Generator
File Key Conversions Help
Key
No key.
Actions
Generate a public/private key pair
Generate
Load an existing private key file
Load
Save the generated key
Save public key
Save private key
Parameters
Type of key to generate:
SSH-1 (RSA)
SSH-2 RSA
SSH-2 DSA
Number of bits in a generated key:
2048

Figure 7.1:

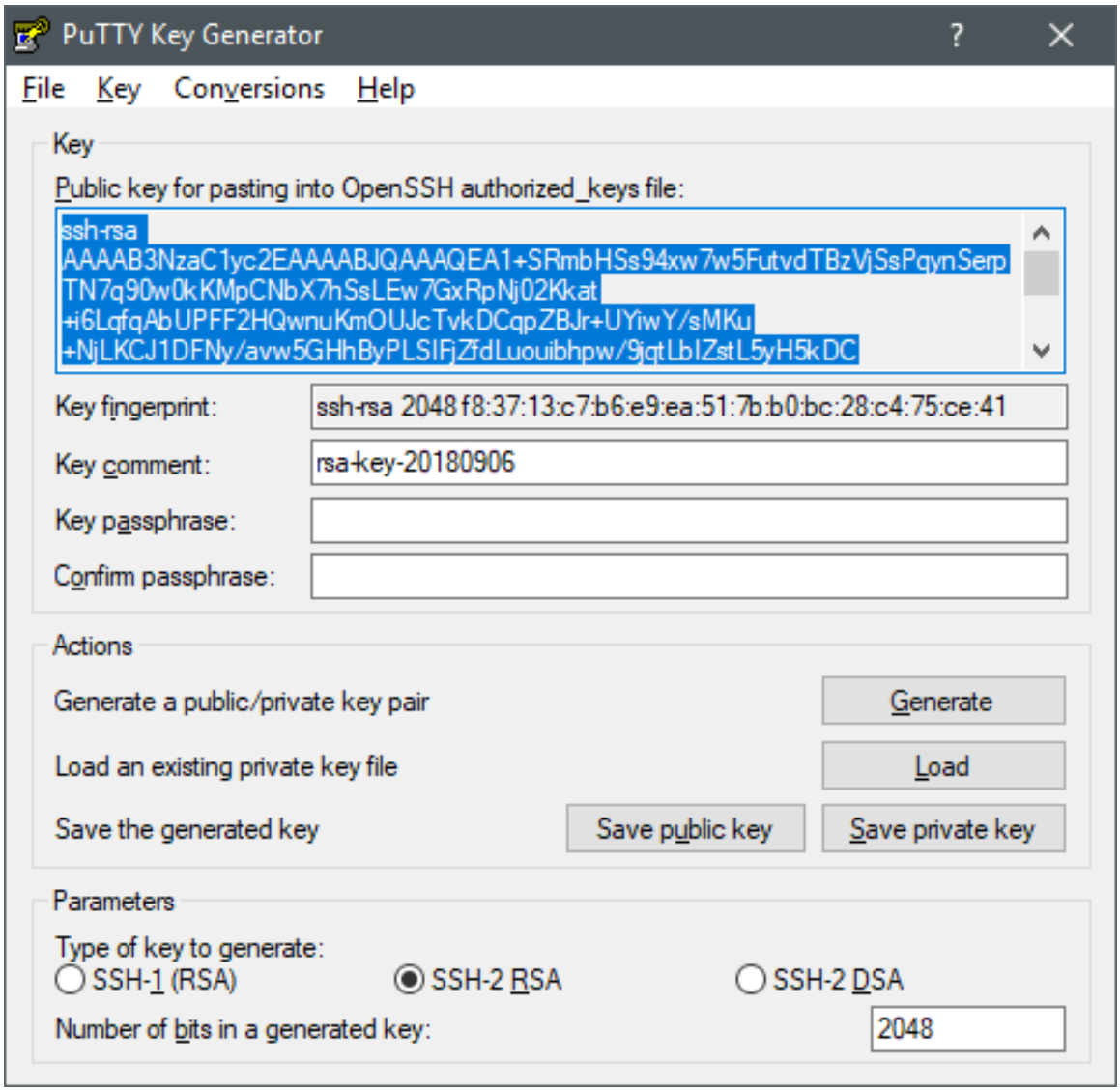
PuTTY Key Generator
File Key Conversions Help
Key
Public key for pasting into OpenSSH authorized_keys file:
ssh-rsa
AAAAB3NzaC1yc2EAAAABJQAAAQEA1+SRmbHSs94xw7w5FutvdTBzVjSsPqynSerp
TN7q90w0kKMpCNbX7hSsLEw7GxRpNj02Kkat
+i6LqfqAbUPFF2HQwnuKmOUJcTvkDCqpZBJr+UYiwY/sMKu
+NjLKCJ1DFNy/avw5GHhByPLSIFjZfdLuouibhpw/9jqtLbIZstL5yH5kDC
Key fingerprint: ssh-rsa 2048 f8:37:13:c7:b6:e9:ea:51:7b:b0:bc:28:c4:75:ce:41
Key comment: rsa-key-20180906
Key passphrase:
Confirm passphrase:
Actions
Generate a public/private key pair Generate
Load an existing private key file Load
Save the generated key Save public key Save private key
Parameters
Type of key to generate:
SSH-1 (RSA) SSH-2 RSA SSH-2 DSA
Number of bits in a generated key: 2048

Figure 7.2:

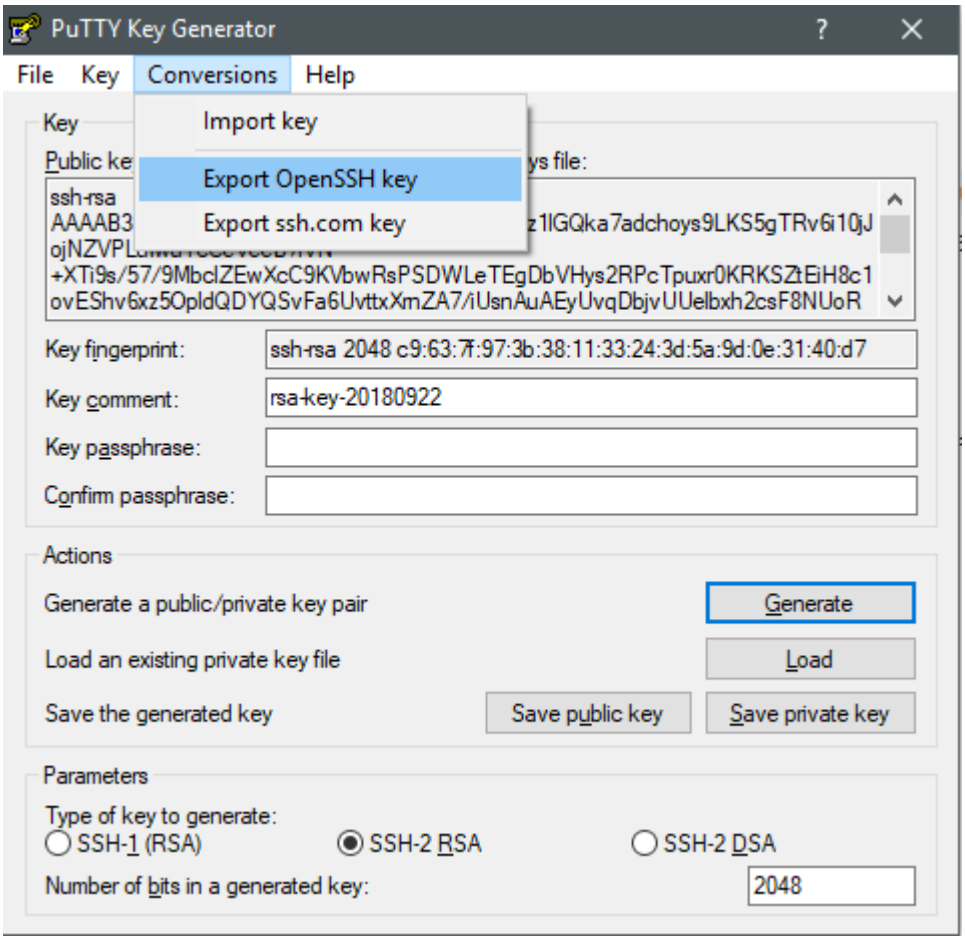

Figure 7.3:

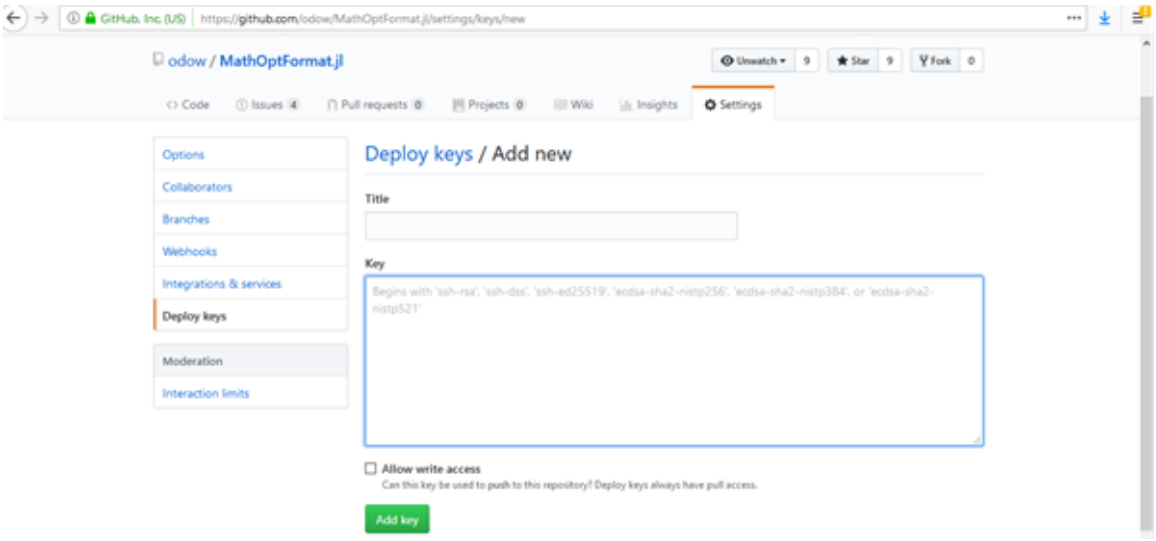

Figure 7.4:

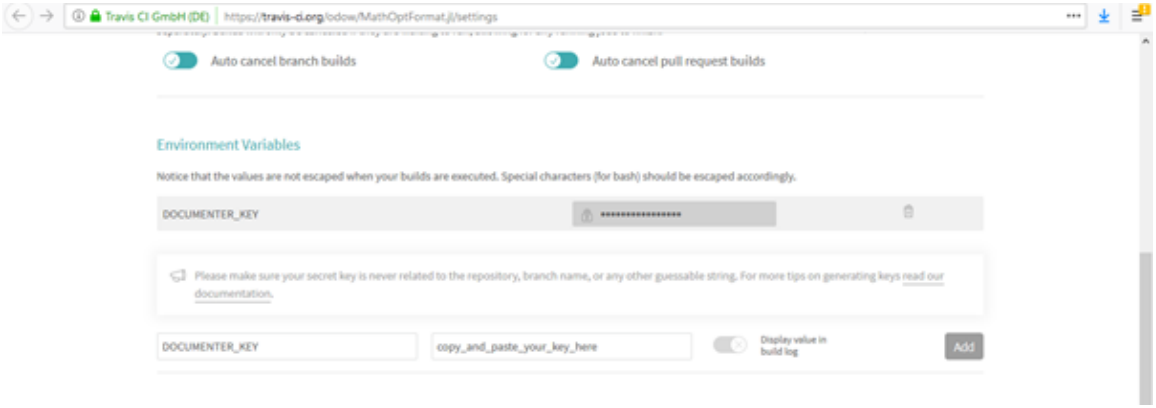

Figure 7.5:

Adding the Public Key to GitHub

In this section, we explain how to upload a public SSH key to GitHub. By this point, you should have generated a public key and saved it to a file. If you haven't done this, go read Generating an SSH Key.

Go to `https://github.com/[YOUR_USER_NAME]/[YOUR_REPO_NAME]/settings/keys` and click "Add deploy key". You should get to a page that looks like:

Now we need to fill in three pieces of information.

1. Have "Title" be e.g. "Documenter".
2. Copy and paste the public key that we generated in the Generating an SSH Key step into the "Key" field.
3. Make sure that the "Allow write access" box is checked.

Once you're done, click "Add key". Congratulations! You've added the public key to GitHub. The next step is to add the private key to Travis or GitHub Secrets.

Adding the Private Key

In this section, we explain how to upload a private SSH key to Travis. By this point, you should have generated a private key and saved it to a file. If you haven't done this, go read Generating an SSH Key.

First, we need to Base64 encode the private key. Open Julia, and run the command

```
julia> using Base64
```

Copy the resulting output.

Go to `https://travis-ci.com/[YOUR_USER_NAME]/[YOUR_REPO_NAME]/settings`. Scroll down to the "Environment Variables" section. It should look like this:

Now, add a new environment variable called `DOCUMENTER_KEY`, and set its value to the output from the Julia command above (make sure to remove the surrounding quotes).

Finally, make sure that the "Display value in build log" is left switched off and then click "Add". Congratulations! You've added the private key to Travis.

Security warning

To reiterate: make sure that the "Display value in build log" option is **OFF** for the variable, so that it does not get printed when the tests run. This base64-encoded string contains the unencrypted private key that gives full write access to your repository, so it must be kept safe. Also, make sure that you never expose this variable in your tests, nor merge any code that does. You can read more about Travis environment variables in Travis User Documentation.

Final Remarks

You should now be able to continue on with the Hosting Documentation.

Chapter 8

Other Output Formats

In addition to the default native HTML output, plugin packages enable Documenter to generate output in other formats. Once the corresponding package is loaded, the output format can be specified using the `format` option in `makedocs`.

8.1 Markdown & MkDocs

Markdown output requires the `DocumenterMarkdown` package to be available and loaded. For Travis setups, add the package to the `docs/Project.toml` environment as a dependency. You also need to import the package in `make.jl`:

```
using DocumenterMarkdown
```

When `DocumenterMarkdown` is loaded, you can specify `format = Markdown()` in `makedocs`. Documenter will then output a set of Markdown files to the `build` directory that can then further be processed with MkDocs into HTML pages.

MkDocs, of course, is not the only option you have – any markdown to HTML converter should work fine with some amount of setting up.

> **Note**
>
> Markdown output used to be the default option (i.e. when leaving the `format` option unspecified). The default now is the HTML output.

The MkDocs `mkdocs.yml` file

A MkDocs build is controlled by the `mkdocs.yml` configuration file. Add the file with the following content to the `docs/` directory:

```
site_name:        PACKAGE_NAME.jl
repo_url:         https://github.com/USER_NAME/PACKAGE_NAME.jl
site_description: Description...
site_author:      USER_NAME

theme: readthedocs

extra_css:
  - assets/Documenter.css

extra_javascript:
```

```
  - https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.1/MathJax.js?config=TeX-AMS_HTML
  - assets/mathjaxhelper.js

markdown_extensions:
  - extra
  - tables
  - fenced_code
  - mdx_math

docs_dir: 'build'

pages:
  - Home: index.md
```

If you have run Documenter and it has generated a `build/` directory, you can now try running `mkdocs build` – this should now generate the `site/` directory. You should also add the `docs/site/` directory into your `.gitignore` file, which should now look like:

```
docs/build/
docs/site/
```

This is only a basic skeleton. Read through the MkDocs documentation if you would like to know more about the available settings.

Deployment with MkDocs

To deploy MkDocs on Travis, you also need to provide additional keyword arguments to `deploydocs`. Your `deploydocs` call should look something like

```
deploydocs(
    repo   = "github.com/USER_NAME/PACKAGE_NAME.jl.git",
    deps   = Deps.pip("mkdocs", "pygments", "python-markdown-math"),
    make   = () -> run(`mkdocs build`)
    target = "site"
```

- `deps` serves to provide the required Python dependencies to build the documentation
- `make` specifies the function that calls `mkdocs` to perform the second build step
- `target`, which specified which files get copied to `gh-pages`, needs to point to the `site/` directory

In the example above we include the dependencies `mkdocs` and `python-markdown-math`. The former makes sure that MkDocs is installed to deploy the documentation, and the latter provides the `mdx_math` markdown extension to exploit MathJax rendering of latex equations in markdown. Other dependencies should be included here.

LaTeX: MkDocs and MathJax

To get MkDocs to display LaTeXequations correctly we need to update several of this configuration files described in the Package Guide.

`docs/make.jl` should add the `python-markdown-math` dependency to allow for equations to be rendered correctly.

```
# ...

deploydocs(
    deps = Deps.pip("pygments", "mkdocs", "python-markdown-math"),
    # ...
```

This package should also be installed locally so that you can preview the generated documentation prior to pushing new commits to a repository.

```
$ pip install python-markdown-math
```

The `docs/mkdocs.yml` file must add the `python-markdown-math` extension, called `mdx_math`, as well as two MathJax JavaScript files:

```
# ...
markdown_extensions:
  - mdx_math
  # ...

extra_javascript:
  - https://cdnjs.cloudflare.com/ajax/libs/mathjax/2.7.1/MathJax.js?config=TeX-AMS_HTML
  - assets/mathjaxhelper.js
# ...
```

Final Remarks

Following this guide and adding the necessary changes to the configuration files should enable properly rendered mathematical equations within your documentation both locally and when built and deployed using the Travis built service.

8.2 PDF Output via LaTeX

LaTeX/PDF output requires the `DocumenterLaTeX` package to be available and loaded in `make.jl` with

```
using DocumenterLaTeX
```

When `DocumenterLaTeX` is loaded, you can set `format = LaTeX()` in `makedocs`, and Documenter will generate a PDF version of the documentation using LaTeX. The `makedocs` argument `sitename` will be used for the `\\title` field in the tex document, and if the build is for a release tag (i.e. when the `"TRAVIS_TAG"` environment variable is set) the version number will be appended to the title. The `makedocs` argument `authors` should also be specified, it will be used for the `\\authors` field in the tex document.

Compiling using natively installed latex

The following is required to build the documentation:

- You need `pdflatex` command to be installed and available to Documenter.
- You need the minted LaTeX package and its backend source highlighter Pygments installed.
- You need the DejaVu Sans and DejaVu Sans Mono fonts installed.

Compiling using docker image

It is also possible to use a prebuilt docker image to compile the `.tex` file. The image contains all of the required installs described in the section above. The only requirement for using the image is that `docker` is installed and available for the builder to call. You also need to tell Documenter to use the docker image, instead of natively installed tex which is the default. This is done with the LaTeX specifier:

```
using DocumenterLaTeX
makedocs(
    format = LaTeX(platform = "docker"),
    ...
)
```

If you build the documentation on Travis you need to add

```
services:
  - docker
```

to your `.travis.yml` file.

Part III

Showcase

This page showcases the various page elements that are supported by Documenter. It should be read side-by-side with its source (`docs/src/showcase.md`) to see what syntax exactly is used to create the various elements.

Chapter 9

Table of contents

A table of contents can be generated with an `@contents` block. The one for this page renders as

- Showcase
  - Table of contents
  - Basic Markdown
  - Docstrings
  - Doctesting example
  - Running interactive code
  - Doctest showcase

Chapter 10

Basic Markdown

Documenter can render all the Markdown syntax supported by the Julia Markdown parser. You can use all the usual markdown syntax, such as **bold text** and italic text and `print("inline code")`.

Code blocks are rendered as follows:

```
This is an non-highlighted code block.
... Rendered in monospace.
```

When the language is specified for the block, e.g. by starting the block with ```` ```julia ````, the contents gets highlighted appropriately (for the language that are supported by the highlighter).

```julia
function foo(x::Integer)
    @show x + 1
```

For mathematics, both inline and display equations are available. Inline equations should be written as LaTeX between two backticks, e.g. ``` ``A x^2 + B x + C = 0`` ```. It will render as $Ax^2 + Bx + C = 0$.

The LaTeX for display equations must be wrapped in a ```` ```math ```` code block and will render like

$$x_{1,2} = \frac{-B \pm \sqrt{B^2 - 4AC}}{2A}$$

By default, the HTML output renders equations with KaTeX, but MathJax can optionally be used as well.

Finally, admonitions for notes, warnings and such:

> **'note' admonition**
>
> Admonitions look like this. This is a `!!! note`-type admonition.
>
> Note that admonitions themselves can contain other block-level elements too, such as code blocks. E.g.
>
> ```
> 
> ```
>
> However, you **can not** have at-blocks, docstrings, doctests etc. in an admonition.
>
> Headings are OK though:

Part IV

Heading 1

Chapter 11

Heading 2

11.1 Heading 3

Heading 4

Heading 5

’info’ admonition

This is a `!!! info`-type admonition. This is the same as a `!!! note`-type.

’tip’ admonition

This is a `!!! tip`-type admonition.

’warning’ admonition

This is a `!!! warning`-type admonition.

’danger’ admonition

This is a `!!! danger`-type admonition.

’compat’ admonition

This is a `!!! compat`-type admonition.

Unknown admonition class

Admonition with an unknown admonition class. This is a `code example`.

11.2 Lists

Tight lists look as follows

- Lorem ipsum dolor sit amet, consectetur adipiscing elit.
- Nulla quis venenatis justo.
- In non sodales eros.

If the lists contain paragraphs or other block level elements, they look like this:

- Morbi et varius nisl, eu semper orci.

  Donec vel nibh sapien. Maecenas ultricies mauris sapien. Nunc et sem ac justo ultricies dignissim ac vitae sem.

- Nulla molestie aliquet metus, a dapibus ligula.

  Morbi pellentesque sodales sollicitudin. Fusce semper placerat suscipit. Aliquam semper tempus ex, non efficitur erat posuere in. Fusce at orci eu ex sagittis commodo.

  > Fusce tempus scelerisque egestas. Pellentesque varius nulla a varius fringilla.

  Fusce nec urna eu orci porta blandit.

Numbered lists are also supported

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
2. Nulla quis venenatis justo.
3. In non sodales eros.

As are nested lists

- Morbi et varius nisl, eu semper orci.

  Donec vel nibh sapien. Maecenas ultricies mauris sapien. Nunc et sem ac justo ultricies dignissim ac vitae sem.

  - Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  - Nulla quis venenatis justo.
  - In non sodales eros.

- Nulla molestie aliquet metus, a dapibus ligula.

1. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
2. Nulla quis venenatis justo.
3. In non sodales eros.

Fusce nec urna eu orci porta blandit.

Lists can also be included in other blocks that can contain block level items

Bulleted lists in admonitions

- Lorem ipsum dolor sit amet, consectetur adipiscing elit.
- Nulla quis venenatis justo.
- In non sodales eros.

Large lists in admonitions

- Morbi et varius nisl, eu semper orci.
  Donec vel nibh sapien. Maecenas ultricies mauris sapien. Nunc et sem ac justo ultricies dignissim ac vitae sem.

  - Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  - Nulla quis venenatis justo.
  - In non sodales eros.
- Nulla molestie aliquet metus, a dapibus ligula.

  1. Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  2. Nulla quis venenatis justo.
  3. In non sodales eros.

  Fusce nec urna eu orci porta blandit.

- Morbi et varius nisl, eu semper orci.
  Donec vel nibh sapien. Maecenas ultricies mauris sapien. Nunc et sem ac justo ultricies dignissim ac vitae sem.

  - Lorem ipsum dolor sit amet, consectetur adipiscing elit.
  - Nulla quis venenatis justo.
  - In non sodales eros.

object	implemented	value
A	✓	10.00
BB	✓	1000000.00

11.3 Tables

With explicit alignment.

object	implemented	value
A	✓	10.00
BB	✓	1000000.00

Tables that are too wide should become scrollable.

object	implemented	value
A	✓	10.00
BBBBBBBBBBBBBBBBBBBB	✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓✓	1000000000000000000000000000000000000000000000000000000000000000000000.00

11.4 Footnotes

Footnote references can be added with the `[^label]` syntax.[1] The footnote definitions get collected at the bottom of the page.

The footnote label can be an arbitrary string and even consist of block-level elements.[2]

11.5 Headings

Finally, headings render as follows

11.6 Heading level 3

Heading level 4

Heading level 5

Heading level 6 To see an example of a level 1 heading see the page title and for level 2 heading, see the one just under this paragraph.

[1] A footnote definition uses the `[^label]: ...` sytax in a block scope.

[2]

> Any sufficiently advanced technology is indistinguishable from magic.
>
> Arthur C. Clarke, Profiles of the Future (1961): Clarke's Third Law.

Headings in sidebars

Level 1 and 2 heading show up in the sidebar, for the current page.

Chapter 12

Docstrings

The key feature of Documenter, of course, is the ability to automatically include docstrings from your package in the manual. The following example docstrings come from the demo `DocumenterShowcase` module, the source of which can be found in `docs/DocumenterShowcase.jl`.

To include a docstrings into a manual page, you needs to use an `@docs` block

````
```@docs
DocumenterShowcase
```
````

This will include a single docstring and it will look like this

> **Missing docstring.**
>
> Missing docstring for `DocumenterShowcase`. Check Documenter's build log for details.

You can include the docstrings corresponding to different function signatures one by one. E.g., the `DocumenterShowcase.foo` function has two signatures – `(::Integer)` and `(::AbstractString)`.

````
```@docs
DocumenterShowcase.foo(::Integer)
```
````

yielding the following docstring

> **Missing docstring.**
>
> Missing docstring for `DocumenterShowcase.foo(::Integer)`. Check Documenter's build log for details.

And now, by having `DocumenterShowcase.foo(::AbstractString)` in the `@docs` block will give the other docstring

> **Missing docstring.**
>
> Missing docstring for `DocumenterShowcase.foo(::AbstractString)`. Check Documenter's build log for details.

However, if you want, you can also combine multiple docstrings into a single docstring block. The `DocumenterShowcase.bar` function has the same signatures as

If we just put `DocumenterShowcase.bar` in an `@docs` block, it will combine the docstrings as follows:

Missing docstring.

Missing docstring for `DocumenterShowcase.bar`. Check Documenter's build log for details.

If you have very many docstrings, you may also want to consider using the `@autodocs` block which can include a whole set of docstrings automatically based on certain filtering options

12.1 An index of docstrings

The `@index` block can be used to generate a list of all the docstrings on a page (or even across pages) and will look as follows

Chapter 13

Doctesting example

Often you want to write code example such as this:

```
julia> f(x) = x^2
f (generic function with 1 method)

julia> f(3)
```

If you write them as a ```` ```jldoctest ```` code block, Documenter can make sure that the doctest has not become outdated. See Doctests for more information.

Script-style doctests are supported too:

```
2 + 2
# output
```

Chapter 14

Running interactive code

`@example` block run a code snippet and insert the output into the document. E.g. the following Markdown

````
```@example
2 + 3
```
````

becomes the following code-output block pair

```
```

```
5
```

If the last element can be rendered as an image or `text/html` etc. (the corresponding `Base.show` method for the particular MIME type has to be defined), it will be rendered appropriately. e.g.:

```
using Main: DocumenterShowcase
DocumenterShowcase.SVGCircle("000", "aaa")
```

This is handy when combined with the `Markdown` standard library

```
using Markdown
Markdown.parse("""
`Markdown.MD` objects can be constructed dynamically on the fly and still get rendered "natively".
```

\texttt{Markdown.MD} objects can be constructed dynamically on the fly and still get rendered "natively".

If the last value in an `@example` block is a `nothing`, the standard output from the blocks' evaluation gets displayed instead

```
```

```
Hello World
```

However, do note that if the block prints to standard output, but also has a final non-`nothing` value, the standard output just gets discarded:

```
println("Hello World")
```

```
42
```

14.1 REPL-type

`@repl` block can be used to simulate the REPL evaluation of code blocks. For example, the following block

```
```@repl
using Statistics
xs = collect(1:10)
median(xs)
sum(xs)
```
```

It gets expanded into something that looks like as if it was evaluated in the REPL, with the `julia>` prompt prepended etc.:

```
julia> using Statistics

julia> xs = collect(1:10)
10-element Array{Int64,1}:
  1
  2
  3
  4
  5
  6
  7
  8
  9
 10

julia> median(xs)
5.5

julia> sum(xs)
```

Chapter 15

Doctest showcase

Currently exists just so that there would be doctests to run in manual pages of Documenter's manual. This page does not show up in navigation.

```
julia> 2 + 2
```

The following doctests needs doctestsetup:

```
julia> Documenter.Utilities.splitexpr(:(Foo.Bar.baz))
```

Let's also try `@meta` blocks:

```
julia> f(2)
```

Part V

Library

Chapter 16

Public

16.1 Public Documentation

Documentation for `Documenter.jl`'s public interface.

See the Internals section of the manual for internal package docs covering all submodules.

Contents

- Public Documentation
  - Contents
  - Index
  - Public Interface
  - DocumenterTools

Index

- `Documenter`
- `Documenter.Deps`
- `Documenter.DocMeta`
- `Documenter.Deps.pip`
- `Documenter.DocMeta.getdocmeta`
- `Documenter.DocMeta.setdocmeta!`
- `Documenter.Writers.HTMLWriter.asset`
- `Documenter.deploydocs`
- `Documenter.doctest`
- `Documenter.hide`
- `Documenter.makedocs`
- `DocumenterTools.generate`
- `DocumenterTools.genkeys`

Public Interface

Documenter – Module.

Main module for `Documenter.jl` – a documentation generation package for Julia.

Two functions are exported from this module for public use:

- makedocs. Generates documentation from docstrings and templated markdown files.
- deploydocs. Deploys generated documentation from Travis-CI to GitHub Pages.

Exports

- Deps
- DocMeta
- KaTeX
- MathJax
- asset
- deploydocs
- doctest
- hide
- makedocs

source

Documenter.makedocs – Function.

```
makedocs(
    root    = "<current-directory>",
    source  = "src",
    build   = "build",
    clean   = true,
    doctest = true,
    modules = Module[],
    repo    = "",
    highlightsig = true,
    sitename = "",
    expandfirst = [],
```

Combines markdown files and inline docstrings into an interlinked document. In most cases makedocs should be run from a `make.jl` file:

```
using Documenter
makedocs(
    # keywords...
```

which is then run from the command line with:

```
$ julia make.jl
```

The folder structure that makedocs expects looks like:

```
docs/
    build/
    src/
    make.jl
```

Keywords

root is the directory from which makedocs should run. When run from a make.jl file this keyword does not need to be set. It is, for the most part, needed when repeatedly running makedocs from the Julia REPL like so:

```
julia> makedocs(root = joinpath(dirname(pathof(MyModule)), "..", "docs"))
```

source is the directory, relative to root, where the markdown source files are read from. By convention this folder is called src. Note that any non-markdown files stored in source are copied over to the build directory when makedocs is run.

build is the directory, relative to root, into which generated files and folders are written when makedocs is run. The name of the build directory is, by convention, called build, though, like with source, users are free to change this to anything else to better suit their project needs.

clean tells makedocs whether to remove all the content from the build folder prior to generating new content from source. By default this is set to true.

doctest instructs makedocs on whether to try to test Julia code blocks that are encountered in the generated document. By default this keyword is set to true. Doctesting should only ever be disabled when initially setting up a newly developed package where the developer is just trying to get their package and documentation structure correct. After that, it's encouraged to always make sure that documentation examples are runnable and produce the expected results. See the Doctests manual section for details about running doctests.

Setting doctest to :only allows for doctesting without a full build. In this mode, most build stages are skipped and the strict keyword is ignore (a doctesting error will always make makedocs throw an error).

modules specifies a vector of modules that should be documented in source. If any inline docstrings from those modules are seen to be missing from the generated content then a warning will be printed during execution of makedocs. By default no modules are passed to modules and so no warnings will appear. This setting can be used as an indicator of the "coverage" of the generated documentation. For example Documenter's make.jl file contains:

```
makedocs(
    modules = [Documenter],
    # ...
```

and so any docstring from the module Documenter that is not spliced into the generated documentation in build will raise a warning.

repo specifies a template for the "link to source" feature. If you are using GitHub, this is automatically generated from the remote. If you are using a different host, you can use this option to tell Documenter how URLs should be generated. The following placeholders will be replaced with the respective value of the generated link:

- {commit} Git branch or tag name, or commit hash
- {path} Path to the file in the repository
- {line} Line (or range of lines) in the source file

For example if you are using GitLab.com, you could use

```
```

highlightsig enables or disables automatic syntax highlighting of leading, unlabeled code blocks in docstrings (as Julia code). For example, if your docstring begins with an indented code block containing the function signature, then that block would be highlighted as if it were a labeled Julia code block. No other code blocks are affected. This feature is enabled by default.

sitename is displayed in the title bar and/or the navigation menu when applicable.

expandfirst allows some of the pages to be expanded (i.e. at-blocks evaluated etc.) before the others. Documenter normally evaluates the files in the alphabetic order of their file paths relative to `src`, but `expandfirst` allows some pages to be prioritized.

For example, if you have `foo.md` and `bar.md`, `bar.md` would normally be evaluated before `foo.md`. But with `expandfirst = ["foo.md"]`, you can force `foo.md` to be evaluated first.

Evaluation order among the `expandfirst` pages is according to the order they appear in the argument.

Experimental keywords

In addition to standard arguments there is a set of non-finalized experimental keyword arguments. The behaviour of these may change or they may be removed without deprecation when a minor version changes (i.e. except in patch releases).

checkdocs instructs `makedocs` to check whether all names within the modules defined in the `modules` keyword that have a docstring attached have the docstring also listed in the manual (e.g. there's a `@docs` blocks with that docstring). Possible values are `:all` (check all names; the default), `:exports` (check only exported names) and `:none` (no checks are performed). If `strict` is also enabled then the build will fail if any missing docstrings are encountered.

linkcheck – if set to `true` `makedocs` uses `curl` to check the status codes of external-pointing links, to make sure that they are up-to-date. The links and their status codes are printed to the standard output. If `strict` is also enabled then the build will fail if there are any broken (400+ status code) links. Default: `false`.

linkcheck_ignore allows certain URLs to be ignored in `linkcheck`. The values should be a list of strings (which get matched exactly) or Regex objects. By default nothing is ignored.

strict – `makedocs` fails the build right before rendering if it encountered any errors with the document in the previous build phases.

workdir determines the working directory where `@example` and `@repl` code blocks are executed. It can be either a path or the special value `:build` (default).

If the `workdir` is set to a path, the working directory is reset to that path for each code block being evaluated. Relative paths are taken to be relative to `root`, but using absolute paths is recommended (e.g. `workdir = joinpath(@__DIR__, "..")` for executing in the package root for the usual `docs/make.jl` setup).

With the default `:build` option, the working directory is set to a subdirectory of `build`, determined from the source file path. E.g. for `src/foo.md` it is set to `build/`, for `src/foo/bar.md` it is set to `build/foo` etc.

Note that `workdir` does not affect doctests.

Output formats

format allows the output format to be specified. The default format is `Documenter.HTML` which creates a set of HTML files.

There are other possible formats that are enabled by using other addon-packages. For examples, the `DocumenterMarkdown` package define the `DocumenterMarkdown.Markdown()` format for use with e.g. MkDocs, and the `DocumenterLaTeX` package define the `DocumenterLaTeX.LaTeX()` format for LaTeX / PDF output. See the Other Output Formats for more information.

See Also

A guide detailing how to document a package using Documenter's `makedocs` is provided in the setup guide in the manual.

source

`Documenter.hide` – Function.

```
hide(page)
```

Allows a page to be hidden in the navigation menu. It will only show up if it happens to be the current page. The hidden page will still be present in the linear page list that can be accessed via the previous and next page links. The title of the hidden page can be overriden using the => operator as usual.

Usage

```
makedocs(
    ...,
    pages = [
        ...,
        hide("page1.md"),
        hide("Title" => "page2.md")
    ]
```

source

```
hide(root, children)
```

Allows a subsection of pages to be hidden from the navigation menu. `root` will be linked to in the navigation menu, with the title determined as usual. `children` should be a list of pages (note that it **can not** be hierarchical).

Usage

```
makedocs(
    ...,
    pages = [
        ...,
        hide("Hidden section" => "hidden_index.md", [
            "hidden1.md",
            "Hidden 2" => "hidden2.md"
        ]),
        hide("hidden_index.md", [...])
    ]
```

source

`Documenter.Writers.HTMLWriter.asset` – Function.

```
```

Can be used to pass non-local web assets to HTML, where `uri` should be an absolute HTTP or HTTPS URL.

It accepts the following keyword arguments:

`class` can be used to override the asset class, which determines how exactly the asset gets included in the HTML page. This is necessary if the class can not be determined automatically (default).

Should be one of: `:js`, `:css` or `:ico`. They become a `<script>`, `<link rel="stylesheet" type="text/css">` and `<link rel="icon" type="image/x-icon">` elements in <head>, respectively.

`islocal` can be used to declare the asset to be local. The `uri` should then be a path relative to the documentation source directory (conventionally `src/`). This can be useful when it is necessary to override the asset class of a local asset.

Usage

```
Documenter.HTML(assets = [
    # Standard local asset
    "assets/extra_styles.css",
    # Standard remote asset (extension used to determine that class = :js)
    asset("https://example.com/jslibrary.js"),
    # Setting asset class manually, since it can't be determined manually
    asset("https://example.com/fonts", class = :css),
    # Same as above, but for a local asset
    asset("asset/foo.script", class=:js, islocal=true),
```

source

Documenter.deploydocs – Function.

```
deploydocs(
    root    = "<current-directory>",
    target = "build",
    repo    = "<required>",
    branch = "gh-pages",
    deps    = nothing | <Function>,
    make    = nothing | <Function>,
    devbranch = "master",
    devurl = "dev",
    versions = ["stable" => "v^", "v#.#", devurl => devurl]
```

Converts markdown files generated by makedocs to HTML and pushes them to `repo`. This function should be called from within a package's `docs/make.jl` file after the call to makedocs, like so

```
using Documenter, PACKAGE_NAME
makedocs(
    # options...
)
deploydocs(
    repo = "github.com/..."
```

When building the docs for a tag (i.e. a release) the documentation is deployed to a directory with the tag name (i.e. `vX.Y.Z`) and to the `stable` directory. Otherwise the docs are deployed to the directory determined by the `devurl` argument.

Required keyword arguments

`repo` is the remote repository where generated HTML content should be pushed to. Do not specify any protocol - "https://" or "git@" should not be present. This keyword must be set and will throw an error when left undefined. For example this package uses the following `repo` value:

```
```

Optional keyword arguments

`deploy_config` determines configuration for the deployment. If this is not specified Documenter will try to autodetect from the currently running environment. See the manual section about Deployment systems.

`root` has the same purpose as the `root` keyword for `makedocs`.

`target` is the directory, relative to `root`, where generated content that should be deployed to `gh-pages` is written to. written to. It should generally be the same as `makedocs`'s `build` and defaults to `"build"`.

`branch` is the branch where the generated documentation is pushed. If the branch does not exist, a new orphaned branch is created automatically. It defaults to `"gh-pages"`.

`deps` is the function used to install any additional dependencies needed to build the documentation. By default nothing is installed.

It can be used e.g. for a Markdown build. The following example installed the `pygments` and `mkdocs` Python packages using the `Deps.pip` function:

```
```

`make` is the function used to specify an additonal build phase. By default, nothing gets executed.

`devbranch` is the branch that "tracks" the in-development version of the generated documentation. By default this value is set to `"master"`.

`devurl` the folder that in-development version of the docs will be deployed. Defaults to `"dev"`.

`forcepush` a boolean that specifies the behavior of the git-deployment. The default (`forcepush = false`) is to push a new commit, but when `forcepush = true` the changes will be combined with the previous commit and force pushed, erasing the Git history on the deployment branch.

`versions` determines content and order of the resulting version selector in the generated html. The following entries are valied in the `versions` vector:

- `"v#"`: includes links to the latest documentation for each major release cycle (i.e. `v2.0`, `v1.1`).
- `"v#.#"`: includes links to the latest documentation for each minor release cycle (i.e. `v2.0`, `v1.1`, `v1.0`, `v0.1`).
- `"v#.#.#"`: includes links to all released versions.
- `"v^"`: includes a link to the docs for the maximum version (i.e. a link `vX.Y` pointing to `vX.Y.Z` for highest `X`, `Y`, `Z`, respectively).
- A pair, e.g. `"first" => "second"`, which will put `"first"` in the selector, and generate a url from which `"second"` can be accessed. The second argument can be `"v^"`, to point to the maximum version docs (as in e.g. `"stable" => "v^"`).

See Also

The Hosting Documentation section of the manual provides a step-by-step guide to using the `deploydocs` function to automatically generate docs and push them to GitHub.

source

`Documenter.Deps` – Module.

Exported module that provides build and deploy dependencies and related functions.

Currently only `pip` is implemented.

source

`Documenter.Deps.pip` – Function.

```
pip(deps)
```

Installs (as non-root user) all python packages listed in `deps`.

Examples

```
using Documenter

makedocs(
    # ...
)

deploydocs(
    deps = Deps.pip("pygments", "mkdocs", "mkdocs-material"),
    # ...
```

source

`Documenter.doctest` – Function.

```

```

Convenience method that runs and checks all the doctests for a given Julia package. `package` must be the `Module` object corresponding to the top-level module of the package. Behaves like an `@testset` call, returning a testset if all the doctests are successful or throwing a `TestSetException` if there are any failures. Can be included in other testsets.

Keywords

`manual` controls how manual pages are handled. By default (`manual = true`), `doctest` assumes that manual pages are located under `docs/src`. If that is not the case, the `manual` keyword argument can be passed to specify the directory. Setting `manual = false` will skip doctesting of manual pages altogether.

Additional keywords are passed on to the main `doctest` method.

source

```
doctest(source, modules; kwargs...)
```

Runs all the doctests in the given modules and on manual pages under the `source` directory. Behaves like an `@testset` call, returning a testset if all the doctests are successful or throwing a `TestSetException` if there are any failures. Can be included in other testsets.

The manual pages are searched recursively in subdirectories of `source` too. Doctesting of manual pages can be disabled if `source` is set to `nothing`.

Keywords

`testset` specifies the name of test testset (default `Doctests`).

`fix`, if set to `true`, updates all the doctests that fail with the correct output (default `false`).

Warning

When running `doctest(...; fix=true)`, Documenter will modify the Markdown and Julia source files. It is strongly recommended that you only run it on packages in Pkg's develop mode and commit any staged changes. You should also review all the changes made by `doctest` before committing them, as there may be edge cases when the automatic fixing fails.

source

`Documenter.DocMeta` – Module.

This module provides APIs for handling documentation metadata in modules.

The implementation is similar to how docstrings are handled in Base by the `Base.Docs` module — a special variable is created in each module that has documentation metadata.

Public API

- `DocMeta.getdocmeta`
- `DocMeta.setdocmeta!`

Supported metadata

- `DocTestSetup`: contains the doctest setup code for doctests in the module.

source

`Documenter.DocMeta.getdocmeta` – Function.

Returns the documentation metadata dictionary for the module `m`. The dictionary should be considered immutable and assigning values to it is not well-defined. To set documentation metadata values, `DocMeta.setdocmeta!` should be used instead.

source

```
getdocmeta(m::Module, key::Symbol, default=nothing)
```

Return the key entry from the documentation metadata for module `m`, or `default` if the value is unset.

source

`Documenter.DocMeta.setdocmeta!` – Function.

Set the documentation metadata value key for module `m` to `value`.

If `recursive` is set to `true`, it sets the same metadata value for all the submodules too. If `warn` is `true`, it prints a warning when key already exists and is gets rewritten.

source

DocumenterTools

DocumenterTools.generate – Function.

Create a documentation stub in path, which is usually a sub folder in the package root. The name of the package is determined automatically, but can be given with the name keyword argument.

generate can also be called without any arguments, in which case it simply puts all the generated files into a docs directory in the current working directory. This way, if you are already in the root directory of your package, you generally only need to call generate() to generate the documentation stub.

generate creates the following files in path:

```
.gitignore
src/index.md
make.jl
mkdocs.yml
Project.toml
```

Arguments

path file path to the documentation directory to be created (default is "docs").

Keywords Arguments

name is the name of the package (without .jl). If name is not given generate tries to detect it automatically.

format can be either :html (default), :markdown or :pdf corresponding to the format keyword to Documenter's makedocs function, see Documenter's manual.

Examples

```
julia> using DocumenterTools

julia> DocumenterTools.generate("path/to/MyPackage/docs")
```

```
DocumenterTools.generate(pkg::Module; dir = "docs", format = :html)
```

Same as generate(path::String) but the path and name is determined automatically from the module.

> **Note**
> The package must be in development mode. Make sure you run pkg> develop pkg from the Pkg REPL, or Pkg.develop("pkg") before generating docs.

Examples

```
julia> using DocumenterTools

julia> using MyPackage

julia> DocumenterTools.generate(MyPackage)
```

DocumenterTools.genkeys – Function.

Generates the SSH keys necessary for the automatic deployment of documentation with Documenter from a builder to GitHub Pages.

By default the links in the instructions need to be modified to correspond to actual URLs. The optional `user` and `repo` keyword arguments can be specified so that the URLs in the printed instructions could be copied directly. They should be the name of the GitHub user or organization where the repository is hosted and the full name of the repository, respectively.

This method of genkeys requires the following command lines programs to be installed:

- `which`
- `ssh-keygen`

Examples

```
julia> using DocumenterTools

julia> DocumenterTools.genkeys()
[ Info: add the public key below to https://github.com/$USER/$REPO/settings/keys with read/write
↪  access:

ssh-rsa
↪  AAAAB3NzaC2yc2EAAAaDAQABAAABAQDrNsUZYBWJtXYUk21wxZbX3KxcH8EqzR3ZdTna0Wgk...jNmUiGEMKrr0aqQMZEL2BG7
↪  username@hostname

[ Info: add a secure environment variable named 'DOCUMENTER_KEY' to
↪  https://travis-ci.com/$USER/$REPO/settings (if you deploy using Travis CI) or
↪  https://github.com/$USER/$REPO/settings/secrets (if you deploy using GitHub Actions) with
↪  value:

LS0tLS1CRUdJTiBSU0EgUFJJVkFURSBLRVktLS0tLQpNSUlFb3dJQkFBS0NBUUVBNnpiRkdXQVZpYlIy...QkVBRWFjY3BxaW9uNjFLaVdOcDU5T2Y


julia> DocumenterTools.genkeys(user="JuliaDocs", repo="DocumenterTools.jl")
[Info: add the public key below to https://github.com/JuliaDocs/DocumenterTools.jl/settings/keys
↪  with read/write access:

ssh-rsa
↪  AAAAB3NzaC2yc2EAAAaDAQABAAABAQDrNsUZYBWJtXYUk21wxZbX3KxcH8EqzR3ZdTna0Wgk...jNmUiGEMKrr0aqQMZEL2BG7
↪  username@hostname

[ Info: add a secure environment variable named 'DOCUMENTER_KEY' to
↪  https://travis-ci.com/JuliaDocs/DocumenterTools.jl/settings (if you deploy using Travis CI)
↪  or https://github.com/JuliaDocs/DocumenterTools.jl/settings/secrets (if you deploy using
↪  GitHub Actions) with value:
```

```
genkeys(package::Module; remote="origin")
```

Like the other method, this generates the SSH keys necessary for the automatic deployment of documentation with Documenter from a builder to GitHub Pages, but attempts to guess the package URLs from the Git remote.

`package` needs to be the top level module of the package. The `remote` keyword argument can be used to specify which Git remote is used for guessing the repository's GitHub URL.

This method requires the following command lines programs to be installed:

- `which`
- `git`
- `ssh-keygen`

Note

The package must be in development mode. Make sure you run `pkg> develop pkg` from the Pkg REPL, or `Pkg.develop("pkg")` before generating the SSH keys.

Examples

```
julia> using DocumenterTools

julia> DocumenterTools.genkeys(DocumenterTools)
[Info: add the public key below to https://github.com/JuliaDocs/DocumenterTools.jl/settings/keys
↪  with read/write access:

ssh-rsa
↪  AAAAB3NzaC2yc2EAAAaDAQABAAABAQDrNsUZYBWJtXYUk21wxZbX3KxcH8EqzR3ZdTna0Wgk...jNmUiGEMKrr0aqQMZEL2BG7
↪  username@hostname

[ Info: add a secure environment variable named 'DOCUMENTER_KEY' to
↪  https://travis-ci.com/JuliaDocs/DocumenterTools.jl/settings (if you deploy using Travis CI)
↪  or https://github.com/JuliaDocs/DocumenterTools.jl/settings/secrets (if you deploy using
↪  GitHub Actions) with value:
```

Chapter 17

Internals

17.1 Anchors

`Documenter.Anchors` – Module.

Defines the `Anchor` and `AnchorMap` types.

Anchors and AnchorMaps are used to represent links between objects within a document.

source

`Documenter.Anchors.Anchor` – Type.

Stores an arbitrary object called `.object` and it's location within a document.

Fields

- `object` – the stored object.
- `order` – ordering of `object` within the entire document.
- `file` – the destination file, in `build`, where the object will be written to.
- `id` – the generated "slug" identifying the object.
- `nth` – integer that unique-ifies anchors with the same `id`.

source

`Documenter.Anchors.AnchorMap` – Type.

Tree structure representating anchors in a document and their relationships with eachother.

Object Hierarchy

```
id -> file -> anchors
```

Each `id` maps to a `file` which in turn maps to a vector of `Anchor` objects.

source

`Documenter.Anchors.add!` – Method.

```
add!(m, anchor, id, file)
```

Adds a new Anchor to the AnchorMap for a given id and file.

Either an actual Anchor object may be provided or any other object which is automatically wrapped in an Anchor before being added to the AnchorMap.

source

Documenter.Anchors.anchor – Method.

```
anchor(m, id)
```

Returns the Anchor object matching id. file and n may also be provided. An Anchor is returned, or nothing in case of no match.

source

Documenter.Anchors.exists – Method.

```
exists(m, id, file, n)
```

Does the given id exist within the AnchorMap? A file and integer n may also be provided to narrow the search for existance.

source

Documenter.Anchors.isunique – Method.

```
isunique(m, id)
```

Is the id unique within the given AnchorMap? May also specify the file.

source

17.2 Builder

Documenter.Builder – Module.

Defines the Documenter.jl build "pipeline" named DocumentPipeline.

Each stage of the pipeline performs an action on a Documents.Document object. These actions may involve creating directory structures, expanding templates, running doctests, etc.

source

Documenter.Builder.CheckDocument – Type.

Checks that all documented objects are included in the document and runs doctests on all valid Julia code blocks.

source

Documenter.Builder.CrossReferences – Type.

Finds and sets URLs for each @ref link in the document to the correct destinations.

source

Documenter.Builder.Doctest – Type.

Runs all the doctests in all docstrings and Markdown files.

source

Documenter.Builder.DocumentPipeline – Type.

The default document processing "pipeline", which consists of the following actions:

- SetupBuildDirectory
- Doctest
- ExpandTemplates
- CrossReferences
- CheckDocument
- Populate
- RenderDocument

source

Documenter.Builder.ExpandTemplates – Type.

Executes a sequence of actions on each node of the parsed markdown files in turn.

source

Documenter.Builder.Populate – Type.

Populates the ContentsNodes and IndexNodes with links.

source

Documenter.Builder.RenderDocument – Type.

Writes the document tree to the build directory.

source

Documenter.Builder.SetupBuildDirectory – Type.

Creates the correct directory layout within the build folder and parses markdown files.

source

Documenter.Builder.walk_navpages – Method.

```
walk_navpages(visible, title, src, children, parent, doc)
```

Recursively walks through the Documents.Document's .user.pages field, generating Documents.NavNodes and related data structures in the process.

This implementation is the de facto specification for the .user.pages field.

source

17.3 CrossReferences

Documenter.CrossReferences – Module.

Provides the crossref function used to automatically calculate link URLs.

source

Documenter.CrossReferences.crossref – Method.

```
crossref(doc)
```

Traverses a Documents.Document and replaces links containg @ref URLs with their real URLs.

source

Documenter.CrossReferences.find_object – Method.

```
find_object(doc, binding, typesig)
```

Find the included Object in the doc matching binding and typesig. The matching heuristic isn't too picky about what matches and will only fail when no Bindings matching binding have been included.

source

17.4 DocChecks

Documenter.DocChecks – Module.

Provides the missingdocs, footnotes and linkcheck functions for checking docs.

source

Documenter.DocChecks.footnotes – Method.

```
footnotes(doc)
```

Checks footnote links in a Documents.Document.

source

Documenter.DocChecks.linkcheck – Method.

```
linkcheck(doc)
```

Checks external links using curl.

source

Documenter.DocChecks.missingdocs – Method.

```
missingdocs(doc)
```

Checks that a Documents.Document contains all available docstrings that are defined in the modules keyword passed to Documenter.makedocs.

Prints out the name of each object that has not had its docs spliced into the document.

source

17.5 DocMeta

Documenter.DocMeta.initdocmeta! – Function.

source

Documenter.DocMeta.META – Constant.

The unique `Symbol` that is used to store the metadata dictionary in each module.

source

`Documenter.DocMeta.METAMODULES` – Constant.

List of modules that have the metadata dictionary added.

source

`Documenter.DocMeta.METATYPE` – Type.

Type of the metadata dictionary.

source

`Documenter.DocMeta.VALIDMETA` – Constant.

Dictionary of all valid metadata keys and their types.

source

17.6 DocSystem

`Documenter.DocSystem` – Module.

Provides a consistent interface to retreiving `DocStr` objects from the Julia docsystem in both `0.4` and `0.5`.

source

`Documenter.DocSystem.binding` – Method.

Converts an object to a `Base.Docs.Binding` object.

```
binding(any)
```

Supported inputs are:

- `Binding`
- `DataType`
- `Function`
- `Module`
- `Symbol`

Note that unsupported objects will throw an `ArgumentError`.

source

`Documenter.DocSystem.convertmeta` – Method.

```
convertmeta(meta)
```

Converts a `0.4`-style docstring cache into a `0.5` one.

The original docstring cache is not modified.

source

Documenter.DocSystem.docstr – Method.

```
docstr(md; kws...)
```

Construct a DocStr object from a Markdown.MD object.

The optional keyword arguments are used to add new data to the DocStr's .data dictionary.

source

Documenter.DocSystem.getdocs – Function.

```
getdocs(binding)
getdocs(binding, typesig; compare, modules, aliases)
```

Find all DocStr objects that match the provided arguments:

- binding: the name of the object.
- typesig: the signature of the object. Default: Union{}.
- compare: how to compare signatures? Exact (==) or subtypes (<:). Default: <:.
- modules: which modules to search through. Default: all modules.
- aliases: check aliases of binding when nothing is found. Default: true.

Returns a Vector{DocStr} ordered by definition order in 0.5 and by type_morespecific in 0.4.

source

Documenter.DocSystem.getdocs – Function.

```
getdocs(object)
getdocs(object, typesig; kws...)
```

Accepts objects of any type and tries to convert them to Bindings before searching for the Binding in the docsystem.

Note that when conversion fails this method returns an empty Vector{DocStr}.

source

Documenter.DocSystem.multidoc – Function.

Construct a MultiDoc object from the provided argument.

Valid inputs are:

- Markdown.MD
- Docs.FuncDoc
- Docs.TypeDoc

source

17.7 DocTests

`Documenter.DocTests` – Module.

Provides the `doctest` function that makes sure that the `jldoctest` code blocks in the documents and docstrings run and are up to date.

source

`Documenter.DocTests.doctest` – Method.

```
doctest(blueprint, doc)
```

Traverses the pages and modules in the documenter blueprint, searching and executing doctests.

Will abort the document generation when an error is thrown. Use `doctest = false` keyword in `Documenter.makedocs` to disable doctesting.

source

17.8 Documenter

`Documenter.gitrm_copy` – Function.

```

```

Uses `git rm -r` to remove `dst` and then copies `src` to `dst`. Assumes that the working directory is within the git repository of `dst` is when the function is called.

This is to get around #507 on filesystems that are case-insensitive (e.g. on OS X, Windows). Without doing a `git rm` first, `git add -A` will not detect case changes in filenames.

source

`Documenter.git_push` – Function.

```
git_push(
    root, tmp, repo;
    branch="gh-pages", dirname="", target="site", sha="", devurl="dev", deploy_config
```

Handles pushing changes to the remote documentation branch. The documentation are placed in the `devurl` folder (if `git_tag(deploy_config)` returns `nothing`) or the version folder corresponding to `git_tag(deploy_config)` e.g. a `vX.Y.Z` directory.

source

17.9 DocumenterTools

`DocumenterTools.package_devpath` – Function.

```
package_devpath(pkg)
```

Returns the path to the top level directory of a devved out package source tree. The package is identified by its top level module `pkg`.

Generator

DocumenterTools.Generator – Module.

Provides the functions related to generating documentation stubs.

DocumenterTools.Generator.gitignore – Method.

```
gitignore()
```

Contents of the default `.gitignore` file.

DocumenterTools.Generator.index – Method.

```
index(pkgname)
```

Contents of the default `src/index.md` file.

DocumenterTools.Generator.make – Method.

```
make(pkgname; format)
```

Contents of the default `make.jl` file.

DocumenterTools.Generator.mkdocs – Method.

```
mkdocs(pkgname; description, author, url)
```

Contents of the default `mkdocs.yml` file.

DocumenterTools.Generator.project – Method.

```
project(; format)
```

Contents of the default `Project.toml` file.

DocumenterTools.Generator.savefile – Method.

```
savefile(f, root, filename)
```

Attempts to save a file at `$(root)/$(filename)`. `f` will be called with file stream (see open).

`filename` can also be a file in a subdirectory (e.g. `src/index.md`), and then then subdirectories will be created automatically.

17.10 Documents

Documenter.Documents – Module.

Defines Document and its supporting types

- Page
- User
- Internal
- Globals

source

`Documenter.Documents.Document` – Type.

Represents an entire document.

source

`Documenter.Documents.Globals` – Type.

`Page`-local values such as current module that are shared between nodes in a page.

source

`Documenter.Documents.Internal` – Type.

Private state used to control the generation process.

source

`Documenter.Documents.NavNode` – Type.

Element in the navigation tree of a document, containing navigation references to other page, reference to the `Page` object etc.

source

`Documenter.Documents.Page` – Type.

Represents a single markdown file.

source

`Documenter.Documents.User` – Type.

User-specified values used to control the generation process.

source

`Documenter.Documents.getplugin` – Method.

```
```

Retrieves the `Plugin` type for `T` stored in `doc`. If `T` was passed to `makedocs`, the passed type will be returned. Otherwise, a new `T` object will be created using the default constructor `T()`.

source

`Documenter.Documents.navpath` – Method.

Constructs a list of the ancestors of the `navnode` (inclding the `navnode` itself), ordered so that the root of the navigation tree is the first and `navnode` itself is the last item.

source

`Documenter.Documents.populate!` – Method.

```
populate!(document)
```

Populates the `ContentsNodes` and `IndexNodes` of the `document` with links.

This can only be done after all the blocks have been expanded (and nodes constructed), because the items have to exist before we can gather the links to those items.

source

Documenter.Documents.walk – Method.

```
walk(f, meta, element)
```

Calls `f` on `element` and any of its child elements. `meta` is a `Dict` containing metadata such as current module.

source

17.11 DOM

Documenter.Utilities.DOM – Module.

Provides a domain specific language for representing HTML documents.

Examples

```
using Documenter.Utilities.DOM

# `DOM` does not export any HTML tags. Define the ones we actually need.
@tags div p em strong ul li

div(
    p("This ", em("is"), " a ", strong("paragraph."),
    p("And this is ", strong("another"), " one"),
    ul(
        li("and"),
        li("an"),
        li("unordered"),
        li("list")
    )
```

Notes

All the arguments passed to a node are flattened into a single vector rather than preserving any nested structure. This means that passing two vectors of nodes to a `div` will result in a `div` node with a single vector of children (the concatenation of the two vectors) rather than two vector children. The only arguments that are not flattened are nested nodes.

String arguments are automatically converted into text nodes. Text nodes do not have any children or attributes and when displayed the string is escaped using escapehtml.

Attributes

As well as plain nodes shown in the previous example, nodes can have attributes added to them using the following syntax.

```
div[".my-class"](
    img[:src => "foo.jpg"],
    input["#my-id", :disabled]
```

In the above example we add a `class = "my-class"` attribute to the `div` node, a `src = "foo.jpg"` to the `img`, and `id = "my-id" disabled` attributes to the `input` node.

The following syntax is supported within `[...]`:

```
tag["#id"]
tag[".class"]
tag[".class#id"]
tag[:disabled]
tag[:src => "foo.jpg"]
```

Internal Representation

The `@tags` macro defines named `Tag` objects as follows

```

```

expands to

```

```

These `Tag` objects are lightweight representations of empty HTML elements without any attributes and cannot be used to represent a complete document. To create an actual tree of HTML elements that can be rendered we need to add some attributes and/or child elements using `getindex` or `call` syntax. Applying either to a `Tag` object will construct a new `Node` object.

```
tag(...)      # No attributes.
tag[...]      # No children.
```

All three of the above syntaxes return a new `Node` object. Printing of `Node` objects is defined using the standard Julia display functions, so only needs a call to `print` to print out a valid HTML document with all nessesary text escaped.

source

`Documenter.Utilities.DOM.@tags` – Macro.

Define a collection of `Tag` objects and bind them to constants with the same names.

Examples

Defined globally within a module:

```

```

Defined within the scope of a function to avoid cluttering the global namespace:

```
function template(args...)
    @tags div ul li
    # ...
```

source

`Documenter.Utilities.DOM.HTMLDocument` – Type.

A HTML node that wraps around the root node of the document and adds a DOCTYPE to it.

source

`Documenter.Utilities.DOM.Node` – Type.

Represents an element within an HTML document including any textual content, children `Nodes`, and attributes.

This type should not be constructed directly, but instead via `(...)` and `[...]` applied to a `Tag` or another `Node` object.

source

`Documenter.Utilities.DOM.Tag` – Type.

Represents a empty and attribute-less HTML element.

Use `@tags` to define instances of this type rather than manually creating them via `Tag(:tagname)`.

source

`Documenter.Utilities.DOM.escapehtml` – Method.

Escape characters in the provided string. This converts the following characters:

- `<` to `<`
- `>` to `>`
- `&` to `&`
- `'` to `&#39;`
- `"` to `&quot;`

When no escaping is needed then the same object is returned, otherwise a new string is constructed with the characters escaped. The returned object should always be treated as an immutable copy and compared using == rather than ===.

source

`Documenter.Utilities.DOM.flatten!` – Method.

Signatures

```
flatten!(f!, out, x::Atom)
```

Flatten the contents the third argument into the second after applying the function `f!` to the element.

source

17.12 Expanders

`Documenter.Expanders` – Module.

Defines node "expanders" that transform nodes from the parsed markdown files.

source

`Documenter.Expanders.AutoDocsBlocks` – Type.

Parses each code block where the language is `@autodocs` and replaces it with all the docstrings that match the provided key/value pairs `Modules = ...` and `Order = ...`.

````
```@autodocs
Modules = [Foo, Bar]
Order   = [:function, :type]
```
````

source

Documenter.Expanders.ContentsBlocks – Type.

Parses each code block where the language is `@contents` and replaces it with a nested list of all `Header` nodes in the generated document. The pages and depth of the list can be set using `Pages = [...]` and `Depth = N` where `N` is and integer.

````
```@contents
Pages = ["foo.md", "bar.md"]
Depth = 1
```
````

The default `Depth` value is 2.

source

Documenter.Expanders.DocsBlocks – Type.

Parses each code block where the language is `@docs` and evaluates the expressions found within the block. Replaces the block with the docstrings associated with each expression.

````
```@docs
Documenter
makedocs
deploydocs
```
````

source

Documenter.Expanders.EvalBlocks – Type.

Parses each code block where the language is `@eval` and evaluates it's content. Replaces the block with the value resulting from the evaluation. This can be useful for inserting generated content into a document such as plots.

````
```@eval
using PyPlot
x = linspace(-, π)
y = sin(x)
plot(x, y, color = "red")
savefig("plot.svg")
Markdown.parse("![Plot](plot.svg)")
```
````

source

Documenter.Expanders.ExampleBlocks – Type.

Parses each code block where the language is `@example` and evaluates the parsed Julia code found within. The resulting value is then inserted into the final document after the source code.

````
```@example
a = 1
b = 2
a + b
```
````

source

Documenter.Expanders.ExpanderPipeline – Type.

The default node expander "pipeline", which consists of the following expanders:

- TrackHeaders
- MetaBlocks
- DocsBlocks
- AutoDocsBlocks
- EvalBlocks
- IndexBlocks
- ContentsBlocks
- ExampleBlocks
- SetupBlocks
- REPLBlocks

source

Documenter.Expanders.IndexBlocks – Type.

Parses each code block where the language is `@index` and replaces it with an index of all docstrings spliced into the document. The pages that are included can be set using a key/value pair `Pages = [...]` such as

```
```@index
Pages = ["foo.md", "bar.md"]
```
```

source

Documenter.Expanders.MetaBlocks – Type.

Parses each code block where the language is `@meta` and evaluates the key/value pairs found within the block, i.e.

```
```@meta
CurrentModule = Documenter
DocTestSetup = quote
    using Documenter
end
```
```

source

Documenter.Expanders.REPLBlocks – Type.

Similar to the ExampleBlocks expander, but inserts a Julia REPL prompt before each toplevel expression in the final document.

source

Documenter.Expanders.SetupBlocks – Type.

Similar to the ExampleBlocks expander, but hides all output in the final document.

source

`Documenter.Expanders.TrackHeaders` – Type.

Tracks all `Markdown.Header` nodes found in the parsed markdown files and stores an `Anchors.Anchor` object for each one.

source

17.13 Markdown2

Documentation for the private `Markdown2` module.

Index

- `Documenter.Utilities.Markdown2`
- `Documenter.Utilities.Markdown2.List`
- `Documenter.Utilities.Markdown2.MD`
- `Documenter.Utilities.Markdown2.MarkdownBlockNode`
- `Documenter.Utilities.Markdown2.MarkdownInlineNode`
- `Documenter.Utilities.Markdown2.MarkdownNode`
- `Documenter.Utilities.Markdown2.Paragraph`
- `Documenter.Utilities.Markdown2.ThematicBreak`
- `Base.convert`
- `Documenter.Utilities.Markdown2.walk`

Docstrings

`Documenter.Utilities.Markdown2` – Module.

Provides types and functions to work with Markdown syntax trees.

The module is similar to the Markdown standard library, but aims to be stricter and provide a more well-defined API.

> **Note**
>
> Markdown2 does not provide a parser, just a data structure to represent Markdown ASTs.

Markdown nodes

The types in this module represent the different types of nodes you can have in a Markdown abstract syntax tree (AST). Currently it supports all the nodes necessary to represent Julia flavored Markdown. But having this as a separate module from the Markdown standard library allows us to consistently extend the node type we support (e.g. to support the raw HTML nodes from CommonMark, or strikethrough text from GitHub Flavored Markdown).

Markdown nodes split into to two different classes: block nodes and inline nodes. Generally, the direct children of a particular node can only be either inline or block (e.g. paragraphs contain inline nodes, admonitions contain block nodes as direct children).

In Markdown2, this is represented using a simple type hierarchy. All Markdown nodes are subtypes of either the `MarkdownBlockNode` or the `MarkdownInlineNode` abstract type. Both of these abstract types themselves are a subtype of the `MarkdownNode`.

Additional methods

- The `Base.convert(::Type{Markdown2.MD}, md::Markdown.MD)` method can be used to convert the Julia Markdown standard libraries ASTs into Markdown2 ASTs.
- The `walk` function can be used for walking over a `Markdown2.MD` tree.

source

`Documenter.Utilities.Markdown2.List` – Type.

If `.orderedstart` is `nothing` then the list is unordered. Otherwise is specifies the first number in the list.

source

`Documenter.Utilities.Markdown2.MD` – Type.

The root node of a Markdown document. Its children are a list of top-level block-type nodes. Note that `MD` is not a subtype of `MarkdownNode`.

source

`Documenter.Utilities.Markdown2.MarkdownBlockNode` – Type.

Supertype for all block-level Markdown nodes.

source

`Documenter.Utilities.Markdown2.MarkdownInlineNode` – Type.

Supertype for all inline Markdown nodes.

source

`Documenter.Utilities.Markdown2.MarkdownNode` – Type.

Supertype for all Markdown nodes.

source

`Documenter.Utilities.Markdown2.Paragraph` – Type.

Represents a paragraph block-type node. Its children are inline nodes.

source

`Documenter.Utilities.Markdown2.ThematicBreak` – Type.

A block node represeting a thematic break (a <hr> tag).

source

`Base.convert` – Method.

Converts a Markdown standard library AST into a Markdown2 AST.

source

`Documenter.Utilities.Markdown2.walk` – Function.

Calls `f(element)` on `element` and any of its child elements. The elements are assumed to be `Markdown2` elements.

source

17.14 MDFlatten

`Documenter.Utilities.MDFlatten` – Module.

Provides the `mdflatten` function that can "flatten" Markdown objects into a string, with formatting etc. stripped.

Note that the tests in `test/mdflatten.jl` should be considered to be the spec for the output (number of newlines, indents, formatting, etc.).

source

`Documenter.Utilities.MDFlatten.mdflatten` – Method.

Convert a Markdown object to a `String` of only text (i.e. not formatting info).

It drop most of the extra information (e.g. language of a code block, URLs) and formatting (e.g. emphasis, headers). This "flattened" representation can then be used as input for search engines.

source

17.15 Selectors

`Documenter.Utilities.Selectors` – Module.

An extensible code selection interface.

The `Selectors` module provides an extensible way to write code that has to dispatch on different predicates without hardcoding the control flow into a single chain of `if` statements.

In the following example a selector for a simple condition is implemented and the generated selector code is described:

```
abstract type MySelector <: Selectors.AbstractSelector end

# The different cases we want to test.
abstract type One    <: MySelector end
abstract type NotOne <: MySelector end

# The order in which to test the cases.
Selectors.order(::Type{One})    = 0.0
Selectors.order(::Type{NotOne}) = 1.0

# The predicate to test against.
Selectors.matcher(::Type{One}, x)    = x === 1
Selectors.matcher(::Type{NotOne}, x) = x !== 1

# What to do when a test is successful.
Selectors.runner(::Type{One}, x)    = println("found one")
Selectors.runner(::Type{NotOne}, x) = println("not found")

# Test our selector with some numbers.
for i in 0:5
    Selectors.dispatch(MySelector, i)
```

`Selectors.dispatch(Selector, i)` will behave equivalent to the following:

```
function dispatch(::Type{MySelector}, i::Int)
    if matcher(One, i)
        runner(One, i)
    elseif matcher(NotOne, i)
        runner(NotOne, i)
    end
```

and further to

```
function dispatch(::Type{MySelector}, i::Int)
    if i === 1
        println("found one")
    elseif i !== 1
        println("not found")
    end
```

The module provides the following interface for creating selectors:

- `order`
- `matcher`
- `runner`
- `strict`
- `disable`
- `dispatch`

source

`Documenter.Utilities.Selectors.AbstractSelector` – Type.

Root selector type. Each user-defined selector must subtype from this, i.e.

```
abstract type MySelector <: Selectors.AbstractSelector end

abstract type First  <: MySelector end
```

source

Documenter.Utilities.Selectors.disable – Method.

Disable a particular case in a selector so that it is never used.

```
```

source

Documenter.Utilities.Selectors.dispatch – Method.

Call `Selectors.runner(T, args...)` where T is a subtype of `MySelector` for which `matcher(T, args...)` is `true`.

```
```

source

Documenter.Utilities.Selectors.matcher – Function.

Define the matching test for each case in a selector, i.e.

```
Selectors.matcher(::Type{First}, x)  = x == 1
```

Note that the return type must be `Bool`.

To match against multiple cases use the `Selectors.strict` function.

source

Documenter.Utilities.Selectors.order – Function.

Define the precedence of each case in a selector, i.e.

```
Selectors.order(::Type{First})  = 1.0
```

Note that the return type must be `Float64`. Defining multiple case types to have the same order will result in undefined behaviour.

source

Documenter.Utilities.Selectors.runner – Function.

Define the code that will run when a particular `Selectors.matcher` test returns `true`, i.e.

```
Selectors.runner(::Type{First}, x)  = println("`x` is equal to `1`.")
```

source

Documenter.Utilities.Selectors.strict – Method.

Define whether a selector case will "fallthrough" or not when successfully matched against. By default matching is strict and does not fallthrough to subsequent selector cases.

```
# Adding a debugging selector case.
abstract type Debug <: MySelector end

# Insert prior to all other cases.
Selectors.order(::Type{Debug}) = 0.0

# Fallthrough to the next case on success.
Selectors.strict(::Type{Debug}) = false

# We always match, regardless of the value of `x`.
Selectors.matcher(::Type{Debug}, x) = true

# Print some debugging info.
```

source

17.16 TextDiff

Documenter.Utilities.TextDiff.splitby – Method.

```
splitby(reg, text)
```

Splits `text` at regex matches, returning an array of substrings. The parts of the string that match the regular expression are also included at the ends of the returned strings.

source

17.17 Utilities

Documenter.Utilities – Module.

Provides a collection of utility functions and types that are used in other submodules.

source

Documenter.Utilities.Object – Type.

Represents an object stored in the docsystem by its binding and signature.

source

Documenter.Utilities.assetsdir – Method.

Returns the path to the Documenter `assets` directory.

source

Documenter.Utilities.check_kwargs – Method.

Prints a formatted warning to the user listing unrecognised keyword arguments.

source

Documenter.Utilities.currentdir – Method.

Returns the current directory.

source

`Documenter.Utilities.doccat` – Method.

Returns the category name of the provided `Object`.

source

`Documenter.Utilities.docs` – Function.

```
```

Returns an expression that, when evaluated, returns the docstrings associated with `ex`.

source

`Documenter.Utilities.filterdocs` – Method.

```
```

Remove docstrings from the markdown object, `doc`, that are not from one of `modules`.

source

`Documenter.Utilities.get_commit_short` – Method.

```
get_commit_short(dir)
```

Returns the first 5 characters of the current git commit hash of the directory `dir`.

source

`Documenter.Utilities.isabsurl` – Method.

```
```

Checks whether `url` is an absolute URL (as opposed to a relative one).

source

`Documenter.Utilities.issubmodule` – Method.

```
```

Checks whether `sub` is a submodule of `mod`. A module is also considered to be its own submodule.

E.g. `A.B.C` is a submodule of `A`, `A.B` and `A.B.C`, but it is not a submodule of `D`, `A.D` nor `A.B.C.D`.

source

`Documenter.Utilities.mdparse` – Method.

```
```

Parses the given string as Markdown using `Markdown.parse`, but strips away the surrounding layers, such as the outermost `Markdown.MD`. What exactly is returned depends on the `mode` keyword.

The `mode` keyword argument can be one of the following:

- `:single` (default) – returns a single block-level object (e.g. `Markdown.Paragraph` or `Markdown.Admonition`) and errors if the string parses into multiple blocks.
- `:blocks` – the function returns a `Vector{Any}` of Markdown blocks.

- `:span` – Returns a `Vector{Any}` of span-level items, stripping away the outer block. This requires the string to parse into a single `Markdown.Paragraph`, the contents of which gets returned.

source

`Documenter.Utilities.nodocs` – Method.

Does the given docstring represent actual documentation or a no docs error message?

source

`Documenter.Utilities.object` – Method.

```
```

Returns a expression that, when evaluated, returns an `Object` representing ex.

source

`Documenter.Utilities.parseblock` – Method.

Returns a vector of parsed expressions and their corresponding raw strings.

Returns a `Vector` of tuples `(expr, code)`, where `expr` is the corresponding expression (e.g. a `Expr` or `Symbol` object) and `code` is the string of code the expression was parsed from.

The keyword argument `skip = N` drops the leading `N` lines from the input string.

If `raise=false` is passed, the `Meta.parse` does not raise an exception on parse errors, but instead returns an expression that will raise an error when evaluated. `parseblock` returns this expression normally and it must be handled appropriately by the caller.

source

`Documenter.Utilities.relpath_from_repo_root` – Method.

```
relpath_from_repo_root(file)
```

Returns the path of `file`, relative to the root of the Git repository, or `nothing` if the file is not in a Git repository.

source

`Documenter.Utilities.repo_root` – Method.

```
```

Tries to determine the root directory of the repository containing `file`. If the file is not in a repository, the function returns `nothing`.

The `dbdir` keyword argument specifies the name of the directory we are searching for to determine if this is a repostory or not. If there is a file called `dbdir`, then it's contents is checked under the assumption that it is a Git worktree or a submodule.

source

`Documenter.Utilities.slugify` – Method.

Slugify a string into a suitable URL.

source

`Documenter.Utilities.srcpath` – Method.

Find the path of a file relative to the `source` directory. `root` is the path to the directory containing the file `file`.

It is meant to be used with `walkdir(source)`.

source

`Documenter.Utilities.submodules` – Method.

Returns the set of submodules of a given root module/s.

source

`Documenter.Utilities.withoutput` – Method.

Call a function and capture all `stdout` and `stderr` output.

```
withoutput(f) --> (result, success, backtrace, output)
```

where

- `result` is the value returned from calling function `f`.
- `success` signals whether `f` has thrown an error, in which case `result` stores the `Exception` that was raised.
- `backtrace` a `Vector{Ptr{Cvoid}}` produced by `catch_backtrace()` if an error is thrown.
- `output` is the combined output of `stdout` and `stderr` during execution of `f`.

source

17.18 Writers

`Documenter.Writers` – Module.

A module that provides several renderers for `Document` objects. The supported formats are currently:

- `:markdown` – the default format.
- `:html` – generates a complete HTML site with navigation and search included.
- `:latex` – generates a PDF using LuaLaTeX.

source

`Documenter.Writers.render` – Method.

Writes a `Documents.Document` object to `.user.build` directory in the formats specified in the `.user.format` vector.

Adding additional formats requires adding new `Selector` definitions as follows:

```
abstract type CustomFormat <: FormatSelector end

Selectors.order(::Type{CustomFormat}) = 4.0 # or a higher number.
Selectors.matcher(::Type{CustomFormat}, fmt, _) = fmt === :custom
Selectors.runner(::Type{CustomFormat}, _, doc) = CustomWriter.render(doc)
```

source

Documenter.Writers.MarkdownWriter – Module.

A module for rendering Document objects to markdown.

source

Documenter.Writers.HTMLWriter – Module.

A module for rendering Document objects to HTML.

Keywords

HTMLWriter uses the following additional keyword arguments that can be passed to Documenter.makedocs: authors, pages, sitename, version. The behavior of HTMLWriter can be further customized by setting the format keyword of Documenter.makedocs to a HTML, which accepts the following keyword arguments: analytics, assets, canonical, disable_git, edit_branch and prettyurls.

sitename is the site's title displayed in the title bar and at the top of the *navigation menu. This argument is mandatory for HTMLWriter.

pages defines the hierarchy of the navigation menu.

Experimental keywords

version specifies the version string of the current version which will be the selected option in the version selector. If this is left empty (default) the version selector will be hidden. The special value git-commit sets the value in the output to git:{commit}, where {commit} is the first few characters of the current commit hash.

HTML Plugin options

The HTML Documenter.Plugin provides additional customization options for the HTMLWriter. For more information, see the HTML documentation.

Page outline

The HTMLWriter makes use of the page outline that is determined by the headings. It is assumed that if the very first block of a page is a level 1 heading, then it is intended as the page title. This has two consequences:

1. It is then used to automatically determine the page title in the navigation menu and in the <title> tag, unless specified in the .pages option.
2. If the first heading is interpreted as being the page title, it is not displayed in the navigation sidebar.

source

Documenter.Writers.HTMLWriter.HTML – Type.

Sets the behavior of HTMLWriter.

Keyword arguments

prettyurls (default true) – allows toggling the pretty URLs feature.

By default (i.e. when prettyurls is set to true), Documenter creates a directory structure that hides the .html suffixes from the URLs (e.g. by default src/foo.md becomes src/foo/index.html, but can

be accessed with via `src/foo/` in the browser). This structure is preferred when publishing the generate HTML files as a website (e.g. on GitHub Pages), which is Documenter's primary use case.

If `prettyurls = false`, then Documenter generates `src/foo.html` instead, suitable for local documentation builds, as browsers do not normally resolve `foo/` to `foo/index.html` for local files.

To have pretty URLs disabled in local builds, but still have them enabled for the automatic CI deployment builds, you can set `prettyurls = get(ENV, "CI", nothing) == "true"` (the specific environment variable you will need to check may depend on the CI system you are using, but this will work on Travis CI).

`disable_git` can be used to disable calls to `git` when the document is not in a Git-controlled repository. Without setting this to `true`, Documenter will throw an error and exit if any of the Git commands fail. The calls to Git are mainly used to gather information about the current commit hash and file paths, necessary for constructing the links to the remote repository.

`edit_branch` specifies which branch, tag or commit the "Edit on GitHub" links point to. It defaults to `master`. If it set to `nothing`, the current commit will be used.

`canonical` specifies the canonical URL for your documentation. We recommend you set this to the base url of your stable documentation, e.g. `https://juliadocs.github.io/Documenter.jl/stable`. This allows search engines to know which version to send their users to. See wikipedia for more information. Default is `nothing`, in which case no canonical link is set.

`analytics` can be used specify the Google Analytics tracking ID.

`assets` can be used to include additional assets (JS, CSS, ICO etc. files). See below for more information.

`sidebar_sitename` determines whether the site name is shown in the sidebar or not. Setting it to `false` can be useful when the logo already contains the name of the package. Defaults to `true`.

`highlights` can be used to add highlighting for additional languages. By default, Documenter already highlights all the "Common" highlight.js languages and Julia (`julia`, `julia-repl`). Additional languages must be specified by" their filenames as they appear on CDNJS for the highlight.js version Documenter is using. E.g. to include highlighting for YAML and LLVM IR, you would set `highlights = ["llvm", "yaml"]`. Note that no verification is done whether the provided language names are sane.

`mathengine` specifies which LaTeX rendering engine will be used to render the math blocks. The options are either KaTeX (default) or MathJax, enabled by passing an instance of `KaTeX` or `MathJax` objects, respectively. The rendering engine can further be customized by passing options to the `KaTeX` or `MathJax` constructors.

Default and custom assets

Documenter copies all files under the source directory (e.g. `/docs/src/`) over to the compiled site. It also copies a set of default assets from `/assets/html/` to the site's `assets/` directory, unless the user already had a file with the same name, in which case the user's files overrides the Documenter's file. This could, in principle, be used for customizing the site's style and scripting.

The HTML output also links certain custom assets to the generated HTML documents, specifically a logo and additional javascript files. The asset files that should be linked must be placed in `assets/`, under the source directory (e.g `/docs/src/assets`) and must be on the top level (i.e. files in the subdirectories of `assets/` are not linked).

For the **logo**, Documenter checks for the existence of `assets/logo.{svg,png,webp,gif,jpg,jpeg}`, in this order. The first one it finds gets displayed at the top of the navigation sidebar. It will also check for `assets/logo-dark.{svg,png,webp,gif,jpg,jpeg}` and use that for dark themes.

Additional JS, ICO, and CSS assets can be included in the generated pages by passing them as a list with the `assets` keyword. Each asset will be included in the <head> of every page in the order in which they are given. The type of the asset (i.e. whether it is going to be included with a <script> or a <link> tag) is determined by the file's extension – either `.js`, `.ico`[1], or `.css` (unless overridden with `asset`).

Simple strings are assumed to be local assets and that each correspond to a file relative to the documentation source directory (conventionally `src/`). Non-local assets, identified by their absolute URLs, can be included with the `asset` function.

source

`Documenter.Writers.HTMLWriter.ASSETS` – Constant.

The root directory of the HTML assets.

source

`Documenter.Writers.HTMLWriter.ASSETS_SASS` – Constant.

The directory where all the Sass/SCSS files needed for theme building are.

source

`Documenter.Writers.HTMLWriter.ASSETS_THEMES` – Constant.

Directory for the compiled CSS files of the themes.

source

`Documenter.Writers.HTMLWriter.MDBlockContext` – Constant.

`MDBlockContext` is a union of all the Markdown nodes whose children should be blocks. It can be used to dispatch on all the block-context nodes at once.

source

`Documenter.Writers.HTMLWriter.THEMES` – Constant.

List of Documenter native themes.

source

`Documenter.Writers.HTMLWriter.HTMLContext` – Type.

`HTMLWriter`-specific globals that are passed to `domify` and other recursive functions.

source

`Documenter.Writers.HTMLWriter.KaTeX` – Type.

An instance of the KaTeX type can be passed to `HTML` via the `mathengine` keyword to specify that the KaTeX rendering engine should be used in the HTML output to render mathematical expressions.

A dictionary can be passed via the `config` argument to configure KaTeX. It becomes the options argument of `renderMathInElement`. By default, Documenter only sets a custom `delimiters` option.

By default, the user-provided dictionary gets merged with the default dictionary (i.e. the resulting configuration dictionary will contain the values from both dictionaries, but e.g. setting your own `delimiters` value will override the default). This can be overridden by setting `override` to `true`, in which case the default values are ignored and only the user-provided dictionary is used.

source

[1] Adding an ICO asset is primarily useful for setting a custom `favicon`.

Documenter.Writers.HTMLWriter.MathJax – Type.

An instance of the MathJax type can be passed to HTML via the mathengine keyword to specify that the MathJax rendering engine should be used in the HTML output to render mathematical expressions.

A dictionary can be passed via the config argument to configure MathJax. It gets passed to the MathJax.Hub.Config function. By default, Documenter set custom configuration for tex2jax, config, jax, extensions and Tex.

By default, the user-provided dictionary gets merged with the default dictionary (i.e. the resulting configuration dictionary will contain the values from both dictionaries, but e.g. setting your own tex2jax value will override the default). This can be overridden by setting override to true, in which case the default values are ignored and only the user-provided dictionary is used.

source

Documenter.Writers.HTMLWriter.asset – Method.

Can be used to pass non-local web assets to HTML, where uri should be an absolute HTTP or HTTPS URL.

It accepts the following keyword arguments:

class can be used to override the asset class, which determines how exactly the asset gets included in the HTML page. This is necessary if the class can not be determined automatically (default).

Should be one of: :js, :css or :ico. They become a <script>, <link rel="stylesheet" type="text/css"> and <link rel="icon" type="image/x-icon"> elements in <head>, respectively.

islocal can be used to declare the asset to be local. The uri should then be a path relative to the documentation source directory (conventionally src/). This can be useful when it is necessary to override the asset class of a local asset.

Usage

```
Documenter.HTML(assets = [
    # Standard local asset
    "assets/extra_styles.css",
    # Standard remote asset (extension used to determine that class = :js)
    asset("https://example.com/jslibrary.js"),
    # Setting asset class manually, since it can't be determined manually
    asset("https://example.com/fonts", class = :css),
    # Same as above, but for a local asset
    asset("asset/foo.script", class=:js, islocal=true),
```

source

Documenter.Writers.HTMLWriter.collect_subsections – Method.

Returns an ordered list of tuples, (toplevel, anchor, text), corresponding to level 1 and 2 headings on the page. Note that if the first header on the page is a level 1 header then it is not included – it is assumed to be the page title and so does not need to be included in the navigation menu twice.

source

Documenter.Writers.HTMLWriter.copy_asset – Method.

Copies an asset from Documenters `assets/html/` directory to `doc.user.build`. Returns the path of the copied asset relative to `.build`.

source

`Documenter.Writers.HTMLWriter.domify` – Method.

Converts recursively a `Documents.Page`, `Markdown` or Documenter `*Node` objects into HTML DOM.

source

`Documenter.Writers.HTMLWriter.fixlinks!` – Method.

Replaces URLs in `Markdown.Link` elements (if they point to a local `.md` page) with the actual URLs.

source

`Documenter.Writers.HTMLWriter.get_url` – Method.

Returns the full path corresponding to a path of a `.md` page file. The the input and output paths are assumed to be relative to `src/`.

source

`Documenter.Writers.HTMLWriter.get_url` – Method.

Returns the full path of a `Documents.NavNode` relative to `src/`.

source

`Documenter.Writers.HTMLWriter.getpage` – Method.

Returns a page (as a `Documents.Page` object) using the `HTMLContext`.

source

`Documenter.Writers.HTMLWriter.mdconvert` – Method.

Convert a markdown object to a `DOM.Node` object.

The `parent` argument is passed to allow for context-dependant conversions.

source

`Documenter.Writers.HTMLWriter.navhref` – Method.

Get the relative hyperlink between two `Documents.NavNode`s. Assumes that both `Documents.NavNode`s have an associated `Documents.Page` (i.e. `.page` is not `nothing`).

source

`Documenter.Writers.HTMLWriter.navitem` – Method.

`navitem` returns the lists and list items of the navigation menu. It gets called recursively to construct the whole tree.

It always returns a `DOM.Node`. If there's nothing to display (e.g. the node is set to be invisible), it returns an empty text node (`DOM.Node("")`).

source

`Documenter.Writers.HTMLWriter.open_output` – Method.

Opens the output file of the navnode in write node. If necessary, the path to the output file is created before opening the file.

source

Documenter.Writers.HTMLWriter.pagetitle – Method.

Tries to guess the page title by looking at the <h1> headers and returns the header contents of the first <h1> on a page (or nothing if the algorithm was unable to find any <h1> headers).

source

Documenter.Writers.HTMLWriter.pretty_url – Method.

If prettyurls for HTML is enabled, returns a "pretty" version of the path which can then be used in links in the resulting HTML file.

source

Documenter.Writers.HTMLWriter.relhref – Method.

Calculates a relative HTML link from one path to another.

source

Documenter.Writers.HTMLWriter.render_html – Function.

Renders the main <html> tag.

source

Documenter.Writers.HTMLWriter.render_page – Method.

Constructs and writes the page referred to by the navnode to .build.

source

Documenter.Writers.HTMLWriter.render_settings – Method.

Renders the modal settings dialog.

source

Documenter.Writers.HTMLWriter.JS – Module.

Provides a namespace for JS dependencies.

source

Documenter.Writers.HTMLWriter.JS.highlightjs! – Function.

Add the highlight.js dependencies and snippet to a RequireJS declaration.

source

Documenter.Writers.LaTeXWriter – Module.

A module for rendering Document objects to LaTeX and PDF.

Keywords

LaTeXWriter uses the following additional keyword arguments that can be passed to makedocs: authors, sitename.

sitename is the site's title displayed in the title bar and at the top of the navigation menu. It goes into the `\title` LaTeX command.

authors can be used to specify the authors of. It goes into the `\author` LaTeX command.

source

`Documenter.Writers.LaTeXWriter.LaTeX` – Type.

Output format specifier that results in LaTeX/PDF output. Used together with `makedocs`, e.g.

```
makedocs(
    format = LaTeX()
```

The `makedocs` argument `sitename` will be used for the `\title` field in the tex document, and if the build is for a release tag (i.e. when the `"TRAVIS_TAG"` environment variable is set) the version number will be appended to the title. The `makedocs` argument `authors` should also be specified, it will be used for the `\authors` field in the tex document.

Keyword arguments

platform sets the platform where the tex-file is compiled, either `"native"` (default) or `"docker"`. See Other Output Formats for more information.

source

`Documenter.Plugin` – Type.

Any plugin that needs to either solicit user input or store information in a `Documents.Document` should create a subtype of `Plugin`. The subtype, `T <: Documenter.Plugin`, must have an empty constructor `T()` that initialized T with the appropriate default values.

To retrieve the values stored in T, the plugin can call `Documents.getplugin`. If T was passed to `makedocs`, the passed type will be returned. Otherwise, a new T object will be created.

source

Part VI

Contributing

This page details the some of the guidelines that should be followed when contributing to this package.

Chapter 18

Branches

From `Documenter` version `0.3` onwards `release-*` branches are used for tagged minor versions of this package. This follows the same approach used in the main Julia repository, albeit on a much more modest scale.

Please open pull requests against the `master` branch rather than any of the `release-*` branches whenever possible.

18.1 Backports

Bug fixes are backported to the `release-*` branches using `git cherry-pick -x` by a JuliaDocs member and will become available in point releases of that particular minor version of the package.

Feel free to nominate commits that should be backported by opening an issue. Requests for new point releases to be tagged in `METADATA.jl` can also be made in the same way.

18.2 `release-*` branches

- Each new minor version `x.y.0` gets a branch called `release-x.y` (a protected branch).
- New versions are usually tagged only from the `release-x.y` branches.
- For patch releases, changes get backported to the `release-x.y` branch via a single PR with the standard name "Backports for x.y.z" and label "Type: Backport". The PR message links to all the PRs that are providing commits to the backport. The PR gets merged as a merge commit (i.e. not squashed).
- The old `release-*` branches may be removed once they have outlived their usefulness.
- Patch version milestones are used to keep track of which PRs get backported etc.

Chapter 19

Style Guide

Follow the style of the surrounding text when making changes. When adding new features please try to stick to the following points whenever applicable.

19.1 Julia

- 4-space indentation;
- modules spanning entire files should not be indented, but modules that have surrounding code should;
- no blank lines at the start or end of files;
- do not manually align syntax such as = or `::` over adjacent lines;
- use `function ... end` when a method definition contains more than one toplevel expression;
- related short-form method definitions don't need a new line between them;
- unrelated or long-form method definitions must have a blank line separating each one;
- surround all binary operators with whitespace except for `::`, `^`, and `:`;
- files containing a single `module ... end` must be named after the module;
- method arguments should be ordered based on the amount of usage within the method body;
- methods extended from other modules must follow their inherited argument order, not the above rule;
- explicit `return` should be preferred except in short-form method definitions;
- avoid dense expressions where possible e.g. prefer nested `if`s over complex nested `?`s;
- include a trailing `,` in vectors, tuples, or method calls that span several lines;
- do not use multiline comments (#= and =#);
- wrap long lines as near to 92 characters as possible, this includes docstrings;
- follow the standard naming conventions used in `Base`.

19.2 Markdown

- Use unbalanced # headers, i.e. no # on the right hand side of the header text;
- include a single blank line between toplevel blocks;
- unordered lists must use * bullets with two preceding spaces;
- do not hard wrap lines;
- use emphasis (*) and bold (**) sparingly;
- always use fenced code blocks instead of indented blocks;
- follow the conventions outlined in the Julia documentation page on documentation.