Package Documentation

Vincent Picaud

May 5, 2018

Contents

1 Introduction

J4Org.jl is a Julia package that allows Julia doc generation into an Org-Mode document. The goal is to be able to code and document Julia packages without leaving Emacs and to reduce as much as possible the burden of documentation. This package depends on Tokenize.jl, to tokenize Julia code.

1.1 Minimal requirements

You need Org-Mode plus ob-julia.el, which has ESS as dependence, to be installed.

1.2 Getting started with a minimal example

The following is a minimal example you can reproduce to have a taste of what this package do.

1.2.1 Emacs configuration

You first need a minimal `init.el` file to configure Emacs.

```
(package-initialize)

(require 'ess-site)
;; if required
;; (setq  inferior-julia-program-name "/path/to/julia-release-basic")

(require 'org)
;; *replace me* with your own ob-julia.el file location
(add-to-list 'load-path "~/GitLab/WorkingWithOrgMode/EmacsFiles")
;; babel configuration
(setq org-confirm-babel-evaluate nil)
(org-babel-do-load-languages
 'org-babel-load-languages
 '((julia . t)))
```

1.2.2 A documented Julia `Foo` module

Then you need a documented module:

```
module Foo

export Point, foo

import Base: norm

#+Point L:Point_struct
# This is my Point structure
```

```julia
#
# *Example:*
#
# Creates a point p of coordinates (x=1,y=2).
#
# #+BEGIN_SRC julia :eval never :exports code
# p=Point(1,2)
# #+END_SRC
#
# You can add any valid Org mode directive. If you want to use
# in-documentation link, use [[norm_link_example][]]
#
struct Point
    x::Float64
    y::Float64
end

#+Point
# Creates Point at origin (0,0)
Point() = Point(0,0)

#+Enum
# An enum
@enum Alpha A=1 B C # just for example

#+Point,Method L:norm_link_example
# A simple function that computes √(x²+y²)
#
# *Example:*
#!p=Point(1.0,2.0);
#!norm(p)
#
# See: [[Point_struct][]]
#
norm(p::Point)::Float64 = sqrt(p.x*p.x+p.y*p.y)

# +Method,Internal
# An internal function
#
# For symbol that are not exported, do not forget the "Foo." prefix:
# !p=Point(1.0,2.0)
# !Foo.foo(2.0,p)
foo(r::Float64,p::Point) = Point(r*p.x,r*p.y)

end
```

I wanted to reduce the documentation process as much as possible. The template is very simple. Before each item you want to document add these comment lines:

```
#+Tag1,Tag2,... L:an_extra_link_if_required
#
# Here you can put any Org mode text, for instance $sin(x)$
#
#!sin(5) # julia code to be executed
#
# [[internal_link][]]
struct A_Documented_Struct
...
end
```

- **#+Tag1,Tag2,...** is mandatory, "#+" is followed by a list of tags. Later when you want to extract doc you can do filtering according these tags.
- **L:an_extra_link_if_required** is **not** mandatory. It defines a reference if you want to create doc links. The previous statement defines a link **target** named `an_extra_link_if_required`.
- [[**internal_link**][]] creates a link to a previously defined **L:internal_link**.
- **!sin(5)** will execute Julia code and include the output in the doc. If you only want to include Julia code without executing it, simply use Org mode source block:

```
# #+BEGIN_SRC julia :eval never :exports code
# sin(5)
# #+END_SRC
```

1. Support for "# +" and "# !" (since v0.2.0)

   As you can see (foo() function comments), "# +", "#+" and "# !", "#!" are synonyms. The motivation is a better integration with poporg Emacs package. With this Emacs package you can edit comment under OrgMode mode without being bothered by the "#" characters.

1.2.3 Minimal OrgMode document

This is the `foo.org` file.

```
#+PROPERTY: header-args:julia :session *my_session* :exports code :eval no-export
#+OPTIONS: ^:{}
#+TITLE: Getting Started with a minimal example

#+BEGIN_SRC julia :results output none :eval no-export :exports none
push!(LOAD_PATH,pwd())
#+END_SRC

#+BEGIN_SRC julia :results output none :eval no-export :exports none
using J4Org
initialize_boxing_module(usedModules=["Foo"])
documented_items=create_documented_item_array("Foo.jl")
#+END_SRC

* Example

Prints all documented items, except those tagged with "Internal"
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
print_org_doc(documented_items,tag_to_ignore=["Internal"],header_level=0)
#+END_SRC
```

- **push!(LOAD_PATH,pwd())** tells Julia where it can find our local `Foo` module. This statement is only required if the documented module is in an unusual place.
- **using J4Org** uses this package
- **initialize_boxing_module(usedModules=[”Foo”])** defines what are the modules to use when executing Julia code extracted from the doc (the ”#!” statements). Here we are documenting the `Foo` module, hence we must use it. Note that you can also use any number of extra modules for instance with `["Foo", "ExtraModule", ...]`. See initialize_boxing_module(...) for further details.
- **create_documented_item_array(”Foo.jl”)** creates the list of documented items from file ”Foo.jl”. You can use a list of files and a directory, see create_documented_item_array(...) for further details.
- **print_org_doc(...,tag_to_ignore=[”Internal”],..)** prints all documented items, except those tagged with ”Internal”, see print_org_doc(...) for further details

1.2.4 Generating the doc

To check that it works you can start a fresh emacs with

```
emacs -q --load init.el foo.org &
```

then type:

- `C-c C-v b` + `RET` to execute all source code blocks
- `C-c C-e h o` to html-export the file
- `C-c C-e l o` to pdf-export the file

You should get this minimal_example/foo.html file.

1.2.5 Improving exported document style

This was a minimal example, you can have a better look for the exported documents by including css theme, etc. This is the approach we used to generate **this** document (also see the main.pdf PDF file). Another example is DirectConvolution.jl documentation.

2 More examples

We still use our `Foo` module to provide more examples. The complete API is detailed after.

2.1 `print_org_doc` options

The print_org_doc(...) function has several options, let's see some usage examples

2.1.1 `header_level`

This integer can have these values:

- **-1**: do not print header nor index, see header_level=-1
- **0**: print header beginning with "-", see header_level=0.
- **l>0** create subsection of level **l**, for instance **header_level=3** creates subsections beginning with **3** stars. See header_level=5. **Caveat:** for **l>0** AFAIK there is a bug in OrgMode, because a residual **:RESULT:** is printed.

1. header_level=-1

```
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
documented_items=create_documented_item_array("minimal_example/Foo.jl");
print_org_doc(documented_items,tag="Method",header_level=-1)
#+END_SRC
```

This will generate:

```
foo(r::Float64,p::Point)
```

An internal function

For symbol that are not exported, do not forget the "Foo." prefix:

```
p=Point(1.0,2.0)
Foo.foo(2.0,p)
```

```
Foo.Point(1.0, 2.0)
Foo.Point(2.0, 4.0)
```

Foo.jl:46

```
norm(p::Point)::Float64
```

A simple function that computes $\sqrt{x^2+y^2}$

Example:

```
p=Point(1.0,2.0);
norm(p)
```

```
2.23606797749979
```

See: struct Point

Foo.jl:35

2. header_level=0

```
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
documented_items=create_documented_item_array("minimal_example/Foo.jl");
print_org_doc(documented_items,tag="Method",header_level=0)
#+END_SRC
```

This will generate:

Index: [**f**] foo [**n**] norm

- **`foo`**

```
foo(r::Float64,p::Point)
```

An internal function
For symbol that are not exported, do not forget the "Foo." prefix:

```
p=Point(1.0,2.0)
Foo.foo(2.0,p)
```

```
Foo.Point(1.0, 2.0)
Foo.Point(2.0, 4.0)
```

Foo.jl:46, back to index

- **`norm`**

```
norm(p::Point)::Float64
```

A simple function that computes $\sqrt{x^2+y^2}$
Example:

```
p=Point(1.0,2.0);
norm(p)
```

```
2.23606797749979
```

See: struct Point

Foo.jl:35, back to index

3. header_level=5

```
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
documented_items=create_documented_item_array("minimal_example/Foo.jl");
print_org_doc(documented_items,tag="Method",header_level=5)
#+END_SRC
```

This will generate:

:RESULTS:

Index: [**f**] foo [**n**] norm

(a) **foo**

```
foo(r::Float64,p::Point)
```

An internal function
For symbol that are not exported, do not forget the "Foo." prefix:

```
p=Point(1.0,2.0)
Foo.foo(2.0,p)
```

```
Foo.Point(1.0, 2.0)
Foo.Point(2.0, 4.0)
```

Foo.jl:46, back to index

(b) **norm**

```
norm(p::Point)::Float64
```

A simple function that computes $\sqrt{x^2+y^2}$
Example:

```
p=Point(1.0,2.0);
norm(p)
```

```
2.23606797749979
```

See: struct Point

Foo.jl:35, back to index

2.1.2 `tag`, `tag_to_ignore`, `identifier`

These options allow to select items to include:

- `tag` a string or an array of strings, collects all items with at least one tag in this `tag` option.
- `tag_to_ignore` a string or an array of strings, ignore all items with at least one tag in this `tag_to_ignore` option.
- `identifier` a string that stands for the structure, abstract type or function name. Collects all items with this `identifier` name.

For instance we can print `norm` identifier, restricted to `Point` tag, as follows:

```
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
documented_items=create_documented_item_array("minimal_example/Foo.jl");
print_org_doc(documented_items,identifier="norm", tag="Point",header_level=-1)
#+END_SRC
```

This will generate:

```
norm(p::Point)::Float64
```

A simple function that computes $\sqrt{x^2 + y^2}$

Example:

```
p=Point(1.0,2.0);
norm(p)
```

```
2.23606797749979
```

See: struct Point

Foo.jl:35

2.1.3 `complete_link`

If you look back at `tag`, `tag_to_ignore`, `identifier` you can see, at the end of the `norm` function documentation, that the Point_struct link is not active. The reason is that the `Point` structure is not present. The `complete_link` option, if set to `true` will try to fix all dangling links by including all the required documented items. For instance, with:

```
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
documented_items=create_documented_item_array("minimal_example/Foo.jl");
print_org_doc(documented_items,identifier="norm", tag="Point",header_level=-1,
              complete_link=true)
#+END_SRC
```

This will generate:

```
struct Point
```

This is my Point structure

Example:

Creates a point p of coordinates $(x = 1, y = 2)$.

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

You can add any valid Org mode directive. If you want to use in-documentation link, use norm(...)

Foo.jl:8

```
norm(p::Point)::Float64
```

A simple function that computes $\sqrt{x^2 + y^2}$

Example:

```
p=Point(1.0,2.0);
norm(p)
```

```
2.23606797749979
```

See: struct Point

Foo.jl:35

you see that the `Point` structure is included to make the struct_Point link active.

2.1.4 `link_prefix`

You can create link from your OrgMode document to Julia documented items that have defined a "L:link_target". However like these items can be extracted at several places in your OrgMode document you need to define a prefix to avoid multiple targets with the same name.

For instance, chose a prefix, here "my_prefix" and use:

```
print_org_doc(documented_items,...,link_prefix="my_prefix_")
```

then you can create a regular OrgMode link to this item using [[**my_prefix_link_target**][**some_text**]].

2.1.5 `case_sensitive`

When set to true, generates an index as follows:

```
[A] ..., [B] ...,[a] ..., [b] ...,
```

When set to false, do not split upper/lower cases and group all A,a;B,b together:

```
[A] ..., [B] ...
```

2.1.6 `boxingModule`

Comments starting with "#!" are executed in a boxed environment

```
module MyBoxing
using RequiredPackage_1,RequiredPackage_2,...
end
```

```
using MyBoxing

# execute "#!" statements here
```

This boxing is defined by the initialize_boxing_module(...) function:

```
initialize_boxing_module(boxingModule="MyBoxing",
                         usedModules=["RequiredPackage_1","RequiredPackage_2",...⌋
                         ↪  ])
```

This `boxingModule` option allows you to chose your boxing environment:

```
print_org_doc(documented_items,boxingModule="MyBoxing",...)
```

2.2 Error reporting

Error reporting is performed as OrgMode comment. For instance if you execute:

```
#+BEGIN_SRC julia :results output drawer :eval no-export :exports results
documented_items=create_documented_item_array("minimal_example/Foo.jl");
print_org_doc(documented_items,tag="Method",header_level=-1)
#+END_SRC
```

you will get:

```
#+RESULTS:
:RESULTS:
# =WARNING:= Link target ("Point_struct", "") not found
...
:END:
```

2.3 Compatibility with docstring / documenter.jl

You can still use something like:

```
"""
    foo()

foo function ...
"""
#+Tags...
# foo function ...
foo() = ...
```

3 API

The API is simple, with very few functions:

Index: [c] create_documented_item_array, create_documented_item_array_dir [i] initialize_boxing_module [p] print_org_doc

- **`create_documented_item_array`**

```
function create_documented_item_array(filename::String)::Array{Documented_Item,1}
```

Reads a Julia code file and returns an array of documented items.

documented_item.jl:91, back to index

```
function create_documented_item_array(filename_list::Array{String,1})::Array{Docu⌋
↪ mented_Item,1}
```

Reads an array of Julia code files and returns an array of documented items.

Usage example:

```
create_documented_item_array(["file1","file2",...])
```

Note: instead of a list of files you can also specify a directory, see create_documented_item_array_dir(...)

documented_item.jl:130, back to index

- **`create_documented_item_array_dir`**

```julia
function create_documented_item_array_dir(dirname::String)
```

Reads all *.jl files in a directory and returns an array of documented items.

documented_item.jl:152, back to index

- **`initialize_boxing_module`**

```julia
function initialize_boxing_module(;
                                  boxingModule::String="BoxingModule",
                                  usedModules::Vector{String}=String[],
                                  force::Bool=false)::Void
```

Initialize a boxing module. This module is used to run Julia comment code snippet (tagged by ”#!” or by ”# !”)

Example:

```julia
initialize_boxing_module(boxingModule="MyBoxing",
                         usedModules=["RequiredPackage_1",
                                      "RequiredPackage_2",...])
```

creates

```julia
module MyBoxing
using RequiredPackage_1,RequiredPackage_2,...
end
```

and future ”# !” statements are executed after using MyBoxing:

```julia
using MyBoxing
# !statements
```

evaluate.jl:18, back to index

- **`print_org_doc`**

```julia
function print_org_doc(di_array::Array{Documented_Item,1};
                       tag::Union{String,Array{String,1}}="",
                       tag_to_ignore::Union{String,Array{String,1}}="",
                       identifier::String="",
                       header_level::Int=0,
                       link_prefix::String=randstring(),
                       complete_link::Bool=false,
                       case_sensitive::Bool=true,
                       boxingModule::String="BoxingModule")
```

Prints generated documentation to be exported by OrgMode, this is the main function of the `J4Org` package.

Org-Mode Usage example:

```julia
#+BEGIN_SRC julia :results output drawer :eval no-export :exports
↪  results
documented_items =
↪  create_documented_item_array_dir("~/GitLab/MyPackage.jl/src/");
print_org_doc(documented_items,tag="API",header_level=0)
#+END_SRC
```

Arguments:

- `tag`: tags to collect when generating the documentation
- `tag_to_ignore`: tags to ignore when generating the documentation
- `identifier`: generates documentation for this "identifier". Can be a function name, a structure name, etc...
- `link_prefix`: allows to add a prefix to extra link (#+tag L=extra_link). this is can be useful to avoid link name conflict when performing local doc extraction.
- `complete_link`: if true, try to fix link without target by adding extra items
- `case_sensitive`: case sensitive index.
- `boxingModule`: specifies the context in which "#!" code will be executed. See initialize_boxing_module(...) for details.

main.jl:350, back to index

4 Unit tests

```
Test Summary: | Pass  Total
J4Org         |  102    102
```