Finalizing Your Julia Package: Documentation, Testing, Coverage, and Publishing

Christopher Rackauckas[1]

[1]Affiliation not available

April 17, 2023

the WINNOWER

COMPUTER SCIENCES

Finalizing Your Julia Package: Documentation, Testing, Coverage, and Publishing

CHRISTOPHER RACKAUCKAS

CORRESPONDENCE:
me@chrisrackauckas.com

DATE RECEIVED:
August 18, 2018

DOI:
10.15200/winn.153459.99339

ARCHIVED:
August 18, 2018

CITATION:
Christopher Rackauckas, Finalizing Your Julia Package: Documentation, Testing, Coverage, and Publishing, *The Winnower* 5:e153459.99339 , 2018 , DOI: 10.15200/winn.153459.99339

In this tutorial we will go through the steps to finalizing a Julia package. At this point you have some functionality you wish to share with the world... what do you do? You want to have documentation, code testing each time you commit (on all the major OSs), a nice badge which shows how much of the code is tested, and put it into metadata so that people could install your package just by typing Pkg.add("Pkgname"). How do you do all of this?

Note: At anytime feel free to checkout my package repository DifferentialEquations.jl which should be a working example.

GENERATE THE PACKAGE AND GET IT ON GITHUB

First you will want to generate your package and get it on Github repository. Make sure you have a Github account, and then setup the environment variables in the git shell:

```
$ git config --global user.name "FULL NAME"
$ git config --global user.email "EMAIL"
$ git config --global github.user "USERNAME"
```

Now you can generate your package via

```
using PkgDev
PkgDev.generate("PkgName","license")
```

For the license, I tend to use MIT since it is quite permissive. This will tell you where your package was generated (usually in your Julia library folder). Take your function files and paste them into the /src folder in the package. In your /src folder, you will have a file PkgName.jl. This file defines your module. Generally you will want it to look something like this:

```
module PkgName

#Import your packages
using Pkg1, Pkg2, Pkg3
import Base: func1 #Any function you add dispatches to need to be imported directly

abstract AbType #Define abstract types before the types they abstract!

include("functionsForPackage.jl") #Include all the functionality

export coolfunc, coolfunc2 #Export the functions you want users to use

end
```

Now try on your computer using PkgName. Try your functions out. Once this is all working, this means you have your package working locally.

WRITE THE DOCUMENTATION

For documentation, it's recommended to use Documenter.jl. The other packages, Docile.jl and Lexicon.jl, have been deprecated in favor of Documenter.jl. Getting your documentation to generate starts with writing docstrings. Docstrings are strings in your source code which are used for generating documentation. It is best to use docstrings because these will also show up in the REPL, i.e. if someone types ?coolfunc, your docstrings will show here.

To do this, you just add strings before your function definitions. For example,

```
"Defines a cool function. Returns some stuff"
function coolFunc()
  ...
end

"""
Defines an even cooler function. ``LaTeX``.

```math
SameAs$$LaTeX
```

Returns
 * Markdown works in here
"""
function coolFunc2()
  ...
end
```

Once you have your docstrings together, you can use them to generate your documentation. Install Documenter.jl in your local repository by cloning the repository with Pkg.clone("PkgLocation"). Make a new folder in the top directory of your package named /docs. In this directory, make a file make.jl and add the following lines to the file:

```
using Documenter, PkgName

makedocs(modules=[PkgName],
        doctest=true)

deploydocs(deps   = Deps.pip("mkdocs", "python-markdown-math"),
    repo = "github.com/GITHUBNAME/GITHUBREPO.git",
    julia  = "0.4.5",
    osname = "linux")
```

Don't forget to change PkgName and repo to match your project. Now make a folder in this directory named /src (i.e. it's /docs/src). Make a file named index.md. This will be the index of your documentation. You'll want to make it something like this:

```
#Documentation Title

Some text describing the package.

## Subtitle

More text

## Tutorials

```@contents
Pages = [
    "tutorials/page1.md",
    "tutorials/page2.md",
    "tutorials/page3.md"
```
```

```
    ]
Depth = 2
```

Another Section
```@contents
Pages = [
    "sec2/page1.md",
    "sec2/page2.md",
    "sec2/page3.md"
    ]
Depth = 2
```

Index

```@index
```

At the top we explain the page. The next part adds 3 pages to a "Tutorial" section of the documentation, and then 3 pages to a "Another Section" section of the documentation. Now inside /docs/src make the directories tutorial and sec2, and add the appropriate pages page1.md, page2.md, page3.md. These are the Markdown files that the documentation will use to build the pages.

To build a page, you can do something like as follows:

Title

Some text describing this section

Subtitle

```@docs
PkgName.coolfunc
PkgName.coolfunc2
```

What this does is it builds the page with your added text/titles on the top, and then puts your docstrings in below. Thus most of the information should be in your docstrings, with quick introductions before each page. So if your docstrings are pretty complete, this will be quick.

BUILD THE DOCUMENTATION

Now we will build the documentation. cd into the /docs folder and run make.jl. If that's successful, then you will have a folder /docs/build. This contains markdown files where the docstrings have been added. To turn this into a documentation, first install mkdocs. Now add the following file to your /docs folder as mkdocs.yml:

```
site_name:          PkgName
repo_url:           https://github.com/GITHUBUSER/PkgName
site_description:   Description
site_author:        You
theme:              readthedocs

markdown_extensions:
  - codehilite
  - extra
  - tables
  - fenced_code
  - mdx_math # For LaTeX

extra_css:
  - assets/Documenter.css
```

```
extra_javascript:
  - https://cdn.mathjax.org/mathjax/latest/MathJax.js?config=TeX-AMS_HTML
  - assets/mathjaxhelper.js

docs_dir: 'build'

pages:
- Introduction: index.md
- Tutorial:
  - Title 1: tutorials/page1.md
  - Title 2: tutorials/page2.md
  - Title 3: tutorials/page3.md
- Another Section:
  - Title 1: sec2/page1.md
  - Title 2: sec2/page2.md
  - Title 3: sec2/page3.md
```

Now to build the webpage, cd into /docs and run `mkdocs build`, and then `mkdocs serve`. Go to the local webserver that it tells you and check out your documentation.

TESTING

Now that we are documented, let's add testing. In the top of your package directory, make a folder /test. In there, make a file runtests.jl. You will want to make it say something like this:

```
#!/usr/bin/env julia

#Start Test Script
using PkgName
using Base.Test

# Run tests

tic()
println("Test 1")
@time @test include("test1.jl")
println("Test 2")
@time @test include("test2.jl")
toc()
```

This will run the files /test/test1.jl and /test/test2.jl and work if they both return a boolean. So make these test files use some of your package functionality and at the bottom make sure it returns a boolean saying whether the tests passed or failed. For example, you can have it make sure some number is close to what it should be, or you can just put `true` on the bottom on the file. Now use

```
Pkg.test("PkgName")
```

And make sure your tests pass. Now setup accounts at Travis CI (for Linux and OSX testing) and AppVoyer (for Windows testing). Modify .travis.yml to be like the following:

```
# Documentation: http://docs.travis-ci.com/user/languages/julia/
language: julia
os:
  - linux
  - osx
julia:
  - nightly
  - release
  - 0.4.5
matrix:
  allow_failures:
    - julia: nightly
notifications:
```

```
  email: false
script:
#  - if [[ -a .git/shallow ]]; then git fetch --unshallow; fi
  - julia -e 'Pkg.clone(pwd())'
  - julia -e 'Pkg.test("PkgName",coverage=true)'
after_success:
  - julia -e julia -e 'Pkg.add("Documenter")'
  - julia -e 'cd(Pkg.dir("PkgName")); include(joinpath("docs", "make.jl"))'
  - julia -e 'cd(Pkg.dir("PkgName")); Pkg.add("Coverage"); using Coverage; Codecov.submit(Codecov.process_folder())'
  - julia -e 'cd(Pkg.dir("PkgName")); Pkg.add("Coverage"); using Coverage; Coveralls.submit(process_folder())'
```

If you are using matplotlib/PyPlot you will want to add

```
ENV["PYTHON"]=""; Pkg.build("PyCall"); using PyPlot;
```

before Pkg.test("PkgName",coverage=true). Now edit your appvoyer.yml to be like the following:

```
environment:
  matrix:
  - JULIAVERSION: "julialang/bin/winnt/x86/0.4/julia-0.4-latest-win32.exe"
  - JULIAVERSION: "julialang/bin/winnt/x64/0.4/julia-0.4-latest-win64.exe"
matrix:
  allow_failures:
    - JULIAVERSION: "julianightlies/bin/winnt/x86/julia-latest-win32.exe"
    - JULIAVERSION: "julianightlies/bin/winnt/x64/julia-latest-win64.exe"
branches:
  only:
    - master
    - /release-.*/

notifications:
  - provider: Email
    on_build_success: false
    on_build_failure: false
    on_build_status_changed: false

install:
# Download most recent Julia Windows binary
  - ps: (new-object net.webclient).DownloadFile(
        $("http://s3.amazonaws.com/"+$env:JULIAVERSION),
        "C:\projects\julia-binary.exe")
  - set PATH=C:\Miniconda3;C:\Miniconda3\Scripts;%PATH%
# Run installer silently, output to C:\projects\julia
  - C:\projects\julia-binary.exe /S /D=C:\projects\julia

build_script:
# Need to convert from shallow to complete for Pkg.clone to work
  - IF EXIST .git\shallow (git fetch --unshallow)
  - C:\projects\julia\bin\julia -e "versioninfo();
      Pkg.clone(pwd(), \"PkgName\"); Pkg.build(\"PkgName\")"

test_script:
  - C:\projects\julia\bin\julia --check-bounds=yes -e "Pkg.test(\"PkgName\")"
```

ADD COVERAGE

I was sly and already added all of the coverage parts in there! This is done by the commands which add Coverge.jl, the keyword coverage=true in Pkg.test, and then specific functions for sending the coverage data to appropriate places. Setup an account on Codecov and Coveralls.

FIX UP README

Now update your readme to match your documentation, and add the badges for testing, coverage, and

docs from the appropriate websites.

UPDATE YOUR REPOSITORY

Now push everything into your Git repository. `cd` into your package directory and using the command line do:

```
git add --all
git commit -m "Commit message"
git push origin master
```

or something of the like. On Windows you can use their GUI. Check your repository and make sure everything is there. Wait for your tests to pass.

PUBLISH YOUR PACKAGE

Now publish your package. This step is optional, but if you do this then people can add your package by just doing `Pkg.add("PkgName")`. To do this, simply run the following:

```
Pkg.update()
using PkgDev
PkgDev.register("PkgName")
PkgDev.tag("PkgName")
PkgDev.publish()
```

This will give you a url. Put this into your browser and write a message with your pull request and submit it. If all goes well, they will merge the changes and your package will be registered with METADATA.jl.

That's it! Now every time you commit, your package will automatically be tested, coverage will be calculated, and documentation will be updated. Note that for people to get the changes you made to your code, they will need to run `Pkg.checkout("PkgName")` unless you tag and publish a new version.