How to build a package in Julia

Giulio Benedetti

21/03/2022

Overview

This tutorial is meant to walk you through the package development workflow in Julia. I will base my explanations on my own experience building FdeSolver.jl and MicrobiomeAnalysis.jl as well as valuable resources by others that I came across online.

Basics

To start off, write down some content in a julia script, say `example.jl`; for instance, it may entail a working example of the functionality you'd like to offer through your new package. Once you realise that some of the code you wrote could be wrapped up into one or more functions, define those functions and move them into a separate file, say `main.jl`.

You can now import the utilities from `main.jl` to `example.jl` with:

```
include("main.jl")
```

But if `main.jl` lives in a subdirectory, you'll have to let Julia know about it. One option is to specify the relative path of the file in the `include` command. A better option is to push a custom path in the loading path of Julia:

```
push!(LOAD_PATH, "subdirectory")
```

Also, if you want to use external utilities, call them in `main.jl` and they will also be exported in `example.jl`:

```
# lazy load module
import FdeSolver
# load module and export all functions
using MicrobiomeTools
```

All of the above works as a good temporary solution, as long as you don't need external utilites or multiple source scripts. In such case, you might want to define a module within a new file `SomeModule.jl`, wich will function as the core of your new package:

```
# define module to load all ingredients you need for your package to work
module SomeModule

# 1. load externals
import FdeSolver
using MicrobiomeTools

# 2. include internals
include("main1.jl")
include("main2.jl")

# 3. export internals
export my_function1
```

```
export my_function2

end
```

Now you can run something like `using SomeModule` in your `example.jl` script and hopefully you'll have access to your functions without having to deal with errors. If you do run into errors, try to troubleshoot with `push!` or reboot Julia.

Learn more about modules at JuliaLang and JuliaNotes.jl.

Make a package template

It might be time-consuming to build your own architecture from scratch, but thanks to PkgTemplates.jl this procedure is automatic:

```
# create a package template named "MyPkg"
using PkgTemplates
t = Template()
t("MyPkg")
```

There is a bunch of additional plugins and features that you can insert in your package draft, such as CI and documentation, but I suggest that you don't overdo with them, because they make it more difficult to find your way among the many generated files. More often than not I just create a basic template with the command above and then add more features manually.

Also, it is good practice to generate the draft package in an empty directory and only then you import the modules and examples that you've already written. You can put the former in the `src` and the latter in the `examples` directories, respectively.

Learn more about package templates at PkgTemplates.jl and JuliaLang.

Manage dependencies

You might wonder how your package keeps track of its credentials, such as name, uuid, authors and name, as well as more practical stuff like dependencies, campatibility requirements and similar things. All this information is stored in the `Project.toml`.

Especially when it comes to dependencies, it is not a smart idea to manually modify the keys and values within this file; it is much more efficient to leave this job to the Julia package manager, which you can access by typing the special key `]`. For instance, you can activate the current project and automatically add new deps to it as follows:

```
# switch to the package manager command line
]
# activate the project defined by the "Project.toml" in the current directory
activate .
# check project status and current dependencies
status
# add new dependencies to the "Project.toml"
add FdeSolver, MicrobiomeAnalysis
```

If you then check the `Project.toml`, you'll see the new deps listed under `[deps]`.

Another important matter is addressed by the `[compat]` section of this file; namely, it accounts for the compatibility requirements of your package utilities. Again, those keys don't have to be inserted manually, since Julia provides the so-called CompatHelper bot which takes care of the compat requirements for you by making pull requests in the GitHub repo. To trigger the bot, all you need to do is include the following GitHub Actions workflow as `.github/workflows/CompatHelper.yml`:

```
name: CompatHelper
on:
  schedule:
    - cron: 0 0 * * *
  workflow_dispatch:
jobs:
  CompatHelper:
    runs-on: ubuntu-latest
    steps:
      - name: Pkg.add("CompatHelper")
        run: julia -e 'using Pkg; Pkg.add("CompatHelper")'
      - name: CompatHelper.main()
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          COMPATHELPER_PRIV: ${{ secrets.DOCUMENTER_KEY }}
        run: julia -e 'using CompatHelper; CompatHelper.main()'
```

A word of care should be made that the CompatHelper doesn't check whether the deps you added make sense, it just finds the best compat requirement, so you are responsible to provide only those dependencies that are meaningful for your package.

Learn more about package components and dependencies, the Julia REPL and the CompatHelper at Pkg.jl, Julia REPL and CompatHelper.jl.

Develop new features

From now on, I suggest that you follow these steps every time you enter Julia and want to further build your package, here named "ProjectName":

```
# switch to shell command line
;
# set project as working directory
cd ProjectName.jl/
[delete-key]
# switch to package manager command line
]
# activate current project
activate .
[delete-key]
# load project utilities
using ProjectName
```

Initially, many different errors might arise due to precompilation issues and local conflicts between different versions of the package being developed. There is no single and easy solution for this and sometimes you just have to live with it until the package gets registered, but note that those are mainly local issues and everything (the package building) should run smoothly on the GitHub Actions host machine, once you set it up.

Still, you can try to start Julia with a few optargs to skip precompilation and directly activate your project:

```
cd ProjectName.jl
julia --startup-file=no --project=.
```

Learn more about setting up project environments in this article.

Implement unit testing

Tests can be found in the `test` directory. The package template has already prepared the draft in `runtests.jl` that you can fill with `@test` macros, which check if the conditionals in the same line are true or false. For example:

```
using Test
using ProjectName

@testset "ProjectName.jl" begin

    var1 = 2 + 2
    var2 = 2 * 2

    @test var1 == var2

end
```

You can then run the tests locally from the package manager command line as follows:

```
]
activate .
# run tests defined in "test" directory
test
```

This is quite nice. However, the amount of tests you make will increase exponentially as you add more features to your package, and you might want to split the tests into multiple files to avoid confusion and conflicts among dependencies (if some utilities depend on different externals). For this, `SafeTestSets.jl` comes into play:

```
using SafeTestsets

@safetestset "Run first test file" begin include("test1.jl") end

@safetestset "Run second test file" begin include("test2.jl") end

@safetestset "Run third test file" begin include("test3.jl") end
```

After which, you can write tests with the same syntax as in two chunks upstream and run them as in the second-to-last chunk, with the `@safetestset` being defined in `runtests.jl` and all other `@testset` in as many files as you like. Also, don't forget to add `SageTestSets.jl` to your extras and targets in the `Project.toml`.

Learn more about unit testing and macros at Test and JuliaLang.

Analyse code coverage

It's time to automatise unit testing and code coverage analysis. There are several options out there, but personally I fancy Codecov, because it's free open source and doesn't give too many problems. Begin with adding the following workflow to your `.github/workflows/` directory, maybe as `CI.yml`:

```
name: CI
on:
  - push
  - pull_request
jobs:
  test:
    name: Julia ${{ matrix.version }} - ${{ matrix.os }} - ${{ matrix.arch }} - ${{ github.event_name }}
```

```
    runs-on: ${{ matrix.os }}
    strategy:
      fail-fast: false
      matrix:
        version:
          - '1.6'
          - 'nightly'
        os:
          - ubuntu-latest
          - macOS-latest
        arch:
          - x64
    steps:
      - name: Check out
        uses: actions/checkout@v2

      - name: Set up Julia
        uses: julia-actions/setup-julia@v1
        with:
          version: ${{ matrix.version }}
          arch: ${{ matrix.arch }}

      - name: Load cache
        uses: actions/cache@v1
        env:
          cache-name: cache-artifacts
        with:
          path: ~/.julia/artifacts
          key: ${{ runner.os }}-test-${{ env.cache-name }}-${{ hashFiles('**/Project.toml') }}
          restore-keys: |
            ${{ runner.os }}-test-${{ env.cache-name }}-
            ${{ runner.os }}-test-
            ${{ runner.os }}-
      - name: Build package
        uses: julia-actions/julia-buildpkg@v1

      - name: Run tests
        uses: julia-actions/julia-runtest@v1

      - name: Process code coverage
        uses: julia-actions/julia-processcoverage@v1

      - name: Run codecov action
        uses: codecov/codecov-action@v1
        with:
          file: lcov.info
        env:
          CODECOV_TOKEN: ${{ secrets.CODECOV_TOKEN }}
```

The last line up here will look for a secret token stored in your repository to connect and upload the code coverage reports to Codecov, so go to your Codecov account, enable access to your repository and store the secret token provided by Codecov in the settings of your repo. After a couple of commits, you should start to see percent coverage and other statistics on the Codecov page of your repo and can also add the Codecov badge to its README.

Learn more about code coverage at this tutorial and at Codecov Quick Start.

Write and deploy documentation

Now we'll look into the `docs` directory, which is responsible for the building and deployment of the documentation. This directory has its own `Project.toml`, therefore you can activate it and add new deps just like you did for the main project. At least, it should contain `Documenter` (the documentation generator for Julia) and your main project as `[deps]` entries.

The package template has already prepared the files `src/index.md`, which serves as the home page of the deployed docs, and `make.jl`, which tells Julia how to build the docs, what dependencies to load, what files to include and in which order they should appear. The latter should look something like this:

```
using ProjectName, Documenter
ENV["GKSwstype"] = "100"

base_url = "https://github.com/OrganisationName/ProjectName.jl/blob/main/"

makedocs(format=Documenter.HTML(),
         authors = "authors",
         sitename = "ProjectName.jl",
         modules = [ProjectName],
         pages=[
             "Home" => "index.md",
             "Examples" => "examples.md"
               ])

deploydocs(repo="github.com/OrganisationName/ProjectName.jl", push_preview=true)
```

After you've added some content to the `index.md` and maybe also to `examples.md` (down below we see worthwhile material to include), try to build the docs locally with the following code:

```
# switch to the shell command line
;
# set "docs" as working directory
cd docs
# build docs
julia make.jl
```

After a while, you'll see a `build` directory appearing in `docs`, click on `index.html` to open the local deployment of the docs.

The docs should primarily showcase the descriptions of the utilities defined in `src` just next to the main module. Those descriptions are called docstrings and you can annotate them upstream the corresponding function in the `src/` files. For example:

```
"""
    sum(a::Float64, b::Float64)
Computes the sum of `a` and `b`.
# Arguments
- `a::Float64`: first arg.
- `b::Float64`: second arg.
"""
sum(a::Float64, b::Float54) = a + b
```

The docstring enclosed between the triple quotes can then be rendered in the docs with the following syntax:

```
sum
function2
function3
```

Code chunks with examples can also be run inside the docs and the produced images can be added in the form of md links:

```
using ProjectName
using Plots

out = sum(2, 3)
plot(out, 2, 3)

savefig("plot1.png"); nothing # hide
```

And then `![plot1](plot1.png)` in the next line. Note that this might take some troubleshooting before it actually works out.

Learn more about generating documentation in Julia and docstring syntax at Documenter.jl.

Host documentation online

Deploying the docs online should go quite smoothly if you managed to build them locally. In case you didn't, deployment on a GitHub Actions host server could actually solve the issues and conflicts you had locally, because it helps run the code in an isolated environment where conflicts between package versions and dependencies are less likely to happen, and if they do happen, that ensures you that is a global problem that inevitably needs to be solved and is not due to your own machine.

First off, add this workflow to your `.github/workflows` directory, maybe as `Documenter.yml`:

```
name: Documenter
on:
  - push
  - pull_request
jobs:
  docs:
    name: Documentation
    runs-on: ubuntu-latest
    steps:
      - name: Check out
        uses: actions/checkout@v2

      - name: Set up Julia
        uses: julia-actions/setup-julia@v1
        with:
          version: '1.7'

      - name: Build package
        run: |
          julia --project=docs -e '
            import Pkg; Pkg.add("Documenter")
            using Pkg
            Pkg.develop(PackageSpec(path=pwd()))
            Pkg.instantiate()'
      - name: Run doctests
        run: |
          julia --project=docs -e '
```

```
            import Pkg; Pkg.add("Documenter")
            using Documenter: doctest
            using ProjectName
            doctest(ProjectName)'
      - name: Deploy documentation
        run: julia --project=docs docs/make.jl
        env:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          DOCUMENTER_KEY: ${{ secrets.DOCUMENTER_KEY }}
```

Next, commit the changes to main. After the workflow has run, a new branch "gh-pages" should be automatically created. In case it isn't, you have to go to the repo settings and manually set gh-pages as the default branch for deployment. From now on, every time you push a new tag to main, the latest version of the docs will be rendered at "https://OrganisationName.github.io/ProjectName.jl/stable/". Practically, here's what you need to do to trigger deployment:

```
# commit changes locally
git commit -m "Add new features"
# create a new tag locally
git tag -a v0.1.0 -m "Release version 0.1.0"
# push changes to origin/main
git push origin main
# push tags to origin
git push --tag
```

Then the `Documenter.jl` workflow should be triggered and will be followed by another workflow named "pages build and deployment", which will present you the link to your new website. It is nice to wrap it up into a badge and showcase it in the README.

Keep in mind that reusing tags, pushing commits but no new tags or tags but no new commits won't trigger deployment (yes, I tried all of that). Sometimes you might need to delete older tags both locally and on GitHub, because they tend to cluster very quickly and you can't come up with any new names for the tags. Sometimes you also need to clear the cookies from your browser before you're able to see the latest version of the docs.

Learn more about hosting documentation at JuliaNotes.jl and Documenter.jl.

Register the new package

Before registering your package, consider the following checklist:

- package successfully compiles on a local machine
- functions and utilities are loaded and callable
- docs are rendered and include utility docstrings
- code coverage lies above 70% and tests don't fail

If so, then the next step is to install `Registrator.jl` on your GitHub repository through these instructions and add a new workflow as `.github/workflows/TagBot.yml`:

```
name: TagBot
on:
  issue_comment:
    types:
      - created
  workflow_dispatch:
jobs:
  TagBot:
    if: github.event_name == 'workflow_dispatch' || github.actor == 'JuliaTagBot'
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: JuliaRegistries/TagBot@v1
        with:
          token: ${{ secrets.GITHUB_TOKEN }}
          ssh: ${{ secrets.DOCUMENTER_KEY }}
```

The workflow above opens a pull request in the General Registry every time you write the comment `@JuliaRegistrator register` inside an issue of your repo. After registration is triggered, you can check if the pull request passes all the checks and can therefory be automatically merged after a stopwatch time of about 3 days. Usually, a few fixes might be necessary, such as:

- setting compat requirements for dependencies and julia itself (I recommend "1" for the latter)
- changing the package name because it's too short or too similar to other package names

For the first case, just update the `Compat.toml`, commit and retrigger the registrator with the comment; the pull request will be updated. For the second case, you'll have to change the name of your repo and triggering registration will open a new pull request. The change of name might also cause some issues for the docs deployment, so make sure that you changed all occurrences of the name in the `Documenter.yml` as well.

New versions can also be registered with this method, but the version has to be manually upgraded in the `Project.toml` and should follow the guidelines for semantic versioning. In addition, packages can also be submitted via the JuliaHub interface, which is more user-friendly for those who are not familiar with GitHub issues.

Learn more about package registration, the General Registry and semantic versioning at JuliaHub, standard and guidelines and this specification.

Find more answers

If you're looking for an answer, the community is the best place to get some help very quickly. My first choice is always the JuliaLang Forum. If that doesn't work, the Julia Language Slack is next door. But the docs are also ok, if you're either certain or desperate to find your answer there. There is also this great video tutorial that helped me get started developing packages in Julia.