Photo by Guillaume Henrotte (https://unsplash.com/@guigui1410) on Unsplash (https://unsplash.com/)

Package Registration and Tests

12 minute read

In this lesson we will learn how to **write tests** to check if a package is working properly and how to automatically check if those tests are passed after every commit to the master branch of your project. Furthermore, we will learn how to **publish a finished package** on the **Julia Registry**.

For this lesson we will write tests and see the procedure to register a package which I have been developing recently, `MultiQuad.jl` (https://github.com/aurelio-amerio/MultiQuad.jl/). `MultiQuad` is a convenient wrapper around some integration routines provided by `QuadGK` (https://github.com/JuliaMath/QuadGK.jl) and `Cuba` (https://github.com/giordano/Cuba.jl). It lets you compute 1D, 2D and 3D finite integrals with custom integration bounds.

Your package will have to be hosted on GitHub (https://github.com/) on a public repository if you want to perform integration tests with Travis and be able register it as an official package.

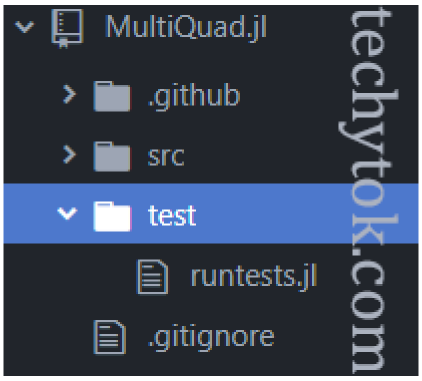
Writing tests

When you write your code, you should think of some tests which it should pass. For example, in the case of `MultiQuad.jl`, since we want to perform numerical integration, we should check if the numerical results reasonably coincide with the analytical results.

`MultiQuad` exports three functions: `quad`, `dblquad` and `tplquad`: we want to write some tests for all those functions. When we are writing tests, we should try to **test every function and option provided** by the package.

In Julia, in order to write tests for a package we need to first create a package using `Pkg.generate("PkgName")`. For more information on how to create packages, please refer to this lesson (https://techytok.com/lesson-packages/).

Once we have finished writing a part of the package, we can start writing tests. In the package folder, create a folder called `test` and inside that folder create a file called `runtests.jl`. Your project tree should look like this:



Please open the `runtests.jl` file, which is where we will write the tests for our package.

We need to first activate the project using `Pkg.activate()` and then we need to add the `Test` package to our project using `Pkg.add("Test")`. In order to write tests, we need to import our package, the `Test` package and every other package which will be used. In our case we will write:

```julia
using MultiQuad, Test, Unitful
```

We can now start writing some tests.

A test is an expression which should evaluate to true or false: true means that the test is successful and false means that the test has failed. To indicate to Julia that a particular expression is a test, we use the `@test` macro, for example:

```julia
>>>@test 42>4
Test Passed

>>>@test sin(3)>5
Test Failed at ---:1
  Expression: sin(3) > 5
  Evaluated: 0.1411200080598672 > 5
```

It is possible to write a set of tests for a common field, for example we can write a series of tests for a function. In the case of `MultiQuad` I have written three test tests, one for each function, which test all the possible arguments for the functions.

To create a test set, use the `@testset` macro:

```julia
@testset "name of the test set" begin
    # write some tests
    @test 3>1
    # .... etc
end
```

In the case of `MultiQuad` we want to check if the result of an integration is correct, within the accuracy we have chosen. Thus I compute a numerical integration and compare the result to the correct analytic solution (computed using Wolfram Mathematica (https://www.wolfram.com/mathematica/)):

```julia
 1  @testset "quad" begin
 2      rtol = 1e-10
 3  
 4      xmin = 0
 5      xmax = 4
 6      func1(x) = x^2 * exp(-x)
 7      res = 2 - 26 / exp(4)
 8      int, err = quad(func1, xmin, xmax, rtol = rtol)
 9      @test isapprox(
10          int,
11          res,
12          atol = err,
13      )
14  end
```

At line 1 we choose the relative tolerance at which the integration routine should stop (which is proportional to the error on the final result). At line 7 we define the analytical result and at line 8 we compute the integral and store the result and the error estimation on the result. At line 9 I use the function `isapprox` which tests whether the two arguments are equal within the given tolerance, which equals to checking whether
$|a - b| < tolerance$

Next we need to write many more tests to check all the options available. I will not deal with all the tests for this package, but I want to focus on two: checking whether units of measurement are handled properly and checking that an error is thrown when I pass an unavailable integration method.

In order to check if the units are handled properly, I add this piece of code to the `quad` test set:

```julia
1  xmin2 = 0 * u"m"
2  xmax2 = 4 * u"m"
3  func2(x) = x^2
4  res2 = 64 / 3 * u"m^3"
5  int2, err2 = quad(func2, xmin2, xmax2, rtol = rtol)
6  @test isapprox(int2, res2, atol = err2)
```

As you can see at line 4, now the result has a unit of measurement. If the test succeeds, then units are likely to be handled properly.

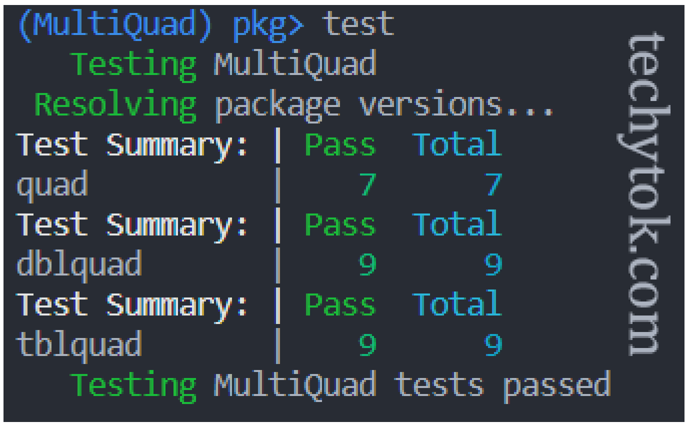
To check whether the proper error is thrown by a function (in our case an `ErrorException` when we try to call `quad` with an integration method which is not available), we use the `@test_throws` macro:

```julia
>>>@test_throws ErrorException quad(func1, 0, 4, method = :none)
Test Passed
Thrown: ErrorException
```

You can find the file with all the tests for `MultiQuad.jl` here (https://github.com/aurelio-amerio/MultiQuad.jl/blob/master/test/runtests.jl).

To run all the tests written inside the `runtests.jl` file, you can type in the REPL

```
] test
```



```
(MultiQuad) pkg> test
   Testing MultiQuad
 Resolving package versions...
Test Summary: | Pass  Total
quad          |    7      7
Test Summary: | Pass  Total
dblquad       |    9      9
Test Summary: | Pass  Total
tblquad       |    9      9
   Testing MultiQuad tests passed
```

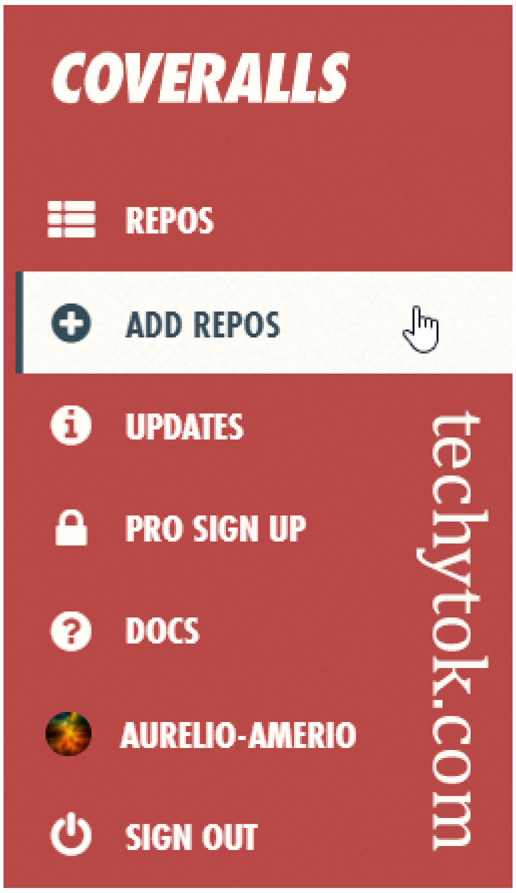
We shall now move on to automating those tests.

Setting up Coveralls

Coveralls (https://coveralls.io/) is a free platform which helps you understand the percentage of your code which is covered by your tests. Ideally, you should test all the parts of your code, but concretely this is not easy to do. Coveralls gives you a metric and some tools to determine how much of your code is covered by your tests.

You can register for free with your GitHub account here (https://coveralls.io/sign-up).

Once you are logged in, you can select the “add repos” page from the sidebar:



Or simply go to this page (https://coveralls.io/repos/new).

You should now enable the repository for which you want to test the coverage:

ON
AURELIO-AMERIO / MultiQuad.jl
techytok.com
DETAILS
GITHUB

After this step the setup of Coveralls is finished, we will see later on how Travis can automatically submit coverage reports to Coveralls automatically after a build.

An example report of a coverage test can be found here (https://coveralls.io/github/aurelio-amerio/MultiQuad.jl).

Setting up Codecov

Codecov (https://codecov.io/) is a free platform which performs coverage tests similar to Coveralls. Although it is not mandatory, if you log in with your GitHub account you can have an overview of all your repository for which coverage data is available.

There is no setup required and Travis will automatically submit coverage data to codecov at build time.

An example report of a coverage test can be found here (https://codecov.io/gh/aurelio-amerio/MultiQuad.jl).

You can choose to use either or both Codecov and Coveralls, but at least one coverage test platform should be included in your project.

Setting up Travis

Travis CI (https://travis-ci.com/) is a hosted continuous integration service used to build and test software projects hosted at GitHub. Travis is free for public repositories, which is fine since the Julia package we plan to register has to be hosted on a public repository.

Travis can be used to run the tests we have written on `runtests.jl` on many versions of Julia and on different operating systems. If all the tests are successful and if our code coverage is high (>90%) we can be pretty confident that our code will work properly on every configuration a user may have.

Travis uses `.yml` configuration files which contain all the information needed to perform the build tests. We shall now create a file called `.travis.yml` at the root of our package directory and paste the following code:

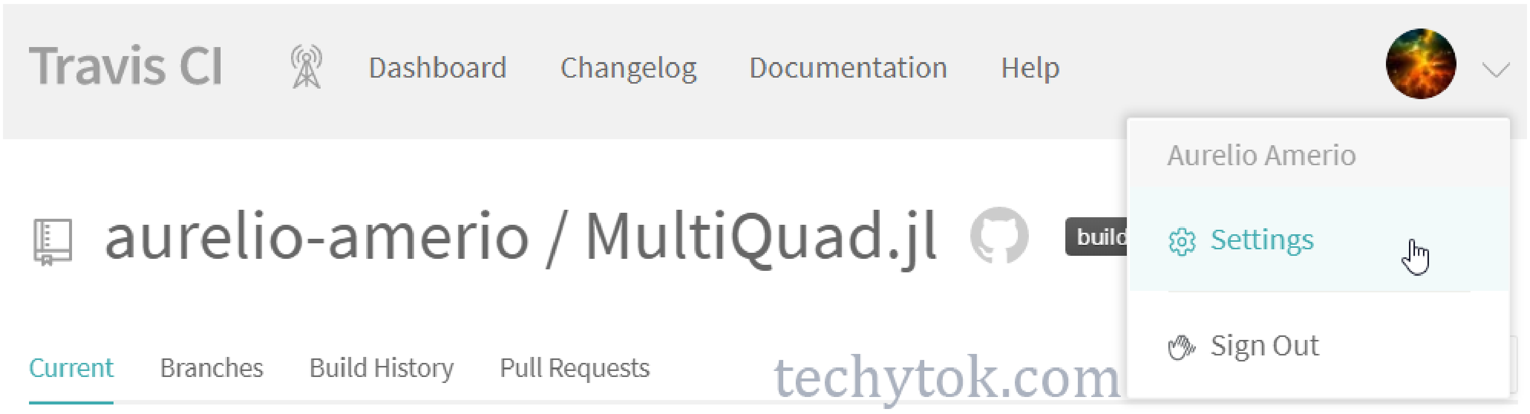
```
 1  language: julia
 2  os:
 3    - windows
 4    - linux
 5    - osx
 6  julia:
 7    - 1.0
 8    - 1.1
 9    - 1.2
10    - 1.3
11    - nightly
12  matrix:
13    allow_failures:
14      - julia: nightly
15  notifications:
16    email: false
17  after_success:
18    # push coverage results to Coveralls
19    - julia -e 'using Pkg; cd(Pkg.dir("MultiQuad")); Pkg.add("Coverage"); using Coverage;
20  Coveralls.submit(Coveralls.process_folder())'
21    # push coverage results to Codecov
  - julia -e 'using Pkg; cd(Pkg.dir("MultiQuad")); using Coverage; Codecov.submit(process_folder())'
```

This is a pretty generic configuration file which should work for every project. I want to test my package on Windows, Linux and OSX (lines 3 to 5) and I want it to work on Julia from 1.0 to 1.3 (lines 7 to 10). We also test our package on the nightly version of Julia (line 11) but we allow this build to fail (line 13-14). If our build fails on the nightly version, we have to consider that in the future we may have to change our package for it to continue working.

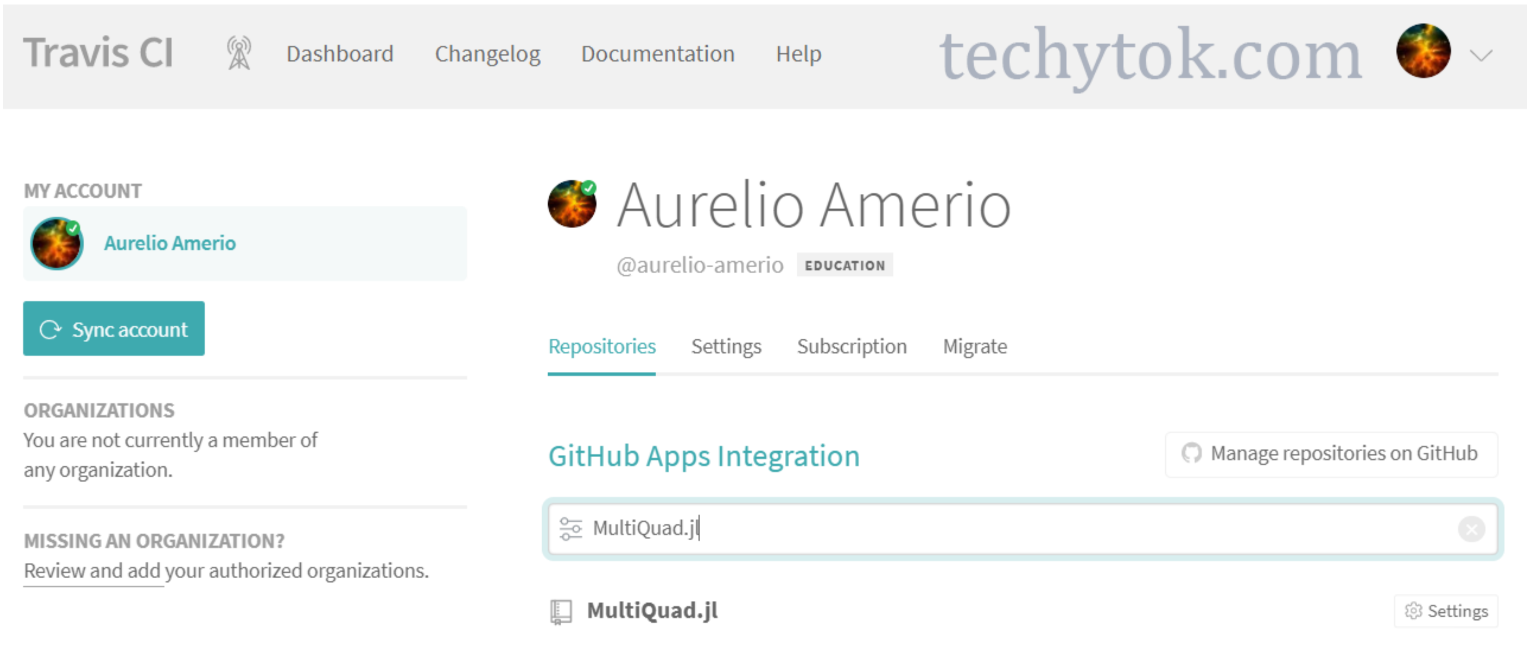
From line 17 to 21 we state what to do after a successful build: we want to **push the coverage results to Coveralls and Codecov**.

We will now register a free Travis account in order to create builds. Go to travis-ci.com (https://travis-ci.com/) and log in with your GitHub account.

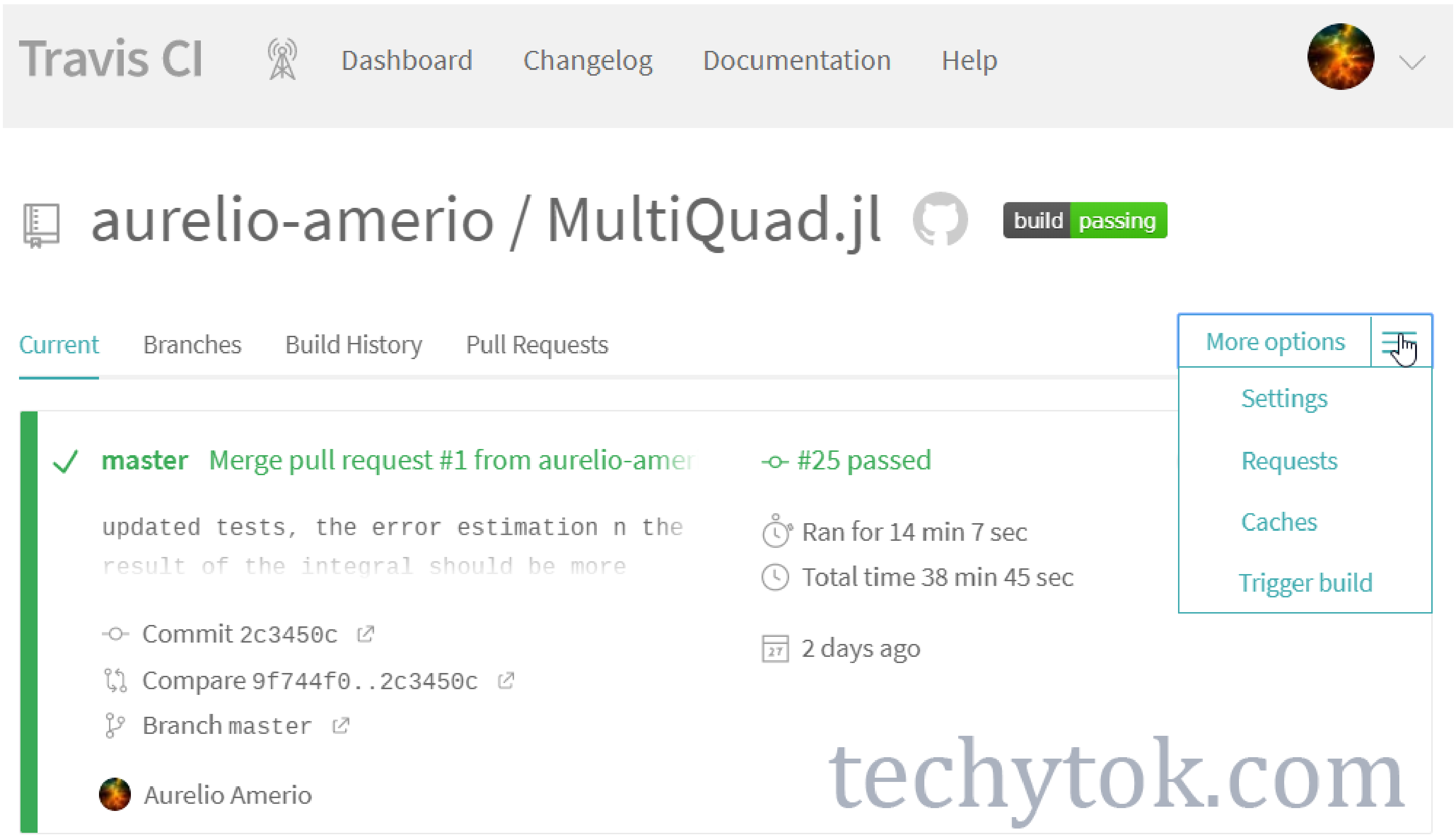
Go to your account settings page (https://travis-ci.com/account/repositories):



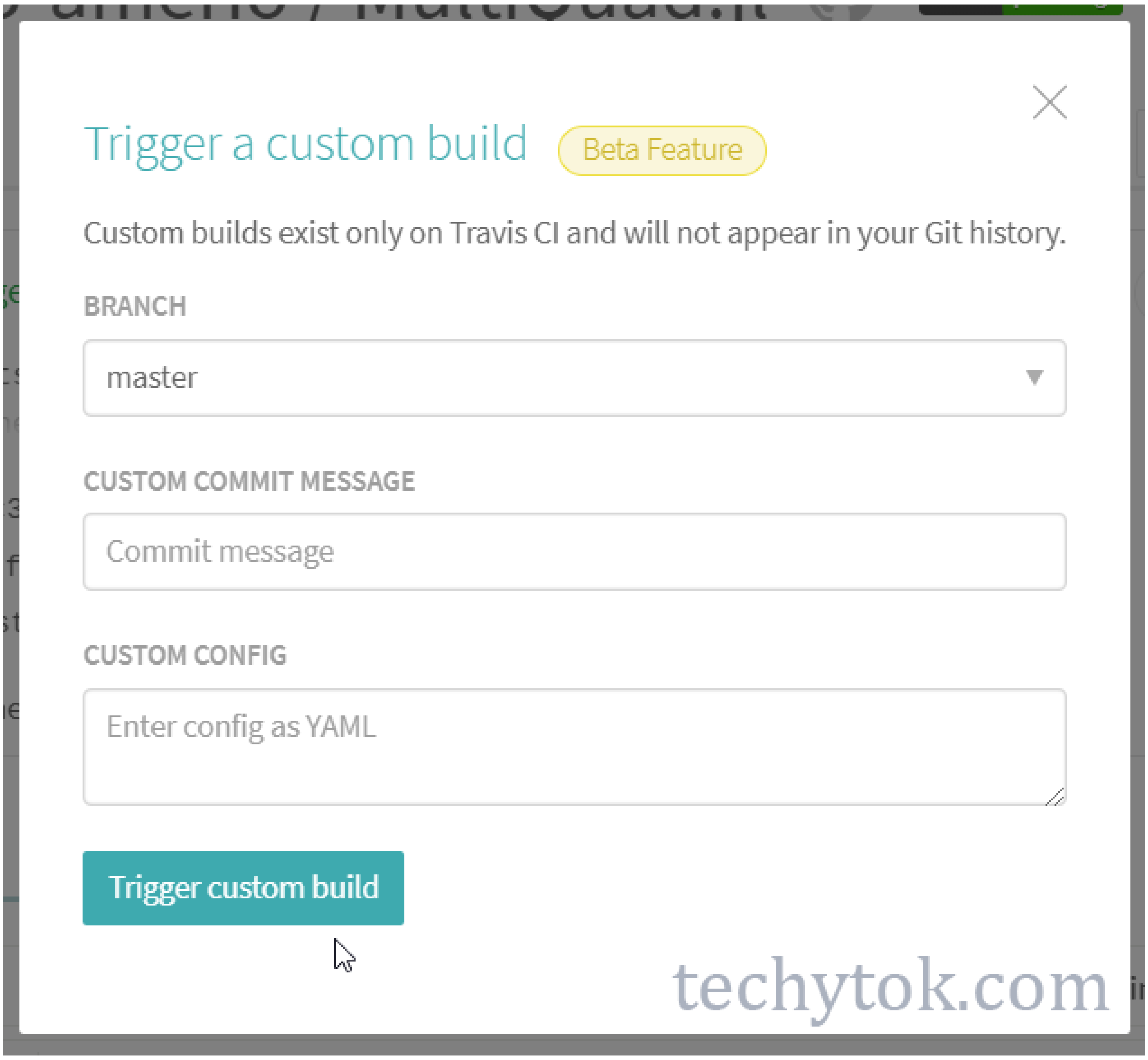
Type the name of the repository that you want to add to Travis (in my case `MultiQuad.jl`)



and click on the repository, this should bring you to the repository's page on Travis, where you can click on the "more options" button:



Now click on "Trigger build" to initiate the first build process, and on the next popup window click "Trigger custom build":


Trigger a custom build Beta Feature

Custom builds exist only on Travis CI and will not appear in your Git history.

BRANCH

master

CUSTOM COMMIT MESSAGE

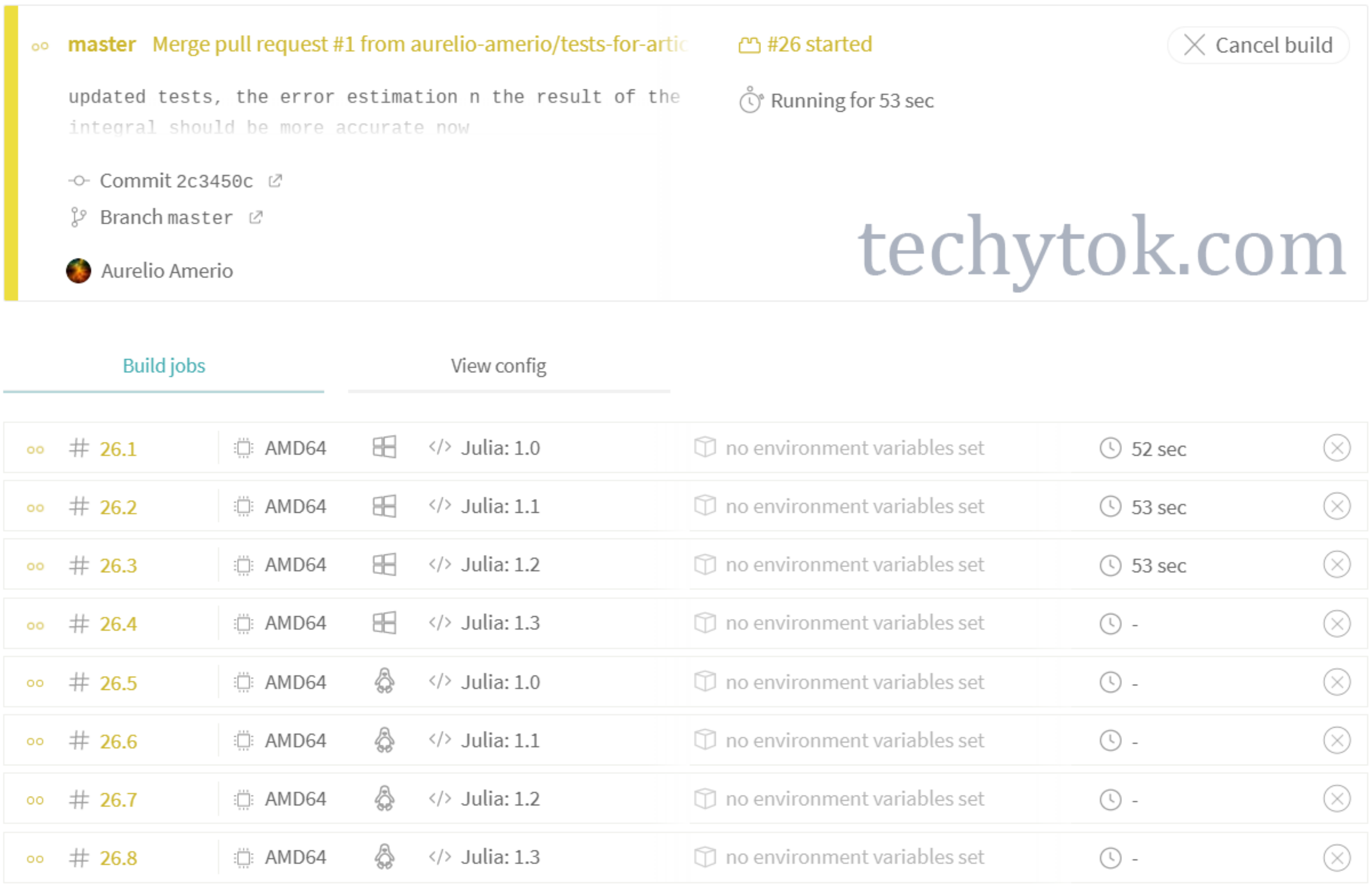
Commit message

CUSTOM CONFIG

Enter config as YAML

Trigger custom build

You should now see a somewhat lengthy page like this one:



master Merge pull request #1 from aurelio-amerio/tests-for-artic

#26 started

Cancel build

updated tests, the error estimation n the result of the integral should be more accurate now

Running for 53 sec

Commit 2c3450c

Branch master

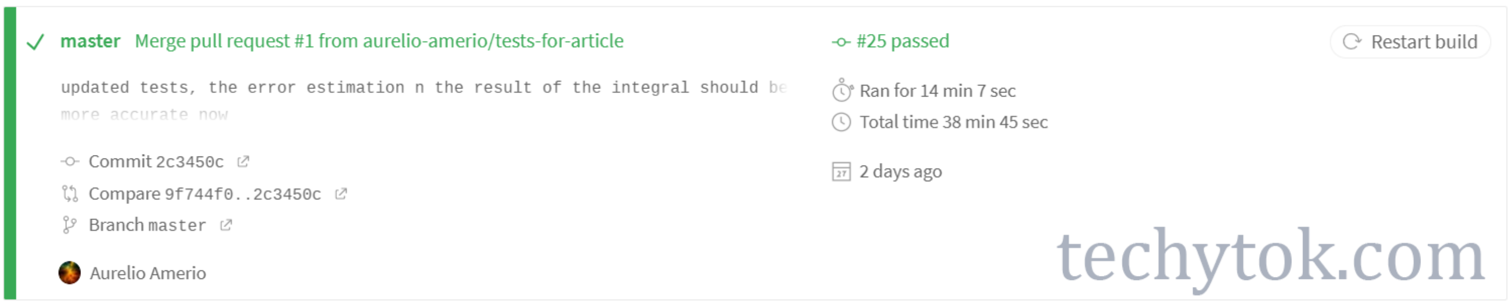
Aurelio Amerio

Build jobs | View config

# 26.1	AMD64	Windows	Julia: 1.0	no environment variables set	52 sec	⊗
# 26.2	AMD64	Windows	Julia: 1.1	no environment variables set	53 sec	⊗
# 26.3	AMD64	Windows	Julia: 1.2	no environment variables set	53 sec	⊗
# 26.4	AMD64	Windows	Julia: 1.3	no environment variables set	-	⊗
# 26.5	AMD64	Linux	Julia: 1.0	no environment variables set	-	⊗
# 26.6	AMD64	Linux	Julia: 1.1	no environment variables set	-	⊗
# 26.7	AMD64	Linux	Julia: 1.2	no environment variables set	-	⊗
# 26.8	AMD64	Linux	Julia: 1.3	no environment variables set	-	⊗

The build process will usually take some time, in the case of `MultiQuad.jl` it will take approximately 38 minutes.

If the build has been successful, you should see something like that:



✓ master Merge pull request #1 from aurelio-amerio/tests-for-article

#25 passed

Restart build

updated tests, the error estimation n the result of the integral should be more accurate now

Ran for 14 min 7 sec

Total time 38 min 45 sec

2 days ago

Commit 2c3450c

Compare 9f744f0..2c3450c

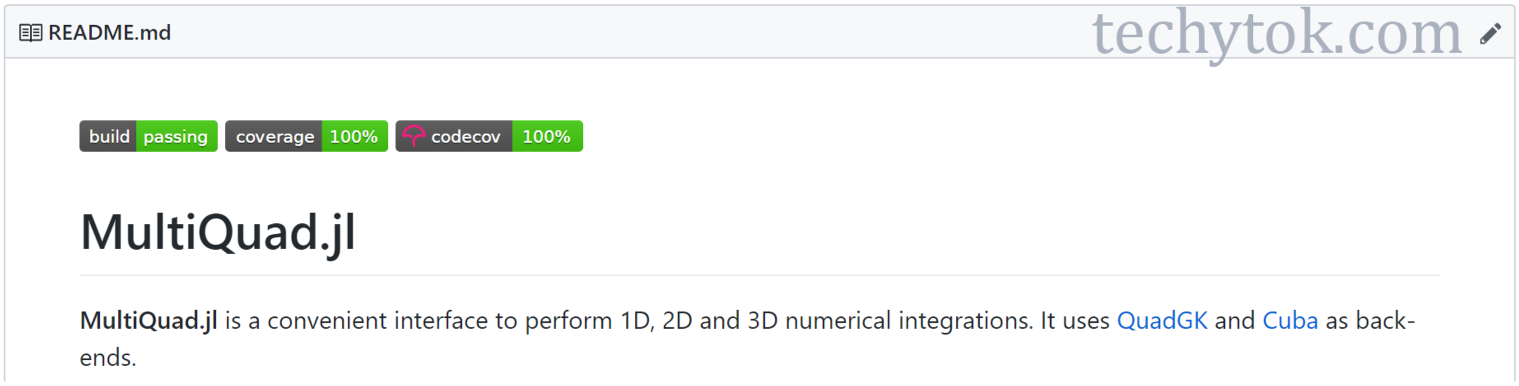
Branch master

Aurelio Amerio

From now on, every time you push a commit to your repository, Travis will run your tests and notify you if they failed. The results can be viewed directly from the GitHub repository page.

Adding badges to the README

Once the build is finished (and if it has been successful) you can add some nice badges to the `README` of your repository to show that your code is good (hopefully):



MultiQuad.jl

MultiQuad.jl is a convenient interface to perform 1D, 2D and 3D numerical integrations. It uses QuadGK and Cuba as back-ends.

To achieve a result similar to this one, you have to add the following code at the top of your `README.md` file:

```
[![Build Status](https://travis-ci.com/aurelio-amerio/MultiQuad.jl.svg?branch=master (https://travis-ci.com/aurelio-amerio/MultiQuad.jl.svg?branch=master))](https://travis-ci.com/aurelio-amerio/MultiQuad.jl (https://travis-ci.com/aurelio-amerio/MultiQuad.jl))
[![Coverage Status](https://coveralls.io/repos/github/aurelio-amerio/MultiQuad.jl/badge.svg?branch=master (https://coveralls.io/repos/github/aurelio-amerio/MultiQuad.jl/badge.svg?branch=master))](https://coveralls.io/github/aurelio-amerio/MultiQuad.jl?branch=master (https://coveralls.io/github/aurelio-amerio/MultiQuad.jl?branch=master))
[![codecov.io](https://codecov.io/github/aurelio-amerio/MultiQuad.jl/coverage.svg?branch=master (https://codecov.io/github/aurelio-amerio/MultiQuad.jl/coverage.svg?branch=master))](https://codecov.io/github/aurelio-amerio/MultiQuad.jl?branch=master (https://codecov.io/github/aurelio-amerio/MultiQuad.jl?branch=master))
```

Just replace `aurelio-amerio/MultiQuad.jl` with `your-username/repository-name` .

Package settings

Before we can publish our package on the Julia registries, we need to take care of the `Project.toml` file.

First of all, check that the first lines of your `Project.toml` file contain the `name` , `uuid` , `authors` and `version` fields.

For the first release of your package, make sure that the version of you package is either `0.1.0` (if this is a beta release) or `1.0.0` (if the package is already mature and complete). `0.1.0` is the default version for the first release of the package.

After we have set the version of the package (for example `0.1.0`) we should specify the compatibility bounds of your package. In particular you need to specify the maximum (and optionally minimum) version of the packages you use for which your package will work.

If you are not sure of the version of the packages you are using, you can check which packages are added to your project by looking at the `Project.toml` file. In the case of `MultiQuad.jl` , I can see in the `[deps]` section that I use:

```toml
[deps]
Cuba = "8a292aeb-7a57-582c-b821-06e4c11590b1"
QuadGK = "1fd47b50-473d-5c70-9696-f719f8f3bcdc"
Test = "8dfed614-e22c-5e08-85e1-65c5234f0b40"
Unitful = "1986cc42-f94f-5a68-af5c-568840ba703d"
```

We then to look up the version of those packages in the `Manifest.toml` file. If your project works with the packages currently installed on your machine, it is safe to use those package versions as upper compatibility bounds. Thus we can write the `[compat]` (compatibility) entry of the `Project.toml` file as follows:

```toml
1  [compat]
2  Cuba = "2.0.0"
3  QuadGK = "2.3.1"
4  Unitful = "0.18.0"
5  julia = "^ 1.0.0"
```

If we assume that subsequent minor releases of a Package are compatible, we can adopt the notation used at line 5 (`^ version`), which specifies that the package will be compatible with every Julia release up to Julia version 2.0 excluded.

> The step of adding the `[compat]` entry to the `Project.toml` file is mandatory in order to have your package accepted.

For more information on the `[compat]` entry and for a list of all the other available options, see the official documentation page (https://julialang.github.io/Pkg.jl/v1/compatibility/). In order to better manage the `[compat]` entry, it is furthermore suggested to install the CompatHelper (https://github.com/bcbi/CompatHelper.jl).

`PkgTemplates`

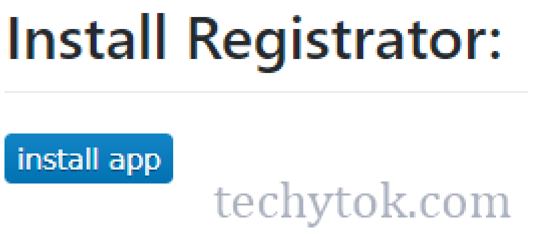
The creation of a package and the integration with Travis, Codecov and Coveralls can be automated using `PkgTemplates.jl` (https://github.com/invenia/PkgTemplates.jl). `PkgTemplates` has undergone large internal changes, and at present the user-facing API is still catching up. Once version 0.7 is released I will update this article to include an example on how to create a package using `PkgTemplates` .

Submitting the package

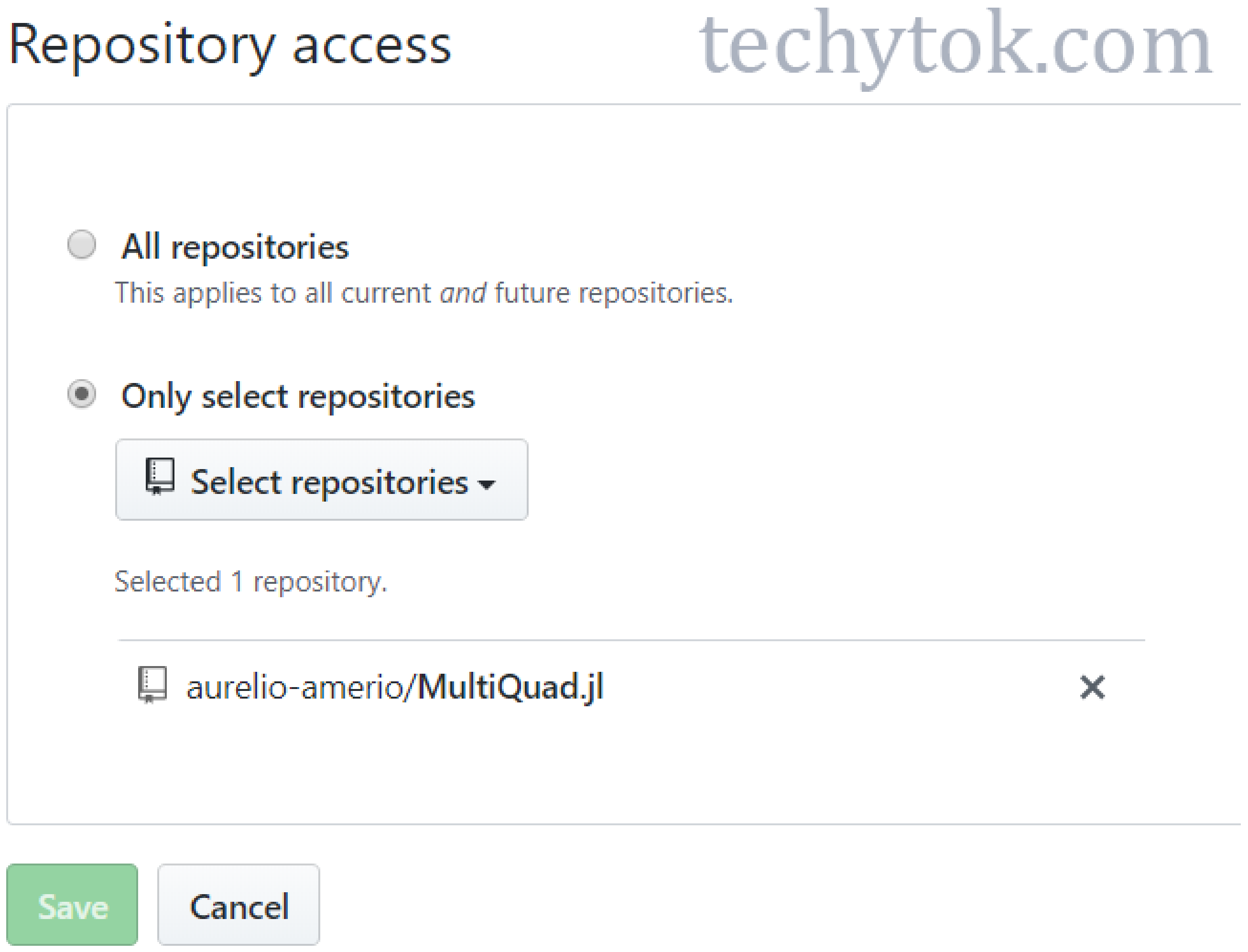
Install the Registrator App

In order to submit your package to the registry, it is recommended to use the official package registrator (https://github.com/JuliaRegistries/Registrator.jl).

We need to first install the GitHub app (https://github.com/apps/juliateam-registrator/installations/new):


Install Registrator:

Now you should give the registrator permission to access your repository:


Repository access

( ) All repositories
This applies to all current *and* future repositories.

(●) Only select repositories

Selected 1 repository.

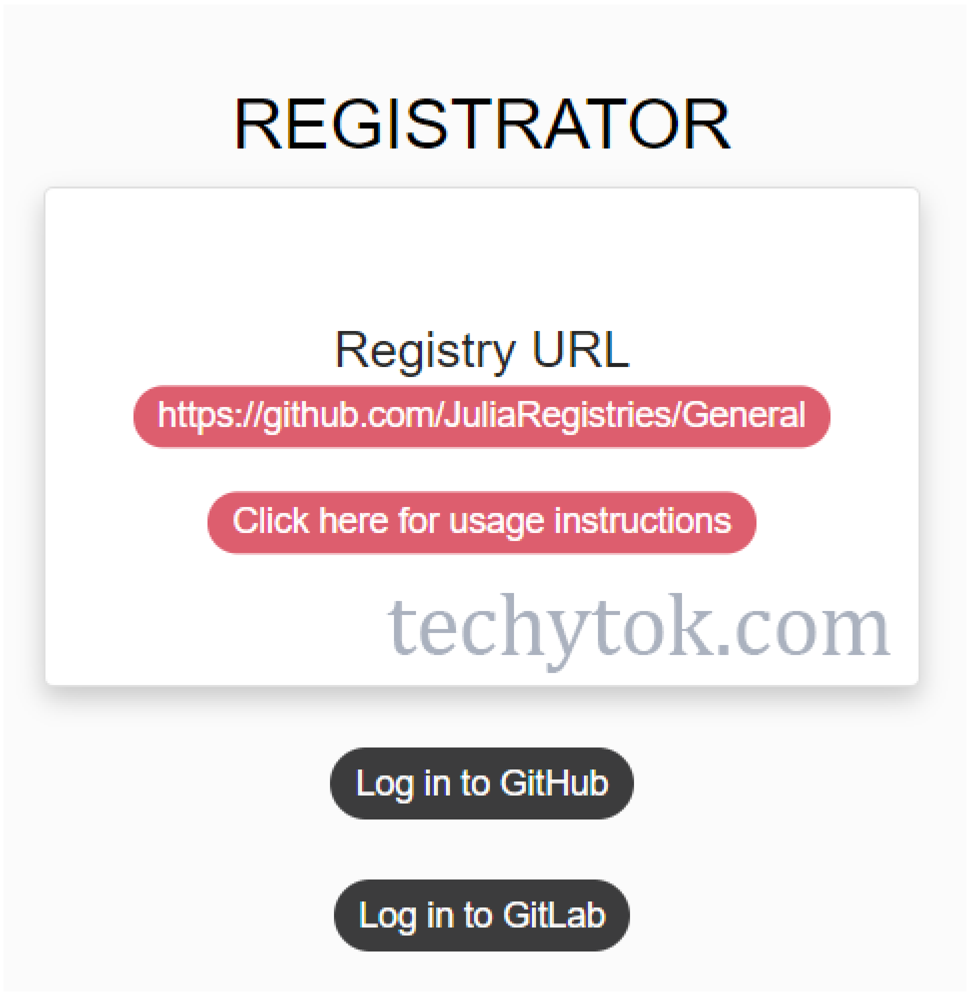
aurelio-amerio/**MultiQuad.jl** ×

If you would like to let Julia automatically create releases and tags once your package is added to the registry, you can install TagBot (https://github.com/JuliaRegistries/TagBot) and give it permission to access your repository in a similar way to what you did for the registrator app.

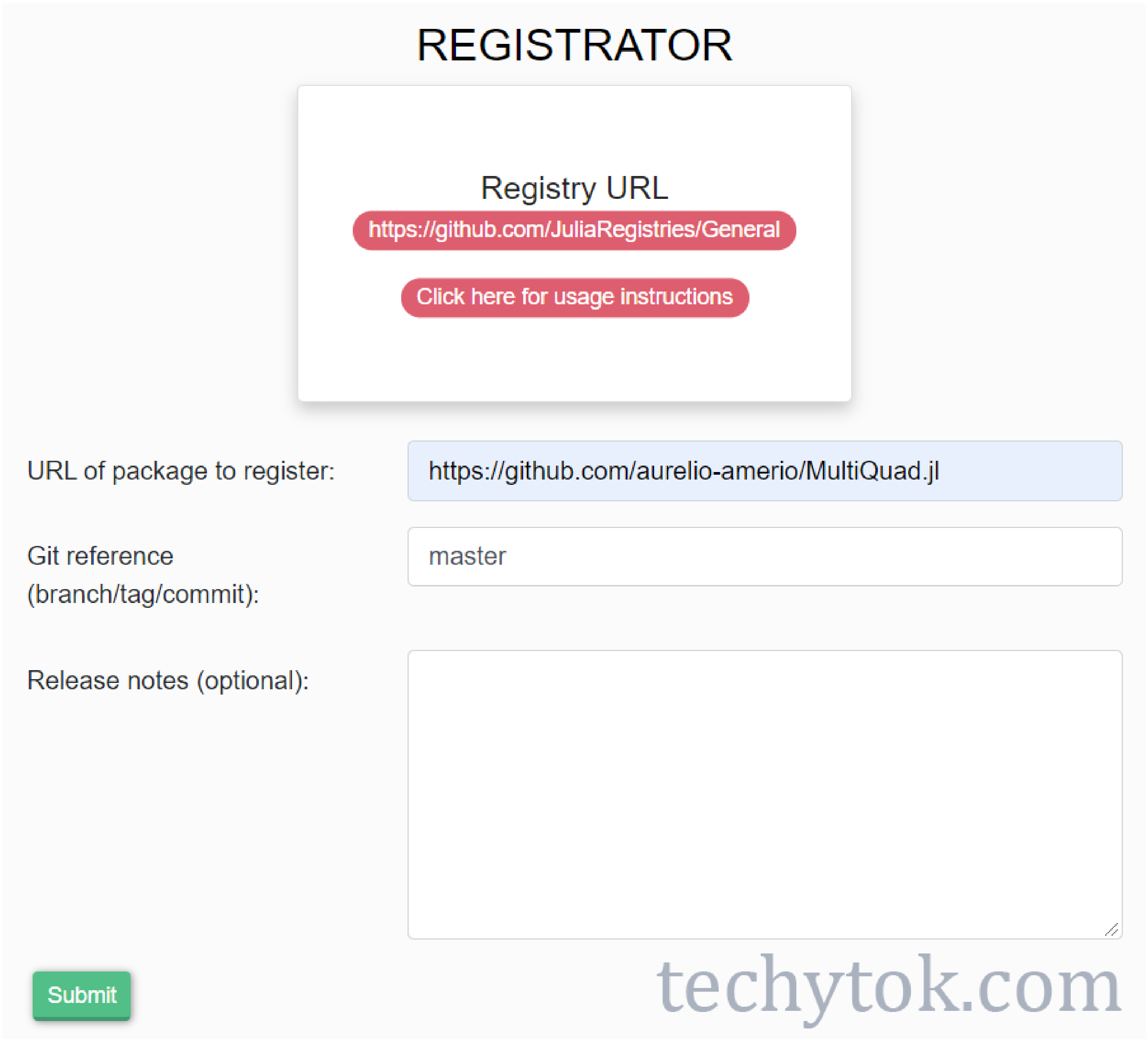
Register your package

The following steps are for educational purposes only, please don't try to register a package if it is not ready to be shared with the public yet.

We shall now go to the package registration page: pkg.julialang.org/registrator (https://pkg.julialang.org/registrator/)


REGISTRATOR
Registry URL
https://github.com/JuliaRegistries/General
Click here for usage instructions
techytok.com
Log in to GitHub
Log in to GitLab

Log in with your GitHub account and you should reach this page:


REGISTRATOR

Registry URL

https://github.com/JuliaRegistries/General

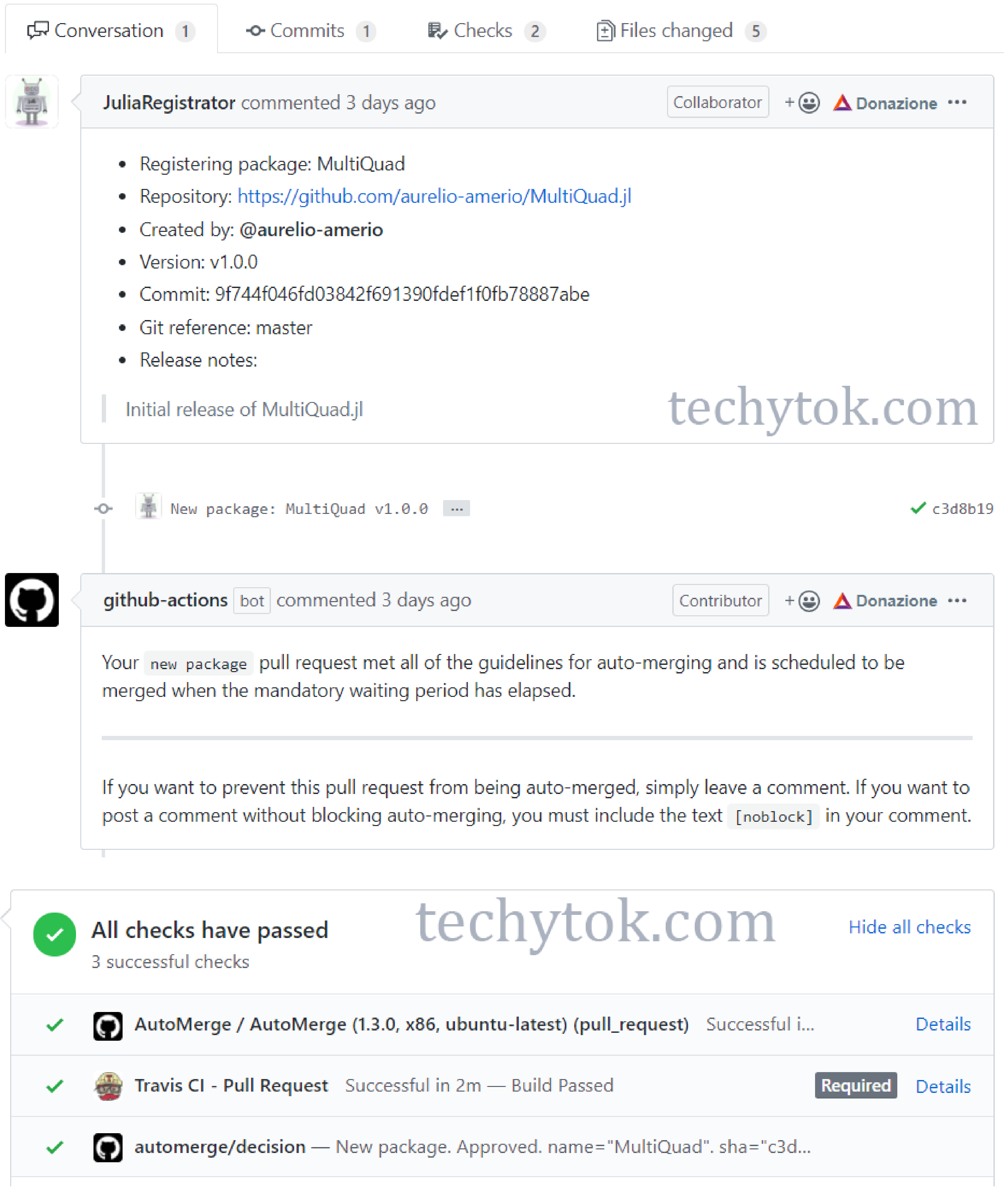
URL of package to register: https://github.com/aurelio-amerio/MultiQuad.jl

Git reference (branch/tag/commit): master

Release notes (optional):

Now type the URL of your GitHub repository and select the branch (usually you are going to use the **master** branch). You can add a message regarding the release (those are the release notes).

Once you are ready, you can submit your package for reviewal: a pull request will be opened on your behalf at the Julia Registry GitHub repository and your code will be checked: if there are no build issues and your package meets all the specifications, it will be scheduled to be merged.


Conversation 1 · Commits 1 · Checks 2 · Files changed 5

JuliaRegistrator commented 3 days ago · Collaborator

- Registering package: MultiQuad
- Repository: https://github.com/aurelio-amerio/MultiQuad.jl
- Created by: @aurelio-amerio
- Version: v1.0.0
- Commit: 9f744f046fd03842f691390fdef1f0fb78887abe
- Git reference: master
- Release notes:

> Initial release of MultiQuad.jl

```
New package: MultiQuad v1.0.0 ...                                    c3d8b19
```

github-actions bot commented 3 days ago · Contributor

Your `new package` pull request met all of the guidelines for auto-merging and is scheduled to be merged when the mandatory waiting period has elapsed.

If you want to prevent this pull request from being auto-merged, simply leave a comment. If you want to post a comment without blocking auto-merging, you must include the text `[noblock]` in your comment.

All checks have passed
3 successful checks

Hide all checks

- ✓ **AutoMerge / AutoMerge (1.3.0, x86, ubuntu-latest) (pull_request)** Successful i... Details
- ✓ **Travis CI - Pull Request** Successful in 2m — Build Passed · Required · Details
- ✓ **automerge/decision** — New package. Approved. name="MultiQuad". sha="c3d...

Updating the package

In the future, when you want to update your code and release a new version, you must change the version of you package in the `Project.toml` file (for example, move from version 0.1.0 to version 0.1.1). In Julia the **`semver`** conventions are as follows:

- When you move to the next version, it should be a standard increment and not skip versions (i.e. don't go from version 0.1.0 to 0.3.0 or from 0.1.0 to 0.1.3 or from 0.1.0 to 0.2.3)
- If you perform bug fixes without changing the interface, move from version `0.x.0` to `0.x.1`
- If you perform upgrades to the code which are retro-compatible, i.e. no breaking changes, but introduce new functionalities and you change the code for various functions, move from version `0.1.x` to `0.2.0`
- If you perform major changes which lead to a variation of the interface, add deprecation warnings (https://docs.julialang.org/en/v1/base/base/#Base.@deprecate) and move from version `0.x.y` to `1.0.0`

Once you have changed the version of your package in the `Project.toml` file, you can repeat the procedure described in the "Register your package" section.

Conclusions

In this lesson we have learned how to **write tests for our code** and how to test our package on many different operating systems and configurations using **Travis**. Finally we have learned how to **submit a pull request to the Julia Registry**, in order to register our package.

If you liked this lesson and you would like to receive further updates on what is being published on this website, I encourage you to subscribe to the **newsletter** (https://techytok.com/newsletter/)! If you have any **question** or **suggestion**, please post them in the **discussion below**!

Thank you for reading this lesson and see you soon on TechyTok!

Tags: Julia Travis

Categories: Lessons Tutorial

Updated: January 10, 2020

LEAVE A COMMENT