H62PEP Computing - Worksheet 7 - Testing, Documention and Automation

P. Evans and K. Cools

Overview

This week we will discuss all aspects of software development that go beyond the writing and sharing of the code base. Nevertheless these aspects will turn out to be indispensable to all but the simplest of software projects. We will discuss:

- **Testing**: In order to guarantee correctness of your code, it is absolutely necessary to complement your code with a number of test programmes. The output of these programmes will not only warn you when errors are introduced in your code, but will also assist you in thinking about what it means for your code to be correct. Because tests are so important and are required to be run many times over it is advantageous to use systems that will run these tests upon every change in code and that will inform you of the outcome using easy to oversee dashboards and notifications.
- **Documentation**: For small projects documentation can be done implicitly by using well chosen class, method, and variable names. For larger projects not all relevant information can be included in the name. Programmers typically try to compensate using comments scattered throughout the code. This is not acceptable for larger projects as this approach requires the client programmer to browse your library code in often hard to guess locations. To alleviate this problem documentation generators are used that crawl the code base for documentation left behind by the library developers. This documentation is summarised in an easy to access form (usually a website or another hyperlink enabled document format).
- **Packaging**: You may have encountered already some of the challenges encountered during deployment and execution of the application you are developing. Environment variables need to be set, libraries need to be present in the correct location… Even then, it is often only possible to launch the app if you know the exact directory the executable was placed in. For many applications (think Word for examples) this is not acceptable. The user cannot be expected to have the technical baggage required. To mitigate this problem deployment technologies that vary from system to system have been developed. These tools will gather all prerequisites needed for the execution of a programme in an installer or archive. At the client site these installers provide an easy self-contained tool to set up the programme.

The tools we will be using to help us with these aspect of software development are:

- `ctest`: ctest is a tool that is part of CMake and that provides two services: (i) it allows you to easily run all tests that you supply with your package (a test is simply an executable that returns 0 on success and 1 on failure), and (ii) it summarises the results in a so called dashboard that can be submitted to a web server (public or private) for easy inspection by the dev-team.
- `travis-ci`: a continuous integration service that can be set up to run a script every time a code change is committed to the repository. This script typically builds the binaries, and runs the tests. *Note*: You will have to log-in at `travis-ci.com` (**not** `travis-ci-org`). The commercial site allows testing of private repos. You get 100 builds for free, which should see you through this project.
- `doxygen`: doxygen is a tool that looks through your source code for specially formatted commnents meant to be included in the package documentation. Because doxygen understands the c++ syntax it is able to supplement these docstrings with relational information on the various classes and methods that make up your package. The final results is a web page, pdf document, … bundling all documentation.

- `cpack`: cpack is yet another component of the cmake ecosystem. It provides a convenient way to gather all the executables, libraries, etc. of your package and copies them in the default directory structure for the desired platform. Next it store this directory structure in an archive together with instructions on how to unpack it at the client site. The delivery medium can be as simple as a zip-file, but more user friendely options such as the very familiar Windows style NSIS installer are also an option.

Writing and running tests

As memtioned in the introduction, tests are simply programmes that return 0 on success and non-zero on failure. Because most tests are similar in structure, and because many test can share common setup and initialisation steps, there are a number of so-called unit test frameworks that can assist in the writing of such test programmes. For this project, we will write all our tests *by hand*. An example test for a simple mathematical library looks, for example, as follows:

```
// test1.cpp
#include <iostream>
#include "maths.h"

int main(int argc, char *argv[])
{
  int a = 2;
  int b = 5;
  int c = add(a,b);
  return (c == 7) ? 0 : 1;
}
```

Nothing out of the ordinary. The relevant part of the CMakeLists.txt file, repsonsible for building the corresponding executable reads:

```
# CMakeLists.txt (fragment of)
cmake_minimum_required(VERSION 2.8.0)
project(MathsLib)
include(CTest)

# ...
# description of executables and libraries
# ...

add_executable(test1 test1.cpp)
target_link_libraries(test1 maths)

add_test(test1 test1)
```

The two things to notice here are:

- The inclusion of `include(CTest)` after the `project` statement. This will enable testing for this project and will make sure the corresponding targets are created during build system generation. The precise names of these targets and how to invoke them will depend on the generator type (makefile, Visual Studio project, CodeBlocks,…).
- The line `add_test(test1 test1)`. The first argument is the name of the test, the second argument is the name of the executable that will be run as part of this test. Here the two arguments coincide but one could imagine situations where multiple tests could entail running the same executable but with different arguments; all such invocations would need their own unique test name.

Building a project containing tests is done exactly like building any other project. Instructions below are to obtain a Unix Makefile on Linux.

```
$ mkdir build && cd build
$ cmake .
```

```
$ make
[ 16%] Building CXX object src/CMakeFiles/maths.dir/maths.cpp.o
[ 33%] Linking CXX static library libmaths.a
[ 33%] Built target maths
[ 50%] Building CXX object src/CMakeFiles/test1.dir/test1.cpp.o
[ 66%] Linking CXX executable test1
[ 66%] Built target test1
[ 83%] Building CXX object src/CMakeFiles/calc.dir/calc.cpp.o
[100%] Linking CXX executable calc
[100%] Built target calc
```

Note that in addition to the libraries and executables that make up the payload of your package, also the executables implementing the tests (in this case `test1`) are built upon running `make` (or `msbuild` when using Visual Studio).

During configuration, `cmake` defined, in addition to the default targets, a target named `test`. Building this target will not actually built additional binaries, but will call a script that will run all the tests you defined in `CMakeLists.txt`.

```
$ make test
Running tests...
Test project /home/user/MathsLib/build
    Start 1: test1
1/1 Test #1: test1 ............................   Passed    0.00 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.00 sec
```

In this case only a single test was defined but in general `make test` will run all the supplied tests and will print the results in an easy to interpret summary.

Just to see what would happen, let's change the corresponding line in `test1.cpp` to

```
// test1.cpp
  return (c == 6) ? 0 : 1;
```

This is actually a bug in the test, not in the library code… Running the test now results in

```
make test
Running tests...
Test project /home/user/MathsLib/build
   Start 1: test1
1/1 Test #1: test1 ............................***Failed    0.00 sec

0% tests passed, 1 tests failed out of 1

Total Test time (real) =   0.00 sec

The following tests FAILED:
   1 - test1 (Failed)
Errors while running CTest
Makefile:149: recipe for target 'test' failed
make: *** [test] Error 8
```

Using `ctest` (under the hood `make test` calls `ctest`!) already made our life easier. We have simple means of running all our tests during code development. At this point, we want to automate this process even further: we want to make it so that, everytime someone submits code to the central repository, the code is built, all tests are run, and we are - if needed - informed of the outcome.

It turns out this service is provided by a free-to-use tool: travis-ci. We can set things up such that the following events will take place everytime a commit is pushed to the github repository:

- Travis spins up a virtual machine (both Linux and MacOS machines can be launched).

- All prerequisite tools and packages are installed on that virtual machine.
- Your repository is cloned on that machine and the most recent commit is checked out.
- The build and test instructions as defined in `.travis.ylm` (at the root of your repo) are being ran
- Travis optionally reports any failure

With this infrastructure in place, developers are essentially forced to make sure their contributions work before submitting them to the central repository.

What follows are the steps to set up this automatic testing for your repository:

- Log in at `travis-ci.com`. One of the provided options is to log in with your github account. This will bring you to a page with a number of *flip switches* that allows you to select for which of your repositories you would wish to use Travis. **Activate the service for your repository.**
- To describe what actually will happen when a new commit is submitted to the repository, we need to supply a `.travis.yml` file in the root of the repo. These `yml` (pronounced yammel) files are text based configuration files (like `ini` files or `json` files). The format is designed to be (i) unambigious and (ii) easy to read. The most important section is `script:`. In this section you find the instructions required to build the code and run the tests.

```
# .travis.yml
sudo: false

branches:
  except:
    - gh-pages

# Build your code
script:
  - cmake .
  - make
  - make test
```

Important: the .yml format uses indentation as an integral part of the script, i.e. the number of spaces/tabs before the start of each line in the file must be the same as shown above!

- If something goes wrong, Travis will send you an email notification. By default, succesful builds will not trigger notifications. To help you and your users monitor the status of the code, it is common to include a *badge* in your `README.md` file. A badge is just an image that indicates if the last attempt at a build was successful or not. **Modify the markdown code below to match your credentials/repo name and add it to your Readme.md file**.

```
[![Build Status](https://travis-ci.com/YOUR_USER_NAME/YOUR_REPO_NAME.svg?branch=master)]
(https://travis-ci.org/YOUR_USER_NAME/YOUR_REPO_NAME)

A fully tested code repository!
```

Note: It is very difficult to write unit tests for the graphical portions of your programme. Not only is it not trivial to install the required prerequisites on Travis, it is very difficult to emulate a human user and decide upon what constitutes success or failure. It is okay for this project to limit yourself to testing only the mathematical backend of your software - you just need to add a few test programs to your group repository that verify key parts of your code. For the brave: https://github.com/richelbilderbeek/travis_cpp_tutorial

Documentation

Doxygen and documenting your code

Documentation of code, both targeted at end users and client developers, can best be done whilst writing the code subject to this documentation. Not only does this result in a work flow containing a minimal number of *context*

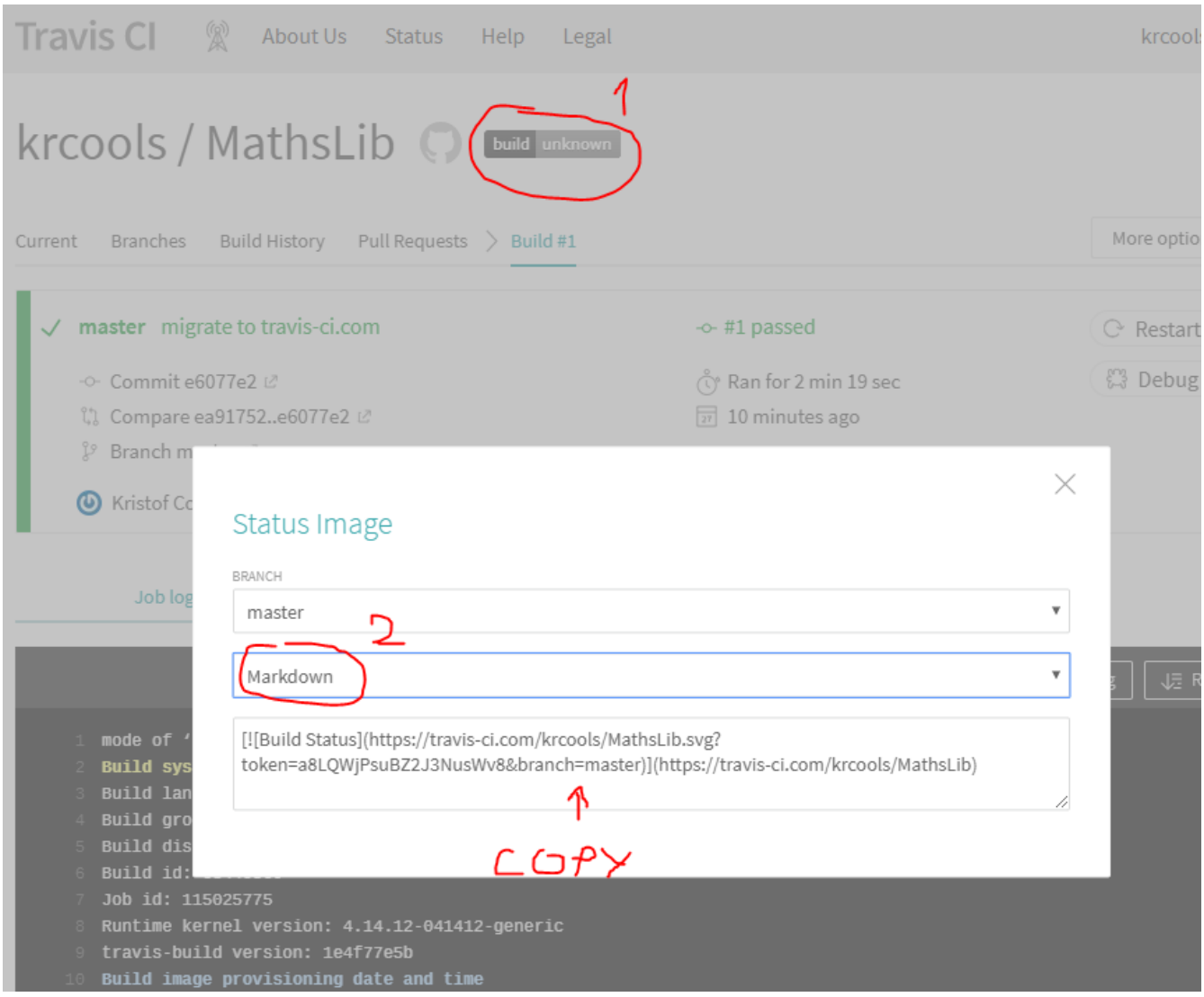

Figure 1: You can get the correct markdown snippet for your Travis badge from the Travis project page.

switches (and thus in higher efficiency), it also decreases the risk of the code (which is ever changing) and the documentation to go out of sync.

To facilitate documenting a number of tools have emerged. These tools are based on the idea that the coder can insert essentially anything in the source code as long as it's part of a comment (delineated by `//` or `/* ... */` in c++). Comments provided in a certain structure are recognised by this system and extracted by the documentation generator. Because comments can include hyperlinks and because the documentation generator understands the syntax and structure of the code base, the extracted documentation can be organised in a structure that reflects the relationships between e.g. namespaces, classes, methods, and variables. Several formats for the final document can be chosen, but given the importance of links between entities and the ability to navigate the document efficiently, a set of html files is an excellent choice. The added advantage is that the output can be served as a web site which can be accessed by the entire team in its most up-to-date incarnation.

The documentation generator we will be using is `doxygen` (take some time to enjoy this extremely clever portmanteau). It is still the most popular solution to document c++ heavy projects and you will have more than likely encountered the output of this system online.

Exercise: Install doxygen on your system. If you are on Windows, it might be a good idea to add the installation directory to the system path. On Linux or MacOS your package manager or `brew` respectively will go through all steps required.

Here is an example of how to decorate your code with comments that will be considered documentation strings by `doxygen`:

```
// maths.h
#ifndef MATHSLIB_MATHS_H
#define MATHSLIB_MATHS_H

/** @file
 * This file contains the declarations of all exported functions.
 */

/** Brief description
 *  This description will hopefully be picked up on by doxygen
 */
int add(int a, int b);

#endif
```

A comment meant for processing by Doxygen sets itself apart from any old comment by the opening sequence `/**`. In addition there are a number of keywords that help doxygen interpret the role and meaning of the documentation comment. In this example, `@file` indicates that this docstring contains information on the file itself (as oppossed to one of its comprising entities). The second docstring in this example above is by virtue of its placement (directly above a function declaration) interpreted as a description of that function. As a result, doxygen will place this comment in the appropiate location in its output.

Important: If no `/** @file ... */` block is provided, doxygen does not pick up on any of the function documentation comments in the file. This is because there is no natural place for that documentation to be included in the output. Documenting the file will generate a corresponding page in the output and all documentation for functions in that file will be included on that page.

The first step in using doxygen is to generate a configuration file at the root of your project:

```
$ doxygen -g
```

Open this file and change the following settings:

```
# To specify that the source files to be parsed reside in this directory
INPUT = $(TRAVIS_BUILD_DIR)/src
# To disable the generation of documentation in the .tex formats
GENERATE_LATEX = no
```

With these settings, simply run (from within the root of your project, i.e. where the Doxyfile configuration was created):

```
$ doxygen
Searching for include files...
Searching for example files...
Searching for images...
Searching for dot files...
Searching for msc files...
Searching for dia files...
Searching for files to exclude
Searching INPUT for files to process...
Searching for files in directory
...
...
Generating hierarchical class index...
Generating graphical class hierarchy...
Generating member index...
Generating file index...
Generating file member index...
Generating example index...
finalizing index lists...
writing tag file...
Running dot...
lookup cache used 1/65536 hits=1 misses=1
finished...
```

A subdirectory named `html` should have been created under your repo's root. This is a good time to add the html subdirectory to your `.gitignore` file. Doing this will avoid the generated documentation to be included in your git commits.

Inspect the `html` subdirectory file explorer (or your system's version of it). Double click `index.html` to open that file in your browser. There is not much there (because we only included two docstrings) but you already get an idea about how your inline comments are processed by doxygen.

Automatic Publication of Your Project Documentation

Just like we want to be informed asap when a code update breaks one of our tests, we want to always share the most up-to-date documentation of our project with the team.

In order to automate the process we will leverage the ability to execute any shell script we want on the virtual machines spun up by travis.

The resulting publication will be published on GitHub. This is possible because GitHub creates a publicly visible web page for every repository at:

```
https://<username>.github.io/<reponame>
```

The content GitHub puts on that web page is the tree on the specially named branch `gh-pages`. Typically there is little or no overlap between the contents of the code repository and what you'd want to display on the project's home page. Even though this is not what git is designed to do, there is nothing stopping you from putting wildly different content on two separate git branches. To explain to the system that `gh-pages` shares no content and thus no history with `master` and the other branches we designate this branch as *orphaned*: this means there is no parent commit to the first commit on this branch. In other words it does not share history with another branch. We can create and push a branch like that as follows (see Doxygen/gh-pages tutorial):

```
$ cd /path/to/repo
$ git checkout --orphan gh-pages
$ git rm -rf .
```

Note the line `git rm -rf`. This line effectively removes all files from the working tree. This is what you want since this tree needs to be populated with the output of Doxygen. It is safe because the source code is securely stowed away in the `master` branch, which is unaffected by whatever happens on this orphaned branch. At this point you'll want to make sure all files have been deleted from your repository. `git rm -rf` will only have deleted files that were tracked by Github so you'll need to make sure you delete any other files, either by selecting and deleting them in the file browser or using `del`/`rm`.
Once you are happy the gh-pages branch is empty you'll need to add a README to the branch, add, commit and push.

```
$ echo "gh-pages branch" > README.md
$ git add .
$ git commit -a -m "Clean gh-pages branch"
$ git push origin gh-pages
```

Exercise: Add a simple `index.html` file and commit it to `gh-pages`. Push to the GitHub hosted remote and inspect the result at `https://<username>.github.io/<reponame>`. This site is public; don't upload content unfit for distribution!

Important: After finishing the exercise, switch back to the master branch (`git checkout master`). The tools used for documentation generation will be ran on the source code of your project, so any related scripts and configuration files need to be stored alongside, in the master branch. The output of the documentation generator, on the other hand, will be committed on the `gh-pages` branch, preparing it for publishing.

Now we want to set things up so that every time Travis builds the code and runs the tests, it also builds the documentation and pushes it to the `gh-pages` branch of the repository. It is as if Travis were a member of your team with privileges to push! Because Travis is a bot, not a person, we will take a slightly different approach. GitHub allows for the creation of security tokens that allow access to the repository for certain pre-agreed purposed. In this case, we will have GitHub generate a token that we will then communicate to Travis so it can authenticate itself as a good bot.

If you are currently logged into your GitHub account, you can find the page to generate tokens at (https://github.com/settings/tokens). Check `Full control of private repositories` in the permissions section of the page. Upon generation of the token, copy it to the clipboard. Be careful not to copy something else over it. If you loose the token this stage you will have to generate a new one.

Now we have to hand this token over to Travis. Go to the repository page on travis-ci.com. Here you can define the environment variables that Travis will set before running your script. We will have Travis set the security token as an environment variable. Since the virtual machine Travis will create is well isolated from the outside world this poses no security risks. Placing sensitive information on your local machine in an environment variable, on the other hand, would be a very very foolish thing to do. *Don't do it.*

It's important to name the token `GH_REPO_TOKEN`. We will refer to the variable by this name in the `.travis.yml` script:

```
# .travis.ylm
# Modified travis script that generates and submits doxygen documentation
sudo: false

branches:
  except:
    - gh-pages

# Modify these variables that will be used in generateDocumentationAndDeploy.sh
# (see below). They tell Doxygen where to find the source on and where to send
# the generated documentation.
env:
  global:
    - GH_REPO_NAME: [YOUR_REPO_NAME]
    - DOXYFILE: $TRAVIS_BUILD_DIR/Doxyfile
    - GH_REPO_REF: github.com/[YOUR_USERNAME]/[YOUR_REPO_NAME].git
```

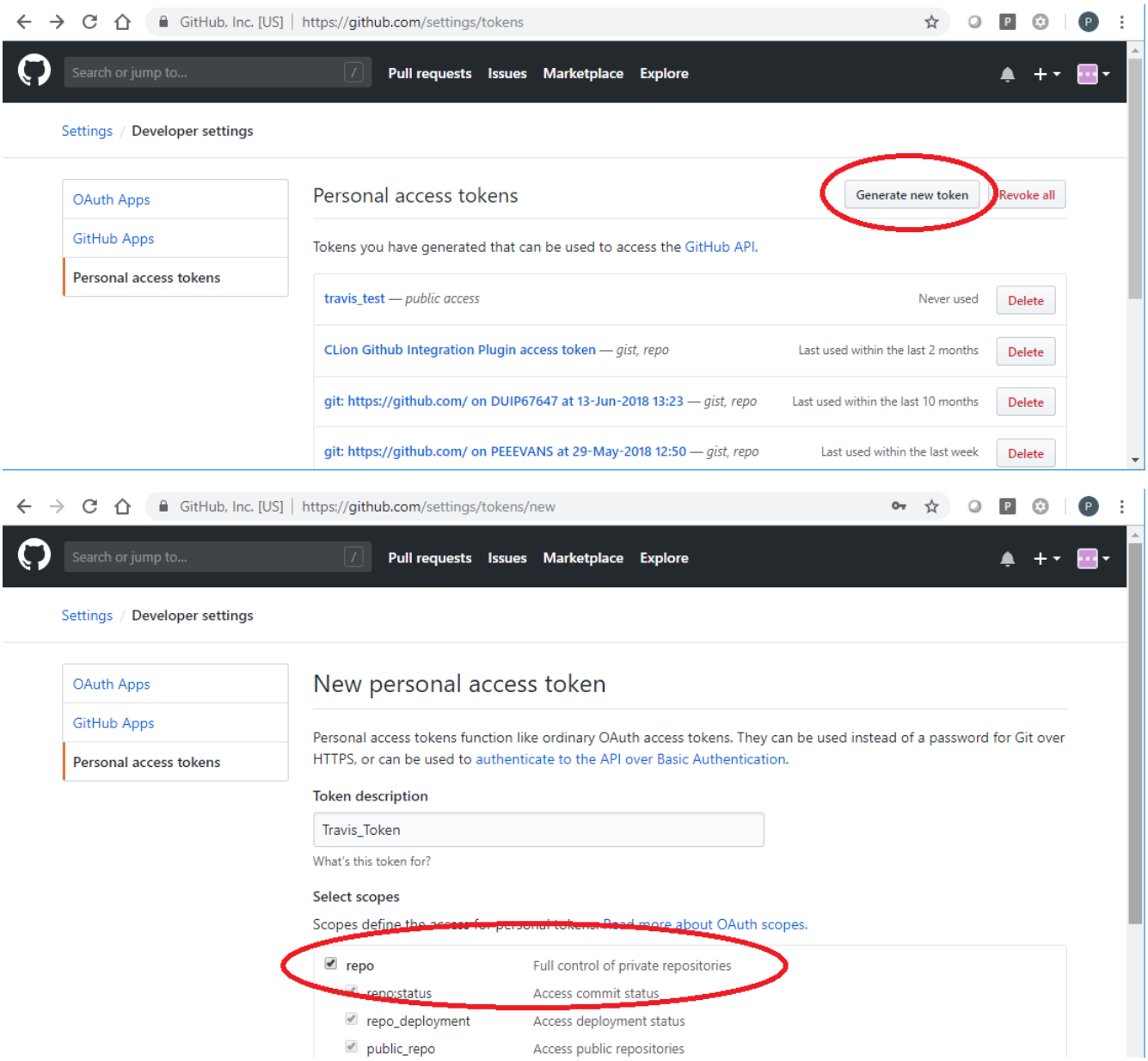

Figure 2: You'll need to generate and access token from within Github, go to https://github.com/settings/tokens when logged in.

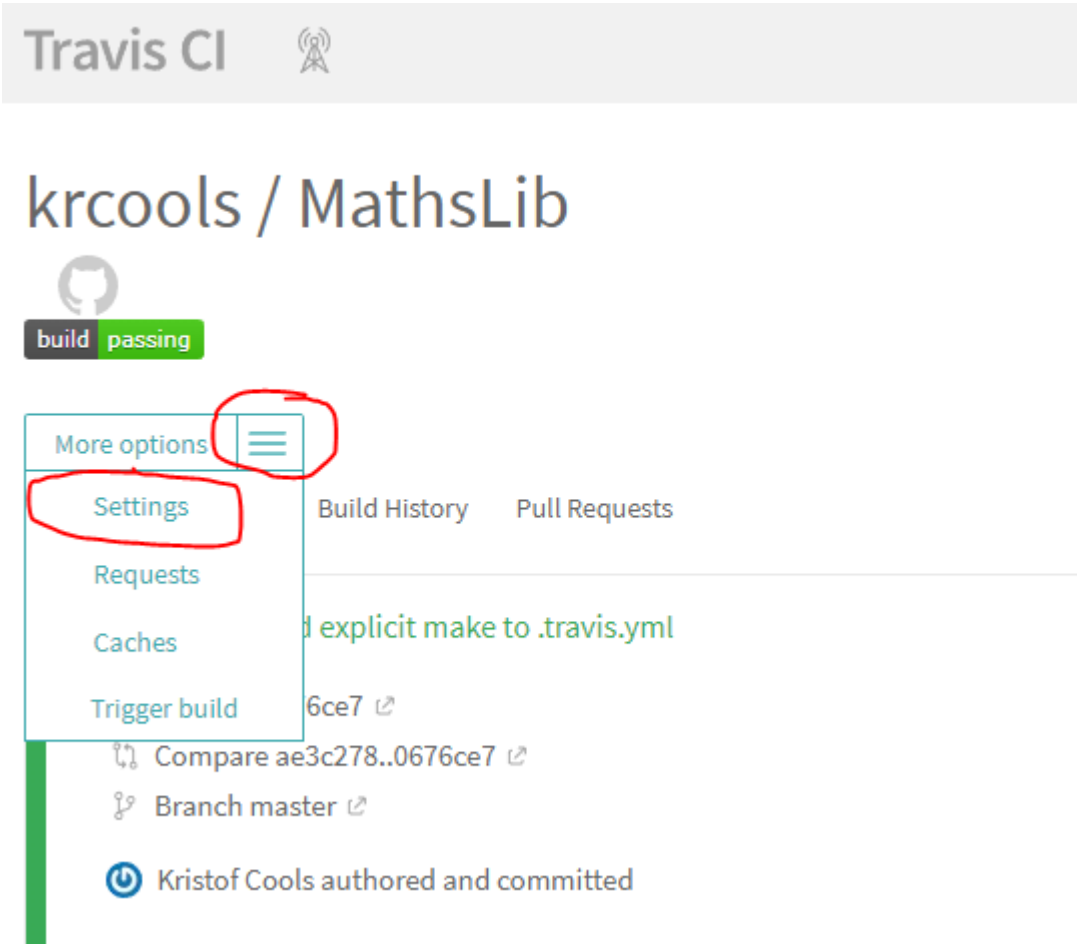

Figure 3: Tokens/Keys can added to Travis on the settings page of your repo on Travis-ci. These keys will allow Travis to access Github

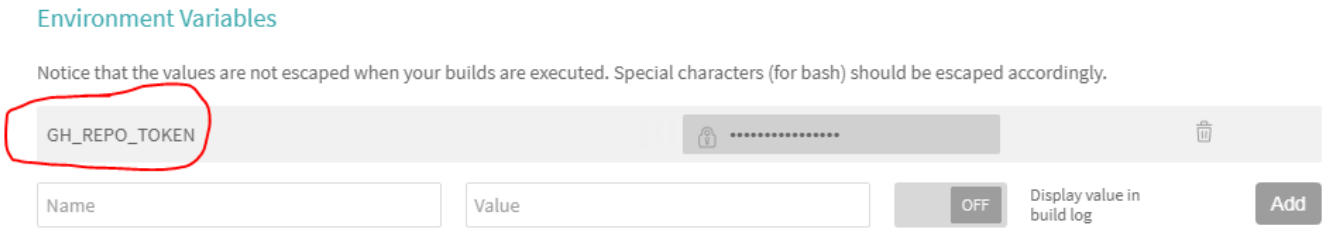

Figure 4: The token copied from Github is added as an environment variable in Travis so it can be accessed by scripts that you will add later. Make sure to name the variable `GH_REPO_TOKEN` as the scripts below will expect the token variable to have been given this name.

```
# The virtual machines created by Travis do not by default have doxygen
# installed. This is fortunately not difficult on a Debian flavoured Linux
# system.
addons:
  apt:
    packages:
      - doxygen
      - graphviz

# Finally we define the script that Travis will run every time code is submitted
# to the repository.
script:
  - cmake .
  - make
  - make test

# After successfully building and testing your software we give Doxygen the
# instruction to scan the code for docstrings and generate the html output.
# The bulk of these instructions is encapsulated in a script.
after_success:
  - cd $TRAVIS_BUILD_DIR
  - chmod +x generateDocumentationAndDeploy.sh
  - ./generateDocumentationAndDeploy.sh
```

You need to add this script (`.travis.yml`) to your repository and edit it to reflect your Github username and repository name (i.e. replace [YOUR_REPO_NAME] and [YOUR_USERNAME] with your Github repository name and Github username.

A second script, `generateDocumentationAndDeploy.h` must also be added to the repository, this script automates the running of Doxygen on the Travis-ci server. Note that even if you develop on a Windows machine, you should fill this script with a Linux styled shell script. This script will be run on the Travis virtual machine; not your local development machine!

For the original script, see (https://gist.github.com/vidavidorra/548ffbcdae99d752da02).

```
#!/bin/sh
__AUTHOR__="Jeroen de Bruijn"
set -e

mkdir code_docs
cd code_docs

# Use the Token to gain access to the private repo.
git clone -b gh-pages "https://${GH_REPO_TOKEN}@${GH_REPO_REF}"
cd $GH_REPO_NAME

git config --global push.default simple
git config user.name "Travis CI"
git config user.email "travis@travis-ci.com"

rm -rf *
echo "" > .nojekyll
doxygen $DOXYFILE 2>&1 | tee doxygen.log
if [ -d "html" ] && [ -f "html/index.html" ]; then

    git add --all
    git commit -m "Deploy docs: ${TRAVIS_BUILD_NUMBER}" -m "Commit: ${TRAVIS_COMMIT}"
```

```
        git push --force "https://${GH_REPO_TOKEN}@${GH_REPO_REF}" > /dev/null 2>&1
    else
        echo '' >&2
        echo 'Warning: No documentation (html) files have been found!' >&2
        echo 'Warning: Not going to push the documentation to GitHub!' >&2
        exit 1
    fi
```

Summarised, this script does the following:

- Clone the `gh-pages` branch of your repository. This is not done to get its content (this will be deleted momentarily) but to have a local repo to push from.
- Remove the previously generated docs, giving Doxygen a clean directory to start from.
- Running Doxygen using the configuration file we provided as part of the repository
- If Doxygen succeeded, commit the generated docs to `gh-pages` and push the new commit back to GitHub. This is where we need the security token.

That's it. This might seem complicated at first, but it all makes sense. Travis monitors GitHub for new commits. If a new commit is placed, Travis clones your repo, builds the code, and runs the tests. If successful, it will clone your repo again, this time limiting itself to the `gh-pages` branch only. On that branch it will save the output of Doxygen (which is run on the `master` branch, not on `gh-pages`). The resulting set of `html` files is committed to `gh-pages` and pushed back to GitHub. Becuase GitHub serves the most recent content of `gh-pages` as a website, you and your team members get to surf the up-to-date documentation of your project.