

First steps on GitHub

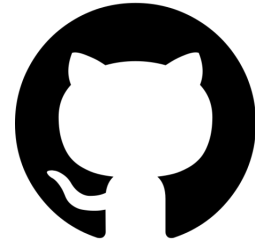
Natalí S. M. de Santi (@natalidesanti)

All steps below are for passionate by terminal on **Linux** operational system.

1 Step 1 - What is GitHub?

GitHub is a code hosting platform for version control and collaboration using **git**. It lets you and others work alone or together on projects from anywhere.

Git is a distributed version-control system for tracking changes in source code during its development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity and support for distributed non-linear workflows.



GitHub logo.

2 Step 2 - Creating a GitHub account

To start your experience on **GitHub** you need to make an account on: <https://github.com>. You will just need to inform your e-mail account, choose an username and create a password. Now, instead of using your password to do your git things, you will need to use your token. The instructions to do this are available in the page.

3 Step 3 - Installing git in your computer

You need to install **git** in your computer. If you are using **Ubuntu** you just need to run OS and package updates:

```
$ sudo apt-get update
```

And install **git** giving the following command:

```
$ sudo apt-get install git-core
```

You may be asked to confirm the download and installation. **Git** should be installed and ready to use. If you can confirm it you can just run the **git** version command:

```
$ git --version
```

Mine is:

```
$ git version 2.25.1.
```

4 Step 4 - Creating a repository

After creating an account you need to start a first (or new) **repository** on **GitHub**. A repository is usually used to organize a **project**. Repositories can contain anything your project needs: folders, files, images, videos, data sets, etc.

You can create your repository in the **GitHub** site, giving to it the following features:

- Name: `project_name`.
- Description: “This project has the objective to...”.
- Privacy: `public` (anyone can see this repository, but you can choose who can commit) or `private` (you choose to see and commit to this repository);
- README;
- LICENSE;

The last two options are really usefull if the project is not only yours, or even if you want to share it with anyone else. I prefer alway create this things after, because then you will end up in a page that gives you the **link** (and some instructions) of your project, that you will use in your computer, to indicates where it will be online! But, as the link is the same as the project one, fell free to create these files whatever time that you want.

Just to know, the README, i.e, a file with information about your project, which is written in markdown. The `license` gives the rights about what other people can do or not with your work. Some directions about what license chose can be found in the page and the instructions to add one in your repo is here.

Then, you need to to start a first (or new) repository giving, inside the directory, in your computer, chosen by you, the command:

```
$ git init
```

Now you need to say who you are for your **git**. Write in the terminal:

```
$ git config --global user.email <you@example.com>
$ git config --global user.name <your_name>
```

You need to pay attention that `--global` means that you are using **git** on your computer. If you omit this option you are logging in only on the local folder.

If you want to see the user and the configuration of your **git**, acess the file `.gitconfig` and the information will be like:

```
[user]

email = you@example.com

name = your_name
```

5 Step 5 - Including files in your repository

You need to add the files that you wish to keep your changes on **git**, giving the command:

```
$ git add <file_name>
```

Remember that, in other times (beyond the first one), when you are adding the changes you need to write stage, instead add, in the command above, like:

```
$ git stage <file_name>
```

6 Step 6 - Committing

On **GitHub**, saved changes are called **commits**. Each commit has an associated **commit message**, which is a description explaining why a particular change was made. Commit messages captures the history of your changes, so you and other contributors can understand what you've done and why.

After adding/staging you need to commit it:

```
$ git commit -m "<message>"
```

You can do as many commits as you want, giving the command above and writing a message to warn you about your changes, in same documents, or about new documents added in your repository.

7 Step 7 - Uploading your files and changes to GitHub

It's time to upload your **git** on **GitHub**. Now you need to log in on **GitHub** site, access your project and upload your **git** giving the commands sequence:

```
$ git remote add origin https://github.com/user/project_name.git  
$ git push -u origin master
```

GitHub will ask for your username and password¹:

```
Username for 'https://github.com': username  
Password for 'https://username@github.com': password
```

And then, you can see which files and commits were upload by you:

```
Counting objects: N, done.  
Delta compression using up to M threads.  
Compressing objects: 100% (N/N), done.  
Writing objects: 100% (O/O), 1.46 KiB | 374.00 KiB/s, done.  
Total N (delta 0), reused 0 (delta 0)  
To https://github.com/username/project_name.git  
* [new branch] master -> master  
Branch 'master' set up to track remote branch 'master' from 'origin'.
```

The **git**'s magic works in this way: you create or add some files, modify them, commit your changes and upload the files on **GitHub**. But there is a lot more!

¹Remember to use your token instead your password!

8 Step 8 - Not uploading some files and changes to GitHub

9 Step 9 - Creating a branch

Now you need to know that all that you have done above was made in the master branch. Ops, what I'm talking about?

Branching is the way to work on different versions of a repository at one time. By default your repository has one branch named **master** which is considered to be the *definitive branch*. We use branches to experiment and make edits before committing them to master. In other words, in **git** you can have a tree history of your code. As a tree you have not only one, but a lot of branches. In the branches you can do changes in your files, save them and use it in your master branch as you want.

When you create a branch off the master branch, you're making a copy of master as it was at that point in time. If someone else made changes to the master branch while you were working on your branch, you could **pull** in those updates. But it is a story that I will tell to you in the next steps.

First, we are going to create a branch:

```
$ git branch <name_of_branch>
```

Second, you need to get to that branch:

```
$ git checkout <name_of_branch>
```

Then, you can stage, commit and push files to that branch, using the same commands used right above. Just notice that, when you do your push you just need to change master by your current branch name_of_branch:

```
$ git push -u origin <name_of_branch>
```

You can add as many branches as you want, like direct and different branches from main or even branches starting in other branches. Just remember that, for each branch that you create, the files in that branch will be the same ones in the previous branch as you start your modifications and commits on it.

You can see all branches of your project writing in the terminal:

```
$ git branch
```

and the present branch that you are working in will be presented with a `*` at the left side as:

```
master
```

```
branch 1
```

```
branch 2
```

```
branch 3
```

If you want to delete some branch you just need to write:

```
$ git branch -d <name_of_branch>.
```

10 Step 10 - Merging

As I have described in the previous step, after all changes and commits in your branch, you can put the modifications in the main one. You just need to merge the main with the branch that you are using. Thus, in the main branch you can give the line command:

```
$ git merge name_of_branch
```

Finally, you have the main branch completely changed by your other branch changes!

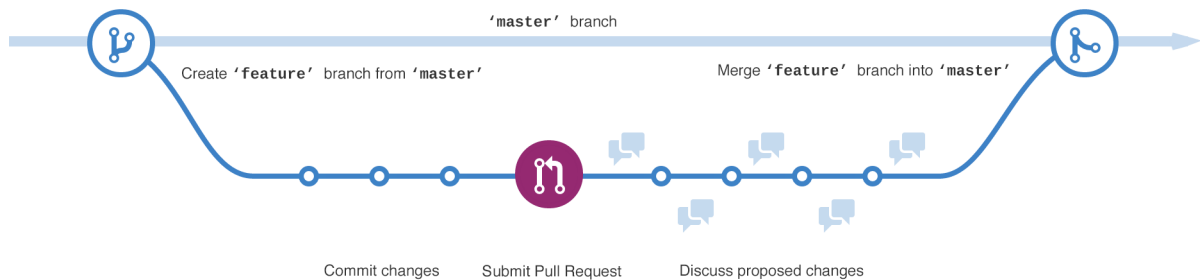


Figure 1: Master branch side by side another branch in which there are many commits, pull requests and discussion before merging into the main one. This image is from Hello Word project on GitHub's site.

Pay attention that, the command `merge`, merges the branch that you are in with the other branch you chose, i. e., you can add the changes of the chosen branch in the branch that you are in, and not just for the main branch and the other one desired branch.

If you need to verify in which branch you are, what files were be uploaded or not and how they are in your **GitHub**'s page, you can just write in the terminal:

```
$ git status.
```

11 Step 11 - Pull Requests

Pull Requests are the heart of collaboration on **GitHub**. When you open a pull request, you're proposing your changes and requesting that someone, really anyone, review and pull in your contribution and merge them into their branch. Pull requests show diffs, or differences, of the content from both branches. The changes, additions, and subtractions are shown in **green** and **red** in your **GitHub**'s page.

As soon as you make a commit, you can open a pull request and start a discussion, even before the code is finished.

By using **GitHub**'s `@mention` system in your pull request message, you can ask for feedback from specific people or teams.

You can even open pull requests in your own repository and merge them yourself. It's a great way to learn the **GitHub** flow before working on larger projects.

Again, there is a lot of other things to do on **git** and **GitHub**.

12 Step 12 - Changing

If you are using **git** you certainly will do changes in your project. Then, you can use the following commands:

- To show the differences in file that are not realized yet:
\$ git diff;
- To show the differences in staged files and its last versions:
\$ git diff --staged;
- Unselect the file, preserving its content:
\$ git reset <file_name>;
- To remove the file:
\$ git rm <file_name>;
- To remove the file, preserving it locally:
\$ git rm --cached <file_name>;
- To change the file name and to select it to a new commit:
\$ git mv <original_file_name> <new_file_name>;
- Undo all commits after the specified commit, keeping the local changes:
\$ git reset <commit>;
- Discards every history and changes for the specified commit:
\$ git reset --hard <commit>.

13 Step 13 - Saving fragments

You can still save and restoring incomplete changes using:

- Keeps, temporally, all modified files:
\$ git stash;
- Restores all recent files that have been stashed:
\$ git stash pop;
- List all changes in stash:
\$ git stash list;
- Discards every recent settle of changes in stash:
\$ git stash drop.

14 Step 14 - Reviewing history

You can make a review of the evolution of all files in some **git** project using:

- List the versions history in the local branch:
\$ git log;
- List the history versions for specific file, including name alterations:
\$ git log --follow <file_name>;
- Show the differences between the content among two branches:
\$ git diff <first_branch> ...<second_branch>;
- Show the changes in the metadata and content for the specif commit:
\$ git show <commit>.

15 Step 15 - Forking/Cloning a repository

If you are surfing into **GitHub**'s site and find a great project that you like you can **fork/clone** it for you. Then, you will have this repo into your computer to run, to modify and to do whatever you want without affecting the original project. To fork/clone some repo you need to follow two steps:

15.1 Forking:

Navigate until the **GitHub** project that you liked and, in the top-right corner of the page, click **Fork!**

15.2 Cloning:

To be “connected” with that repo and receive the last actualization's of it, when the owner do some modifications, i.e., to keep your fork synced, you just need to write in the terminal:

```
$ git clone <link>.
```

This command do the download of the project with its complete historic version. As a simple example, you can clone this tutorial writing:

```
$ git clone <https://github.com/natalidesanti/first_steps_on_github>.
```

Remember to clone some repo in some location that you want into your computer.

If you have interest to make a pull request in this repo you can give a:

```
$ git pull
```

to see the last alterations into this repo before proceed to make your pull request!

16 Basic git commands

I would like to finish this manuscript listing some basic **git commands**:

17 Acknowledgments and references

To write the **First steps on GitHub** I really appreciate the Nicolas Morazotti (@Morazotti help, Patricia Novais (@pnovais) tutorial, the Hello World project and the Wikipedia pages for GitHub and git.

Git task	Notes	Git commands
Adding files	“*” means all files	git add <file_name> git add *
Branches	Create a new branch and switch to it or commit	git checkout -b name_of_branch
	Switch from one branch to another	git checkout name_of_branch
	List all the branches in your repo and tell you what branch you are in	git branch
	Delete the specific branch	git branch -d name_of_branch
Create a new local repository		git init
Commit		git commit -m “<message>”
Merge	To merge a different branch into your active branch	git merge name_of_branch
Push	Send changes to the master branch	git push origin master
	Push the branch to your repo	git push origin name_of_branch
Status	List the files you’ve changed and those you still need to add or commit	git status

Tell git who you are	Configure the author name and email address to be used with your commits	<pre>git config --global user.email <you@example.com> git config --global user.name <your_name></pre>
Undo	Undo the most recent commit	<pre>git reset HEAD~1</pre>