Git tutorial

Authors: Guillaume Oliviéro
Date: 2023-02-17
Contact: oliviero@cenbg.in2p3.fr

Table of Contents

Goal of the tutorial

- Set-up git SSH keys for cloning and push
- Learn the git flow
- Learn how to use properly a git repository
- Understand the branching concepts

Introduction

What is git?

`Oh baby, don't loose me. Don't loose me. No more`. Git is a powerful and free versioning system usually used for coordinating work among programmers collaboratively developing source code during software development.

Why using git?

- Backup and save your work
- Powerful version system if you want to roll back
- Most used collaborative tool for code
- You can use it for your codes, manuals BUT also your presentations AND your thesis ;)

Set-up SSH keys

You must follow this small github documentation guide:

https://docs.github.com/en/authentication/connecting-to-github-with-ssh/generating-a-new-ssh-key-and-adding-it-to-the-ssh-agent#generating-a-new-ssh-key

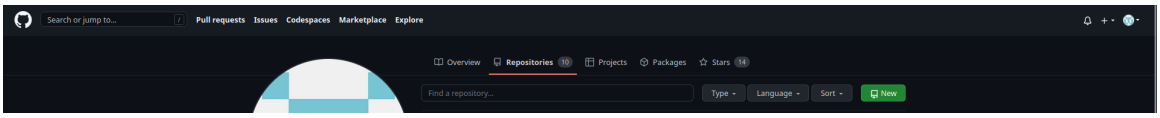
Bash commands

```
$ ssh-keygen -t ed25519 -C "your_email@example.com"

$ eval "$(ssh-agent -s)"

$ open ~/.ssh/config

$ touch ~/.ssh/config # if the config file doesn't exist

$ ssh-add --apple-use-keychain ~/.ssh/id_ed25519
```

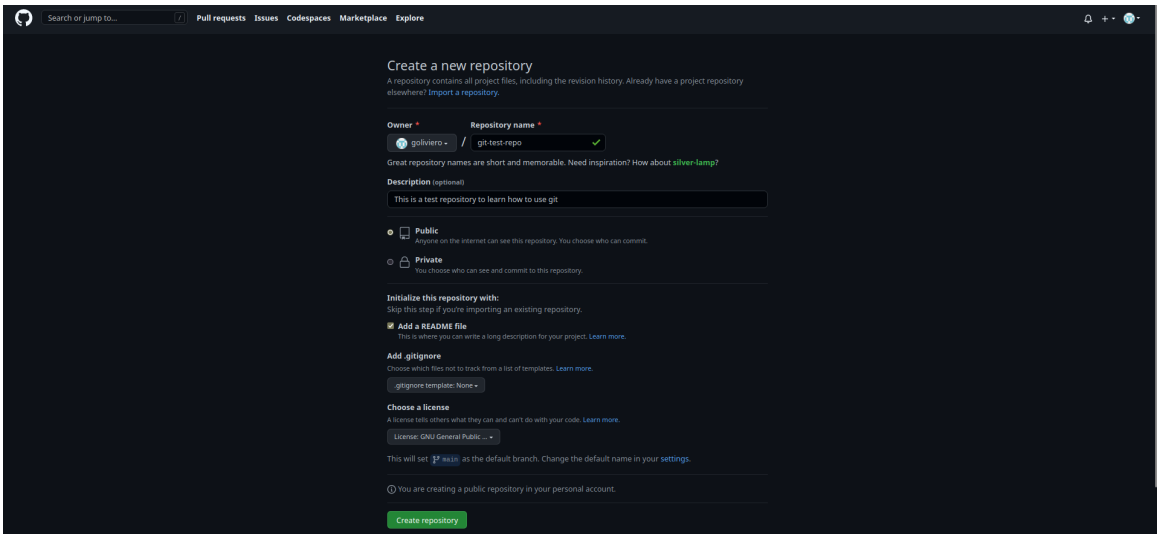
Now instead of using HTTPS to clone a repository OR each time you want to commit something, it will be done through SSH keys.

Git step-by-step

Step 1: Creating your first repository on the github interface



Under your account, click on Repositories and New

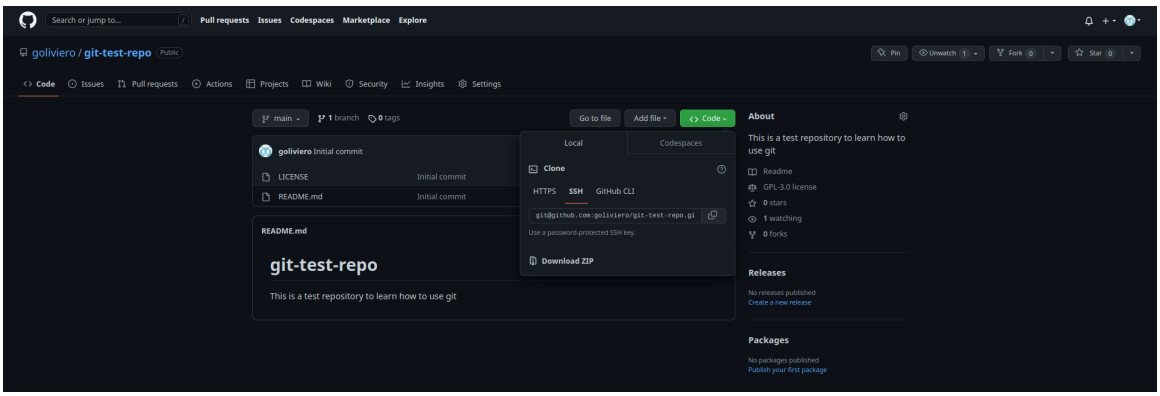


Give a name and a short description to your repository. Tick the `Add a README file` *box and choose a convenient License. GNU General Public License can be chosen*

Step 2: Cloning your repository

Cloning your new repository into your `/home/user/~`:

```
$ cd ~
$ git clone https://github.com/YOURUSERNAME/git-test-repo.git
# Replace username and the name of your new git repo, mine is under goliviero/git-test-repo.git
$ cd git-test-repo/
```



Cloning using SSH keys

Step 3: Stage new files

Creating a src directory:

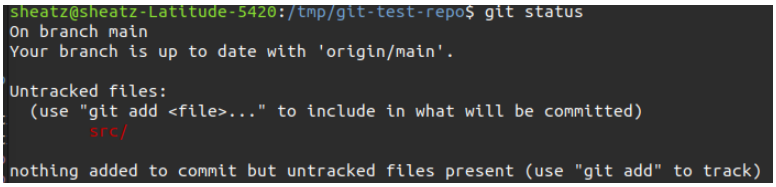
```
$ mkdir src/
```

Creating an empty cxx program:

```
$ touch src/test.cxx
```

See the status of your repository:

```
$ git status
```



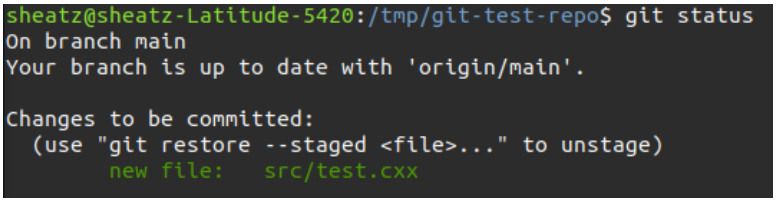
Git status initial before stage and commit

For now, the file is only existing locally. We want to add your first CPP program to your repository. First, we will `stage` (track) the file we want:

```
$ git add src/test.cxx
```

See the status again of your repository:

```
$ git status
```

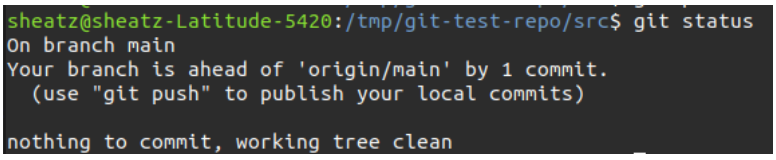


Git status after `git add` *while staged*

Step 4: Commit new staged files

After stagging your file, we want to commit your new file to the branch we are working on. For now, we are on the `main` branch.

```
$ git commit -m "Add a test cpp program to my repository"
```



Git status after `git commit` *and before the push*

The `-m` option allows you to do an inline commit message. Otherwise it will open an editor inside the terminal but you can give more details about your commit.

HOW TO RIGHT A GOOD COMMIT MESSAGE - few recommandations:

- Keep it short
- Use the imperative mood
- Add a short title
- Add a body (explain WHAT the change is, but especially WHY the change was needed)

- Good examples:
    - *Enable Logging Globally*
    - *Add Account Delete Route*

      Needed for account deletion workflow on frontend
- Bad examples:
    - *debugging*
    - *update*
    - *I've added a delete route to the accounts controller*

Step 5: Push files remotely

Once stagged and commit, we want to push the file to the online remote repository:

```
$ git push
```

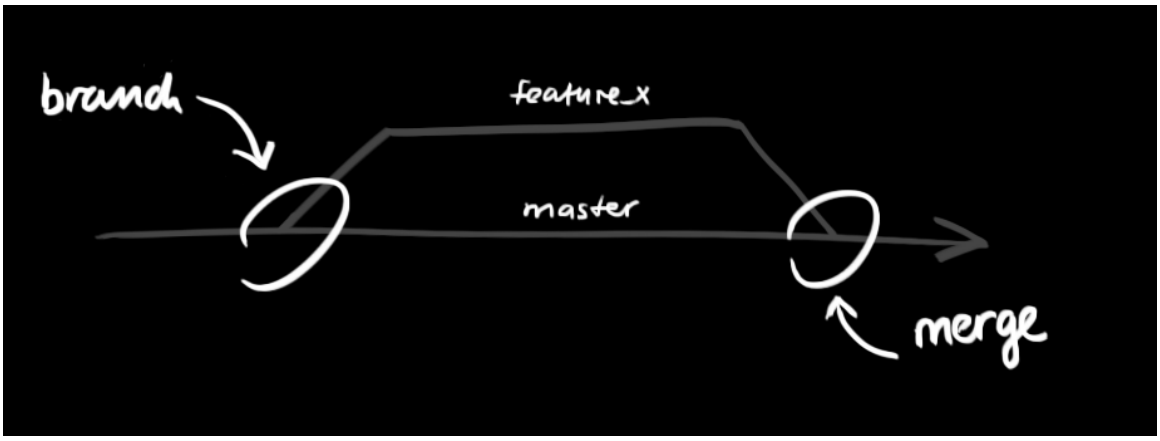
```
sheatz@sheatz-Latitude-5420:/tmp/git-test-repo/src$ git push
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (4/4), 371 bytes | 371.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:goliviero/git-test-repo.git
   864b65e..70b3bb8  main -> main
```

Git push on the `main` *branch*

Your file has been pushed to your `main` branch.

Step 6: Create a new branch and push it

Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is `master` or `main`.



First we start from our `main` branch:

```
sheatz@sheatz-Latitude-5420:/tmp/git-test-repo/src$ git branch
* main
```

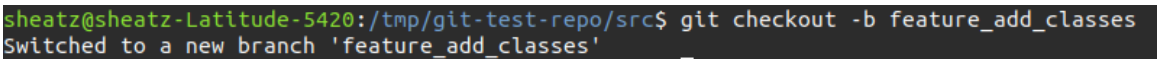
Make sure the starting point is the main branch in most cases

Creating a new branch for a dedicated feature. Here we will add two empty classes in our src/ directory.

First we have to create the new branch `feature_add_classes`.

```
$ git checkout -b feature_add_classes
```

the `-b` option allow us to create a branch and switch (i.e `git checkout`) directly on it.



Switching to the new feature branch just created

Then create the two empty classes named `foo` and `bar` on this new branch.

```
$ cd src
$ touch foo.cpp foo.hpp bar.cpp bar.hpp
```

Stage all the files at once in the src directory on branch `feature_add_classes`:

```
$ git add *
```

Stage all new files at once with `git add *`

Before commit, check we are in the right branch (i.e: feature_add_classes):

```
$ git branch
```

Checking we are on the good branch before commit and push

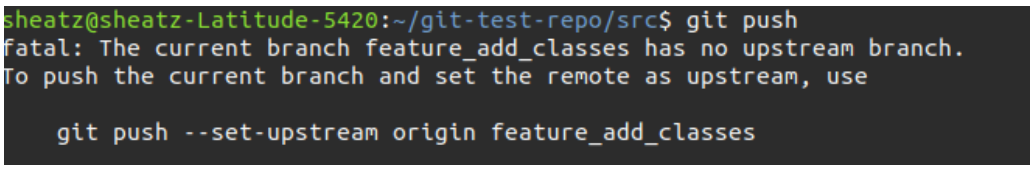
Commit the two classes to the `feature_add_classes` branch with a good and explicit commit message:

```
$ git commit -m "Add two empty classes named foo and bar"
```

First push to the upstream branch. If you try to just:

```
$ git push
```

You'll see a fatal error message:



Push error message because the current local branch as no upstream branch

The current branch `feature_add_classes` is only existing on your local machine and has no `upstream` branch remotely. We should set the `remote` as `upstream` using:

```
$ git push --set-upstream origin feature_add_classes
```

```
sheatz@sheatz-Latitude-5420:~/git-test-repo/src$     git push --set-upstream origin feature_add_classes
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 331 bytes | 331.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'feature_add_classes' on GitHub by visiting:
remote:      https://github.com/goliviero/git-test-repo/pull/new/feature_add_classes
remote:
To github.com:goliviero/git-test-repo.git
 * [new branch]      feature_add_classes -> feature_add_classes
Branch 'feature_add_classes' set up to track remote branch 'feature_add_classes' from 'origin'.
```

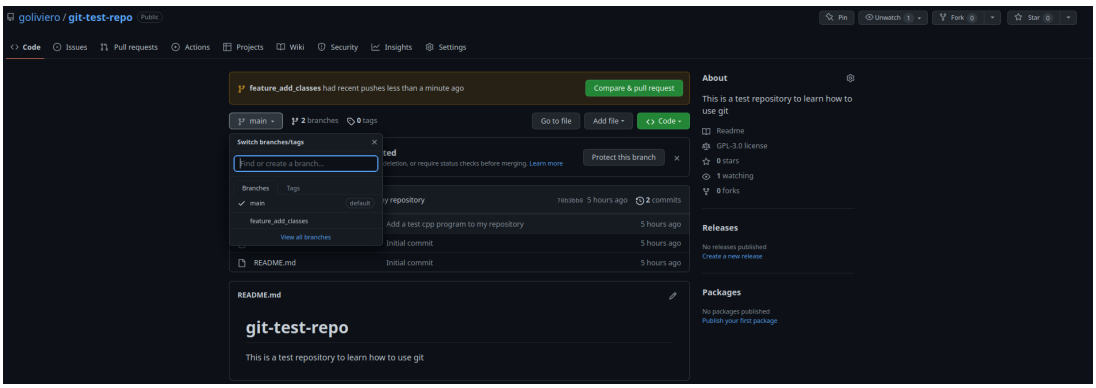
Commit and push the new branch complete

For the next pushes on this branch it will be set so you can just use `$ git push`.

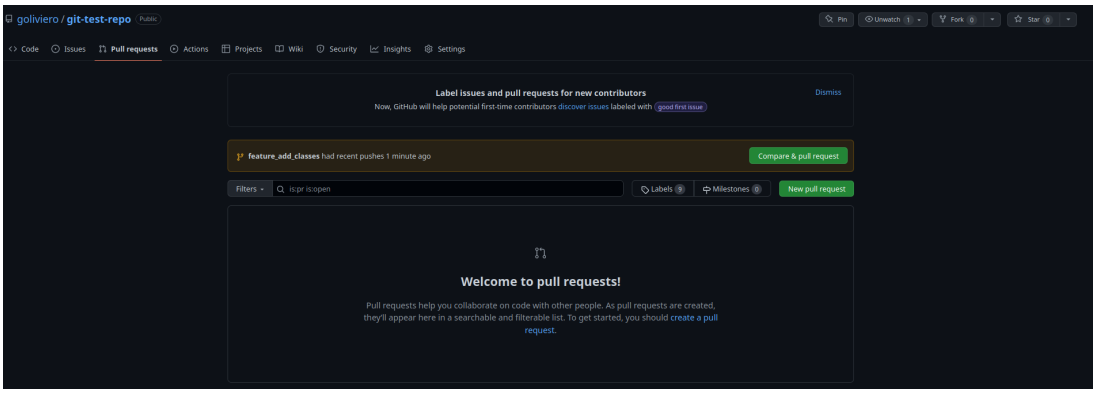
BRANCH NAMING CONVENTIONS: as for the commits, you should be concise and explicit about what you want to do with a branch. You can indicate if you want to add a new feature with the prefix `feature-`, fix a bug with `bugfix-` prefix, test with `test-` and so on. Then you should describe briefly the purpose.

Step 7: Open a merge/pull request

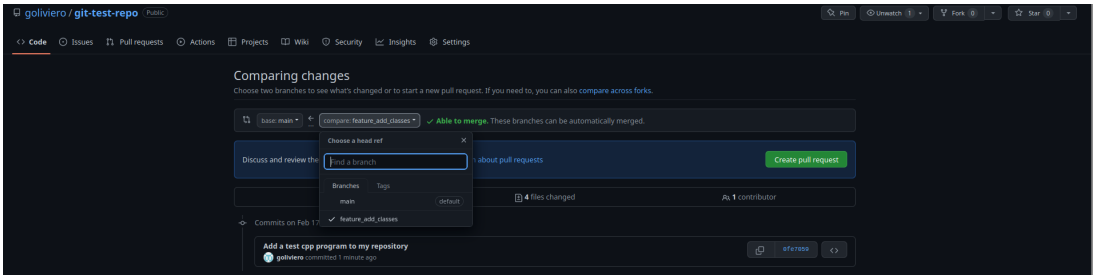
Opening a merge request through the git interface.



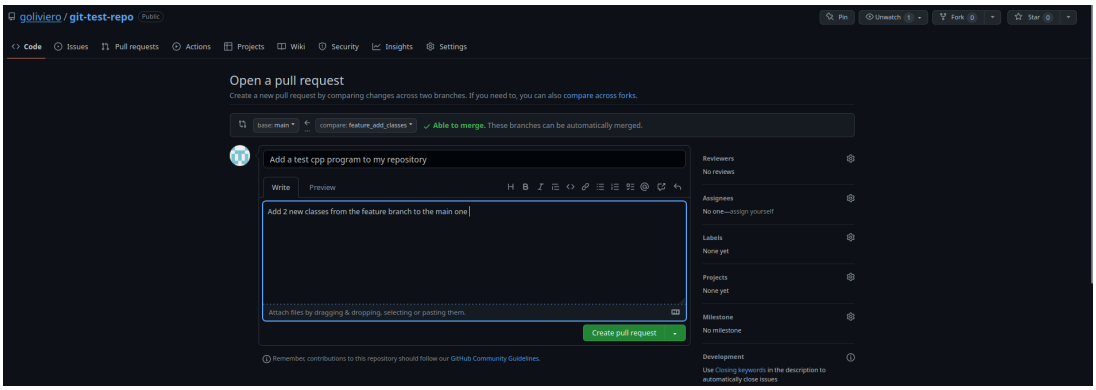
Git interface for the repository. Click on Pull Requests and you will open the interface where you can easily open one



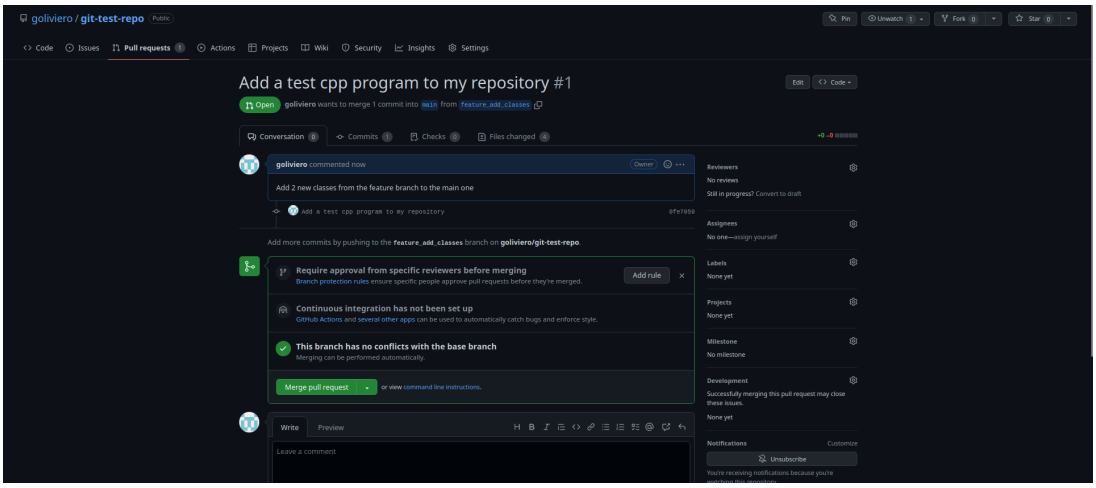
Interface to create a new Pull Request



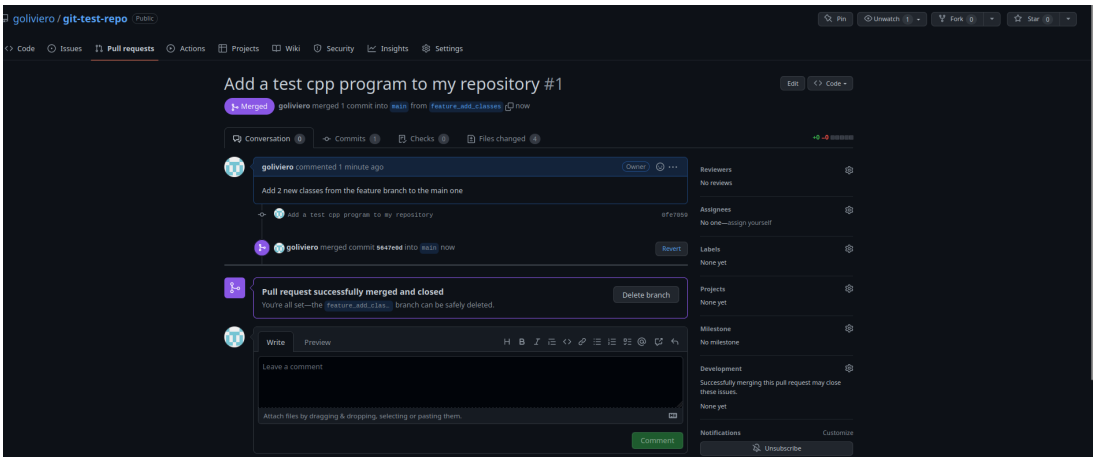
Choose the feature branch you want to merge into the main branch



Describe what your changes will do to the code



The merge request is now open and someone else (or you) can crosshcheck your changes and then accept or not your merge request. It is a space of discussion where someone can ask you to do some modifications and so on.



Accepted merge request (`Merged`)

Note 1: merge request and pull request are the same thing.

Note 2: we can do it with the command line but it is much less convenient. I'll let you look online for this.

Step 8: Pull and update your repository

Pull the changes in your main local branch from remote

```
sheatz@sheatz-Latitude-5420:~/git-test-repo/src$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

Checkout the `main` branch after the `feature` branch was merged into the `main` on the remote git interface

```
$ git pull --all
```

The `--all` option allows you to pull the commits as well as all of the branches from the remote.

The changes you made on the feature branch are now on the main branch and the 2 new classes `foo` and `bar` are available.

Step 9: Delete your branch localy and remotely

Deleting the branch you worked on (i.e `feature_add_classes` branch).

You can delete it through the git interface after accepting the merge request.

But you can also delete it manually and push this from local to remote. First of all, git won't let you remove the branch you're sitting on so you must make sure to checkout a branch that you are NOT deleting:

```
$ git checkout main
```

Delete the branch locally:

```
$ git branch -d feature_add_classes
```

```
sheatz@sheatz-Latitude-5420:~/git-test-repo$ git branch -d feature_add_classes
Deleted branch feature_add_classes (was 0fe7059).
sheatz@sheatz-Latitude-5420:~/git-test-repo$ git br
* main
```

Deleting localy the `feature` branch

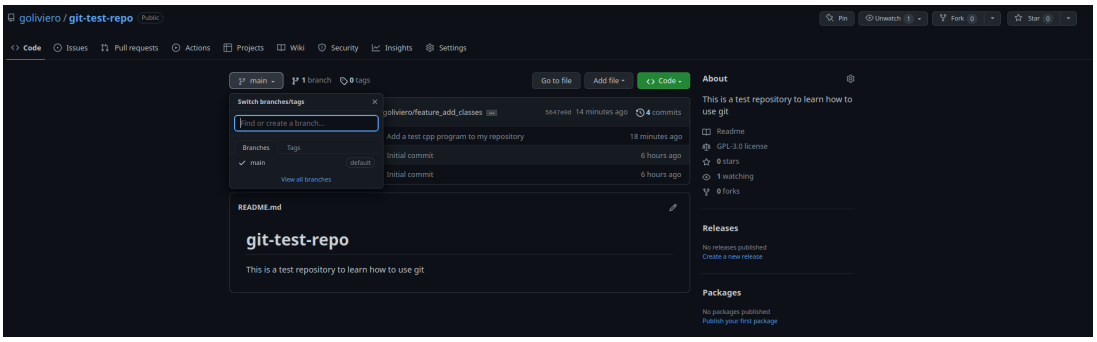
Then propagate it remotely:

```
$ git push origin --delete feature_add_classes
```

```
sheatz@sheatz-Latitude-5420:~/git-test-repo$ git push origin --delete feature_add_classes
To github.com:goliviero/git-test-repo.git
 - [deleted]         feature_add_classes
```

Deleting remotely the `feature` branch

and see the result on your git repository interface:



Git repository interface without the feature branch because we deleted it

Useful git commands

In this section I will provide some useful git commands.

Reset a commit not pushed to remote:

```
$ git reset HEAD~1
```

Reset the last commit pushed to remote:

```
$ git revert HEAD
```

Git has the ability to tag specific points in a repository's history as being important. Typically, people use this functionality to mark release points (v1.0, v2.0 and so on). List all the tags of the repository:

```
$ git tag -l
```

Create a new annotated tag (`-a` option) with a tagging message (`-m` option):

```
$ git tag -a v2.0 -m "my version 2.0"
```

Add some your email, name and some aliases to your `~/.gitconfig`:

```
$ emacs ~/.gitconfig

# Once in your gitconfig file you can put this basic gitconfig file:

[alias]
```

```
  co = checkout
  br = branch
  ci = commit
  st = status

[user]
  email = youremail@yourdomain.com
  name  = yourusername
```

Resources

- Official git scm (source code mirror) documentation: https://git-scm.com/book/en/v2
- Git - the simple guide: https://rogerdudler.github.io/git-guide/