

Lab 01 – Git

Group: Tobias Bayer, Jonas Mantay, Jonas Trenkler

Our repository: https://github.com/mantayjon/info03_lab01

Part 2: Set up a central GitHub repo and try out:

We set up a GitHub repository by clicking ‘new repository’ on the GitHub website. Our next step was to read the recommended instructions on the website. To create a new repository on the command line, it told us to:

1. `echo "# test">> README.md`

- writes ‘# test’ into a file. If the file does not exist, it is created first.

2. `git init`

- initializes a git repository. A hidden .git folder is created, which contains information about the repository.

3. `git add README.md`

- `git add file.extension` adds a file to the staging area, ready to be committed. We could have written `git add .` to add all files in the current working directory to the commit.

4. `git commit -m "first commit"`

- `git commit -m message` creates a commit with a message, e.g “add very important thing”

5. `git branch -M main`

- `git branch -M main` is short for `git branch -move -force main`. This means that a new branch will be created as main no matter what.

6. `git remote add origin https://github.com/abductedRhino/test.git`

- this adds a repository to pull from and push to. The name is origin.

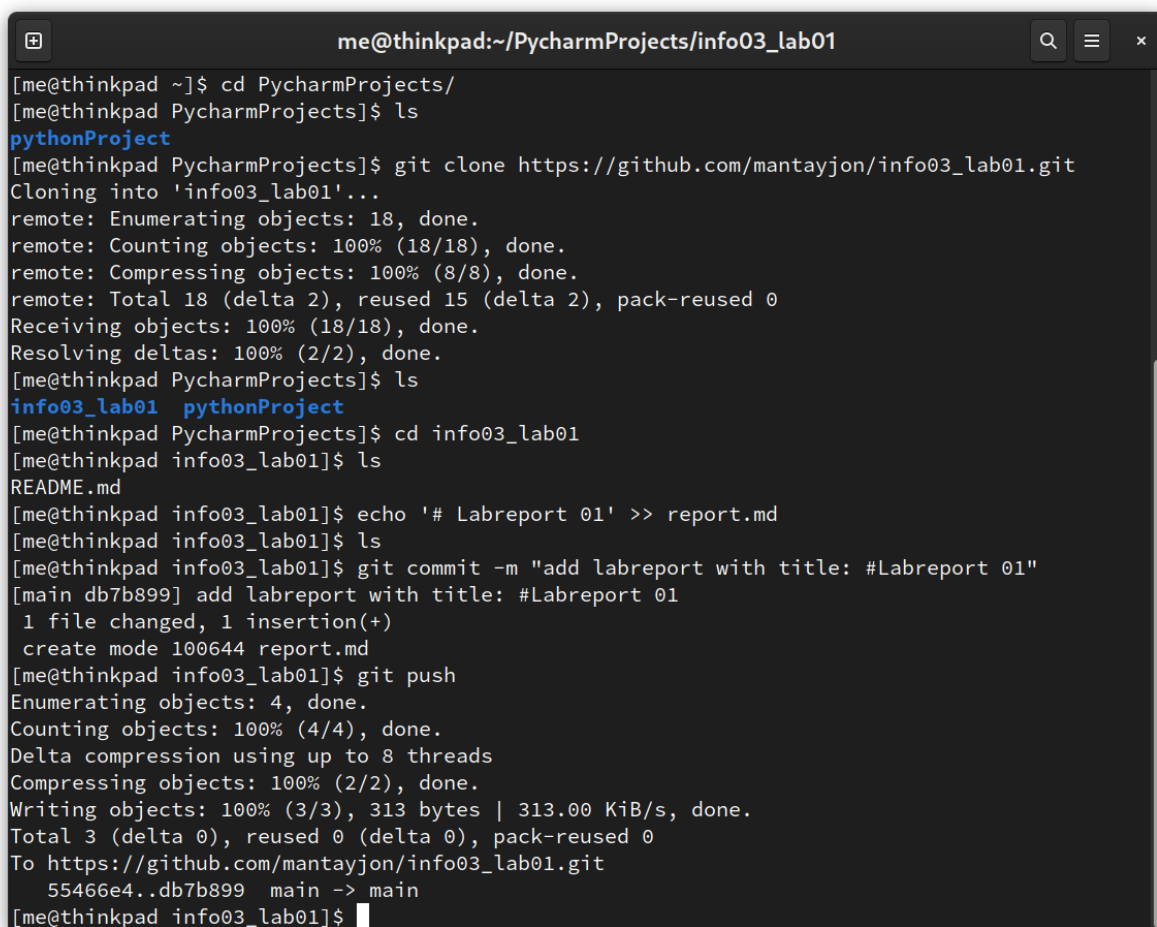
7. `git push -u origin main`

- this creates an upstream tracking branch and pushes it.

With the suggested steps, we were able to create and push a GitHub repository. We wrote a list called “Einkaufsliste” into our `README.md` and collaboratively pushed items to it. Again, we chose to use the Terminal:

```
1 mkdir info3
2 cd info3
3 git clone https://github.com/mantayjon/info03_lab01.git
4 cd info03_lab01
```

1. make a new directory named 'info3'
2. change directory to 'info3'
3. copy the git repository from https://github.com/mantayjon/info03_lab01.git
4. change directory to 'info03_lab01'



```
me@thinkpad:~/PycharmProjects/info03_lab01
[me@thinkpad ~]$ cd PycharmProjects/
[me@thinkpad PycharmProjects]$ ls
pythonProject
[me@thinkpad PycharmProjects]$ git clone https://github.com/mantayjon/info03_lab01.git
Cloning into 'info03_lab01'...
remote: Enumerating objects: 18, done.
remote: Counting objects: 100% (18/18), done.
remote: Compressing objects: 100% (8/8), done.
remote: Total 18 (delta 2), reused 15 (delta 2), pack-reused 0
Receiving objects: 100% (18/18), done.
Resolving deltas: 100% (2/2), done.
[me@thinkpad PycharmProjects]$ ls
info03_lab01  pythonProject
[me@thinkpad PycharmProjects]$ cd info03_lab01
[me@thinkpad info03_lab01]$ ls
README.md
[me@thinkpad info03_lab01]$ echo '# Labreport 01' >> report.md
[me@thinkpad info03_lab01]$ ls
[me@thinkpad info03_lab01]$ git commit -m "add labreport with title: #Labreport 01"
[main db7b899] add labreport with title: #Labreport 01
1 file changed, 1 insertion(+)
create mode 100644 report.md
[me@thinkpad info03_lab01]$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes | 313.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/mantayjon/info03_lab01.git
55466e4..db7b899  main -> main
[me@thinkpad info03_lab01]$
```

Figure 1: Screenshot: Setting up the lab report in the terminal

To push an item to the list in `README.md`, we open the file with any text-editor and append a new list-item in Markdown syntax, e.g. – `Salz` and hit save. We then return to our Terminal:

```
1 git add .
2 git commit -m "add item to list"
```

3 `git push`

1. add every new or changed file in current working directory to git staging area. In this case, just `README.md`
2. create a commit with message 'add item to list'
3. push, update the remote commit with the commit we just created.

With the above, we were able to change file collaboratively, but we created conflicts while working on the same file simultaneously. When trying to push to the remote repository, git warned us that there is a conflict that needs to be resolved before pushing. The same problem occurs when trying to pull the upstream branch. The CLI gave us hints how to proceed, and we chose to pull and merge the branch locally. This opened `vimdiff` as the default merge tool. For someone not used to vim a GUI merge tool like VS Code might be easier to use.

The merge conflicts can be prevented by:

- working in different branches
- working on different files

This copies the remote repo, makes a file with a line of text, adds it to the staging area, creates a commit with a message and updates the remote with the local commit. This report was created in a similar way as the `git init` above.

When in doubt

`git status` and `git log --oneline` helped to get an overview of what is going on in the repository. There are a lot of different options for `git log` that are worth exploring, this is just a convenient short form. `git fetch` was needed to get the current status of the upstream repository as well, `git pull` implicitly calls that and whatever strategy for dealing with conflicts is configured. Whenever possible the default is to just fast-forward, i.e., merge without creating a new commit. The other option is to merge after resolving the conflict manually. Finally, a rebase offers the most flexibility and `git rebase -i` is an interactive way to more or less rewrite the repositories' history. The following screenshot shows the contents of a rebase-todo, and all its options, during an interactive rebase. If a merge conflict is detected during the rebase, it can be resolved manually and the rebase is continued with `git rebase --continue`. Again, the CLI gave hints how to proceed.

```
1 pick 6b0fb09 added Milch to Einkaufsliste
1 pick be5827b add important stuff
2 pick 55466e4 added one more Ingredient
3 pick db7b899 add labreport with title: #Labreport 01
4
5 # Rebase 674d02c..db7b899 onto 674d02c (4 commands)
6 #
7 # Commands:
8 # p, pick <commit> = use commit
9 # r, reword <commit> = use commit, but edit the commit message
10 # e, edit <commit> = use commit, but stop for amending
11 # s, squash <commit> = use commit, but meld into previous commit
12 # f, fixup [-C | -c] <commit> = like "squash" but keep only the previous
13 #                               commit's log message, unless -C is used, in which case
14 #                               keep only this commit's message; -c is same as -C but
15 #                               opens the editor
16 # x, exec <command> = run command (the rest of the line) using shell
17 # b, break = stop here (continue rebase later with 'git rebase --continue')
18 # d, drop <commit> = remove commit
19 # l, label <label> = label current HEAD with a name
20 # t, reset <label> = reset HEAD to a label
21 # m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
22 #       create a merge commit using the original merge commit's
23 #       message (or the oneline, if no original merge commit was
24 #       specified); use -c <commit> to reword the commit message
25 # u, update-ref <ref> = track a placeholder for the <ref> to be updated
26 #                       to this position in the new commits. The <ref> is
27 #                       updated at the end of the rebase
28 #
29 # These lines can be re-ordered; they are executed from top to bottom.
30 #
31 # If you remove a line here THAT COMMIT WILL BE LOST.
32 #
33 # However, if you remove everything, the rebase will be aborted.
34 #
```

NORMAL db7b899 .git/rebase-merge/git-rebase-todo 12 Top 1:4 08:40

Figure 2: Screenshot of a git rebase