Git Exercise

Jens Lechtenbörger

August 18, 2020

1 Introduction

I suppose that you went through the Git Introduction and installed and set up Git before you start here.

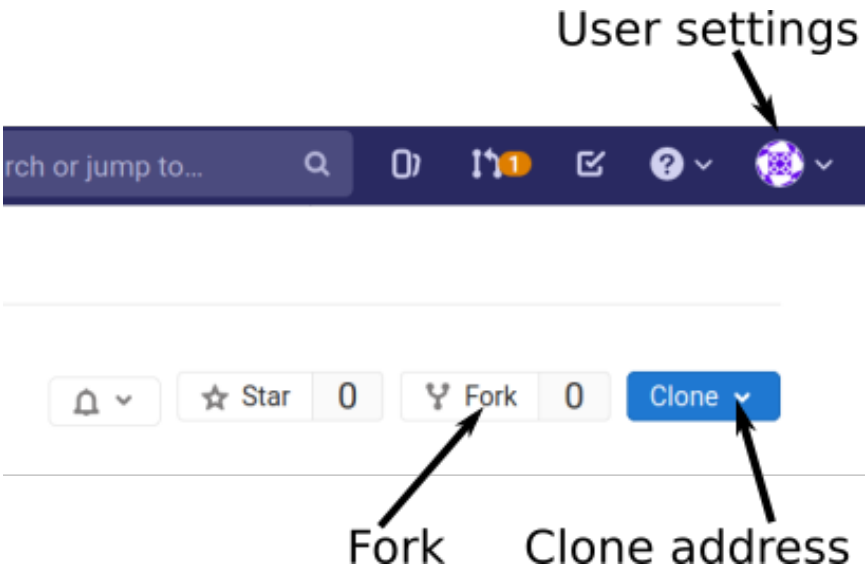
Please be aware that this is a new exercise, which we are going to perform jointly for the first time, so there may be rough edges. Please do not hesitate to ask early. Your first steps with Git will seem unfamiliar, but my goal is to help you in this unfamiliar terrain. Git is a powerful tool with many commands and options (GitLab even more so), but the basic workflow should not be hard to follow. (As an aside: If you really go for Git, you probably want to do that embedded in your daily work environment, where the most important Git operations are available through some UI. My daily work environment is GNU Emacs with Magit.)

On our GitLab server, I will assign each of you as "Reporter" to my project "cacs-2020", and GitLab will notify you once that happened. (This will take some time because our admins need to create your accounts first, before I assign you manually ...) Then, you can start with this task, which is meant to practice the feature branch workflow mentioned in the Git Introduction.

2 Tasks

This task is part of a group exercise. In the following, M1, M2, ... indicate different group members.

2.1 Part 1



1. Determine group member M1 who forks project "cacs-2020".

2. M1: Go to my project and perform the fork (see screenshot, which shows my browser's upper right). Default settings should be fine.

   In your fork, assign the remaining group members as Developers or Maintainers and me (my username is lechtej) as Guest or higher. (Maintainers can do everything in a project, while Developers cannot push to the `master` branch, enforcing to work with other branches. As Guest, I can open so-called Issues to offer feedback if necessary, but I can neither clone your fork nor see your branch. That would be possible in other roles.) To assign members in the fork, to the **left**, go to the project's "Settings → Members" (*not* part of screenshot), search members and assign roles.

   The newly added project members (including me) receive e-mails with the address of the forked project.

3. All: Check your notification settings (see screenshot, User settings → Notifications), where you can configure for what events to receive e-mails. Maybe "Watch" your own project; at least, make sure that you are notified if I open an issue.

   Setup SSH keys (see slide on SSH for pointers). If you are not sure whether SSH setup was successful, execute this:

`ssh -T git@wiwi-gitlab.uni-muenster.de`

If you see a success message from the previous command, you are good to go. Otherwise, add option `-v` for verbose output:

`ssh -v -T git@wiwi-gitlab.uni-muenster.de`

Pay attention to lines about `identity file`. Those ending in `-1` imply that `ssh` tried to access a key that does not exist. At least one must exist. Ask.

Clone the fork to directory "cacs-2020". The necessary address (starting with `@git`) is visible under the Clone button (see screenshot) under "Clone with SSH".

4. M2: Create a new branch that collects your group work for this task, say "g42-task-1". Create a new sub-directory (under "cacs-2020") for your group, e.g., "G42" if you are group G42.

   Please restrict all your group's changes to that sub-directory.

   As this was missed by several groups in the first run, let me **repeat**: Create a **new branch** and a **new sub-directory**. This task is meant to familiarize you with branches, and the directory provides a scope for your file names. (Several groups created files named `test.txt`...)

   If you do not create a directory, I will reject your merge requests later on, asking you to move your files. (Then, you may want to read `git help mv`.) If you do not use a branch, then you need to read about protected branches at GitLab; note that `master` is protected by default, and group members with the role Developer are not allowed to push.

   Add a first document (which will be **shared** via my project with the **entire class** subsequently, so, please, take some care), either a text file or some lightweight markup language; maybe start to document your experiences, questions, and answers. Commit (maybe more than once) and push the branch.

5. M3: Pull and check out your branch, change the file more, commit, and push.

6. M4: Create a merge request in GitLab for your branch into my `master` branch. (On the top of your fork's project you should see a suggestion to create a merge request for your branch. Maybe refresh the page.)

   When you create merge requests, please fill out the title (e.g., Group 42 - Task 1). The description may remain empty here; in a real project you

would explain what you propose to be merged. Note that the option "Delete source branch when merge request is accepted" is checked by default. Maybe uncheck it to keep your branch.

2.2 Part 2

After I merged your branch, my project is ahead of yours. To bring your project up-to-date (possibly including changes of other groups as well), please do the following in your project (see slide on "git remote").

```
git remote add upstream <URL of original project>
git fetch upstream
git checkout master
git rebase upstream/master
git push
```

Note that by default only Maintainers (M1, maybe others) can push to the `master` branch of repositories.

During your work, you may see error messages, such as `failed to push some refs`. In particular, `pull` and `push` only succeed if one project is strictly behind or ahead of the other. Usually, Git provides some hints how you may proceed.

One important command may be `git pull --rebase`, which you can execute if additional commits happened at the remote end as well as at yours. In this situation, `git push` is rejected. Try `git pull --rebase` first, then push again. Also, in this situation, `git pull` would offer to merge changes, leading to an additional commit. With option `--rebase`, however, remote commits are fetched, and your local commits are re-executed on top of that (see slide on "git rebase"). Afterwards, `git push` should succeed. In general, for questions that I have related to Git, Stack Overflow has answers. You may want to ask in a course-related forum, though.

We work on a playground project here, and it is unlikely that you destroy things (well, `git reset` and options that "force" actions may destroy state; you should not need them). If your local repository seems to be in an inconsistent state, you could just give that up and clone again (depending on the amount of work that you did locally; you can also copy files from the inconsistent state to a new clone).

Again, please do not hesitate to ask in a course-related forum.

3 Further Suggestions

Create and resolve a conflict, to see what this is about: Two team members, say Alice and Bob, pull their fork's most recent state. Both edit the same file on the same branch simultaneously, commit, and try to push. Clearly, both commits are based on the same parent commit, which implies that they are not nicely ordered. Hence, one push operation will be rejected, say Bob's.

Given the rejected push, Bob could now `git pull` (don't do that yet—read on), which is really `fetch` followed by `merge`, ultimately creating a merge commit. Note that you see two branches on the slide, while you may believe to work on one. That belief would be wrong: You work on your local branch with your commits, which you try to combine with a second remote branch that potentially includes other commits.

To avoid the merge commit, Bob could `git pull --rebase` to rebase own commits on the current remote state as suggested in the previous section.

In both cases (merge and rebase), Bob has to deal with the fact that the same file was edited simultaneously at different places, maybe with conflicting or even contradictory contents. If such conflicts occur, Git inserts so-called conflict markers to highlight conflicting regions and tells as much on the command line, with suggestions for follow-up commands. It is up to Bob to decide what to do with those regions. Sometimes, Alice's part may be better than his, sometimes the other way round, sometimes both changes may complement each other. In any case, he produces a version in which no conflict markers are left, commits, and pushes.

If Bob is lucky, his push succeeds. If not, somebody, maybe Alice again, was faster to commit and push in between, and Bob has to start over. This does not look like an effective workflow, does it?

So, in your team, you also may want to apply the Feature Branch Workflow, where each team member works on an own branch (or really on branches for specific purposes/features) to commit and push freely, only to merge or rebase with the team's main branch when stable states have been reached. In this case, it is a good practice to rebase the individual branch on the most recent state of the team's main branch before merging.

Now, try the above?

4 What's next

You have experienced Git as sample communication and collaboration system for distributed teams. In the next session, you will learn to position Git

as an example of so-called *distributed systems*, and we will revisit the notions of consistency and conflict in that larger context.

Upcoming presentations contain "Review Questions". Manage your answers in additional branches ("g42-task-2", ...) and submit a merge request once you are done with the presentation. Make sure that different group members participate with commits and merge requests.

I'll provide feedback for merge requests.

Have fun!