Linköping University
Department of Computer and Information Science (IDA)
The UPP group 2015-09-01

A thorough look at `git`

Aim

Your goal should be to learn how to use `git` day-to-day and how to handle the most common situations. Complete the tasks in this laboration in order.

`Git` itself is best described by reading `https://git-scm.com/book/en/v2`.

Preparation

Read this laboration in it's entirety.

Read chapter 1.3 in `https://git-scm.com/book/en/v2`. Learn about *snapshots*, *repository* and *staging*. Quickly scroll through chapters 1,2,3,6,7,8 and A3. Just look at the headlines and commands to get an idea of what topics they cover. Later on it will be easier to look for something you *know* you've seen "somewhere".

Tasks

In most task you will see a hint. The hint is not a complete command, you have to read about what it does and adapt it to your needs.

Basic git

1. Visit `https://gitlab.ida.liu.se/` and setup your personal git account. Click around to get accustomed to the interface. Hover the mouse to get tooltips. Make sure to set up ssh-keys according to `https://gitlab.ida.liu.se/help/ssh/ssh.md`. At this point you only need a working login to `gitlab.ida.liu.se` in order to access the repository in the next step.
2. Clone the `git-laboration` repository at
   `https://gitlab.ida.liu.se/klaar36/git-laboration.git` (https) or
   `git@gitlab.ida.liu.se:klaar36/git-laboration.git` (ssh).
   This will create a local repository for you and populate it with some files. [Hint: `git clone`]
3. The (only) important thing for this step is that you *modify* at least one existing file and *create* one new file. But you also have the chance to share the questions you have.

(a) Change to the repository folder and use Emacs to investigate the files in the `questions` folder.
(b) Create a new file in the folder. The file should contain a one line question about git. You do not have to know the answer, but may add answer options if you like to. You may also add more options to existing questions.
(c) Modify `CONTRIB.TXT` by adding your information as suitable.

4. Check the status of your repository. `Git` will produce a list of the files you've changed or created. [Hint: `git status`]
5. To save your changes in your local repository you must tell `git` which files and changes you want to keep. You do this by *adding* your new or modified files to the *staging area*. Everything in the staging area will be included in the next commit.
   This way you can craft and fine-tune your *commit* before actually committing. Compare to a chef at a restaurant, he prepare the food out in the kitchen using pots and pans, then he *stages* a meal by placing it nicely on a plate, before finally *committing* by bringing the plate to the customer. You prepare the files in your working folder using Emacs, then stage the files before finally committing to your local repository.
   You can also choose to split your changes over several commits (just as the chef may choose to deliver the meal as teaser, main course and dessert instead of serving it all at once). If you split your commit in several parts it will be easier to find a certain change later.
   Add *one* of the files you modified to the staging area. [Hint: `git add`]
6. Check the status again. `Git` will list files to be committed and modified files not staged for commit.
7. It is tedious to add each and every file. Add all changed files at once to the staging area. [Hint: `flag -A`]
8. Check status. Stop! Do you really want to commit all files listed? Normally emacs create backup files (ending in `~`) that we want to keep out of the repository. Remove all backup files from the staging area. [Hint: `git reset`]
9. Check status again. It is painful to manually keep all unwanted files (emacs backup files, LaTex generated files, compiler generated files or any other auto generated or temporary files) out of the repository. To tell `git` to automatically ignore files you can add ignore-patterns in special configuration files. Go ahead and add `*~` to your `~/.config/git/ignore` (for global enforcement) or to `.gitignore` in each local folder (to tailor each folder individually).
10. Check status. Now the backup file will not appear anymore. Make sure the list only includes files you want to maintain in the repository.
11. View how your working copy differs from the last committed snapshot [Hint: `git diff HEAD`].
12. Keep working, add some more question or answer option(s).
13. Now your working copy may be different from your staged copy, that in turn will be different from your repository. View how the working copy differs from the staged copy [Hint: `git diff`].
14. Add your latest changes to the staging area and commit your files. You need to specify a message to describe the set of changes you commit. [Hint: `git commit`]

Remote repository

This assumes you just solved the tasks in previous section.

1. Check status and differences again. What if you forgot to mention something important in the commit message? What if you just remembered another change that should have gone in the same commit? One way is to simply do a second commit, but it is better to keep the repository clean. You can amend files to your commit or change the commit message if you remember something you forgot just after committing. Try to amend your last commit message. [Hint: `flag --amend`]
2. So far you've been working with your local snaphot and committed to your local repository. Now it's time to cooperate. You need to make your changes known to the world. This is done by pushing your local repository to a shared remote repository. When you clone a repository as you did in the beginning of this laboration, the *origin* remote repository is automatically set. View which remote repositories you have configured. [Hint: `git remote show`]
3. You created your own remote repository on `https://gitlab.ida.liu.se/` in the first step. Log in an create a new project *git-tutorial* (the `+` button in the upper right area).
4. Add *git-tutorial* as a new remote repository (to the local repository where you work on this laboration). To find the git-url of your new *git-tutorial* repository, click *Dashboard* in the upper left, then *Projects*, then *Personal* and finally the project you're searching. [Hint: `git remote add`]
5. Push your changes to your new remote repository. [Hint: `git push`]

Conflicting files

This assumes you just solved the tasks in previous section.

1. Ask one of your friends for the url of his/her repository and add it as remote repository. Pull his/her changes to incorporate with your own. **At this point you may encounter an error regarding conflicting changes. If you do, skip the next point.** [Hint: `git pull`]
2. **Do this only if the previous step succeeded.** Help each other to introduce a conflicting change in some file. Typically you change a line in your snaphot and your friend changes the same line in a different way. Decide on a remote repository to share and set it up as remote as needed. Have your friend commit his change and then push it to the *shared* remote repository. Then commit and push your own change with a different commit message to the *shared* remote repository. The last push will be refused due to the collision. Pull your friends conflicting change. `Git` will state in which file(s) the conflict is. [Hint: `git pull`]
3. **At this point you should be stuck with a conflict.** Open the conflicting file in Emacs to review the differences. Edit the file and save a correct version. Then add the file to the next commit and perform the commit. [Hint: `git add`]

4. Push your changes to your remote repository. It should work, but if you have new conflicts you know how to resolve them.
5. View the commit log as a simple ascii graph. [Hint: `git log --oneline --decorate --graph --all`]

Branching and tagging

This assumes you just solved the tasks in previous section.

1. Pretend you have an idea for a new cool slogan to remember a good way to work with git, for example *Branch, Develop, Save and Merge*. You want to use the right work-process to add a question about it. Create a new branch. [Hint: `git branch`]
2. Switch to your new branch. [Hint: `git checkout`]
3. Work on your new question. While working you discover a spelling/phrasing error in a previous question. It's a small quick correction and you want to correct it immediately. At this point you want to switch back to your master branch and correct the spelling/phrasing. But if you do a checkout you'll lose all uncommitted changes in the current branch. You could commit them first, but commit is not a good solution when you want (and you do want) a clean repository without half-done work. Instead you should stash your half-done work for later completion. [Hint: `git stash`]
4. Switch switch back to your master branch, correct the spelling/phrasing, and commit the correction with an appropriate message.
5. Switch back to your branch and continue work. [Hint: `git stash apply`]
6. When you're done you should try out both what to do if the work (in this case the question) turned out good (you want to keep it) or inappropriate (you want to abandon it). You may want to create one slightly different branch for each of those cases to try both.
   - To incorporate the branch in the main code, switch to master and merge. [Hint: `git merge`]
   - To abandon the changes you can switch to master and delete the branch. [Hint: `git branch -d`]
7. Finally, you have work ready for release (or let's at least pretend you do). Now you want to remember the release version. Do so by creating an annotated tag. [Hint: `git tag -a`]
8. Tags are not included automatically when you push, you have to push it specifically. Push your tag to your remote repository.