Git Notes

Filippo Valmori

9th June 2024

1. Installation

- Installation procedure (tested on *Windows 10* OS):
  - download and launch installer from Git website (free and open-source);
  - set *C:\Program Files\Git* as installation path;
  - tick *Open Git Bash here* and untick *Open Git GUI here* within *Windows Explorer integration*;
  - select *Use Visual Studio Code as Git's default editor*;
  - select *Let Git decide* about default branch naming;
  - select *Git from command line and also from 3rd-party software*;
  - select *Use bundled OpenSSH*;
  - select *Use the OpenSSL library*;
  - select *Checkout Windows-style, commit Unix-style line endings*;
  - select *Use MinTTY*;
  - select *Fast-forward or merge* as pull-command behavior;
  - select *Git Credential Manager*;
  - tick *Enable file system caching*
  - skip the *Experimental options* window and start the installation.

2. Setup

- User's name and email configuration:
  - to configure Git username ⇔ *git config --global user.name "<name>"* (e.g. *Filippo Valmori*);
  - to configure Git email ⇔ *git config --global user.email "<email>"* (e.g. *filippo.valmori@gmail.com*);
  - **NB #1**: for the last two commands, *--system* or *--local* could be used in place of *–global* to (however that's in general not recommended, see Coursera's training for more details);
  - to readback set user's name and email ⇔ *git config user.name* and git config user.email (or all at once via *git config [--global] --list*);
  - to avoid line-ending issues among team members working with different OSs and automatically convert 'CRLF' (typical of Windows) line-endings into 'LF' (typical of Linux and macOS) when adding a file to the index (and vice versa when it checks out code onto your filesystem) ⇔ *git config --global core.autocrlf true* (in particular, when asserted on Windows machines, this converts 'LF' endings into 'CRLF' when you check out code);
  - **NB #2**: 'CR' = '\r' = *Carriage Return* character | 'LF' = '\n' = *Line Feed* character.

- SSH key generation for encrypting and authenticating communication from/to server (assuming ED25519 algorithm):
  - open Git bash (anywhere);
  - type *ssh-keygen -t ed25519 -C "<email>"*;
  - empty-ENTER until completion;
  - check public (.pub) and private keys have been successfully created in the specified (hidden) folder *.ssh* (e.g. *C:\Users\Filippo\.ssh*);

- from internet browser, go to *github website* > *profile icon* > *settings/manage-account* > *ssh keys* and upload the authentication public-key just created (simply drag-and-drop it);
- as a confirmation, type on bash *git gui* to open Git GUI and here go to *help* > *show ssh key* and verify the key has been successfully loaded;
- **NB #3**: setting up the SSH key is helpful because it allows to automatically authenticate yourself when accessing the remote server, thus without the need of supplying your username and password at each visit. For further details see *here*.

3. Locations & Syntax

- Locally on each developer's PC the so-called *project directory* folder contains:
  - ***working tree*** (WT), where the actual project files and directories (relative to a single commit at each time) are placed and can be edited;
  - ***staging area*** (SA, sometimes called also *index*), where the file planned to be part of the next commit are stored (aka *staged*);
  - ***local repository*** (LR), storing all the commits of the project (thus, representing its versioning history).

- Note locally SA and LR are located inside the hidden sub-directory *.git*. Thus, removing this folder means removing all the project version history locally.

- The ***remote repository*** (RR), unlike the other three mentioned above, is located remotely in a single data-center or cloud and represents the common interaction point among all developers (thus it identifies the official state of the project at any time). When LR and RR are synchronized, they contain exactly the same commits.

- The general syntax for commands is *git command [--flags] [arguments]* or, more in detail, *git command (-f|--flag) [<id>] [<paths> ...]*, where:
  - | = alternative;
  - *[]* = optional values;
  - - or -- = command flag or option;
  - <> = required values (aka *placeholder*);
  - *()* = grouping (for better clarity and disambiguation);
  - ... = multiple occurrences possible.

4. Create new local repository

- If the repository does not exist remotely on GitHub website yet:
  - create a folder where to place all repositories (e.g. *repos_folder*);
  - move inside the folder and create an empty sub-folder named after the new project to be created (e.g. *proj_xyz*);
  - move inside the sub-folder, open a bash and initialize the LR ⇔ *git init*;
  - verify the *.git/* hidden folder has been successfully ⇔ *ls -a*;
  - now work on the repository by adding, modifying and/or removing files;
  - **NB #1**: for new repositories it is good practise to add a *README.md* file to project main path (if not existing yet).

5. Commit to local repository

- Useful commands for committing:
  - to check if there are files/directories modified or untracked in respect of the last commit ⇔ *git status [-s]*;
  - to add desired untracked or modified files/directories (and in turn all files inside the latter) to SA ⇔ *git add <files-or-directories>*;
  - alternatively, to add all untracked, modified and deleted files/directories to SA without specifying them one by one ⇔ *git add --all | -A*;
  - to move all staged files/directories from SA to LR (i.e. adding a new snapshot/node representing the current state of the project to the version history) ⇔ *git commit -m "<comment>"*;
  - to verify WT and SA have been cleaned of all committed files/directories ⇔ *git status*;
  - finally check the LR commit-history of the current branch via *git log* (see Section 12.);
  - **NB #1**: to delete a current branch commit in LR ⇔ *git reset --hard <commit_id>* (or *git reset --hard HEAD~X* to cancel last X commits, assuming HEAD is pointing to latest commit of the desired local branch);
  - **NB #2**: to examine changes more in detail and select one by one the files to stage and commit use Git GUI (see Section 13.).

- Not all modified and untracked files in the WT have to be staged and then part of the next commit. Typical examples are the *.o*, *.map*, *.bin*, *.srec* or *.exe* files generated after building, since only *.c* and *.h* source files are actually included in the versioning (assuming not to include them in any *.gitignore* file).

6. Push to remote repository

- In case LR exists (see Section 4. and 5.) and RR does not yet:
  - open an internet browser and create a new empty repository on GitHub website with the same name of that in your LR (e.g. *proj_xyz*);
  - open bash inside LR and link it to RR through its URL address (retrievable at *github website* > *prj_xyz* > *code* > *HTTPS/SSH*) ⇔ *git remote add <prj_name> <url>* (e.g. *<prj_name>* = *origin* or *proj_xyz*, and *<url>* = *https://github.com/AlectoSaeglopur/proj_xyz* or *git@github.com:AlectoSaeglopur/proj_xyz.git*);
  - **NB #1**: using SSH addresses is always recommended if SSH key has been already generated and linked for the user (see Section 2.)];
  - initialize RR according to current LR state ⇔ *git push -u | --set-upstream <rr_name> <branch>* (e.g. *git push -u origin master*, or more specifically *git push --set-upstream proj_xyz master*).

- In case RR already exists and LR does not yet:
  - clone RR to LR ⇔ *git clone <url>* (as already mentioned, SSH addresses are recommended, thus use HTTPS only if SSH does not work);
  - now work on the repository by adding, modifying and/or removing files;
  - commit desired changes to LR (see Section 5.);
  - push current LR state to RR ⇔ *git push*.

7. Objects & IDs

- Git provides four main types of so-called *objects*:

- ***commit*** ⇔ representing a snapshot of the repository at a particular point in time;
- ***annotated tag*** ⇔ representing a permanent reference to a commit (typically used for software releases);
- ***tree*** ⇔ used to create the hierarchy between files and directories within a repository;
- ***blob*** ⇔ where the actual content of project files is stored.

◦ Typically the user has to care only about commits and annotated tags, whereas trees and blobs are handled internally and hiddenly by Git itself.

◦ The specific name of a Git object is called *ID* (aka *hash* or *checksum*), consisting of a 40-character hexadecimal string generated through the SHA-1 encryption algorithm (e.g. the IDs of all commits related to a branch can be seen via *git log*). However, IDs are often shortened to the first seven characters to make their visualization more user-friendly (e.g. as they are shown via *git log --oneline*).

◦ For each commit Git automatically create and associates a unique ID (generated through an avalanche principle part of the SHA-1, i.e. producing massive differences on the hash value even for minimal changes on repository files), so to guarantee consistency-check.

8. References & Tags

◦ A commit can be associated with a ***reference***, namely a user-friendly name (e.g. *HEAD* or *master*) pointing to a commit hash (e.g. 64a0c2b...) or another reference (aka *symbolic reference* in this latter case). Therefore, references can be used instead of hashes for simplicity's sake.

◦ Each branch is assigned with a so-called ***branch-label***, a reference with the same name of the branch (e.g. *develop* or *master*) pointing always to the most recent commit of that branch (aka *tip of the branch*). Note that *master* is the default name of the main branch in every repository (thus also the *master*-reference does exist for any repository).

◦ All references are automatically stored and updated within *.git/refs/heads/* and they are nothing but files named after the corresponding local branch and containing inside their current local hash/commit value. The only exception is *HEAD*, which is kept within *.git/*.

◦ The *HEAD*-reference points to the branch-commit pair currently present in the WT (thus, it can exist only one *HEAD* per repository). By default it is usually equal to the *branch-label* (e.g. *master* or *develop*), but unlike the latter this can be also moved back to previous commits of the branch via *git checkout <commit_id>*. For example, assuming to be in the *master* branch, if using *git log --oneline* returns that *HEAD* points to *master* (i.e. *HEAD* → *master*), then using *git checkout HEAD~* (i.e. *git checkout <previous_commit_id>*) moves *HEAD* one commit back and updates the WT accordingly (thus now *master* becomes one commit ahead of *HEAD*). Executing now *git checkout master* resets *HEAD* equal to *master* (i.e. to the latest commit of the branch).

◦ The '~' and '^' characters can be used to refer to previous commits. In particular, '~' allows to refer to single-parent commits, whereas '^' to multiple-parent commits (i.e. in case of merge-commits). For example:

- to show latest four commits of current branch (e.g. *develop*) ⇔ *git log --online -4*;
- to print detailed info about last commit (assuming *HEAD→master*) ⇔ *git show HEAD* (expected to return the same commit ID shown as 1st entry by the aforementioned *git log* command);
- to print detailed info about second-last commit ⇔ *git show HEAD~1* or *git show HEAD~* (expected to return the same commit shown as 2nd entry by the aforementioned *git log* command);

  - to print detailed info about third-last commit ⇔ *git show HEAD~2* or *git show HEAD~~* (expected to return the same commit shown as 3rd entry by the aforementioned *git log* command);
  - ...

- ***Tags*** are references attached to specific commits, acting as a sort of user-friendly labels for these commits. Thus, tags can be used in place of branch-labels or IDs for Git commands (verifiable via *git show <tag_name>*). There are two types of tags:
  - ***lightweight***, a simple reference to the commit (just like branch-labels or *HEAD*) with no additional information;
  - ***annotated***, a full object referencing the commit (including tag's author, date, message and commit ID), which can be optionally even signed and verified via GPG (aka *GNU Privacy Guard*), and typically used for code-releases.

- Useful commands for tags:
  - to create a new *lightweight tag* ⇔ *git tag <tag_name> [<commit_id>]* (e.g. *git tag v.3.1.8* automatically linked to *HEAD*);
  - to create a new *annotated tag* ⇔ *git tag -a [-m "<message>" | -F <file>] <tag_name> [<commit_id>]* (e.g. *git tag -a -m "release for EMC tests" v.3.1.8* or *git tag -a -F my_message.txt v.3.1.8 f318bd7*);
  - to check the tag has been successfully associated to the desired commit ⇔ *git log --oneline --graph*;
  - to delete a tag locally ⇔ *git tag -d <tag_name>* (e.g. *git tag -d v.1.3.8*);
  - to delete a tag remotely ⇔ *git push <remote> -d <tag_name>* (e.g. *git push origin -d v.1.3.8*);
  - **NB #1**: if not specified, <*commit_id*> is always linked by default to *HEAD* for all tag-commands;
  - to check details of the created tag ⇔ *git show <tag_name>*;
  - to show all repository tags created ⇔ *git tag*.

- Keep in mind the *git push* command does not automatically transfer tags to RR:
  - to transfer a single tag to RR ⇔ *git push <remote> <tag_name>* (e.g. *git commit origin v.3.1.8*);
  - to transfer all of your tags to RR ⇔ *git push <remote> --tags* [NOT RECOMMENDED];
  - note unfortunately tags are not show in GitHub network-graph, but can be displayed by clicking the dedicated button on the main page of the project (from here also formal releases can be created, giving the chance to add the corresponding binaries as well).

9. Ignore

- The purpose of the ***.gitignore*** file is to prevent specific files within the Git repository from being part of the versioning. In particular, to keep some WT files/directories out of Git versioning since the beginning of the project without removing them locally from the WT (e.g. build files generated after compilation, that usually are not versioned), add them as new lines to the *.gitignore* file present in the main page of the WT. For example:

  build/
  source/test.log

- It is possible to create additional *.gitignore* files inside project sub-folders to simplify the handling of the files path to be ignored. For example, to exclude from versioning the file *source/hal/adc.h* you can either add the entry *source/hal/adc.h* within the main *.gitignore* file of the project or add the entry *adc.h* within the additional *.gitignore* file created inside *source/hal/*.

- To remove from versioning some files or directories which were already pushed to RR during previous commits (but still keeping them locally), remember it is not enough to add them to the *.gitignore* file, since, even though you stop pushing them in future commits, they still remain stored in the RR with their previous content. Thus, they shall be removed from RR first. Follow the overall procedure hereafter:
  - be sure your LR branch is up-to-date with RR, there's nothing to commit, and the working tree is clean ⇔ *git fetch && git status*;
  - add files or directories to be excluded from versioning as new entries inside *.gitignore* file;
  - remove files or directories to be excluded from versioning from RR ⇔ *git rm --cached <file>* or *git rm -r --cached <directory>* (e.g. *git rm --cached source/xyz.c* or *git rm -r --cached bin/*);
  - **NB #1**: the previous command can be alternatively executed over the whole LR via *git rm -r cached .* [NOT RECOMMENDED, since it may create issues in case the project contains submodules];
  - stage changes (i.e. files or directories excluded from versioning according to updated *.gitignore*) ⇔ *git add --all* (or more specifically *git add <files-or-directories>*);
  - commit changes ⇔ *git commit -m "apply .gitignore updates"*;
  - push changes to RR ⇔ *git push*;
  - open RR on GitHub website and verify the specified files OR directories have been removed from the project;
  - as a check, try to modify within WT a file just excluded from versioning and then verify via *git status* that it is not reported anymore as *modified*].

- The *--cached* option specifies the removal should happen only on the staging index, leaving WT files untouched. On the other hand, executing *git rm <file>* without the *--cached* option physically deletes the files from WT and automatically stages the change (i.e. it is equivalent to delete the file manually and then execute *git add <file>*).

- Instead, if some project files or directories are no more needed and thus can be completely deleted both locally and remotely, the procedure is easier and similar to usual commits:
  - delete files or directories (i.e. move to recycle bin);
  - stage changes ⇔ *git add <files-or-directories>*;
  - commit changes ⇔ *git commit -m "files removed"*;
  - push changes to RR ⇔ *git push*.

- To create exceptions for specific *.gitignore* entries use the '!' character. For example, adding the following two lines inside the *.gitignore* file cause all *.cpp* files within the repository to be excluded from versioning except for *src/xyz.cpp*:

  **.cpp*
  !src/xyz.cpp

10. Branches

- Every commit belongs to a specific branch (Git's default one is named *master*), which contains the history of all commits related to that branch. Branches are created by reference and they are essential useful for code experimentation, testing, development within a team (since they allow concurrent and independent work on the same project without mutual interference), and for supporting multiple project-versions (in case of need for customization between different applications or clients).

- Branches can be of two types:

  - ***long-lived*** (aka *base*), related to stable and official versions of the project over time (such as *master* or *develop*);
  - ***short-lived*** (aka *topic*), related to tickets that shortly merge back to a *long-lived* branch (such as *bugfix*, *hotfix* or 'feature').

- Useful commands for branches:
  - to show all local branches of the project ⇔ *git branch*;
  - **NB #1**: among the results shown by executing *git branch*, the branch marked with the '*' character represents the one currently pointed by *HEAD*, thus the one present within WT at that time;
  - to create a new branch locally ⇔ *git branch <branch_name>*;
  - **NB #2**: before creating a new branch (e.g. *bugfix_startup_327*) be sure to checkout to the desired source branch (e.g. *develop*);
  - to push a new branch just created in LR to RR ⇔ *git push -u origin <branch_name>* (where *-u* = *--set-upstream*);
  - **NB #3**: new branches can be also created on RR first (go to *github website* > *project_name* > *view all branches* > *new branch* and select the source branch), then locally use *git fetch* to check and retrieve new branches from RR (and now by executing *git branch* it can be verified the new branch is listed locally as well);
  - **NB #4**: a new branch keeps inside also the whole previous history of its source branch (easily verifiable via *git log*).

- The *checkout* command can be used to:
  - switch *HEAD* from the current branch-commit pair to either the tip of another branch-label or another commit ID of the same branch-label ⇔ *git checkout <branch>* or *git checkout <commit_id>*;
  - update the WT with files and directories from the checked out branch or commit;
  - be allowed to commit for that branch locally and remotely (i.e. to both LR and RR);
  - **NB #5**: the commands *git branch <new_branch> && git checkout <new_branch>* can be combined into a single one (assuming <*new_branch*> does not exist yet locally) ⇔ *git checkout -b <new_branch>*.

- Whenever *HEAD* within the current branch does not point to its branch-label (i.e. tip of the branch) but to one of its previous commits, that causes the so-called ***detached HEAD*** situation. Keep in mind if you want to work restarting from a previous commit (e.g. *HEAD~*) you shall first create a dedicated branch and checkout to that, otherwise it would create a *HEAD-detached conflict*.

- Use *git branch -d <branch_name>* to locally delete a branch. Keep in mind deleting a branch actually means deleting its branch-label. Moreover, note the command fails if the branch to be deleted is the one currently pointed by *HEAD* (thus checkout to another first) or is *dangling* (i.e. it has commits which have not been merged back yet to any *long-lived* branch). To solve the latter, you can decide to either merge it back or force its deletion via *git branch -D <branch_name>*. To revert an accidental branch deletion use *git reflog* (showing LR list of recent HEAD commits) to read the commit ID of the deleted dangling branch to be restored and then use *git checkout -b <branch_name> <commit_id>*. Remember that Git periodically checks in background withing the project the presence of dangling commits (i.e. commits not linked to a branch anymore, since the latter has been deleted) and delete (aka *garbage collect*) them automatically. Finally, to delete the branch also remotely on RR use *git push origin -d <branch_name>* (where *-d* = *--delete*).

11. Merging

- Merging allows to combine the work of independent branches (usually from a *short-lived* branch into a *long-lived* one). There exist four types of merging:
  - *fast-forward*;
  - *merge commit*;
  - *squash merge*;
  - *rebase merge*.
- ***Fast-forward*** (FF) moves directly the source branch-label to the tip of the destination branch without causing conflicts (therefore, it is possible only if no overlapping commits or unstaged changes have been added to the source branch in the meanwhile). In this case, after merging both branches contain exactly the same commits, thus the destination branch inherits all the source branch commits (even in case later the destination branch gets deleted). With FF merging the resulting *commit history* (aka *commit graph*) is linear, namely no commits have multiple parents (verifiable visually via *git log --oneline --graph*). Hereafter the procedure to perform an FF merging:
  - commit and push the final changes on the destination branch (aka *active branch*);
  - switch to the source one (aka *passive branch*) and pull latest updates (this is always a good practise, even if nothing is expected to have changed) ⇔ *git checkout <source_branch> && git pull*;
  - checkout back to destination branch (i.e. where merging process is going to be applied);
  - merge source branch into destination branch ⇔ *git merge <source_branch>*;
  - **NB #1**: whenever using the command *merge*, Git specifies the type of merging is trying to execute (e.g. by printing *Fast-forward* on shell);
  - **NB #2**: FF is always the default type of merging attempted by Git at first;
  - verify your destination branch is now marked as *ahead of origin by 1 commit* and push merge-changes to RR ⇔ *git status && git push*;
  - check the source branch commits have been added to the destination branch, *HEAD* points now to both branches, and commit graph is linear ⇔ *git log --oneline --graph*;
  - if not needed anymore, delete the merged source branch both locally and remotely (see Section 10.) - this is not mandatory, but recommended to save space on both disk and GitHub and keep project versioning cleaner;
  - **NB #3**: a merged source branch is never deleted automatically by Git, it has to be done explicitly by the user via command line or GitHub website.

12. Git Log

...

13. Git GUI

...

References

[1] B. Sklar, P. K. Ray, *Digital Communications*, Chap. 4-9, Pearson Education, 2012.