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 21.);
  - **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 22.).

- 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>*.

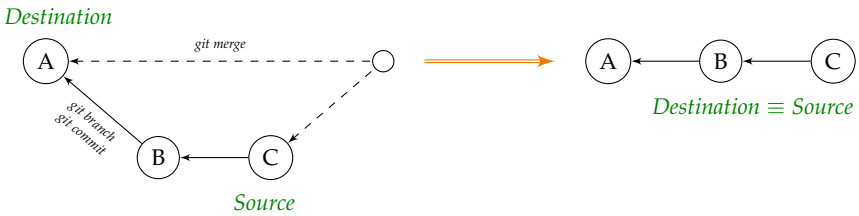
- Whenever *HEAD* within the current branch does not point to its branch-label (i.e. tip of the branch) but to one of its previous commits, that causes the so-called ***detached HEAD*** situation. Keep in mind if you want to work restarting from a previous commit (e.g. *HEAD~*) you shall first create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

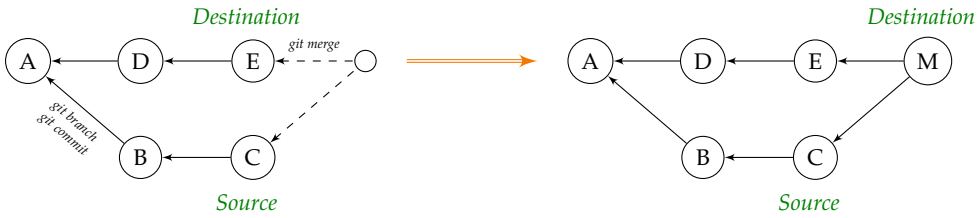
- Commit graph example for FF:



- ***Merge-commit*** (MC) combines the commits at the tips of the two branches to be merged and saves the result into a new commit. A *merge-commit* has always multiple parents, since represents a combination of both branches (and so generating a non-linear commit graph, verifiable via *git log --oneline --graph*). Moreover, MC may cause conflicts if the two branches end up modifying the very same files of the project. The commands to perform an MC merge are the very same already mentioned for FF, with the only difference (assuming, for the moment, that no conflicts occur) that now a new commit is created as a result of the merging process, generating also a default commit message (modifiable if desired) explaining the specific merge details. Finally, remember it is possible to create an MC even

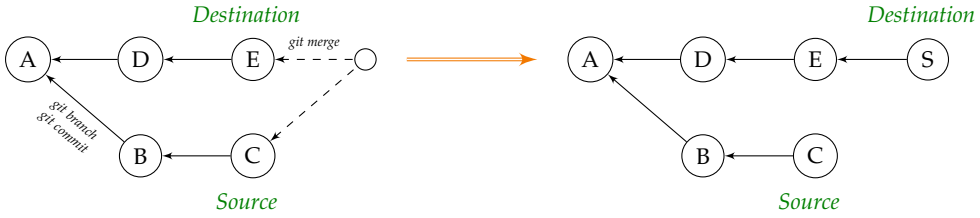
for simpler FF cases by using *git merge --no-ff <source_branch>*. This way, the *merge-commit* will keep track also of the source branch existence. This solution generates exactly the same project files as FF would, the only difference regards the project history information. In particular, MC allows a more accurate history tracking (since source branches are not concealed), whereas FF can help to keep the history tracking cleaner. So usually it is a software team decision which policy to adopt.

- Commit graph example for MC (where commit 'M' contains info from 'B' and 'C' as well and these are included within destination branch history):



- ***Squash-merge*** (SM) merges the source branch tip into the destination branch one (thus, conflict may arise in this case too). The content of the WT after SM is exactly the same as for MC, the only difference regards the final destination branch commit history. In fact, in this case the destination branch discards the source branch history after merging and considers the new commit as a single-parent one. This can help keeping the commit history cleaner, but also means losing detailed tracking information about the destination branch development. Note SM is a form of *history-rewriting*. Hereafter the procedure for SM:
  - follow the same preliminary steps already described for FF;
  - checkout to destination branch ⇔ *git checkout <destination_branch>*;
  - apply SM ⇔ *git merge --squash <source_branch>*;
  - commit (deciding also whether to accept or modify the default SM message prompted).

- Commit graph example for SM (where commit 'S' contains info from 'B' and 'C' as well, but these are not included within destination branch history):



- ***Rebase-merge*** (RM) is treated in Section 15..

- In case of long-running topic-branches, it is always a good practice to periodically merge back to the related base-branches in order not to stray too much from development progression of the latter and avoid a potentially large number of conflicts and operation mismatches.

- Practical case where base-branch (e.g. *develop* acting as source branch) needs to be merged into a completed topic-branch (e.g. *feature_423_bootloader* acting as destination branch) to solve conflicts locally in advance in sight of PR opening:
  - work on topic-branch, then commit and push final changes;
  - checkout to base-branch and pull latest state from RR ⇔ *git checkout <base_branch> && git pull*;
  - move back to topic-branch ⇔ *git checkout <topic_branch>*;

- apply conditional merge (i.e. forcing MC without automatic commit) ⇔ *git merge --no-commit --no-ff <base_branch>*;
- solve conflicts, if any (see Section 12.);
- update submodules (only in case any reference has been changed within base-branch) ⇔ *git submodule update --recursive*;
- **NB #1**: avoid executing the previous command for submodules just updated within the topic-branch (if any, or alternatively remember to execute an additional git pull on these submodules), since topic-branch reference is supposed to be the most recent (otherwise it would mean to revert part of the topic-branch work);
- open Git GUI (and verify that merged files have been already staged but not committed yet), then stage, commit and push all merge changes;
- **NB #2**: to revert the whole merge process (in case of any error) ⇔ *git merge --abort*.

12. Conflicts

- Conflicts arise when trying to merge branches that have modified the same sections (aka *hunks*) of the same files. So, in this case the user is asked to make a decision on what to keep exactly. Note that no conflicts arise if changes have been made on separate files or even on separate hunks of the same files, since Git usually manages to resolve them autonomously.

- Procedure to resolve conflicts:
  - after merging source branch into destination one via *git merge* command, if conflicts have been detected Git shows a message reporting all the project files inside which there are conflicts that cannot be automatically resolved (e.g. *CONFLICT: Merge conflict in file xyz.c*), and modifies the content of these files in the WT to highlight the conflicting sections (see below);
  - by checking now the current state of LR branch via *git status* Git should answer *You have unmerged paths* (giving you also the chance to abort the merge attempt via *git merge --abort*, which restores the WT state to the latest LR destination branch commit);
  - to manually resolve the conflicts open corresponding files one by one with your editor (e.g. *Visual Studio Code*) and solve them by deciding which branch version to keep for each unmerged section, then save and close;
  - stage and commit the fixed files (e.g. via Git GUI);
  - by checking now the current state of LR branch via *git status* Git should answer that WT is clean and LR ahead of origin by 2 commits;
  - push changes to RR;
  - check via *git log --oneline --graph* and in GitHub's *Network graph* the merge has been executed successfully;
  - optionally delete merged source branch (see Section 10. and 11.);

- Git marks files where conflicts occurred while merging as follows:

  ```
  <<<<<<<
  ... section #1 ...   → destination branch / HEAD
  =======              → conflict divider
  ... section #2 ...   → source branch
  >>>>>>>
  ```

- To check which project files contain conflicts to be solved after a merge use *git diff --check* (if none or after solving all of them, the command simply returns no message). Plus, they are labeled as *unmerged* as a result of *git status*. Another way is to simply search for the string *<<<<<<< HEAD* identifying

any pending conflict throughout the project. Note conflicts can arise also due to *git stash pop | apply* commands (see Section 19.); in this case look for the string >>>>>>> *Stashed changes* instead.

13. Tracking-Branches

- ***Tracking-branches*** (TB) are local branches representing remote branches. They are named <*remote*>/<*branch*> (e.g. *origin/master*, where *origin* is a shortcut for remote repository URL/SSH address). Just after cloning a repository, the remote *master* branch on GitHub and the local TB *origin/master* are synchronized (i.e. containing the very same commits). However, remember TBs shall be then updated explicitly by the user to keep the synchronization. For instance, if after cloning the repository another user pushes a new commit on the same branch, RR becomes 1 commit ahead of your LR (aka *decoupled*), but you cannot know this locally (e.g. via *git status*) until you execute a *fetch* or *pull* command, since your *origin/master* TB still points to the previous commit. The opposite example is when you create a new commit locally which has not been pushed yet to RR, where executing *git status* returns *Your branch is ahead of origin/master by 1 commit*.

- By default the command *git branch* shows only the LR branches. Use *git branch --all* to show TBs as well, which are all unique branches stored within *.git/refs/remotes/origin/* except for *HEAD* (by default *remotes/origin/HEAD → origin/master*). The latter is actually a symbolic reference specifying the TB related to the currently checked out branch and allowing to type only <*remote*> instead of the whole <*remote*>/<*branch*> in Git commands (e.g. *origin* instead of *origin/master*).

- Useful commands related to TBs:
  - to show commit-history of a remote branch (again, tracked only indirectly via the corresponding tracking-branch) ⇔ *git log origin/<branch> [--oneline]* (equivalent to simply *git log origin [--oneline]* if <*branch*>, for instance *master*, is set as default remote TB);
  - to change locally the default remote tracking-branch (e.g. from *master* to *develop*, since that is usually the one where the bulk of the work is done by the team) ⇔ *git remote set-head <remote> <branch>* (e.g. *git remote set-head origin develop*, so from now on the command *git log origin [--oneline]* will refer to *develop* commits and *git branch --all* will show *remotes/origin/HEAD → origin/develop*).

- The initial default TB (e.g. corresponding to the default branch when cloning the project) can be changed for all users through *github website* > *settings* > *general* > *default branch*. Note this is always set by default to *master* for any new repository.

- As already mentioned above, the *git status* and *git log --oneline --graph --all* commands include info about TBs status and inform you if the local branch and the corresponding TB are out of sync or not (e.g. *Your branch is up-to-date with origin/master*).

14. Fetch, Pull & Push

- Most Git commands interacts with LR only, but there are also four main *network commands* in charge of communicating with the RR:
  - ***clone*** ⇔ to copy a remote repository locally;
  - ***fetch*** ⇔ to retrieve new objects and references from RR to LR;
  - ***pull*** ⇔ to fetch and merge commits locally;
  - ***push*** ⇔ to add new object and reference from LR to RR.

- The *git fetch* command allows to update local *tracking-branches* with info from RR without merging the changes immediately into WT (so WT is never modified as a result of a *fetch* command), thus does not affect you LR labels and just makes the user aware of the latest RR state (see Section 13.). In case LR is already up-to-date with RR, no info is displayed on shell after executing the command. Plus, executing *git status* before and after *git fetch* command allows to understand if and how LR and RR are out of sync (giving also helpful suggestions, e.g. if branch can be fast-forwarded or not in case RR is ahead by some commits). This command is also essential to get aware of and retrieve info about new branches created remotely from scratch by any user or pushed remotely by other users.

- The *git pull* command combines *git fetch* and *git merge FETCH_HEAD*. If any object is fetched, the tracking-branch is updated and the changes merged automatically into WT (so it may cause conflicts). Hereafter some useful merge options for the *git pull* command are listed:
  - *--ff* ⇔ to fast-forward if possible, otherwise *merge commit* (default if no option specified);
  - *--no-ff* ⇔ to always *merge commit* (even when fast-forward would be possible);
  - *--ff-only* ⇔ to abort if fast-forward is not possible (i.e. avoiding merge-commit);
  - *--rebase [--preserve-merges]* ⇔ see Section 15..

- Remember to always execute *git status* after each *git pull*, in order to check the refreshed state of the repo. This is especially useful in case the project contains submodules to verify if any of them requires to be updated (in case, run *git submodule update --recursive*, see Section 20.).

- There can be multiple situations when executing *git pull*:
  - LR and RR in sync, without WT unstanged/uncommitted/unpushed changes ⇔ no effect;
  - LR and RR in sync, with WT unstanged/uncommitted/unpushed changes ⇔ no effect (WT changes preserved);
  - LR ahead of RR, without WT unstanged/uncommitted/unpushed changes ⇔ no effect;
  - LR ahead of RR, with WT unstanged/uncommitted/unpushed changes ⇔ no effect (WT changes preserved);
  - RR ahead of LR, without WT unstanged/uncommitted/unpushed changes ⇔ fast-forward (no conflicts);
  - RR ahead of LR, with WT unstanged/uncommitted changes not conflicting with latest RR commits ⇔ fast-forward pull update (no conflicts and WT changes preserved);
  - RR ahead of LR, with WT unstanged/uncommitted changes conflicting with latest RR commits ⇔ pull aborted [see *PULL MESSAGE #1*];
  - RR ahead of LR, with LR unpushed commits having no conflicts with latest RR commits ⇔ merge-commit (no conflicts) [see *PULL MESSAGE #2*] - so in this case a new local commit is created representing the automatic combination of the latest RR commit and the unpushed LR commits (plus, a message is prompted asking the user to provide a comment for the merge), and by checking now the local state via *git status* Git informs you that LR is ahead of *origin/master* by 2 commits (i.e. the *pull-merge* is just local for the moment);
  - RR ahead of LR, with LR unpushed commits having conflicts with latest RR commits ⇔ merge conflict to solve [see *PULL MESSAGE #3*] - so in this case Git automatically marks the issues on the conflicting files as already reported in Section 11., waiting for the user to manually resolve them (then add, commit and push) or abort the *pull-merge* (again via *git merge --abort*);
  - **NB #1**: always try to avoid the last situations listed above (namely generating *pull-merge* conflicts) and perform *git pull* with WT clean and no LR unpushed commits (thus create a new topic branch whenever a new bugfix or feature has to be added and never work directly on the base branch).

- **PULL MESSAGE #1** - Aborting:

  Updating b7b6353..bbb97e7
  error: Your local changes to the following files would be overwritten by merge:

xyz.c
Please, commit your changes or stash them before you can merge.
Aborting

- **PULL MESSAGE #2** - Merge commit:

  remote: Enumerating objects: 5, done.
  remote: Counting objects: 100% (5/5), done.
  remote: Compressing objects: 100% (1/1), done.
  remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
  Unpacking objects: 100% (3/3), done.
  From https://github.com/AlectoSaeglopur/proj_16
  4f83692..badcf67 master → origin/master
  Merge made by the 'recursive' strategy.
  xyz.c | 1 +
  1 file changed, 1 insertion(+)

- **PULL MESSAGE #3** - Conflicts:

  Auto-merging abc.txt
  CONFLICT (content): Merge conflict in xyz.c
  Automatic merge failed; fix conflicts and then commit the result.

- The *git push* command allows to forward commits from LR to RR for the current branch. Its basic syntax is *git push [-u | --set-upstream] [<repository>] [<branch>]* (see Section 6. for more details). Keep in mind it is always good practice to fetch/pull before pushing in order to be aware of the latest RR state for that branch. In fact, trying to push when RR is ahead of LR by some commits simply fails (and that's why the work on a single topic branch should be done by one user only at a time for simplicity's sake) returning the following error:

  ! [rejected] master → master (fetch first)
  'error: failed to push some refs to origin'
  ...

15. Rebase & History Rewriting

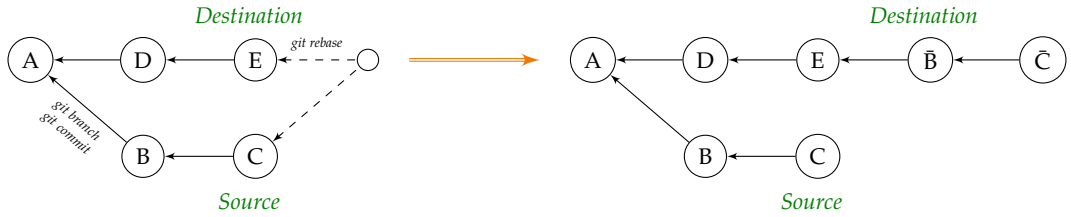
- ***Rebasing*** means moving commits to a new parent, namely unique commits of a branch (source) are reapplied to the tip of another branch (destination). Since ancestor chain has changed, each of the reapplied commits has now a different commit ID. An advantage is that after rebasing the merging process from source branch to destination branch can be fast-forward, thus easier. Note that since Git actually works by estimating and versioning only the differences between consecutive commits (known as *diff* or *patch*), rebasing means reapplying the differences to the new parent commits (process called *commits reapplying*). Of course, since reapplying commits is a form of merging, this is susceptible to conflicts (e.g. if commits 'B' and 'D' modified the same sections of the same files, see the example below).

- Rebasing has both pros and cons:

  + allows to incorporate changes from parent branch more easily;
  + avoids potentially "unnecessary" commits (thus history is cleaner);
  – solved conflicts are harder to track;
  – severe issues if commits are shared with other users;

  - does not preserve the actual commits sequence (thus history is less accurate).

- Commit graph example for rebasing (aka RM) (where commits 'B̄' and 'C̄' add respectively the $diff_{AB}$ and $diff_{BC}$ changes from source to destination branch and are treated as completely separate commits from the original ones in terms of history):



- There are two types of rebasing:
  - ***regular rebase***, which is the one discussed so far;
  - ***interactive rebase***, which allows to rebase (by adding the *-i* option) a series of commits belonging to a specific branch - here commits can be edited, stopped, dropped/deleted, squashed (i.e. combining a specific commit with the previous one into a new single commit, combining also the related messages), fixed-up (like squash, but discarding the comment of the newer commit), reordered, etc.

- Useful commands for *regular rebasing*:
  - to rebase source branch to destination branch (could be even a remote one) ⇔ *git checkout <source_branch> && git rebase <destination_branch>* (e.g. *git checkout bugfix_358 && git rebase develop*);
  - in case of conflict you can either solve them manually (see Section 12. and finally complete the rebase via *git rebase --continue*, instead of the usual *git commit ...*) or abort it (via *git rebase --abort*).

- Useful commands for *interactive rebasing*:
  - to list current branch commits (starting from a specified one) in an editor in order to be modified (e.g. you might want to combine the two most recent commits 'B' and 'C' into a new single one 'Z') ⇔ *git rebase -i <commit_id_after_which_start>*;
  - **NB #1**: for more details refer to Coursera's lesson #16.

- Git gives also the possibility to modify (or simply slightly adjust) the most recent commit (in terms of files, comment, etc) through the ***--amend*** option. Note this also modifies the commit ID associated to that commit as well, thus rewrites history. Use the procedure hereafter to amend the most recent commit:
  - work on the WT by adding, modifying and/or removing files;
  - stage changes ⇔ *git add <files-or-directories>*;
  - amend commit ⇔ *git commit --amend -m "new_comment"* or *git commit --amend --no-edit* (with the *--no-edit* option the original message is preserved).

- **CAUTION**: never rewrite history that is shared with other users!

16. Pull Request

- ***Pull request*** (PR) is a feature provided by most Git hosting services (such as GitHub or BitBucket). Its ultimate goal is to improve team communication and development reliability during a source-to-destination-branch (usually a topic-to-base-branch) merging process. In particular, team members can

send notifications to each others, leave comments or feedbacks, review code and approval or decline its content for merging. Note since pull-request is a feature provided basically by the hosting services (thus not a Git built-in functionality), it is not possible to handle that directly via Git bash.

- Pull requests can be actually opened at any time of a branch life:
  - at the very beginning to allow branch creation;
  - whenever comments or feedbacks on the current progress of the branch is needed;
  - when branch is ready to be reviewed and merged.

- There exist two pull request cases depending on the repositories configuration:
  - ***single-repository*** (SR), where a source branch merges back to a destination branch of the same repository (i.e. the usual case where only one RR is involved);
  - ***multi repository*** (MR), where a *fork-repository* merges back to the *upstream-repository* (thus two RRs are involved).

- Procedure for SR pull request on GitHub:
  - create (or fetch) and checkout to new topic branch (e.g. *bugfix_123*);
  - work on WT, then stage/commit/push final changes;
  - open project on GitHub website and go to *pull requests* > *new pull request*;
  - here select the merging direction for branches (e.g. *develop* ← *bugfix_123*) and click *create pull request* (plus, you can also select the desired merging strategy, e.g. *merge-commit*, *squash*, etc);
  - then on the right side of the webpage you can add other team members as reviewers;
  - in case there are conflicts, you can solve them directly on GitHub website on the same *pull requests* page and then click *merge commit*;
  - now wait for other members to review and approve;
  - on the other hand, the reviewer has to open the same *pull requests* page on GitHub, click on *add your review*, and decide whether to comment, approve or request changes and click *submit*;
  - once approved (depending on the team PR strategy), any team member can click *merge pull request* > *confirm merge* to complete the merging procedure (PR is then automatically closed);
  - finally all users should checkout to the base branch and pull locally the latest changes just merged.

- MR pull requests can be handled either via GitHub website or Git bash (see Section 17.).

- Note that opening a PR without checking for conflicts in advance may cause it to be opened and approved twice (depending on the team PR strategy). In fact, after approving the 1st PR the actual merging process is executed remotely and, if conflicts are detected, the user is asked to solve them and open another PR to get the resolution of conflicts approved too. Thus, it is always a good practice to check and solve conflicts locally and in advance before opening the PR. Hereafter the preliminary procedure to handle topic-to-base-branch merge conflicts is reported, so that the PR on GitHub will surely result conflict-free (see also Section 11. for further details):
  - create (or fetch) and checkout to new topic branch (e.g. *bugfix_123*);
  - work on WT, then stage/commit/push final changes;
  - checkout to base branch (e.g. *develop*) and pull latest changes;
  - checkout back to topic branch;
  - merge base branch into topic branch and resolve conflicts locally (if any);
  - stage/commit/push merge changes;
  - open topic-to-base-branch conflict-free PR on GitHub (as described above).

17. Forking

- *Forking* means copying an existing remote repository (usually referred to as the *source of truth*) to your own GitHub account and ending up to create a second parallel remote repository. Forking workflows is common for open-source projects, so that topic-branches do not need to be shared (thus, for example, rebasing is way easier). However, you shall always pay attention to manually keep the upstream and fork repositories in sync to avoid issues, since this is not handled automatically by Git.

- A fork can be used for several reasons:
  - make experiments with or learn from the upstream repository without affecting its official workflow;
  - work independently on the same project and issue PRs only for merging back to the upstream repository;
  - switch to a different source of truth for the project.

- Useful tips for forking:
  - to create a new fork, open the project page on GitHub website and click *fork this repo*;
  - to synchronize a fork with the latest upstream repo commits, open the project page on GitHub website and click *sync fork* (this creates a merge-commit on the fork RR, so then remember to update you LR as well via *git pull*).

18. Workflows

- The term ***Gitflow*** refers to the standard/official versioning workflow that allows safe and continuous releases of the project. For example, the project versioning should include:
  - two main base-branches, such as *master* (whose tagged commits contain firmware versions to release to customers) and *develop* (for main code development);
  - any number of topic-branches for each new task to fulfill (e.g. *hotfix_123*, *bugfix_345*, *feature_456*, *release_002*, etc);
  - for more details see video *19_git_workflows.mp4* from minute 3:30 on.

- Some Gitflow recommendations:
  - perform only merge-commits on *master* branch (and more in general on every base branch);
  - commit to *master* branch only from *release* or *hotfix* branches;
  - after committing to *master* branch, merge the same changes into *develop* branch too (in order not to lose track of the updates/patches applied during the release/hotfix phase);
  - no direct work shall be done on *master* (and more in general on any base branch).

19. Stash

- *Stash* commands allow to take WT uncommitted changes (both staged and unstaged), store them away for later use, and restore them to WT later:
  - to stash uncommitted changes ⇔ *git stash [push]*;
  - **NB #1**: after pushing a new stash entry, execute *git status* and check WT is now clean;
  - to restore latest stashed changes (liked to the stash entry labeled as *stash@{0}*) to WT and remove them from the stashing area ⇔ *git stash pop*;

  - to restore latest stashed changes to WT but keep them in the stashing area (useful if you need to apply the same stashed changes to multiple branches) ⇔ *git stash apply*;
  - to restore older changes use include the desired stash entry as additional argument ⇔ *git stash pop | apply stash@{<num>}* (.e.g. *git stash apply stash@{3}*).

- Note that by default the *git stash [push]* command stores:
  - staged changes (i.e. the changes that have been added to your index);
  - unstaged changes (i.e. changes made to files that are currently tracked by Git);

  but does NOT store:
  - new files adde within WT that have not been staged yet (aka *untracked*, i.e. not part of the versioning yet) ⇔ add the *-u | --include-untracked* option to include untracked files as well, namely *git stash -u*;
  - ignored files (i.e. part of the *.gitignore* file) ⇔ add the *-a | --all* option to include both untracked files and changes to ignored files as well, namely *git stash -a*.

- The user is not limited to a single stash at a time, but can push multiple times to create multiple stash entries in sequence. By default, stash entries are identified simply with a *WIP* (aka *Work In Progress*) label linked to the branch-commit pair the stash entry was generated from. The *git stash list* command can be used to list all stash entries currently stored within the stashing area. However, after a while it can be difficult to remember what each stash entry contains, because each of them is identified with a string like *stash@{0}: WIP on bugfix_123: 2e16987* or *stash@{4}: WIP on develop: f955ab0*. To provide a clearer and more detailed tracking, it is recommended to include a description to each stash entry through the *save "<message>"* option, namely *git stash save "<message>"* (e.g. *git stash save "stashed changes on xyz.c" -u*).

- To retrieve info about a specific stash, use the *git stash show stash@{<num>}* command and add the *-p* option to display also all differences within modified files (e.g. *git stash show -p stash@{1}*).

- Use *git stash drop stash@{<num>}* to delete a specific stash entry or *git stash clear* to delete all of them at once.

- Of course restoring stashed changes via *pop* or *apply* commands can generate conflicts if in the meanwhile the same sections of the same files have been modified. In this case, just resolve manually the conflicts on each file and then stage the changes as usual (see Section 12.).

- Remember stash entries are actually encoded locally as commit objects. The special reference at *.git/refs/stash* points to the most recent stash, and previous stashes are referenced by the corresponding stash *ref*'s reflogs. This is why you refer to each stash entry as *stash@{<num>}*, since you're actually referring to the n-th reflog entry of the stash *ref*.

- Note stash commands do not apply by default on submodules too. Hereafter some useful commands to stash changes on submodules in parallel (see Section 20. for more details):
  - to stash all changes within both parent project and submodules ⇔ *git stash save "<message>" -u && git submodule foreach --recursive git stash save "<message>" -u*;
  - to reapply all changes stashed in the previous step to both parent project and submodules ⇔ *git stash apply && git submodule foreach --recursive git stash apply* → **CAUTION**: working only if they all have the same stash index;
  - clear all stash entries for both parent project and submodules ⇔ *git stash clear && git submodule foreach --recursive git stash clear*.

20. Submodules

- ...

21. Git Log

...

22. Git GUI

...

References

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