IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

VCS practical work – Let's practice Git

Ingénierie du développement logiciel – DLR-IDL

Correction

Objectives

At the end of the activity, you should be capable of:

- mastering Git basics,
- using Git for most of common use cases,
- learning to learn Git incantations.

Exercises

Preliminary note #1: In the following exercises, the different git commands won't be detailed nor explained. The exercises are a way to discover those commands. Therefore, at the beginning of each exercise, an hint will be given (usually one or several target commands or options). To find any information about a git command, in addition to the Git reference book[1], remember the command `git --help` and to one of the most important UNIX family command: `man`. What does `man` do? `man man`! ☺

A public Git repository is available here:
`https://gitlab.imt-atlantique.fr/jcbach/idl-sample-project` [2].
It will serve as a reference if you need a starting point during this practical work. It contains plain text files which can be easily edited.

Preliminary note #2: This practical work is probably too long to be done in only one session. It is normal and it is OK if you do not finish it. However, you should all be able to finish the first section (*Basics: starting with Git*) during the session.

[1] Pro Git: `https://git-scm.com/book/en/v2`
[2] The former repository is still there: `https://redmine.telecom-bretagne.eu/git/idl-sample-project`

Preliminary note #3: In these exercises, we only consider Git. All the exercises have to be solved with CLI[3]. In a real production process, maybe you will use a terminal, a plugin in your IDE, a web browser, etc. However all those tools might hide the logic of Git, therefore we encourage you to use the terminal to begin with Git.

Preliminary note #4: In this practical work, a personal Git repository is needed. Use any hosting platform you prefer. Note that IMT Atlantique has two platforms[4] allowing students to create projects with a Git or a Subversion repository. It can be a good option for your training. Although both of them are working and are sufficient for our lab session, please prefer the Gitlab one.
How to create a Git repository on the Gitlab platform:

- open `https://gitlab-df.imt-atlantique.fr` in a browser
- log in with your usual login (+ 2FA if already configured)
- click on *New project* (on the upper left)
- select the type of project you want to create (e.g. *Blank project*)
- fill the form: project name, group/user used (only your id is available by default), project slug (project identifier), visibility, other options
- create it by clicking on *Create project*
- the repository URL will be:
  https://gitlab-df.imt-atlantique.fr/group_or_user_id/your_project_slug.git

How to create a Git repository on the Redmine platform:

- open `https://redmine-df.telecom-bretagne.eu` in a browser
- log in (upper right) with your usual login
- click on *projects* (on the upper left)
- click on *New project*
- fill the form (name and id of the project, select Git as your SCM)
- create it by clicking on *create*
- the repository URL will be:
  `https://redmine-df.telecom-bretagne.eu/git/name_of_your_project`

Note: In the correction, `$>` represents the prompt in command line interface, meaning that the text following that symbol is a command typed in the shell.

[3] Command Line Interface

[4] A modern one, Gitlab (`https://gitlab.imt-atlantique.fr/https://gitlab-df.imt-atlantique.fr`, and a legacy one, Redmine(`https://redmine-df.telecom-bretagne.eu`).

Basics: starting with Git

In this first section, you will learn the very basics of Git. They are essential to begin to work with Git. Another objective of this section is to help you in setting up your working environment for this practical work.

Exercise 1 (*Retrieving a Git repository for the first time*)

The objective of this exercise is to show you the first step when having to retrieve an existing Git repository.
Hint: clone, status

- using a terminal, create the directory where you want to work, then move to the directory
- get the URL of the example remote repository (c.f. first preliminary note)
- retrieve it with the adequate Git command (this is the moment to use the *hint*. . .)
- check the newly created directory
- check the state of the project

Creation of the directory, whith parent directory if needed:

```
$> mkdir -p TAF-DL/IDL/

$> cd TAF-DL/IDL

$> git clone https://gitlab.imt-atlantique.fr/jcbach/idl-sample-project.git

$> cd idl-sample-project

$> git status
```

Exercise 2 (*Change URL of remote repository*)

The objective of this exercise is to learn how to change the remote repository. . . and to set up your own remote repository.
Hint: remote

- create a new repository (on the Redmine platform, within a project)
- clone an existing repository (you can also use the repository you have cloned in the previous exercise)
- go to your newly created directory

- check the existing remotes
- change its remote repository in order to set the new repository URL as the origin (by using a git command, of course)
- check the remote has been changed

Create your own repository on redmine-df
```
$> git clone https://gitlab.imt-atlantique.fr/jcbach/idl-sample-project.git
$> cd idl-sample-project
```
Check existing remotes:
```
$> git remote -v
$> git remote set-url origin new_url
```
Check new URL remotes: they should have changed
```
$> git remote -v
```
Another solution would have been to directly edit the configuration file:
```
$> vim .git/config
```
Find the `[remote "origin"]` section and replace the old host url (in url field) by the new host one

Exercise 3 (*Setting up your environment*)

The objective of this exercise is to learn basic setup.
Hint: config, user.email, user.name, --global

- move to your working directory where you cloned and set up your repository
- configure your name (it will appear in the history of this repository)
- configure your email (it will also appear next to your name in the history)
- how would you modify the previous commands to configure globally your Git repositories?

```
$> cd ~/TAF-DLR/IDL/idl-sample-project
$> git config user.name "<Name that should appear in logs and emails>"
$> git config user.email <email@yourdomain.example.com>
$> git config --global user.name "your name"
$> git config --global user.email <email@yourdomain.example.com>
#Optional
$> git config --global color.diff auto
$> git config --global color.status auto
$> git config --global color.branch auto
```

Note: Typing your login and password for each `push` can be boring, therefore is possible to complete your Git credentials configuration: see `https://git-scm.com/docs/gitcredentials`.

Exercise 4 (*Making a change*)

The objective of this exercise is to learn how to make a change in a Git environment. It is also the opportunity to observe the state of your Git repository along the changes.

Hint: status, add, commit, log

- in your working directory, check the current state of your repository
- choose a file and make a change
- check the state of your repository
- specify this change will be part of your next set of related changes
- validate this set of related changes (which is actually only one change), do not forget to add a relevant comment to explain your change
- check the state of your repository
- choose a file and make a change
- check the state of your repository
- specify this change will be part of your next set of related changes
- make another changes in the file you chose, in other files, etc.
- check the state of your repository
- specify this change will be part of your next set of related changes
- validate this set of related changes (is it useful to remind you that you'll have to add a relevant comment?)
- check the state of your repository
- look at the history of the repository

```
$> git status
$> echo "//an important footer" >> aFile.txt
$> git status
$> git add aFile.txt
$> git commit -m "added an footer to aFile"
$> git status
$> echo "//yet another important footer" >> anotherFile.txt
$> git status
$> git add anotherFile.txt
$> echo "//Copyleft 2019" >> anotherFile.txt
$> echo "//yet another important footer" >> yetAnotherFile.txt
$> echo "//Copyleft 2019" >> anotherFile.txt
$> git add anotherFile.txt
$> git add yestAnotherFile.txt
$> git commit -m "yet another 20th century task:  added another footer"
$> git status
$> git log
```

Note: `git add` allows the developer to add changes to the commit. By using the `-p` switch, it is possible to add a part of the changes in a file to a commit. Now, feel the potential of this command modifier: a new world of commits is opening. ☺

Note: From now, we will use the word *commit* for *set of related changes*.

Exercise 5 (*Sharing a change*)

The objective of this exercise is to begin with the collaborative part of Git: this tool is useful for a lone developer working locally, but it is much more useful in a context of collaborative work. We assume you just have completed the previous exercise.

Hint: pull, push

- check the state of your repository
- before sharing your work, check that there was no new change on the remote repository (in the current use case, it should not, but it is a good practice that should become automatic)
- check the state of your repository
- send your changes on the remote repository
- check the state of your repository

```
$> git status
$> git pull
$> git status
$> git push
$> git status
```

At this point of the practical work, you should understand the basics of Git, allowing you to handle a simple and linear process, which is the most common Git use case. It is absolutely mandatory to master this part before going further or contributing to a project using Git as its version control system.

Actual Git use cases

In this section, each exercise represents a typical situation developers can encounter when developing using a VCS such as Git. They will guide you in exploring how Git works.

Exercise 6 (*Differences*)

The objective of this exercise is to show the different "areas" (local directory, staging area and repository).

Hint: diff

- add a file and send it to the local repository
- change the content of the file in the workspace
- get the list of changes
- add the file to the staging area
- change the content of the file
- get the list of changes in workspace
- get the list of changes in staging area

```
$> cd idl-sample-project
$> touch myfile.txt
$> git add myfile.txt
$> git commit myfile.txt -m "a new file to the repository"
$> echo "a change" >> myfile.txt
$> git status .
$> git diff
$> git diff myfile.txt
$> git add myfile.txt
$> echo "another change" >> myfile.txt
$> git diff myfile.txt
$> git diff --staged
```

Exercise 7 (*Merge and fast forward*)

The objective of this exercise is to use branches and to merge them. You can consider a branch as a local "checkpoint" (a reference to a specific commit). Once a developer has written new code in her branch, she can merge it with other branches. Usually, there is at least a branch named master.

Hint: checkout, branch, merge

- First:
  - go to branch *master* or ensure you are already in master branch
  - create branch *newfeature*
  - make changes on branch *newfeature*
  - merge from *newfeature* into *master*

- Second:
  - go to branch *master*
  - create branch *anothernewfeature*
  - make changes on branch *anothernewfeature*
  - make changes on branch *master*
  - merge from *anothernewfeature* into *master*

Note: do not copy/paste the commands, type them. There can be copy problems from pdf to text, especially with some characters (quotes. . .).

```
$> cd idl-sample-project
```

- First

  ```
  $> git checkout master
  $> git branch newfeature
  $> git checkout newfeature
  ```

  The two previous commands can be done in one single command:

  ```
  $> git checkout -b newfeature
  $> echo "a change in B" >> afile.txt
  $> git add afile.txt
  $> git commit -m "add a super feature"
  $> git checkout master
  $> git merge newfeature
  ```

- Second

  ```
  $> git checkout master
  $> git checkout -b anothernewfeature
  $> echo "a change in B" >> afile.txt
  $> git add afile.txt
  $> git commit -m "add a super feature"
  $> git checkout master
  $> echo "another change in A" >> afile.txt
  $> git add afile.txt
  $> git commit -m "add another super feature"
  $> git merge anothernewfeature
  ```

Exercise 8 (*Merge and conflicts*)

The objective of this exercise is to learn to merge files and to handle conflicts (in a previous exercise, the case was to simple to create conflicts).

Hint: no new Git command, but do not forget to use add to construct your commit (and thus to mark a conflict as solved

- create a local branch *feature1*
- create a local branch *feature2*
- create the same file in both branches, with a difference in the same lines
- merge from *feature2* into *feature1* and solve the conflict
- merge from *feature1* into *feature2* and…

```
$> cd idl-sample-project
$> git branch feature1
$> git checkout -b feature2
$> echo "from branch feature2" >> aFile.txt
$> git add aFile.txt
$> git commit -m "minor change in aFile.txt"
$> git checkout feature1
$> echo "from branch feature1" >> aFile.txt
$> git add aFile.txt
$> git commit -m "minor change in aFile.txt"
$> git merge feature2
```

There is a conflict that has to be solved: first, edit the file and choose the correct change, then…

```
$> git add aFile.txt
$> git commit -m "fix merge conflict in aFile.txt"
$> git checkout feature2
$> git merge feature1
```

(There is no conflict here)

Exercise 9 (*pull = fetch + merge*)

The objective of this exercise is to show what is a *pull* command.

- create a branch *feature* (local and remote)
- create 2 working copies from the same repository (we want to simulate two people working on the same branch)
- make changes in a working copy and send to remote
- go to another working copy and make changes in local repository
- update the content: why did one commit about merging appear whereas there was only an update with pulling?

```
$> cd idl-sample-project
$> git checkout -b feature
$> echo "some data in a new file" >> file.txt
$> git add file.txt
$> git commit -m "new file in a new branch"
$> git push -u origin feature
$> cd ..
$> git clone repository wc1
$> git clone repository wc2
$> cd wc1
$> git checkout feature
$> echo "some other data in the file" >> file.txt
$> git add file.txt
$> git commit -m "new file in a new branch"
$> echo "add data" >> file.txt
$> git add file.txt
$> git commit -m "add data in file"
$> git push
$> cd ../wc2
$> echo "some data in a file" >> file.txt
$> git add file.txt
$> git commit -m "new data"
$> git pull
```

Exercise 10 (*Cleaning branches*)

The objective of this exercise is to play with branches (and to keep your git repository clean).

- create local branch *first* and send it to remote
- create local branch *second* B and send it to remote
- create local branch *third* C and send it to remote
- delete branch *first* only in local
- delete branch *second* only in remote
- delete branch *third* in local and in remote

```
$> cd idl-sample-project
$> git checkout -b first
$> git push -u origin first
$> git checkout -b second
$> git push -u origin second
$> git checkout -b third
$> git push -u origin third
$> git branch -d first
```
If you want to force a deletion (and take a risk on losing your work), use `-D` instead of `-d`
```
$> git push origin --delete second
$> git branch -d third
$> git push origin --delete third
```
After a removal of a branch on the remote side, you should run the following command to remove any obsolete tracking branches:
```
$> git fetch --all --prune
```

Modifying changes

The objective of this set of exercises is to learn how to undo changes. Depending on the type of changes, the process does not require the same Git commands.

WARNING: Be very careful when trying to undo changes that have already been made public. There are several ways to modify a public change, however it is strongly discouraged to do it unless using the proper command. In this section, unless any specific indication, we assume we undo local changes which have not been made public.

Starting point for this set of exercises (or for each of them):

- create a local branch
- create files and make changes until having 2 commits in local repository

```
$> cd idl-sample-project
$> git checkout -b mybranch
$> echo "first" >> file.txt
$> git add file.txt
$> git commit file.txt -m "first commit:  add initial file.txt"
$> echo "This a relevant information."  >> file.txt
$> git commit file.txt -m "second commit:  add relevant information"
```

Some Git syntax: Git uses the symbol `~` to express the notion of *parent commit* (a commit always knows its parents). `HEAD` is the current working version, therefore `HEAD~` is the parent of `HEAD`, i.e. the previous commit. By combining `~` with a number, it is possible to go back further in the history. If there are 4 commits and `HEAD` points to the last one, then `HEAD~3` refers to the first commit (3 parents from `HEAD`).

Exercise 11 (*Amending the last change*)

The objective of this exercise is to learn how to handle the easiest changes: modifying the last local change.

Hint: --amend

- make a change
- commit it with a typo in the comment (*e.g.* `git commit -m "add efature #42"`)

- check the log of your commits: number, comments, hash, etc.
- amend your last change (modify only the comment)
- check the log of your commits and compare it to the previous check

```
$> echo "//It was very interesting" >> file.txt
$> git add file.txt
$> git commit -m "add efature #42"
$> git log --oneline
$> git commit --amend -m "add feature #42"
$> git log --oneline
```

- make another change
- modify the content of the last commit in order to integrate that new change
- check the log of your commits and compare it to the previous checks

```
$> echo "//yet another last minute change" >> file.txt
$> git add file.txt
$> git commit --amend -m "add feature #42"
$> git log --oneline file.txt
```
The number of commits did not changed, however the hash of the last commit is different from the one before the modification

Exercise 12 (*Undoing the last change*)

The objective of this exercise is to learn how to undo the last (local) change and to retrieve the previous state of the repository and/or of the working directory. You will discover on command with different options.

Hint: reset, --soft, --mixed, --hard

- make a change and commit it
- check the status and log (also, note the hash of the last commit)
- rollback to the previous state just before the last commit using the `--soft` switch
- check the status and log, compare it to the previous check. Look also at the index

- go forward to the last state by using `git commit -C ORIG_HEAD`

Note: Actually it is not the same state, but it looks like because it is a new commit which uses the metadata of the state before the reset. Check the hash of this last commit and compare it to the former one: they are different. You could also have made any change and commit it with the same comment to obtain a similar result. In real situations, we usually use the `--amend` switch of `commit` instead of a soft reset of the last commit.

- rollback to a previous state before constructing the last commit (and after the files modifications), using the `--mixed` switch (which is the default behavior of the command)
- check the status and log, compare it to the previous check. Look also at the index
- go forward to the last state (just add the pending changes and commit them. If you want to keep previous metadata, reuse `git commit -C ORIG_HEAD`)
- check the status, the log and the hash of your last commit
- make a change in your existing files or add a new file
- check the status and log
- rollback to the last commit, using the `--hard` switch
- check the status and log

```
$> echo "A change without any future" >> file.txt
$> git add file.txt
$> git commit -m "add an unnecessary line"
$> git log --oneline -n 3
$> git status
$> git diff --staged
$> git reset --soft HEAD~
$> git commit -C ORIG_HEAD
```

Note: the sequence of the two previous commands is equivalent to `git commit --amend --no-edit`

```
$> git status
$> git log --oneline -n 3
$> git reset [--mixed] HEAD~
$> git log --oneline -n 3
$> git diff --staged
$> git commit -C ORIG_HEAD
$> git status
$> git log --oneline -n 3
$> echo "This change does not exist" >> file.txt
$> git reset --hard HEAD~
$> git status
$> git log --oneline -n 3
$> git diff --staged
```

Warning: any changes made between the two commits in the working directory will be lost. You should be very cautious with hard resets

Exercise 13 (*Rewriting history*)

In this exercise, we approach the dangerous territory of history rewriting. Be cautious, rewriting Git history is considered as a bad and dangerous practice if the work has already been published, but acceptable for local rewriting. Thus, the objective of this exercise is to learn how to rewrite local history before sharing it. The usual use cases are the following: cleaning bad comments or non-professional comments, removing a faulty commit in the middle of many legit commits, polishing the history to make it more understandable... or to polish the commiter e-reputation. These use cases can be divided into two categories: the cases the developer really wants to rewrite the history and the cases she only wants to undo a change without hiding it.

Hint: revert, rebase

- as usual, go to your working directory
- (optional) create your branch (in order to avoid to break your master branch)
- make many changes in many commits (add file and commit it, modify it and commit it, make a dummy changes and commit it, add few typos and commit, etc.)
- case #1: undoing changes
  - check the history and choose a commit to remove
  - keep the hash of the commit to be deleted
  - remove it
  - check the status of the repository: Git is helping you by proposing possible solutions concerning the operation (aborting or continuing)
  - continue: solve the conflict if needed (remember: if there is a conflict, it is mandatory to edit the file, then to add it to mark it as solved)
  - continue the operation
  - check the state of the repository: is it consistent? how did the history change? Did the faulty commit disappear?

```
$> cd TAF-DL/IDL/idl-sample-project
$> git checkout -b mytestingbranch
$> touch testingFile.txt
$> echo "content of my new file" >> testingFile.txt
$> git add testingFile.txt
$> git commit -m "add a new data file"
$> echo "some data" >> testingFile.txt
$> git add testingFile.txt
$> git commit -m "more data"
$> echo "new corrected data" >> testingFile.txt
$> git add testingFile.txt
$> git commit -m "add consistent data"
$> git log --pretty=oneline
```
Here, copy the *hash* of the commit you want to remove
```
$> git revert commit_hash
```
An error message should be displayed
```
$> get status .
```
Solve the conflict (edit the file and mark it as solved)
```
$> git add testingFile.txt
$> git revert --continue
```
The other solution was:to abort the revert withr `git revert --abort`.

- case #2: rewriting history
  - let's suppose the previous solution is not satisfaying: there are many commits for simple changes, and there is a visible undo. We want to polish our changes (we do not want to make appear the revert in the history)
  - go back to the previous state (before reverting) as seen in the previous exercise
  - check the state of the repository
  - keep the hash of the commit to be deleted (the same as in the first case)
  - reconstruct (interactively) the history: drop the selected commit
  - if there is a conflict, solve it (edit the file and mark it as resolved)
  - continue the operation (note that Git messages are present to help you to choose which command to type)
- finish the operation
- share your work

```
$> git log --pretty=oneline testingFile.txt
$> git reset --hard HEAD~
$> git log --pretty=oneline testingFile.txt
```
Here, one has to copy the commit hash corresponding to the commit
```
$> git rebase -i commit_hash^
```
Choose the commit to *drop* (by default, it is probably marked as *pick*). A wild conflict appears, solve it then type:
```
$> git add file.txt
$> git rebase --continue
$> git push
```

For a fistful of git commands

If you read this, it means you have done all previous exercises. Now, you should be a Git expert. However, there are a lot of other commands to discover: `reflog`, `stash`, etc.. If you are curious enough, you can search what are those commands...

pull request. A "pull request" is not a Git command, but a GitHub feature. However, the expression has been made very popular and a lot of developers assume all people are Git gurus. The consequence is that their FAQ look often like: "How can I contribute to the project? Do a pull request."

A "pull request" is a specific feature of GitHub, but it relies on a Git command: git-request-pull.

For more information, see `https://git-scm.com/docs/git-request-pull`.

stash. The stash is a way to be able to switch branches even if your current work is not finished or in a clean state. Instead of doing a commit of half-done work, you can stash it. For more information, see `https://git-scm.com/book/en/v2/Git-Tools-Stashing-and-Cleaning`.

reflog. Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs". It tracks when references are updated. When rewriting history, commits can be "lost", meaning there is no direct access to it. The reflog can be useful to retrieve them. For more information, see `https://git-scm.com/docs/git-reflog` and `https://www.atlassian.com/git/tutorials/rewriting-history/git-reflog`.