Atelier Git — TD 1

version #

Assistants C/Unix 2023 <assistants@tickets.assistants.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.assistants.epita.fr

1 Git

Git is one of the most important tools you will learn to use at EPITA. It is **crucial** for your education at EPITA and your future professional work. Pay attention to this tutorial, take some time to re-read it, use Git properly and document yourself. Every exercise and project will make use of Git, so you will have plenty of time to practice. Furthermore, we will discuss Git several times during the year, through tutorials, conferences, exercises, etc.

1.1 What is Git?

Git is a *version control system* (abbreviated VCS), a program that backups the evolution of a group of files over time. VCSs can be used in many situations (image/video editing software, note taking, ...) and are particularly relevant in computer science projects where we want to track changes made to source code, tests, documentation and more.

Advantages of VCS include:

- No more tedious old-school backups where you manually copy folders and files to another location. Manipulations are automated, space efficient and reliable.
- Synchronizing your work with multiple persons inside a team.
- A quick and convenient way to navigate between different versions of a file. This also means bringing an entire project back to a stable and functional state.
- Identifying a change that introduced a bug (where? when? who?).
- Visualizing the changes made since the last saved version.

In particular, here are some advantages of Git over other VCSs:

- It is *distributed*, meaning it can work without any interaction with a server acting as the "main repository".
- It is the most used VCS in the world. It is used by large projects you probably already know such as the Linux kernel or Android.
- Although it is mainly developed for Linux, it also works on Windows and macOS. It can thus easily be used for any of your projects.

Going further...

Git was originally developed for the Linux kernel in 2005 by Linus Torvalds. This was motivated by the fact that BitKeeper, the proprietary VCS previously used for the kernel, revoked the free access to the Linux developers. Other free alternatives did not suit Linus needs, thus he created a new free and open source tool: Git, focused on speed, distribution and heavy project workload.

Git will be the only VCS presented during this semester, but you are invited to document yourself about other tools, such as Mercurial or SVN, to form your own opinion on the different existing alternatives. However, remember that Git will be **mandatory** for all your projects and submissions.

1.2 Glossary

VCSs have their own vocabulary. We are going to stop on a few terms you have to know to understand the rest of the explanations and commands:

- **Repository:** name given to the folder of your project managed by Git. Only the files present in this directory (and its sub-directories, obviously) can be tracked by the Git repository of your project.
- **Remote repository:** distant repository accessible by you and your collaborators. You can synchronize your local repository with it. This way you can collaborate and backup your work by pushing and pulling data to and from this remote.
- **Commit:** *"committing"* is recording changes to the repository.
- **Push:** *"pushing"* is propagating your local commits to a remote repository.
- **Revision:** save of your project at a given point in time. For each commit you make, you create a new revision, or version, of your project.
- **Changeset (or SHA-1):** Git needs to attribute a version number to each of your commits. This number should not be "sequential" because of conflicts problems. Therefore, Git attributes a unique SHA-1 (something like *5500bd8e0ae52775c9abf69749cde9c34001650b*) to each of your commits.
- **HEAD:** HEAD is the name given to the currently active revision of your project. It thus often matches your last commit.
- **Working copy/workspace:** the current state of the files of your hard drive. When working on a project, your working copy will differ from HEAD until you commit your changes.

1.3 Git configuration

Before using Git, you must configure it to let it knows at least *who* will make changes. Here are the commands necessary for a basic configuration:

1. `git config --global user.name "Xavier Login"`
2. `git config --global user.email xavier.login@epita.fr`

Git looks for configuration options in three different files, which allow to have different options for individual repositories, users or the whole system. The files, ordered by descending precedence, are:

- `<repository>/.git/config`: settings of the repository.
- `~/.gitconfig`: settings of the user. This is where the settings defined with the `--global` option go.
- `/etc/gitconfig`: settings for the whole system.

There are obviously a lot of configuration options, like aliases, colors, diff formats, pagers and merge tools. If you want more information on what to configure in Git, see `git-config(1)`.

1.4 Adding SSH keys

You have already generated a pair of **SSH keys**. If needed, do not hesitate to go back and read the corresponding section.

This year, you will work on provided remote Git repositories. Thus, you will need to use your SSH keys to connect to the server containing the Git repositories.

To do so, you should copy the content of `id_ed25519.pub` in the corresponding field in your profile on the CRI's Intranet. The `id_ed25519.pub` key is **your public key**, the one you can share with others, and is available in the `$HOME/.ssh/` directory.

Be careful!

Beware not to change or move the generated keys, else you will not be able to use them easily.

You will see that your passphrase will be asked several times during this year. You may find it annoying to write it every time you want to save your work. Thus, you will be able to use a `ssh-agent` that will keep record of your passphrase and will not ask you to fill it every time. This is the procedure to make the ssh-agent work.

```
42sh$ ssh-agent
SSH_AUTH_SOCK=/tmp/ssh-ZzU1iSRPrisK/agent.9796; export SSH_AUTH_SOCK;
SSH_AGENT_PID=9790; export SSH_AGENT_PID;
echo Agent pid 9790;
```

This command shows what variables must be declared and exported. To do so, run the following commands with **your** output of `ssh-agent`.

```
42sh$ SSH_AUTH_SOCK=/tmp/ssh-ZzU1iSRPrisK/agent.9796; export SSH_AUTH_SOCK;
42sh$ SSH_AGENT_PID=9790; export SSH_AGENT_PID;
```

```
42sh$ ssh-add
Enter passphrase for /home/xavier.login/.ssh/id_ed25519:
Identity added: /home/xavier.login/.ssh/id_ed25519(/home/xavier.login/.ssh/id_ed25519)
```

This one will ask you your passphrase in order to save it.

After you have done this, your passphrase is kept in cache until you killed your current session and will not be asked every time.

1.5 The Git commands

1.5.1 `git help`

All of the Git commands that you will type will begin with the name of the binary: `git`. The command you will use the most at first is `git help`. It gives you a non-exhaustive list of the available git commands. For more information about a command, type `git help <command name>`. Do not worry, we will not see all the available commands today, only the ones that will be useful to simply manage your projects and submissions.

1.5.2 `git init`

This command creates a new Git repository. It is normally used to create a new empty repository. Most of the other Git commands cannot be used outside of an initialized repository thus it is usually one of the first commands you would use. However, for school projects this year, you will not initialize your repository because you will work on remote repository already created for you.

1.5.3 `git clone`

During this semester, each project will have a dedicated remote repository hosted by the CRI. To work on them, you will need to obtain a local copy of these repositories on your local computer. This step can be done with the `git clone` command on your machine. This creates a directory named like the repository you are cloning. You are now able to work on it.

1.5.4 `git status`

This command allows you to see the current state of your repository. This command is a list of files divided in categories. Usually, the first one contains the list of files that will be part of the next commit, the second one lists the modified files that will not be part of the next commit, and the last one lists the files untracked by Git.

1.5.5 `git add`

Git does not track any file on your repository when you start. You need to ask it explicitly to track files that will be important for the next commit. To achieve this, we use the command: `git add myfile`.

Moreover, if changes are done on the tracked files, it is important to use this command again to note the changes.

Caveats

- When you call `git add` on a directory, all of its contents are added to the list of files to be committed.
- Git **cannot track an empty directory**.
- Git **cannot track the file permissions** other than the execution bit (`+x`). You will learn more about file permissions in the next tutorial.
- Before every commit, you have to do a `git add` of every file you want to commit in your repository.

Be careful!

You must not push any binary files! You will be sanctioned if we see any in your remote Git repository provided by EPITA. It is considered a bad practice.

Exercise

Clone your repository with the following URL:

```
42sh$ git clone git@git.assistants.epita.fr:p/2025/ateliergit-2025/xavier.login-ateliergit-
↪2025.git
```

Be careful!

Do not forget to change `xavier.login` to your own login.

Go into your repository and try to run the command `ls -la`. You will see a hidden directory named `.git`.

Create a `first_submission` folder containing an empty folder named `empty`, and a `joke.txt` file in your repository.

```
 42sh$ tree
 .
 `-- first_submission
     |-- empty
     `-- joke.txt

2 directories, 1 file
```

1. Add the different files in the repository.
2. Look at the current state of the repository with the `git status` command. The following result should appear:

```
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   first_submission/joke.txt
```

The `empty` folder is not present: Git does not track empty folders.

1.5.6 `git commit`

The commit is one of the most essential commands. It creates a save of the changes you staged for commit with `git add`. Using this command will open your favorite editor[1] and ask you for a commit message: a short comment to describe your commit.

Be careful!

It is **very important** to put a **useful and relevant** comment that explains the goal of each commit! The following commit messages are completely useless and will therefore be **sanctioned**: “update”, “fix”, “commit”, “lol”, “mdr”, “wesh”, “I hate this project”, “chair”, “kebab”, “flan”. Keep in

[1] To set your editor for commits, use `git config`. For example: `git config --global core.editor "vim"`.

mind to always explain exactly what the commit changes, and avoid commit messages like “typo” (345 lines added, 568 lines removed).

The commit messages will be listed in your repository and allow you to know precisely which version is associated to each commit. On a very large project, when you want to find what broke your project through several weeks of commit logs, you probably would rather find commit messages like *“change the behavior of XXX in the YYY feature to avoid potential infinite loops”* than *“lol”*.

Here are some useful guidelines to keep in mind when writing a commit message.

The commit description does not need to be really large, as soon as it is auto sufficient, for instance: `fix the factorial function when N < 0`.

Tips

- The *-v option* show the diff of your commit at the bottom of your commit message
- The *-m option* can take the following argument as your commit message. For example, `git commit -m "fix the factorial function when N < 0"`.

To ensure that you correctly use Git during your projects, **we will check** your repositories. Empty logs, stupid commit descriptions and unjustified low number of commits will be **sanctioned**.

Exercise

1. Commit your previous modifications with the following commit message: *“first_submission: add joke.txt”*.

1.5.7 `git diff`

This command allows you to see all the modifications you have done on a file since the last commit, before adding it and committing its changes. You can use:

- `git diff file.txt` to show the changes for `file.txt` only.
- `git diff` to show the changes for all the files.

We advise you to read `git help diff` to see other options, that will allow you to see changes between two commits[2].

Exercise

1. Write a joke in the `joke.txt` file in the `first_submission/` directory.
2. Add the changes to do a new commit.
3. Change the file again.
4. Look at the `git diff` output on this file.

[2] or even two branches/tags/remotes/...! Branches are an important notion of Git that we will learn about during other tutorials.

1.5.8 `git log`

This command allows you to see the history of your repository as a list of commits, with their author, their date and the description of the commit. This command is really useful to know where we are or to find an old version of your repository that you want to restore.

Exercise

Try the following commands:

- `git log`
- `git log --oneline`
- `git log --stat`
- `git log -p`
- `git log --graph`
- `git log --all --decorate --graph --oneline`

1.5.9 `git tag`

The tags are an important tool provided by Git. They will allow you to identify a revision with a name instead of a SHA-1. For instance, your submissions will have a `submission` tag, so that you know which step of your work you chose to submit for the evaluation. To tag your commit, you will use the following command:

```
git tag -a <tag name> -m <message>
```

Note that you **cannot have** two tags with the same name in a repository. If you want to change the commit pointed by a tag, use the `git tag -f -a <name of your tag> -m <message>` command.

You can list the tags of your repository with the `git tag` command.

Going further...

To check that your tag is on the right commit, you can use the `--decorate` option of `git log`.

You can also check the tag message with `git tag -n`.

Exercise

Tag your latest commit with the tag `exercises-first_submission-v1.0`.

Be careful!

It is important to write the correct tag. We use it to identify and grade your exercises and projects.

1.5.10 `git restore`

This command lets us discard uncommitted local changes. For example, if you changed some code in `file.txt` and want to go back to the clean version of the file that was last committed, you can use `git restore file.txt`.

It is possible to add the specific version with the option `--source`.

Exercise

1. Change the joke in your file.
2. Use a previously explained command to check the changes on `joke.txt`.
3. Restore the previous joke.

1.5.11 `git push`

When you commit your work, Git keeps track of the versions locally. If you want to propagate your versions on the servers from which you cloned your repository, you need to use the `git push` command. This operation will send all of your local commits to the servers.

Tips

In order to push your tags with your commits, you have to add the `--follow-tags` option. Else, you can use the option `--tags` to push only the tags to the remote repository.

Exercise

1. Push your changes to the server by indicating `git push origin master` for the remote server.
2. Push, this time including your tags. You should get a message informing you that it got submitted.

Be careful!

Without a tag, your submission **will not be graded**! Do not hesitate to call an assistant if you failed to submit your work.

1.5.12 `git pull`

Incorporates changes from a remote repository into the current branch. It will allow you to recover any work pushed from any computers.

1.6 Git during the *piscine*

We remind you that during the *piscine* you are only going to see a really basic usage of Git. Once the *piscine* is over, you will learn a lot of other features of Git that will be useful during the rest of the year.

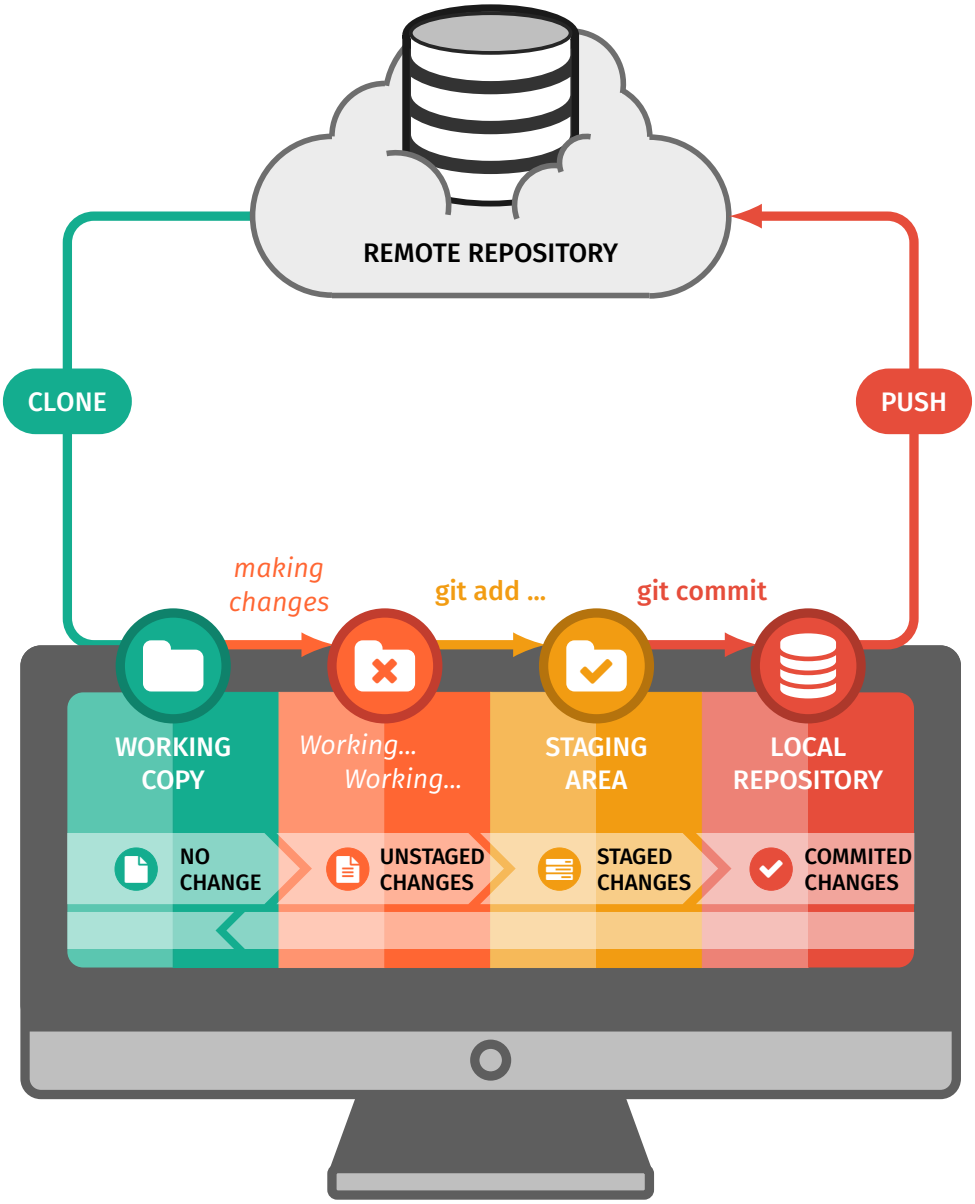

Fig. 1: The workflow we want you to understand and practice.

Knowing the commands shown in this tutorial by heart will be a very good start. The web is full of other tutorials if you want to dive deeper:

- The Git man pages.
- The Pro Git book, freely available online.

- An online interactive cheat sheet.
- And many other resources (tutorials, books, videos, courses, ...).

1.7 Git during the *piscine*

When you will work in teams during the year, if you work all at the same time on different computer, you will see that just `pushing` and `pulling` will not be enough. It will create many conflicts between you and your group. To avoid this, you can use `branches` to work efficiently with your group.

1.7.1 `git branch`

A simple git command that will list every branch currently on your computer. If you specify a name, it will create a branch with this name if the branch does not already created. It is also useful to delete branches.

1.7.2 `git checkout`

A git command that allows you to switch to an existing branch with the name of the branch specified. If you add `-b` option before, it will also create the branch.

1.7.3 `git merge`

This command will allow you to reunite two branches that have different commit. You need to be on the branch you want to continue working after (typically *master* or *main*) and put the name of the other branch after `git merge`.

1.7.4 `git fetch`

Download objects and refs from another repository. It is useful to recover information on branches or tags.

The way is lit. The path is clear. We require only the strength to follow it.