Git Tutorial

Version: 0.5

Anders Nilsson
anders.nilsson@control.lth.se

October 19, 2018

1 Introduction

Why use git, or any other version control software to keep track of files? In short there are at least three big reasons why you should version control your files, two of which are valid also if you work all by yourself. See Figure 1 for an illustration of the first two reasons listed below:

Backups: With version control you always have previous versions of your files available, if you happen to do something stupid like erasing a file. Or just regret a large edit later on.

Organization: Knowing which copy of a project directory is up to date, and which ones are not, saves a lot of time and problems.

Collaboration: Everyone who has tried to collaborate with other people by sending files, or parts of files, over email knows how fragile that is.

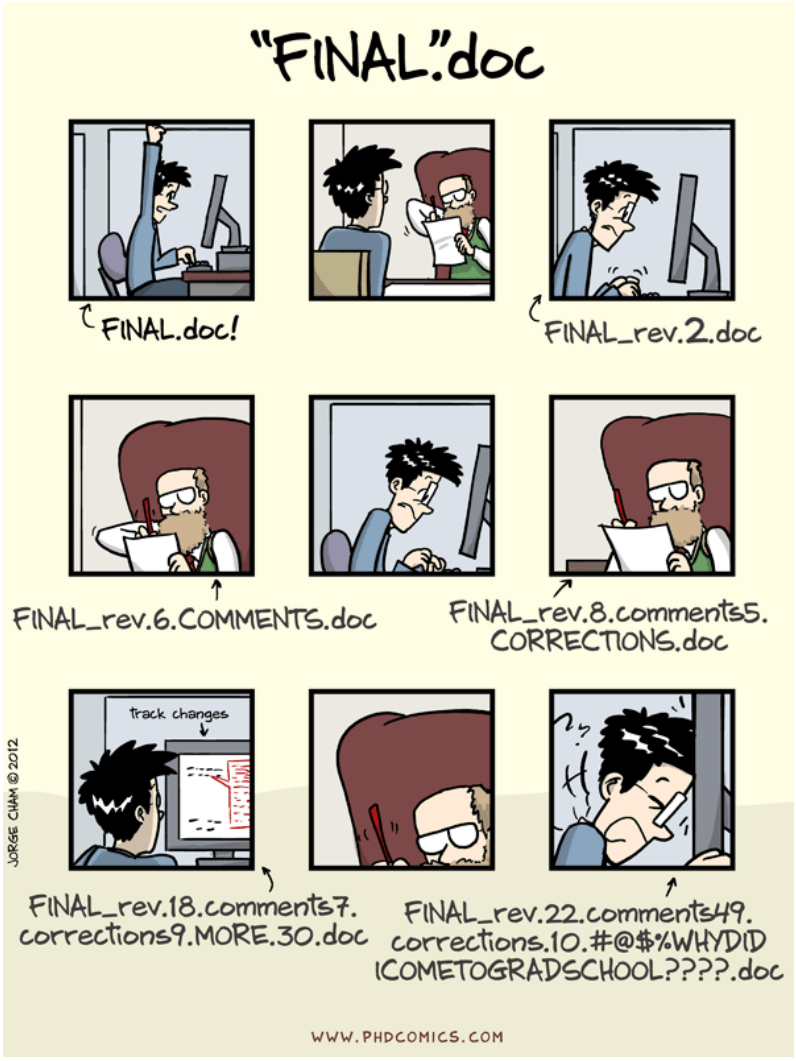

Figure 1: Typical(?) result after version control by renaming.

But why use git instead of CVS, subversion, clearcase, or any other well-known version control tool?

- Git is free, both as in speech and in beer, unlike Clearcase, Team Foundation and other commercial systems.

- Git is distributed. When each user has her own repository copy you do not always need to have contact with a central server.
- There is a large Internet community around git making it easy to find information and/or help when needed.
- It is used in some large high profile open source projects. Most known is the Linux kernel.

2 Git Tools

Depending on which OS you use, there are some different choices of git clients available. For the purpose of this tutorial we will only consider running git from a terminal shell, but you can of course install one of the available gui clients and transform the information given in the tutorial to that gui client.

Linux

Git packages are included in at least all major Linux distributions. Just use your package manager of choice to find and install git.

MacOSX

Get your git client from `http://git-scm.com/download/mac`

Windows

Get your Windows git client from `http://git-scm.com/download/win`.

3 Hands-On Example

In this section we will walk through the basic git operations using this very tutorial as an example project. We will first show the work-flow for a standalone repository, typically what you would have for versioning your own small projects. Then we will show what happens when we want to work in a distributed context, cooperating with other people.

3.1 Standalone Repository

The simplest use case is that of one single developer, one local repository. Files in a project versioned with git go through the following stages as you work on them, see Figure 2:

Untracked: Git does not know anything about untracked files.

Tracked, unmodified: Git knows about these files, and there have been no changes to them since last commit.

Tracked, modified: Files tracked by git, there are changes made to the file since last commit but git does not know whether these changes should go into next commit.

Staged: The changes made to this file will be committed next time you perform a commit.

Help

If you wonder what a git command do or which arguments it takes, just remember that `git help` will list the most common git commands and `git help command` will show the git manual for the given command.

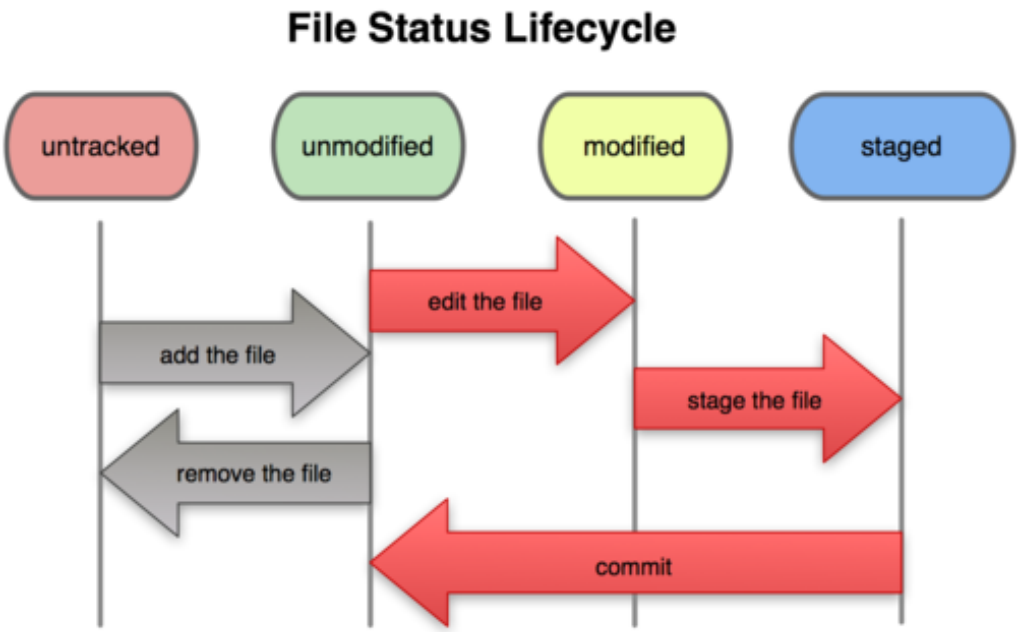

Figure 2: Files go through the following stages as you work on them.

First time configuration

On each computer (or for each user account) you need to tell git your name and email address. These are used in each commit to make it possible to see who has performed which commits to a shared repository.

```
git config --global user.name "Anders Nilsson"
git config --global user.email "andersn@control.lth.se"
```

You can always see which name and/or email, if any, git thinks you have:

```
git config --global --get user.name
```

Create a Repository

Let us say we have a directory with files that we want to version control using git.

```
andersn@stodola: pwd
/local/home/andersn/work/computers/git-tutorial

andersn@stodola: ls
git-tutorial.aux  git-tutorial.pdf    git-tutorial.tex
lifecycle.png  git-tutorial.log  #git-tutorial.tex#
git-tutorial.tex~
```

We can now create a new repository in this directory

```
andersn@stodola: git init
```

The result of this command is a new subdirectory called `.git` that contains the actual git repository. If you look into it you will see various files and subdirectories.

```
andersn@stodola: ls .git/
branches  config  description  HEAD  hooks  info  objects  refs
```

Some of these are human readable while others are not. If you are curious you can navigate the subdirectories and try to look into files. Or don't, there is almost never any need to look into this directory.

Adding Files

As you have seen above there are directory we do not want to version control. The only files here that should go into git are `git-tutorial.tex` and `lifecycle.png`, the rest are either temporary files from emacs and LaTeXor a pdf file which is generated from the LaTeXsource. We will take care of that when it is time to add files to the repository.

```
andersn@stodola: git status
# On branch master
#
# Initial commit
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       git-tutorial.aux
#       git-tutorial.log
#       git-tutorial.pdf
#       git-tutorial.tex
#       git-tutorial.tex~
#       lifecycle.png
nothing added to commit but untracked files present (use "git add" to track)
```

```
andersn@stodola: git add lifecycle.png git-tutorial.tex
andersn@stodola: git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   git-tutorial.tex
#       new file:   lifecycle.png
#
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#
#       .#git-tutorial.tex
#       git-tutorial.aux
#       git-tutorial.log
#       git-tutorial.pdf
#       git-tutorial.tex~
```

Using gitignore

As you have just seen `git status` lists a number of untracked files that we are not interested in versioning, and as the number of unversioned files grow they tend to clutter the view. By creating a file called `.gitignore` in which we list filenames and name pattern that we do not want git to consider the view becomes much more uncluttered.

```
*.aux
*.log
*~
git-tutorial.pdf
```

Adding `.gitignore` to our repository and running `git status` again we see that it now looks much better.

```
andersn@stodola: git add .gitignore
andersn@stodola: git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file:   .gitignore
#       new file:   git-tutorial.tex
#       new file:   lifecycle.png
```

Commit

If we are satisfied with the output of `git status` it is now time to perform a commit operation and create an initial revision of the project in the repository. The text within double quote marks is a revision comment. Good revision comments, that is, meaningful but terse, are important as they can be of good use when you or someone else at some time look back into the revision history and try to find out what happened and when.

```
andersn@stodola: git commit -a -m"Initial commit"
[master (root-commit) c6d834c] Initial commit
 3 files changed, 146 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 git-tutorial.tex
 create mode 100644 lifecycle.png

andersn@stodola: git status
# On branch master
nothing to commit (working directory clean)
```

A brief explanation of the command options. `-a` means that all modified files will be staged before commit, without it we would only commit newly added files. `-m` says that a revision comment will follow after it, without it a text editor (perhaps vi) would start and allow you to edit a more comprehensive comment. If you are not comfortable with using the system default editor (which may be quite understandable if that editor is *vi*) you can easily set your shell to use another editor by entering

```
andersn@stodola: export EDITOR=nano
```

If you by mistake become stuck in vi and do not know how to get out, the sequence `<Esc> <Esc>:q!<enter>` should liberate you.

After performing the first commit, work goes on. New files are added and existing files are edited. Whenever it feels motivated, add new files to the repository and perform new commits with meaningful commit messages.

3.2 Distributed Work

With the distributed model of git (and bzr,hg,darcs,...) each user has his own repository, in contrast to the centralized model of subversion and cvs where there is only one central repository where the version data of a project may be stored. With multiple repository copies we need operations to keep different repositories synchronized.

In this tutorial we will keep to the model with one "master" repository with which all users synchronize. However, this model is by no means enforced by the git tool. Users could just as well synchronized directly with their fellow users' repositories, but practical difficulties with allowing access to all repositories as well as the higher level of discipline needed to keep all repositories synchronized can make this flat hierarchy model cumbersome for all but very experienced users.

Clone

To create a local copy of a remote repository we use `git clone`

```
andersn@fiol: git clone andersn@stodola.control.lth.se:work/computers/git-tutorial
Cloning into 'git-tutorial'...
remote: Counting objects: 11, done.
remote: Compressing objects: 100% (10/10), done.
remote: Total 11 (delta 2), reused 0 (delta 0)
Receiving objects: 100% (11/11), 46.64 KiB, done.
Resolving deltas: 100% (2/2), done.
```

Pull

When we want to check for, and merge into our repository copy, changes made to the master repository, we can use `git pull`

```
andersn@fiol: git pull
Already up-to-date.
```

In this case it was not particularly interesting since the local clone was already up to date with the remote master.

Push

The opposite operation to `git pull` is `git push`, to push local commits to the remote master branch.

```
andersn@fiol: git push
Counting objects: 8, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 3.99 KiB, done.
Total 6 (delta 1), reused 0 (delta 0)
To stodola.control.lth.se:work/computers/git-tutorial
   ebf6fbb..15f6453  master -> master
```

3.3 Other useful commands

In addition to the above listed git operations, there are a few more operations that are good to know right from start.

log

```
andersn@stodola: git log
commit 846bf870955b4eae2a5fd7bb6f5071af6b7a9003
Author: Anders Nilsson <andersn@control.lth.se>
Date:   Wed Mar 20 09:32:35 2013 +0100

    Added chaos figure and some motivation

commit 95390dea5ea41624953f566fc077c8a8bdbfaf3a
Author: Anders Nilsson <anders@angsro14.se>
Date:   Tue Mar 19 23:02:44 2013 +0100

    Last changes for tonight
```

`git log` lists all commits in reverse chronological order. In the example above that the last commits were performed by different users; myself at home and myself at work.

diff

`git diff` shows the differences between two revisions of a file. When no revisions are given as arguments it lists the differences to the latest committed revision.

```
andersn@stodola: git diff git-tutorial.tex
diff --git a/git-tutorial.tex b/git-tutorial.tex
index dfcd562..3e66f5d 100644
--- a/git-tutorial.tex
+++ b/git-tutorial.tex
@@ -370,7 +370,12 @@ users; myself at home and myself at work.

 \subsubsection*{diff}
 \label{diff}
+\verb|git diff| shows the differences between two revisions of a
+file. When no revisions are given as arguments it lists the
+differences to the latest committed revision.

 \subsubsection*{merge}
 \label{merge}
```

blame

If we are interested in finding out exactly who is responsible for some information in a file, and when it was committed, we can use `git blame <file>`. In the example below we only show a small excerpt of the output, the real output is a fully annotated version of the file given as argument.

```
7c8a691e (Anders Nilsson    2013-03-20 11:07:17 +0100 370)
7c8a691e (Anders Nilsson    2013-03-20 11:07:17 +0100 371) \subsubsection*{diff}
7c8a691e (Anders Nilsson    2013-03-20 11:07:17 +0100 372) \label{diff}
00000000 (Not Committed Yet 2013-03-20 11:27:53 +0100 373) \verb|git diff| shows the diffe
00000000 (Not Committed Yet 2013-03-20 11:27:53 +0100 374) file. When no revisions are giv
00000000 (Not Committed Yet 2013-03-20 11:27:53 +0100 375) differences to the latest commi
```

merge

```
andersn@stodola: git pull /tmp/git-tutorial
remote: Counting objects: 5, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (3/3), done.
From /tmp/git-tutorial
 * branch            HEAD       -> FETCH_HEAD
Auto-merging git-tutorial.tex
CONFLICT (content): Merge conflict in git-tutorial.tex
Automatic merge failed; fix conflicts and then commit the result.
```

When we open the file containing the conflict we find a section with conflict markers;
`<<<<<<< local version ======= remote version >>>>>>>`

```
<<<<<<< HEAD
Sometimes it happens that merge conflicts are introduced, the most
common occasion is when two users happen to edit the same lines of a
file.
=======
Bla bla bla bla bla bla
>>>>>>> 82b8fb8e2f4985b12dfb9d402be069eeb070db92
```

Resolve the conflict by editing the file and choose which version, or combination of them, that should be kept. In the above case it seems pretty obvious that the second version is just crap so we remove it as well as the conflict markers. Then we commit the resolved file.

```
andersn@stodola: git commit -am"Resolved merge conflict"
[master ee5645b] Resolved merge conflict
```

4 Further Reading

Some useful links:

- A cheat sheet is good to print out and have at hand next to your keyboard:
  `http://git-scm.com/documentation` or
  `http://www.git-tower.com/files/cheatsheet/Git_Cheat_Sheet_grey.pdf`
- The extensive *Pro Git* handbook is available for free online: `http://git-scm.com/book`

In general, Google is your friend, and there are literally hundreds (if not thousands) of freely available git references, Q&A:s, and tutorials out there.

5 FRTN40 Specifics

Here follows some specific information for students involved in the project course FRTN40.

Repositories

At the department we are running an instance of *Gitlab* (`https://about.gitlab.com/`, a web-based system used for administering git repositories. We have set up master repositories for you to at the department. To create a local clone you write something similar to:

```
andersn@stodola: git clone git@gitlab.control.lth.se:regler/FRTN40/2018/GroupA
Cloning into 'group-A'...
warning: You appear to have cloned an empty repository.
```

Just replace "A" with your group denomination. You should now have an empty (except for .git) directory called `group-A`.

Playground

We have a playground repository set up if you want to try using the gitlab server. Go to `https://gitlab.control` and play around or follow instructions to make a local clone to play around with.

Gitlab accounts

You create an account by going to `https://gitlab.control.lth.se` and following instructions there. You may also use an existing Google account of yours if you would like to.

Issue Tracker

The issue tracker is a quite useful feature of the Gitlab server. Access the tracker for your project by clicking on the corresponding button along the left side of the browser window, as seen in figure 3. Look at an issue as whatever project-related you might scribble down on a piece of paper or a postit note; a bug, a suggested feature, or just something to remember. The tracker is the tool to manage the heap of notes; adding new ones, update with more information, throw away, and even brush through the trash to find that note you accidentally threw away too early by mistake.

ssh or https

When accessing repositories on the gitlab server you can choose between two protocols/authentication mechanisms; *ssh* or *https*. To use *https* does not require any further setup, but the drawback is that you will have to authenticate with your gitlab username/password everytime to access the server (every *push* and *pull*). Using *ssh* is recommended, but that requires you to generate (if

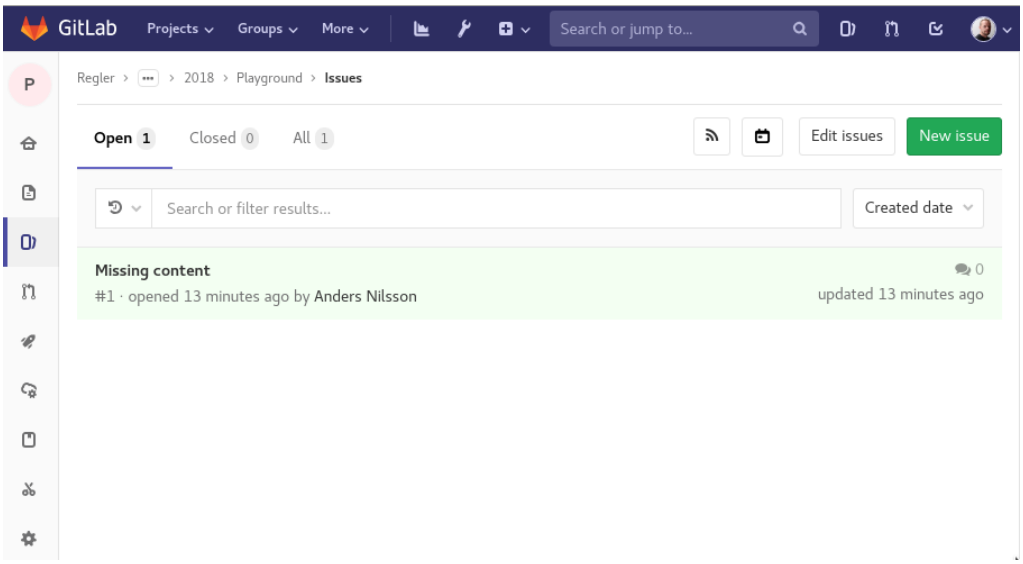

Figure 3: Issue tracker in Gitlab.

not done already) ssh keys and uploading to the gitlab server. Good instructions are available if you go to SSH keys in your gitlab account settings.

ssh problems

If you want to be able to use more than one user account, for example you want to access the master repository both from computers @control and from your private laptop, you should make sure to use the same ssh key pair on all user accounts. Otherwise you will not be allowed access. You do this by copying *both*[1] key files to the other user account you want to use.

[1]Well, you only *need* to copy the *private* key, but it does not really make sense to not also bring with you the public part.