Git for Gas Group: A practical guide

Alvaro Diez

CERN (Universidad de Cantabria)

Abstract. This is a very quick guide aimed to teach anyone (no knowledge required) how to use any git version-control software. The guide is divided into two parts: the rationale behind git, which covers the philosophy behind git version control as well as a more technical approach to what things mean and how they are intended to be used; tnd the second part being a practical usage guide, including the 2-minutes guide to using git, in which the basic work-flow is explained together with the most basic commands needed to use git.

Table of Contents

Abstract 1

Introduction 3
1 Git Rationale 4
1.1 What is version-control and what is NOT version-control 4
1.2 How version control improves software development 4
1.3 How git helps you cope with colleagues 4
1.4 The basic concepts behind git using a practical example 4
1.5 Commits, the necessary evil 4
1.6 Git-ing as a one man company 4
1.7 Branching or how not to screw your group's project 4
2 Extended Practical Guide on Git(Lab) 4
2.1 Installation (Windows and UNIX) 5
2.2 Most basic use case (a.k.a. the only thing you need to read to use git) 5
Configuration 5
Creating a repository 5
Modifying, staging, committing, pushing 6
Pulling changes, i.e. updating 8
2.3 How to fix it when you ALMOST break it 9
You committed what you shouldn't have committed 9
You didn't read above and pushed the wrong code 9
You screwed your local copy of the repository, e.g.: merge failures, modifications before pulling... 9
2.4 Fancier use case (a.k.a. how to make others believe you're a pro) 9
Full repository status reporting 9
Branching, forking and other ways to keep your colleagues happy without bringing cake everyday 9
3 Git'n'Chill (Learn Git in 2minutes) 10

Introduction to version control at CERN

Software changes as much as the weather, anyone that has ever coded even the smallest, simplest parser knows. On top of that it is often the case that more than several people are involved in the development process, either actively contributing or just supervising. In this cases version control is a must, except for those who like to live in a messy past where nothing makes sense and you have to rethink the same things every time you need to modify the software. Git is one of the many version control systems available in the world an it's probably the most widespread of all. When used correctly, it acts as a time-machine / back-up / log-book keeping track of every change done to the software and allowing to go back to older versions, revisit past decision about the development process or structure of the software.

In this short guide we will understand the importance of decent (no need for absolute perfection) version control can improve developing efficiency and revisions of the code as well as helping spot problems in the development process faster or improve anything related to software development. We will also cover the very basics of git version control so that the read will be able to confidently use git version control after reading and following this short, unprofessional guide.

There are three main rules to follow when using any version control that would significantly improve the developing process: Commit frequently, comment extensively, keep it organized. We will explain this three rules in detail in the "rationale" section and are by far the best tips when starting with version control since those three alone would improve the work-flow far more than any other rules imaginable.

Amongst the top on-line services that use git as version control system, GitHub is the most widely used and famous of all. At CERN however GitLab (another extensively used service) is the go-to option with free unlimited accounts provided for all CERN associates. There are many more options including DIY solutions but we will focus mainly on GitLab. GitLab and GitHub have different underlying philosophies but for the most part can be used in the exact same way. Since this is a basic barely-scratching-the-surface guide, everything we will present is similar in GitLab and GitHub (except for the pull/merge-request) making this guide a broad approach that can be used to get started in either platform.

We shall now take a trip into the basic concepts of version control in general and git in particular in a (firstly) theoretical and (at last) practical way. Even-though one should read both sections in the usual order (first 1, then 2) the user might choose to skip any of the two or just go for the super-quick guide, with total confidence that he'll be able to use git version control (albeit not very efficiently) for both sections are self-contained.

1 Git Rationale

1.1 What is version-control and what is NOT version-control

1.2 How version control improves software development

1.3 How git helps you cope with colleagues

1.4 The basic concepts behind git using a practical example

1.5 Commits, the necessary evil

1.6 Git-ing as a one man company

1.7 Branching or how not to screw your group's project

2 Extended Practical Guide on Git(Lab)

For the typical use case scenario, one can either start from an empty repository or from an already existing code on which we want to start version control using git. This two case have a different treatment, but after one has initialized the repository[1] the proceedings are the same and it makes no difference after the first commit, as we will see now.

First of all one should open a browser tab and visit "https://www.gitlab.cern.ch" (or their git service of choice). If not already done, create an account; for CERN's GitLab logging in with your NICE account will automatically create a GitLab account without any necessary steps.

There are many ways to start from scratch but we will cover the easiest for newcomers. In the main page of your GitLab dashboard you should find a button named "New Project", we shall click on it to... (can you guess it?) create a new project repository. Here one should fill the blank spaces with sensible information, the more precise and descriptive, the better. We do not need to worry too much about the description as it can be modified fairly easy later on; the SLUG[2] however is quite important, the name of the project should go here, if it is descriptive, even better.

We should also select "visibility level" or privacy values for our projects. Depending on the target and reach of your project you should be able to

Once we have the repository created, our paths diverge depending on whether we are starting from scratch or from an existing project/code. Worry not, for our paths will converge again before the sun sets over the lamplighter's planet.

[1] Where code is stored
[2] The first blank space on the top, what goes at the end of the URL

2.1 Installation (Windows and UNIX)

2.2 Most basic use case (a.k.a. the only thing you need to read to use git)

So you have installed git in your computer be it Windows or UNIX based. Now it's time to get started on "the git thing". For that we'll take a look at the basic steps you should take to get a git basic git version control working and the simplest workflow that can take advantage of it. As we go we shall explain the terminology and their intended usage refering back to the "Rationale" section of this guide. For a faster no-theory-involved guide, check out the Git'n'Chill section.

Configuration First thing we have to do is configure our git installation. This step is not totally compulsory but since you're probably using your own computer or at least your own user on the computer, it makes sense to have it done. It will also make it less painfull to work on a private repository.

We need to open git by opening a terminal in any UNIX system that has git already installed, or we navigate to our workspace folder in Windows, right click anywhere inside of it and hit "Git Bash". Once there we should type the following commands:

```
git config -global user.name "[YourUserName]"
```

and

```
git config -global user.email "[YourCERNEmail@cern.ch]"
```

Now git know who you are so it can log you in or upload content to your repositories without you having to manually log in every single time.

Creating a repository For creating a new repository on Git there are two main options: using the GUI on their webpage; or using the CLI to do so. The GUI version is very straight forward and it only involves clicking a few buttons and filling up some information about the repository. Since this options is completely hassle free and this guide is not intended as a high-level reference but rather as a "get started" quick guide, we will only focus on the GUI-web version.

One should go to the dashboard of their git service of choice, in the image below one can see an example using CERN's GitLab, but it should be similar in GitHub/BitBucket...

- 1) Find and click on the *+ New Project*
- 2a) Fill the necessary fields, use sensible data that relates to the code you'll be developing. For example if you plan to write an script that fits data to a straight line a proper name for the repository could be *Data fitter* and a wrong name would be *MyScript*.

Always try to explain everything and never fall in the "I think I will remember forever" hole, because nobody does.
- 2b) Should you already have a working repository that you'd like to import to a different service, GitLab provides a very simple way of doing so. Just copy the URL of your previous repository on the corresponding box and let is work.
- Check that everything is right, consider creating a test README document that would help you check that everything is working.

At this point you have a working repository, but it is still only on the Internet, which is not a very practical place to code, specially since Git services have ~~the crappiest~~ not the best text editors. for that we will return to our old friend the terminal. As always for Windows you have to:

- navigate to the folder where your repository folder will be in
- Right click anywhere inside the folder
- Click "Git Bash Here"
- Good job, you can now have a cookie and thank MS for making the process "simple"

In UNIX systems simply open a terminal.

Now that we are at the git-capable terminal we shall run the following command to download the repository information and files:

```
git clone [URL of Repository] [Name of local folder]
```

Where *Name of local folder* is the name of the folder where all the files on your repository will be stored. Git can create that folder for you. If not specified it defaults to a folder with the name of the repository.

The *URL of the repository* can be obtained form the main page of the project on GitLab's web-page (see picture)

Now you have an online repository for your code and have already linked a local folder in your computer to that online repository. This means we are ready to start development. Next up will learn how to stage, save and document changes (commits in git mumbo-jumbo), by far the most powerful and useful feature for version control system broke is such as git.

Modifying, staging, committing, pushing So now you have your brand new repository. You start coding, and start making progress. What do you do now? well, as we saw in the theoretical part one should now set a mark on the changes so that they are stored and accessible later on. The way to do this is by staging and committing the changes. Staging and committing effectively mean storing the state of your code and project at a certain time first of all for safety reasons (as a back-up) and secondly for logging purposes (being able to go back to previous states and to know what and why the previous states of the code were as they are).

To perform such actions git provides de user with a series of commands that directly relate to the processed needed for any back-up/logging activity. The first step is to state which files have interesting changes, i.e. which files to upload; then pack them under a common status report i.e. committing and later on to push these modifications to the cloud repository so that it becomes available for anyone , anywhere (as long as the Wi-Fi works, unlike at CERN).

So getting back to the commands the first command, the one that helps selecting the files that have relevant changes for the project. The command is a very simple and straight forward one:

```
git add [fileName]
```

where *fileName* should be the name of the file(s) that one wished to update. For adding multiple files at once it suffices to concatenate the file names one after the other separated by a white-space character.

Once we have added all the files we wish to update, it's time to commit the changes, whcih is the git mumbo-jumbo for packing the files under a common changelog. For this process there's a simple way and a more flexible option. The easiest way, which is the recommended for small changes that don't need very big explanations is:

```
git commit -m [Commit message
```

where *Commit message* should be a brief description of the changes that where made in the files that you added in the previous commands; and "-m" is the message option that allows the user to input the commit message in the same command.

The second way to commit changes is a more complex and flexible one; it is the same idea at it's core but allow for more complex commit messages:

```
git commit
```

This will bring up a terminal-based text editor allowing you to format commit messages better. First line should be small enough to be the title of the commit message and the rest of the text can be as large as one would like it to be. Keep in mind that commits should happen regularly so if you need a very long commit message you're probably not doing it right.

After the message has been written to the file using the text editor, you should save this file without changing the name of the file. Now the changes have been reported, packed and stored as a sort of landmark within your project. At this point one can keep working on the code without updating the online repository. If you keep committing the changes they will all be stored and they can pushed together to the online repository using the next command. This is not the recommended work-flow because pushing commits to the online git service is a fast process and acts as a reliable and organized back-up.

Pushing the commit(s) to the online git service couldn't be easier. If you want to push your changes to the same branch it is as easy as:

```
git push
```

This will push the commits and upload them to your working branch. Alternatively you can specify where to push the commits by adding such data. A simple example would be when you wish to push your changes to the master branch of the repository from which you initialized your repository:

```
git push origin master
```

After running either command you can get your code up to date in a matter of seconds. This means having a effective version control system that allows you to keep track of the changes and easily revert to previous versions of the software that also doubles as a syncronization tool between different systems and as a back-up for all your code.

Up to this point we have cover in quite some depth the development process in the forward direction, this is all you need for git-controlled development if you were to code, develop and run the software on yur own in just one system. But git is not limited to such restrictions, instead where it shine brighter is in multisystem and collaborative development as we have already seen in the theoretical part. Now we will cover the most basic way to retrieve the latest version of a software and also how to undo commits before pushing then or to go back to previous versions of the code.

Pulling changes, i.e. updating

Let's picture the following scenario. You (or a colleague) have been working on the code from system A, and made some changes. Having read this guide you proceeded as expected: committed the changes, commented them properly and pushed them regularly so that the online repository is up-to date with the latest local changes. But now you are working in a different system, B, that has an out-dated version, ¿what can you do?

Well you could go online to the project's webpage download the modified files and replace the old ones with the new ones, but if you're going to do that, you might as well just type the changes directly into the files. No, what an efficient and handsome/beautiful developer would do is open it's git interface (a terminal or a git bash depending on the OS), navigate to the repository folder on system B and type the following command:

```
git pull
```

then all that's left for this extraordinary programmer is to wait while the `pull` command does its job. What this command does is check the current status of the repository both locally and remotely and update the local version

to the newest online version available. After the command has finished one can start developing with the certainty that they are modifying the latest version of the software, which means that they are actually making progress and not just working in vein.

There's a common problem arising from this command, so typical that is worth mentioning even in an introductory guide. The command might fail if you have made changes to the files in system B but have not committed them to the repository. This problem should not happen to you, because you know better and you commit or delete every change you make to the code, before leaving your machines. But if you find yourself in this situation (because, let's face it, bad colleagues are more common than Italians at CERN) here are the easiest two solutions:

A) Review the changes with respect to the previous and next version of the code and decide whether they shall be merged, ditched... And then proceed accordingly B) Discard the changed files by running

```
git checkout [fileName]
```

and keep working as if those changes were never existent.

2.3 How to fix it when you ALMOST break it

You committed what you shouldn't have committed

You didn't read above and pushed the wrong code

You screwed your local copy of the repository, e.g.: merge failures, modifications before pulling...

Reverting to previous changes

2.4 Fancier use case (a.k.a. how to make others believe you're a pro)

Full repository status reporting `git status`

```
git blame
```

```
git SOMETHINGELSETHATISHOULDGOOGLE1st
```

Branching, forking and other ways to keep your colleagues happy without bringing cake everyday `git status`

```
git diff
git pull
```

3 Git'n'Chill (Learn Git in 2minutes)

We assume you have already created an account on CERN's GitLab (Log in with your NICE account into gitlab.cern.ch) and have downloaded and installed Git for Windows (available from CMF) or Unix (available from their web page and on every package repository). If you have trouble doing this steps, refer back to the installation section above.

- A) Go to you dashboard on GitLab's webpage and create a new project by clicking "+ New Project" button.

- B) Fill the necessary fields with sensible data to match your project

- C) Configure your git by opening a terminal (UNIX) or Right-Click + Git Bash Here (Windows) in the folder where you wish to store your code locally.

- D) Type into the terminal that has just open
  `git config -global user.name "[YourUserName]"`
  `git config -global user.email "[YourCERNEmail@cern.ch]"`

- E) Intitialize your new repository with an empty README
  `git clone https://gitlab.cern.ch/[username]/[repositoryName].git`
  `cd [NameOfTheFolderThatGitHasCreated]`
  `touch README.md`
  `git add README.md`
  `git commit -m "add README"`
  `git push -u origin master`

- F) Now you can start developing your project. Everytime you make a worthy modification you should run: `git add [filesThatYouModified]`
  `git commit -m "[DescriptioOfTheChanges]"`
  `git push`
  If you happen to work in different computers or with more collaborators make sure to run
  `git pull`
  To download all the changes made in the code up to that moment

- G) Enjoy happy organised and backed-up coding. Do not forget to commit your changes regularly. Also, consider taking a look at the full guide for more advance options like branching (so you always have a working version of the code available), forking (so you can develop other people's software in your own way) as well as issue tracking and building a proper wiki or contribution guide.