Git Foundations

by Aviral Janveja

August 9, 2019

1 What is Git?

Git is a software/tool that allows you to save and manage multiple versions of your code/projects. Git makes it easy for developers to collaborate and share work with others.
Git works by recording the changes you make to a project, storing those changes, then allowing you to reference them as needed.

- Git thinks about its data more like a stream of snapshots. Each time a commit is made, git takes a snapshot of how all the project files look like and saves a reference of that snapshot.
- Git references everything in its database not by file name but by the hash value of its contents. The 40 Character SHA-1 hash-code is a string composed of hexa-decimal characters and it is calculated based on the contents of a file or directory structure.

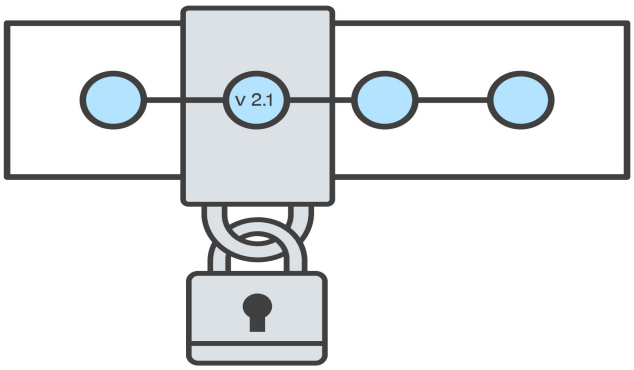

Figure 1: Version Control.

2 Getting Started: Basics

2.1 git init

The word init means initialize. This is a one time command you use during the initial setup of a new repository. Executing this command will create a new '.git' sub-directory in your current working directory. This will also create a new master branch.

2.2 git config −−global user.name "Aviral Janveja"

- The first thing you should do when you install git is to set your user name and email address. This is important because every git commit uses this information.
- You can do this through the git config commands such as **git config −−global user.name "Aviral Janveja"**, **git config −−global user.email aviraljanveja@gmail.com** and **git config −−list**.
- The git config command as shown can be applied on *local(repository)*, *global(OS user)* and *system(machine)* levels.

2.3 Local Workflow of Git

Locally, the git workflow consists of:

1. Editing files in the working directory.
2. Adding edited files to the staging area.
3. Saving these changes to a git repository with a git commit.

Have a look at figure 2 below.

2.4 git status

You will be changing the contents of the working directory. You can check the status of those changes with this command.

2.5 git add filename

We can add a file or many files together to the staging area with this command. As an advanced option **git add -A** can be used to add all edited files to the staging area. As shown in the figure 3 below.

2.6 git diff filename

Since the file is in the 'staging area' and therefore being tracked, we can check the differences between the *working directory* and the *staging area* with this command.

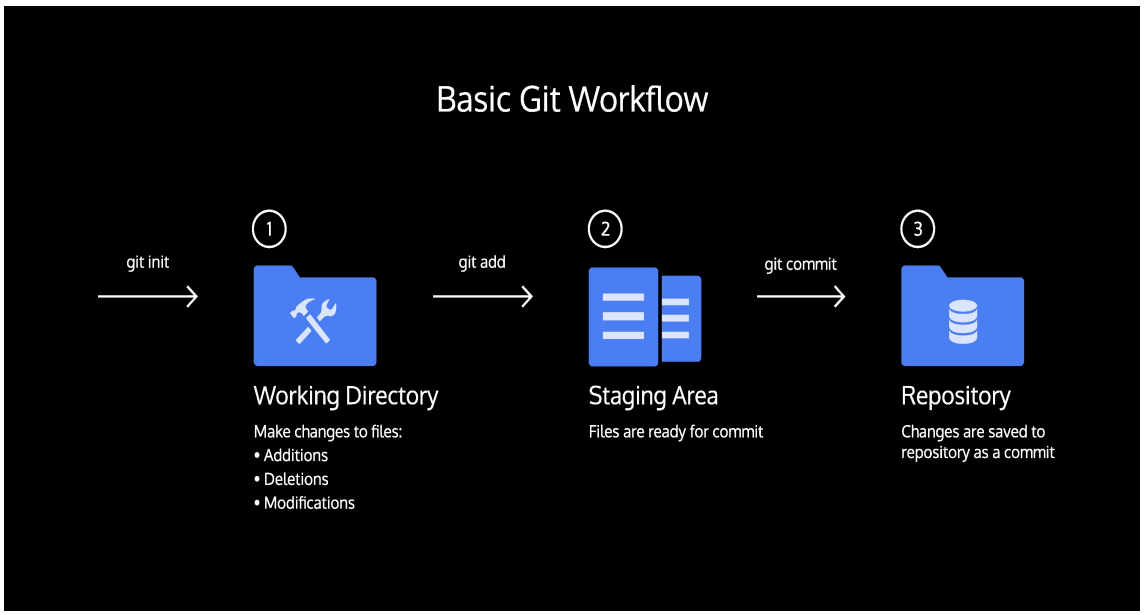

Figure 2: Local Workflow of Git.

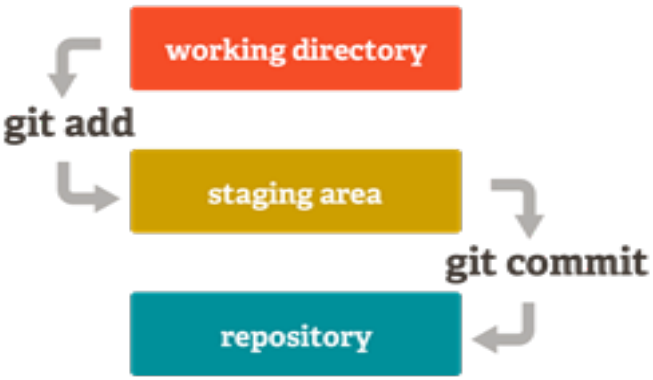

Figure 3: How Git Add works.

2.7 git commit -m "Commit Message"

A commit is the last step in our basic git workflow as shown above in figures 2 & 3 . A commit permanently stores changes from the staging area into the repository.

3 Branching & Merging

3.1 git branch

You can use the command below to answer the question: "which branch am I on?"

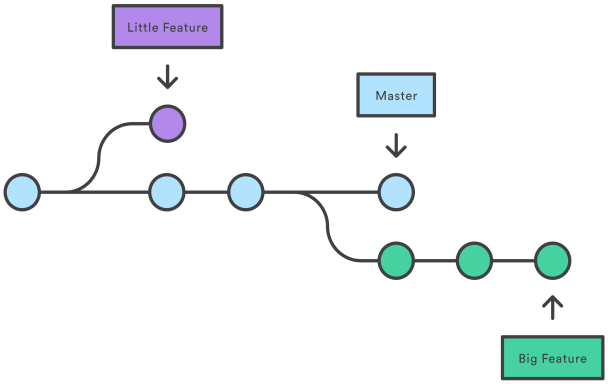

Figure 4: Git Branch.

3.2 git branch new_branch

To create a new branch, use this command. Also, new branch name cannot have white-spaces in between. New Branch is a different version of the Git project. It contains commits from Master but also has commits that Master does not have. It will have no effect on the master branch until you're ready to merge it to the master branch. As shown in figures 4 & 5.

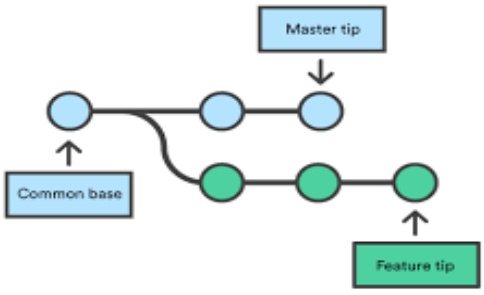

Figure 5: Making a new branch.

3.3 git checkout branch_name

You can switch to the new branch with this command. Once you switch branch, you can now make commits on that branch that have no impact on master.You can continue your workflow, while master stays intact! As shown in figure 5.

3.4 git checkout -b test feature

This command creates a test branch from your feature branch. This command can be used to create branches from any of your existing branches in git.

3.5 git merge branch_name

We can merge all the changes of the new branch into the master branch by *merging* the branch into master. In order to merge, first switch to master(the receiver branch) and then use the above command to merge the branch_name(giver branch) into master. As shown in figure 6.

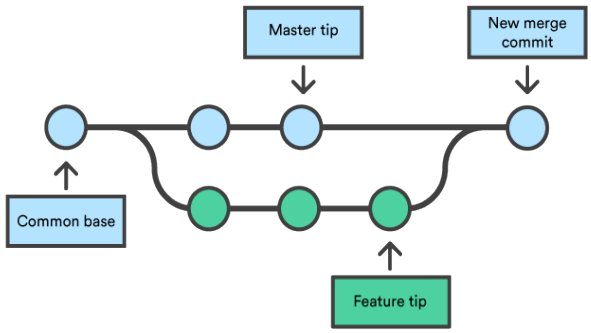

Figure 6: The created branch is now merged back into the master with a new merge commit.

3.6 Handling Merge Conflicts

- The merge was successful because master had not changed since we made a commit on the new branch. Git had to simply update master with changes on the new branch.
- What would happen if there was a commit on master before you merged the two branches? Furthermore, what if that commit altered the same exact part you worked on in the new branch?

Here the Trouble begins! As shown in Figure 7.
There are commits on separate branches that alter the same part of the project in conflicting ways. Now, when you try to merge your branch into master, git will not know which version of the file to keep!

Git therefore uses markings to indicate the master version(HEAD) of the file and the branch version of the file. You can now edit the content according to what you want to keep and make a new commit as shown in figure 8.

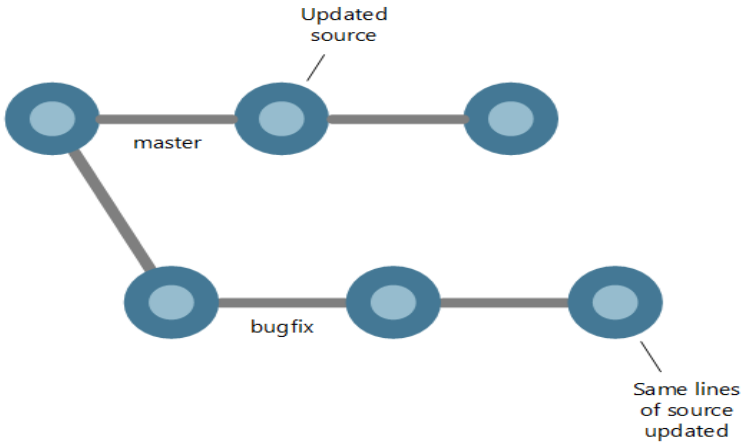

Figure 7: An Example of Merge Conflict.

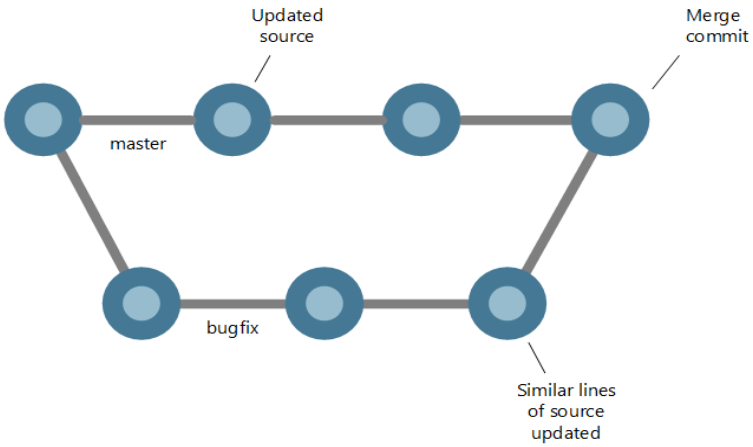

Figure 8: Merge Conflict Resolved with a new Merge Commit.

3.7 git branch -d branch_name

After a branch has been integrated into master, it has served its purpose and can be deleted. This command will delete the specified branch from your git project.

3.8 git branch -D branch_name

This command will allow you to delete the specified branch even if it is not merged into master. This can be useful when you create certain branches for testing/experimentation and then want to delete those branches without merging them into master.

4 Collaborating with a Remote Project

Reference: https://git-scm.com/book/en/v2/Git-Branching-Remote-Branches

4.1 git clone remote_location clone_name

- In order to get your own replica of the remote repository, you will need to clone it with this command.
- "remote_location" tells Git where to go to find the remote. This could be a web address, or a file-path.
- "clone_name" is the name you give to the directory in which Git will clone the repository. This folder/directory is your *local copy* of the Git project repository. If you commit changes to the project here, your team will not know about them.

4.2 git remote -v

Behind the scenes when you clone a remote repository, Git gives the remote address the name 'origin', so that you can refer to it more conveniently. You can see a list of a git project's remotes with this command.

4.3 git fetch

- An easy way to see if changes have been made to the remote and bring the changes down to your local reference with this command
- This command will not merge changes from the remote into your local repository. It brings those changes to your system onto what's called a 'local reference of remote branch'. - for example 'origin/master'.
- These are also called remote-tracking branches and are explained in detail in the the following section.

4.4 Tracking Remote Branches

- 'Remote-tracking branches' (see figure 10) are references to the state of branches on your remote repositories.
- They're local references to the state of remote branches that you can't move. Git moves them for you whenever you do any network communication to make sure they accurately represent the state of your remote repository.
- Remote-tracking branches take the form (remote name)/(branch name). For instance origin/master.

so let's look at an example. Let's say you have a Git server on your network at gitlab.com. If you clone from this -

- Git automatically names it origin for you, pulls down all its data.
- Git creates a pointer to where its master branch is, and names it origin/master locally.
- Git also gives you your own master branch starting at the same place as origin's master branch, so you have something to work from. See Figures 9 and 10.

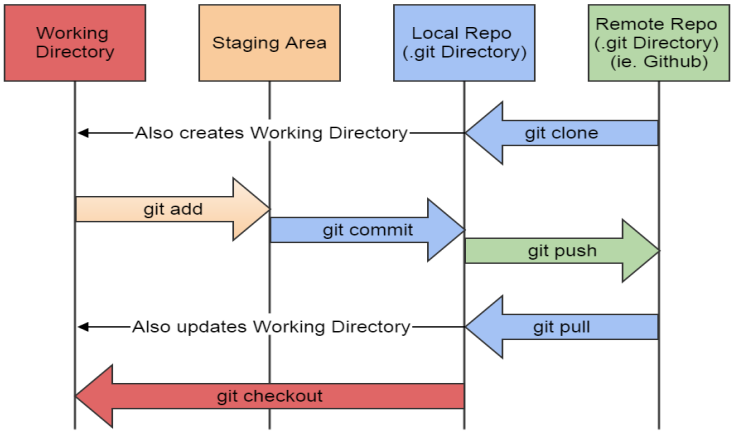

Figure 9: git workflow with remote repository.

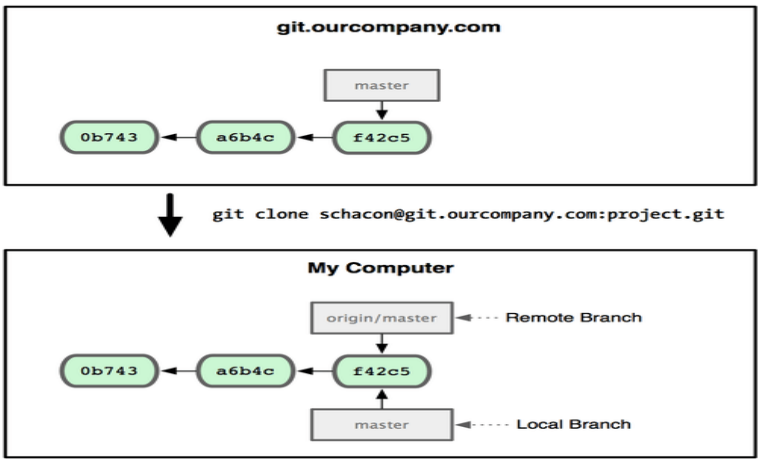

Figure 10: A Git clone gives you your own master branch and origin/master pointing to origin's master branch.

- If you do some work on your local master branch and in the meantime, someone else pushes to gitlab.com and updates its master branch, then your histories move forward differently. Also, as long as you stay out of contact with your origin server, your origin/master pointer doesn't move. See Figure 11.
- To synchronize your work, you run a **git fetch origin** command. This command fetches any data from the origin(gitlab) that you don't yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position. See Figure 12.

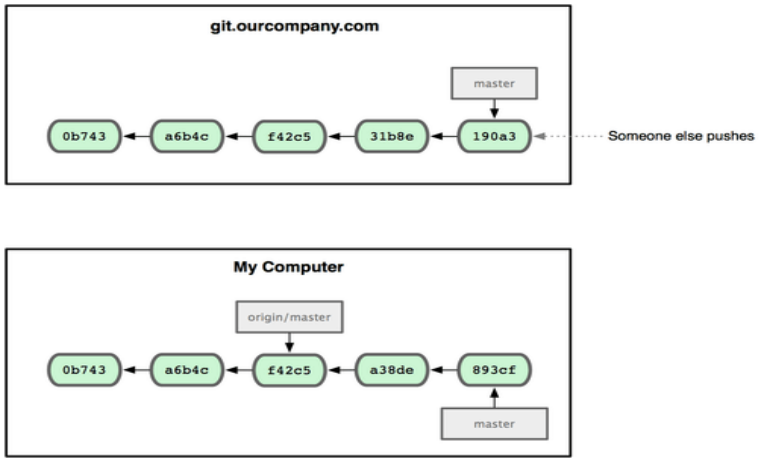

Figure 11: Working locally and having someone push to your remote server makes each history move forward differently.

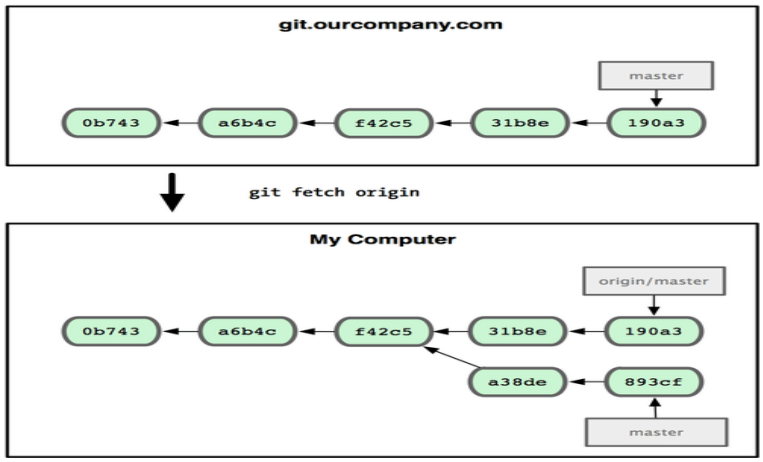

Figure 12: The git fetch command updates your remote references.

4.5 git push

- Your local branches aren't automatically synchronized to the remote. You have to explicitly push the branches you want to share.
- That way, you can use private branches for work you don't want to share, and push up only the topic branches you want to collaborate on.
- If you have a 'issue' branch that you want to work on with others, you can push it up by running **git push origin issue**.
- You can instead run **git push origin issue:awesomebranch** to push the local "issue" branch into remote with a different name. (awesome branch in this case)
- The next time one of your collaborators fetches from the server(remote), they will get a reference to where the server's version of issue is, under the 'remote-tracking branch' - origin/issue.

It is important to note that when you do a fetch that brings down new remote branches, you don't automatically have local, editable copies of them. In other words, in the above case, you don't get a new "issue" branch — you only have an origin/issue pointer (reference) that you can't modify.

- If you want your own local "issue" branch that you can work on, you can base it off your remote branch by running **git checkout -b issue origin/issue**.
- This gives you a local branch that you can work on that starts where origin/issue is.

4.6 Tracking Branches

- Checking out a local branch from a remote branch automatically creates what is called a 'tracking branch'. Tracking branches are local branches that have a direct relationship to a remote branch.
- If you're on a local tracking branch and type 'git pull', Git automatically knows which server to fetch from and which branch to merge in.
- When you clone a repository, it generally automatically creates only a master branch that tracks origin/master.
- If you already have a local branch and want to set it to track a remote branch, or want to change the upstream branch you're tracking. You can explicitly set it at any time using **git branch -u origin/issue**.
- If you want to see what tracking branches you have set up, you can use the command **git branch -vv**. This will list out your local branches with information including what each branch is tracking and if your local branch is ahead, behind or both.
- If you want totally up to date ahead and behind numbers, you'll need to fetch from all your remotes right before running this. You could do that like this: **git fetch −−all; git branch -vv**.

4.7 git pull

- While the 'git fetch' command will fetch all the changes on the server that you don't have yet, it will not modify your working directory at all. It will simply get the data for you and let you merge it yourself.
- However, there is a command called 'git pull' which is essentially a 'git fetch' immediately followed by a 'git merge' in most cases.

4.8 Deleting remote branches

- Suppose you're done with a remote branch — say you and your collaborators are finished with a feature and have merged it into the master.
- You can delete a remote branch using the –delete option to git push. If you want to delete your "issue" branch from the server, you run the following: **git push origin −−delete issue**.

5 Backtracking: Undoing Commits & Changes

5.1 git log

Often with Git, you'll need to refer back to an earlier version of a project. Commits and their related information is stored chronologically in the repository and can be viewed with this command.

5.2 git show HEAD

In Git, the commit you are currently on is known as the HEAD commit. In many cases, the most recently made commit is the HEAD commit. The output of this command will display everything the 'git log' command displays for the HEAD commit, plus all the file changes that were committed.

5.3 git checkout HEAD filename

What if you decide to make a change in the working directory, but then decide you wanted to discard that change? This command will restore the file in your working directory to look exactly as it did when you last made a commit.

5.4 git reset HEAD filename

The file change you don't want to include it in the commit, We can unstage that file from the staging area using this command. This command resets that file in the staging area to be the same as it was in the HEAD commit. It does not discard file changes from the working directory, it just removes them from the staging area.

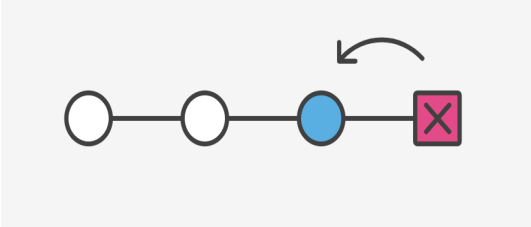

Figure 13: The last commit is no longer part of your project.You have in essence rewound the project's history.

5.5 git reset commit_SHA

Git enables you to rewind to a part of the project before you made the wrong turn. This command works by using the first 7 characters of the SHA of a previous commit. HEAD is now set to that previous commit. The commits that came after the one you reset to are gone as show in in figure 13 above.

6 Some Pro Stuff

6.1 Merge Vs Rebase

In Git, there are two main ways to integrate changes from one branch into another: the 'merge' and the 'rebase'. In this section you'll learn what rebasing is, how to do it, why it's a pretty amazing tool, and in what cases you won't want to use it - https://git-scm.com/book/en/v2/Git-Branching-Rebasing.

6.2 merge conclicts after rebase

6.3 .gitignore

Ignored files are usually build artifacts and machine generated files that can be derived from your repository source or should otherwise not be committed. Ignored files are tracked in a special file named .gitignore that is checked in at the root of your repository. There is no explicit git ignore command: instead the .gitignore file must be edited and committed by hand when you have new files that you wish to ignore. .gitignore files contain patterns that are matched against file names in your repository to determine whether or not they should be ignored.

6.4 git stash

6.5 squashing commits

The End.

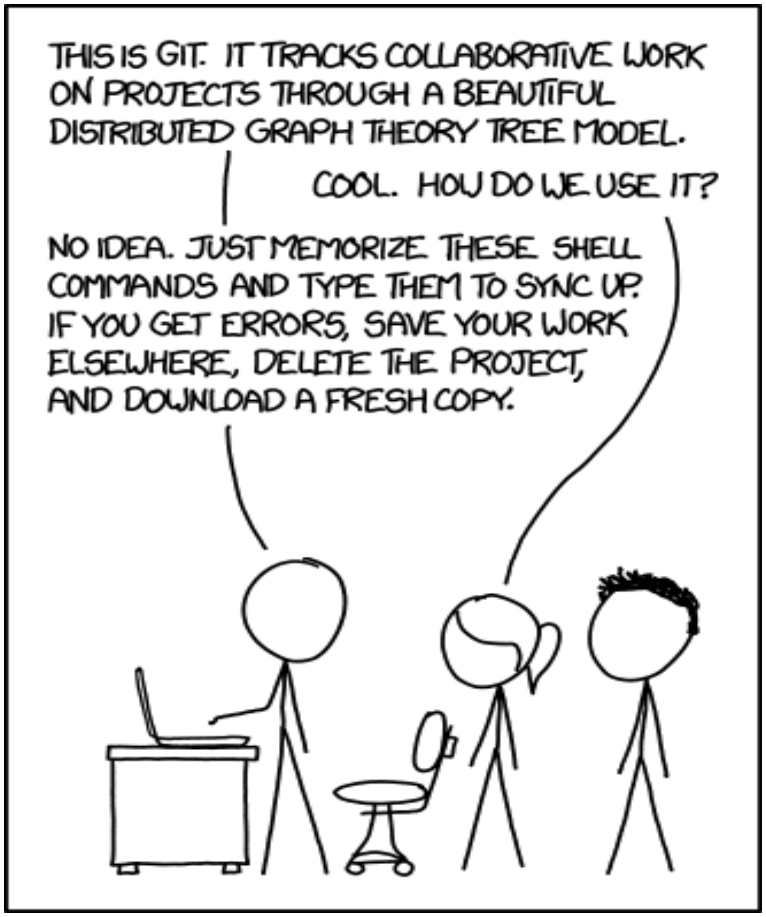

Figure 14: Have Fun With Git!