# Git

*Information found from pages 1-3 comes from the book “Pro Git” produced under the Creative Commons AttributionNonCommercial-ShareAlike 3.0 Unported License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA*

*A copy of the entire book can be found https://progit2.s3.amazonaws.com/en/2015-05-08-a1136/progit-en.482.pdf*

#### What is Git?

Git is a **Version Control System** (VCS) that has the ability to track the different versions of any given document on your computer. If you are a graphic or web designer and want to keep every version of an image or layout (which you would most certainly want to), a Version Control System (VCS) is a very wise thing to use. It allows you to revert files back to a previous state, revert the entire project back to a previous state, compare changes over time, see who last modified something that might be causing a problem, who introduced an issue and when, and more. Using a VCS also generally means that if you screw things up or lose files, you can easily recover. In addition, you get all this for very little overhead.

#### A Brief History

**Local Version Control Systems**

Many people’s version-control method of choice is to copy files into another directory (perhaps a time-stamped directory, if they’re clever). This approach is very common because it is so simple, but it is also incredibly error prone. It is easy to forget which directory you’re in and accidentally write to the wrong file or copy over files you don’t mean to.

To deal with this issue, programmers long ago developed local VCSs that had a simple database that kept all the changes to files under revision control. One of the more popular VCS tools was a system called RCS, which is still distributed with many computers today. Even the popular Mac OS X operating system includes the rcs command when you install the Developer Tools. RCS works by keeping patch sets (that is, the differences between files) in a special format on disk; it can then re-create what any file looked like at any point in time by adding up all the patches.

**Centralized Version Control Systems**

The next major issue that people encounter is that they need to collaborate with developers on other systems. To deal with this problem, Centralized Version Control Systems (CVCSs) were developed. These systems, such as CVS, Subversion, and Perforce, have a single server that contains all the versioned files, and a number of clients that check out files from that central place. For many years, this has been the standard for version control.

This setup offers many advantages, especially over local VCSs. For example, everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it’s far easier to administer a CVCS than it is to deal with local databases on every client.

However, this setup also has some serious downsides. The most obvious is the single point of failure that the centralized server represents. If that server goes down for an hour, then during that hour nobody can collaborate at all or save versioned changes to anything they’re working on. If the hard disk the central database is on becomes corrupted, and proper backups haven’t been kept, you lose absolutely everything – the entire history of the project except whatever single snapshots people happen to have on their local machines.

Local VCS systems suffer from this same problem – whenever you have the entire history of the project in a single place, you risk losing everything.

**Distributed Version Control Systems**

This is where Distributed Version Control Systems (DVCSs) step in. In a DVCS (such as Git, Mercurial, Bazaar or Darcs), clients don’t just check out the latest snapshot of the files: they fully mirror the repository.

Thus if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it. Every clone is really a full backup of all the data. Furthermore, many of these systems deal pretty well with having several remote repositories they can work with, so you can collaborate with different groups of people in different ways simultaneously within the same project. This allows you to set up several types of workflows that aren’t possible in centralized systems, such as hierarchical models.

#### Snapshots, Not Differences

The major difference between Git and any other VCS (Subversion and friends included) is the way Git thinks about its data. Conceptually, most other systems store information as a list of file-based changes. These systems (CVS, Subversion, Perforce, Bazaar, and so on) think of the information they keep as a set of files and the changes made to each file over time.

Git thinks of its data more like a set of snapshots of a miniature file system. Every time you commit, or save the state of your project in Git, it basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot. To be efficient, if files have not changed, Git doesn’t store the file again, just a link to the previous identical file it has already stored. Git thinks about its data more like a stream of snapshots

#### Git Has Integrity

Everything in Git is check-summed before it is stored and is then referred to by that **checksum**. This means it’s impossible to change the contents of any file or directory without Git knowing about it. This functionality is built into Git at the lowest levels and is integral to its philosophy.

You can’t lose information in transit or get file corruption without Git being able to detect it. The mechanism that Git uses for this checksumming is called a **SHA-1 hash**. This is a 40-character string composed of hexadecimal characters (0–9 and a–f) and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

*24b9da6552252987aa493b52f8696cd6d3b00373*

You will see these **hash values** all over the place in Git because it uses them so much. In fact, Git stores everything in its database not by file name but by the hash value of its contents.

#### Git Generally Only Adds Data

When you do actions in Git, nearly all of them only add data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As in any VCS, you can lose or mess up changes you haven’t committed yet; but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see “Undoing Things”.

#### The Three States

Now, pay attention. **This is the main thing to remember about Git if you want the rest of your learning process to go smoothly**. Git has three main states that your files can reside in: committed, modified, and staged.

**Committed** means that the data is safely stored in your local database.

**Modified** means that you have changed the file but have not committed it to your database yet.

**Staged** means that you have marked a modified file in its current version to go into your next commit snapshot.

This leads us to the three main sections of a Git project: the Git directory, the working directory, and the staging area.

#### The Git directory

The **Git directory** is where Git stores the metadata and object database for your project. This is the most important part of Git, and it is what is copied when you clone a repository from another computer.

**The working directory** is a single checkout of one version of the project. These files are pulled out of the compressed database in the Git directory and placed on disk for you to use or modify.

**The staging area** is a file, generally contained in your Git directory, that stores information about what will go into your next commit. It’s sometimes referred to as the “index”, but it’s also common to refer to it as the staging area.

**Basic Git Workflow**

The basic Git workflow goes something like this: ­­

1. You modify files in your working directory.

2. You stage the files, adding snapshots of them to your staging area.

3. You do a commit, which takes the files as they are in the staging area and stores that snapshot permanently to your Git directory.

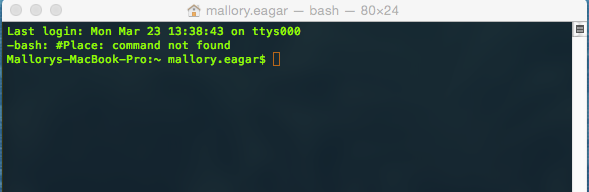
If a particular version of a file is in the Git directory, **it’s considered committed**.

If it’s modified but has been added to the staging area, **it is staged**.

And if it was changed since it was checked out but has not been staged, **it is modified**.

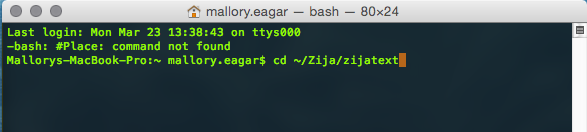
## [40 Terminal Tips and Tricks You Never Thought You Needed](http://computers.tutsplus.com/tutorials/40-terminal-tips-and-tricks-you-never-thought-you-needed--mac-51192) https://computers.tutsplus.com/tutorials/40-terminal-tips-and-tricks-you-never-thought-you-needed--mac-51192

1) When you open your terminal (on Mac):



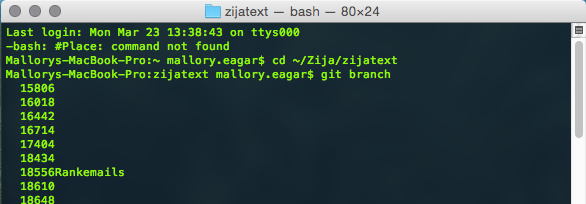
2) Your first step will be to **Change the Directory** to one of the servers (or repositories) we use:

cd ~/Repository/path/[REPOSITORY NAME]



3) Once there, you can either work on a ‘**branch’** you’ve already created or **create a new branch**, for a new issue.

a) To see branches you’ve been working on, type: git branch



b) To **create a new** branch, type: git checkout –b [branch name- that is the issue number]

git checkout -b [BRANCH NAME]

c) To transfer to a different branch:

git checkout [BRANCH NAME]

When you have finished working on a file and you now want to “commit”

d) Type the following on Git

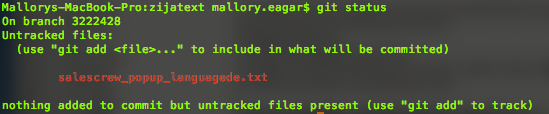
git status

Macintosh HD:Users:mallory.eagar:Desktop:Screen Shot 2015-03-24 at 10.34.10 AM.png

#### Uploading files.

At this point, a list of files will appear telling you what needs to be staged so you can commit it.

1->So, you make a change to a file and save it: type git status to see which files have been changed:



2->then, type:

git add --allMacintosh HD:Users:mallory.eagar:Desktop:Screen Shot 2015-03-24 at 10.58.09 AM.png

( note if you don’t want to add file (it’s the wrong file, added by mistake, etc) you can undo this by typing:

git checkout [filename] )

3-> commit:

git commit

Macintosh HD:Users:mallory.eagar:Desktop:Screen Shot 2015-03-24 at 10.59.46 AM.png

This will bring you to a screen where you must use VI commands:

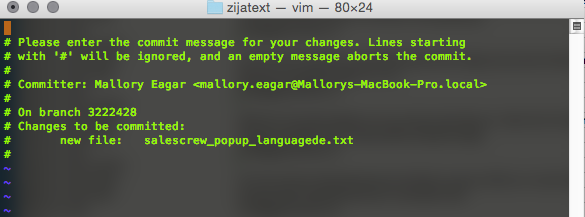
dw to delete the # in front of the files changed

I to insert a brief descriptive message

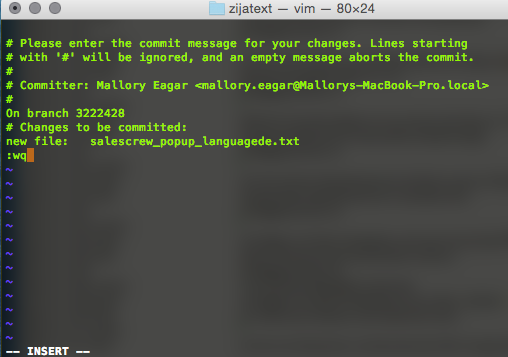
:wq to close out the window and go back to the main page

Any line with a preceding # will be hidden / removed from the commit message

Before:



After:



This will then return you to the main page with this message:

4-> Next step is to push it to the repository. Type:

git push

#### Other things:

To remove a file from the database:

rm [file name]

To see files you adjusted in the past:

git log

To Rename your branch

git branch -m [newname]

To Delete a branch

git branch -d [branch name]

If you push a file and it doesn’t work (but doesn’t conflict out, You’ll know the difference)

rm -rf ~/.ssh/known\_hosts

this command removes a file called known\_hosts on your local machine. .ssh is the type of way you are logging on to your a server.. and instead of running through sequence of reinputting your key every time you login to the server.. known\_hosts remember's that you did already

#### CONFLICTS:

**NEVER WORK ON THE SAME FILE USING TWO DIFFERENT BRANCHES**- it will conflict when you push your branch, not only for you, **but for everyone** **else**. (I’ve done this before…) Wait until a file has been pushed live (master) to work on it using a different branch.