Git FAQ

Created 2021-02-04 by J.D. Muehlbauer (jmuehlbauer@usgs.gov)

1. What is Git?
   Git is version tracking software. It is a handy way of controlling changes to documents, code, software, and data without making full backups. Rather than having several versions of a large document floating around, for instance, Git handles the version control incrementally and makes it easy to revert to prior versions if needed.

2. Why should I use Git?
   Although Git is designed for version control and it is very good at that, this benefit tends to be secondary for me. My major motivation in using Git is for its implementation with GitHub and GitLab, which allow hosting of Git repositories ("repos") and therefore file storage online. These platforms make it exceedingly easy to share and distribute documents, code, software, and data, and for multiple people to work on them collaboratively.

3. What are GitHub and GitLab?
   GitHub and GitLab are online platforms that utilize Git. There are few meaningful differences between how you interact with these two platforms; it's kind of like using Firefox vs. Chrome. The main differences in why to use one over the other are that GitHub is shared publicly, unless you have paid for private repos (we haven't). GitLab, in contrast, has a dedicated USGS site (code.usgs.gov), which is private to us and DOI collaborators. This is good and bad. The general public can't see our in-progress work on GitLab, which is generally good, but collaborators such as AZGFD also can't see anything we put there, which is generally bad. In practice when I want to share software like the foodbase package, I create a repo on GitHub. When I want to share code I'm working on just within our lab group, I put it on GitLab.

4. Do I just work directly on GitHub or GitLab online?
   This might be possible, but is not ideal in practice. What I do is create local git repos on my computer, then "clone" those repos to a "remote origin" on GitLab or GitHub. So I always have what I need locally and online, and I use "push" and "pull" routines to make sure I'm always working with the latest versions.

5. How do I get started?
   a. Have IT install for you the latest Git GUI from https://github.com/git-for-windows/git/releases. The default options are likely fine. Make sure Git Bash is installed as an option.
   b. Git plays nicely with RStudio if you use that (I'm old school and don't). Mike Dodrill would be a good resource to ask about RStudio and Git integration.
   c. To create a repo, I find it easiest to go to whatever folder I want to make into a Git repo, and right click in the folder, then select "Git Bash here". This opens a Git window and

automatically sets the directory to that location. You can use the “cd” command to navigate to that directory from the Git GUI, but why would you?

d. To create a git repo at that location, in the Git GUI just type “git init” (no quotes).
e. To stage every file in the repo, type “git add -A” (no quotes). If you only want to stage some files, you can type the file or subfolder names explicitly. You can also start typing a file name then hit the tab button and Git will fill out the rest of the file name for you. You generally only do this when creating a new repo or when creating new files.
f. If there are files or subfolders you know you will want to ignore and not want Git to track, then you can create a file in the repo called “.gitignore” Open that file in Notepad or similar and type in on a new line every file or subfolder you want Git to ignore. This is optional.
g. It is good practice to create a file called “README.md” in your repo. This readme tells users what the repo contains and what it is used for. GitLab and GitHub display this readme text prominently on the front page of the repo. You can also use what is called markdown text to include bolding, italics, emoji, etc.
h. To commit files, type ‘git commit -a -m “some message”’ (without the outer quotes). “-a” means commit all the files Git is tracking, “-m” tells Git to include the message that follows in the commit.” Common practice on your first commit, for instance, would be ‘git commit -a -m “Initial commit.”’ If you use the -a shortcut then you don’t have to do the “git add” step above every time you commit. This is optional though, and you could do “git add” and ‘“git commit -m “some message”’ separately. I rarely do this, but the latter case is how you would commit only some, rather than all, of the files you have changed after your last commit.
i. To set up a remote repo on GitLab or GitHub, just click “New project” on one of those platforms and follow the instructions. This only has to be done once. I suggest you do the bare minimum and not initialize a README or any other stuff on those platforms. Do it locally instead.
j. Once you set up the remote repo, type “git remote add origin https://whatever” (no quotes). The https link will be the link of your repo, which GitHub and GitLab will provide you when you set it up. This only has to be done once.
k. To push to the remote repo, type “git push origin master”. You may get SSL certificates errors the first time you try to do this, if so ask me for help. You will likely need to authenticate the first time you do this as well. On GitLab authentication is your USGS password and PIN, on GitHub it is whatever your login is for that site.
l. If you ever want to revert back to an old version, you can use git checkout to compare files in the Git GUI. I find I easier to use the comparison options on the GitLab/GitHub platforms though, which make it very easy to compare changes. If you revert to an old file using GitHub/GitLab, then do a pull when you are done to get your local repo back to the same state.

6. What jargon and commands do I need to know to use Git?
   When using the Git GUI, nearly every command you type will need to start with "git " (no quotes. For instance, if I want to see the status of files in my Git repo, I would type "git status". Here are some common commands and other jargon:

   repo: Short for "repository." In a Windows framework this is your main folder in which files and subfolders are housed, although the name you give the repo needn't be identical to the Windows folder name.

   commit: A commit is a point in time at which you tell Git to save your work. There is no downside to committing as often as you like, although it takes a little longer than hitting ctrl+s to save a file. Common practice is to commit at the end of the day and whenever you cross a milestone ("I finished a chunk of code!", etc), or whenever you add new content ("I added a new spreadsheet."). Best practice is to annotate your commits so if you ever have to go back and revert changes it is easy to figure out which commit you want to go back to.

   push/pull: In Git parlance you either "push" or "pull" to/from a repo to/from a remote origin elsewhere. You can think of pushing commits out to the repo to update everyone, or if you are out of date then pulling commits from a repo to update yourself. Unless I am actively collaborating with someone I only push commits (as in, I make changes on my local repo, then push them to GitHub).

   stage: Git keeps track of all the files in your repo (unless you explicitly tell it to ignore a file or folder by calling it out in a .gitignore file you create). But it does not commit all these changes unless you "stage" them. In other words, you have to explicitly tell Git which files you want to update during a commit, and which ones you want to remain "unstaged". Often if I am working on multiple files within a repo I may want to commit only one file that I know is in ship-shape, and leave the others unstaged so I don't end up with a bunch of commits of files in known "bad" condition. Git allows you to do this.

   remote origin: Git's way of talking about GitHub or GitLab repos. You set the remote origin once when you initialize a local git repo, and then push or pull from that remote.

   master/branch/fork/checkout: One of the cool things about Git is that it allows you to create "parallel universes" of code within the same repo. Say, for instance, that I created a repo and was working on it alone for a while, but am now working with a colleague is working on the same code. I would be working from a "master", but we may want to create branches for our code (called "forking" a repo) so that we can each work and commit independently for a while, without influencing the workflow or files on the other branch. Then when my colleague has something they think is good, they submit a "pull request" for me to "merge" their branch with my master branch. I can "checkout" both branches, to compare differences. If I like what they have done, then it is simple for me to update my "master" branch. This sounds more complicated than it is, and the reality for our near-term usage is that you are likely to only be working form the master.