Git: The Ultimate Tool for Collaborative Coding

Cameron Rider

December 3, 2017

If you've ever browsed the web looking to try out some software, perhaps something from a more niche field or a program that hadn't been fully finished yet, you may have stumbled upon a site called GitHub. Delving into a GitHub repository, if you're unfamiliar with how the site works, can seem strange in a lot of ways. To act out a few: I just wanted the program this guy said he was working on, so why is this website giving me access to all his code with no way to run it? I've been searching forever and I still can't find the download link! Why do I have to scroll past ALL of these files just to find the instructions?

Believe it or not, these evidently disastrous design mistakes are completely intended. As the website's main page prominently states, GitHub is built for developers (as in, not consumers). It enables programmers who are interested at looking at your code to browse it with ease, and if they're so inclined to try and improve upon what you've done, they can work on it remotely without directly changing your code. That's, of course, just scraping the surface of what you can do. It turns out programmers have made a lot of nifty tools for themselves over the years. As an online interface for grabbing files, GitHub is just the shiny wrapping paper . for the powerhouse that allows developers across the world to work together, Git.

If you've never worked with Git before, or only use it for, say, turning in your assignments, you may be asking: what's so special about Git that makes it worth my time? The answer is, everything. Not only does Git allow you to access versions of your programs as they change over time, it gives you the unique ability to code collaboratively with others without any of the hassle of copy-pasting. How does that work? Let me show you, right after we go through…

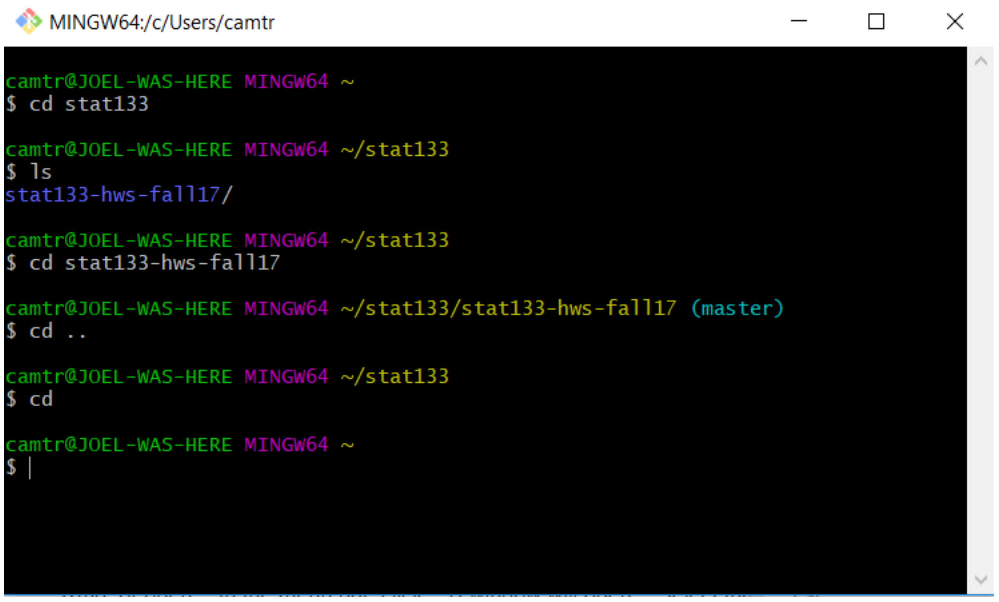
Installing Git

On the chance that you don't have Git on your computer, the best place to start is the Git website. Simply click whichever operating system is on your computer and follow the instructions until it's done.

There's one small catch: Git is not a traditional program in the sense that you open it up on your computer. If you'd like to do so, you can use a GUI (I won't go into these, but feel free to look them up) - however, most simply use their computer's command line to run git commands. These can be easily accessed on Mac and Windows by opening up Terminal or Command Prompt respectively. Alternatively, if you're a Windows users, Git will also automatically install a program called Git Bash which effectively does the same thing (and looks a little nicer!).

If you're unused to navigating through your files with your command line, here are a few quick pointers:

- When you first open Terminal/Command Prompt/Git Bash, you will automatically start in your User folder (in my case, C:). You can also open up multiple windows of your command line program if you want to visit different folders at the same time.
- If you want to enter a folder, the command is **cd *name of folder**". If you're ever confused about where you are, you can always open up a window with all your folders and look at how your computer is set up. The path that you're in is also displayed above the command line.
- To go to the parent folder of the one you're in, the command is **cd ..**. To go all the way back to your User folder, simply type in **cd**.
- If you want to view a quick list of everything in the current folder without opening up your file finder, the command is **ls**.


```
MINGW64:/c/Users/camtr

camtr@JOEL-WAS-HERE MINGW64 ~
$ cd stat133

camtr@JOEL-WAS-HERE MINGW64 ~/stat133
$ ls
stat133-hws-fall17/

camtr@JOEL-WAS-HERE MINGW64 ~/stat133
$ cd stat133-hws-fall17

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ cd ..

camtr@JOEL-WAS-HERE MINGW64 ~/stat133
$ cd

camtr@JOEL-WAS-HERE MINGW64 ~
$ |
```

Now, we can finally start with Git!

All Things Git

Before we do anything else, we have to set up a git repository. First, navigate to the folder with the code that you'd like to keep track of - in my case, it's all of my course files. Then type **git init**. You should get a message along the lines of:

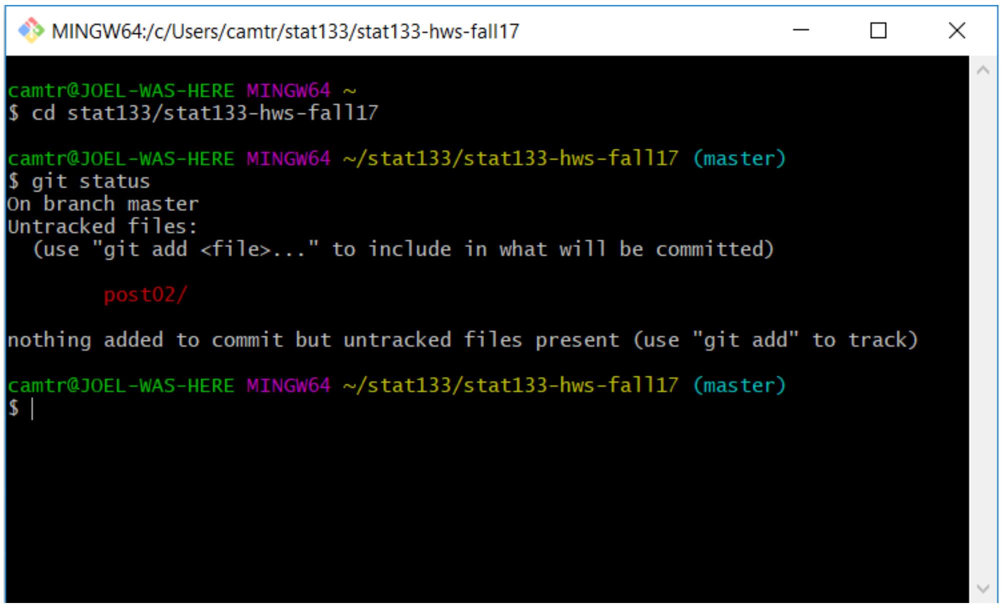
"Initialized empty Git repository in *file directory*/.git/"

If so, congratulations! You did it. If it didn't work, make sure that your folder is not already a git repository. If you did it correctly, your file directory should also have the word master in parenthesis after it - this indicates that the folder is the master branch of your git repository.

There are plenty of benefits to using git before you even connect online! It's easier and more fun to explain in commands and images than text, so let's get started!

git status

Git status allows you to see exactly what's going on in your repository (shows you the status of your repository, if you will). Try typing in **git status** in any folder within your repository. It doesn't actually change anything, but if anything has changed in your repository since the last time you worked with git, it will highlight the folders with changes in bright red:



```
MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17

camtr@JOEL-WAS-HERE MINGW64 ~
$ cd stat133/stat133-hws-fall17

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        post02/

nothing added to commit but untracked files present (use "git add" to track)

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ |
```

git add

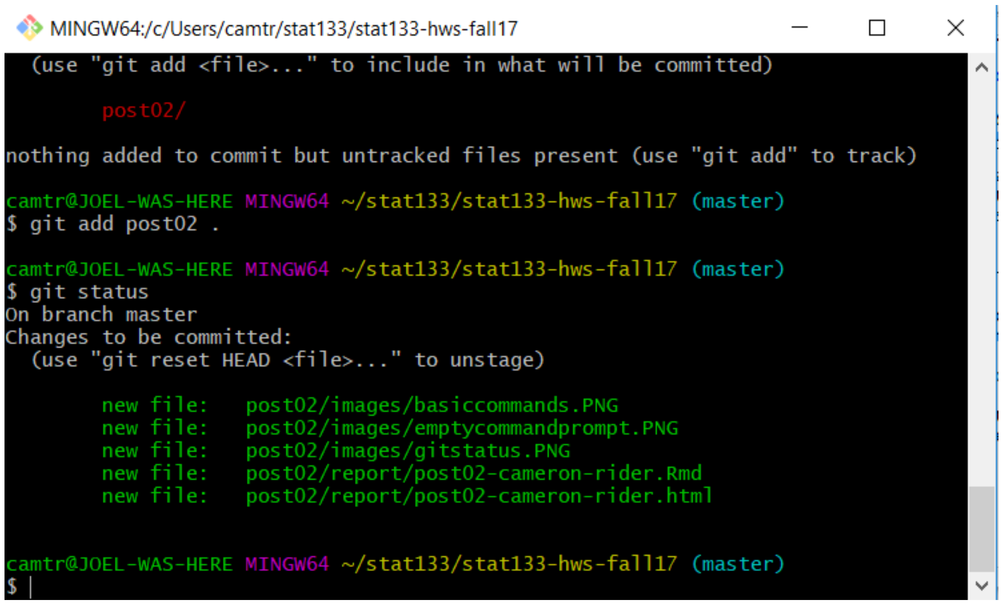
Now, you may have noticed the message "nothing added to commit but untracked files present". Git doesn't automatically grab our files and keep track of them: however, that gives us the flexibility to choose which files we ask Git to track and what versions we want Git to have. Say I've just finished a major part of my stats homework and want git to grab it for me. By typing in:

git add *file name here*

Or, in the case of a folder where you want to track all the files within it:

git add *folder name here* .

That doesn't give us a prompt, but if we use git status again, we see:

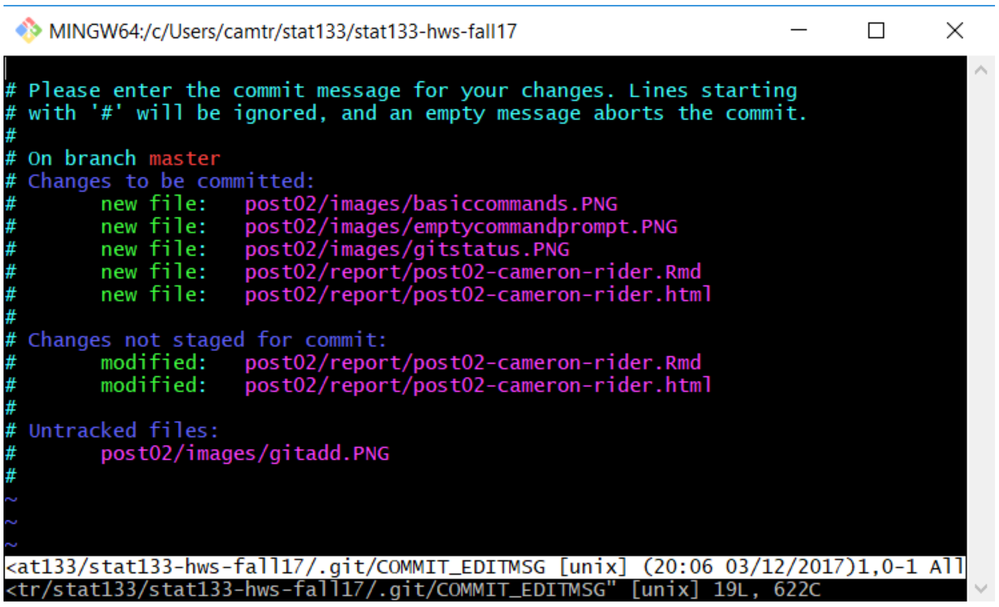


```
MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17
  (use "git add <file>..." to include in what will be committed)

        post02/

nothing added to commit but untracked files present (use "git add" to track)

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git add post02 .

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   post02/images/basiccommands.PNG
        new file:   post02/images/emptycommandprompt.PNG
        new file:   post02/images/gitstatus.PNG
        new file:   post02/report/post02-cameron-rider.Rmd
        new file:   post02/report/post02-cameron-rider.html


camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ |
```

Everything is green now! This means git is now set to track all of the files listed.

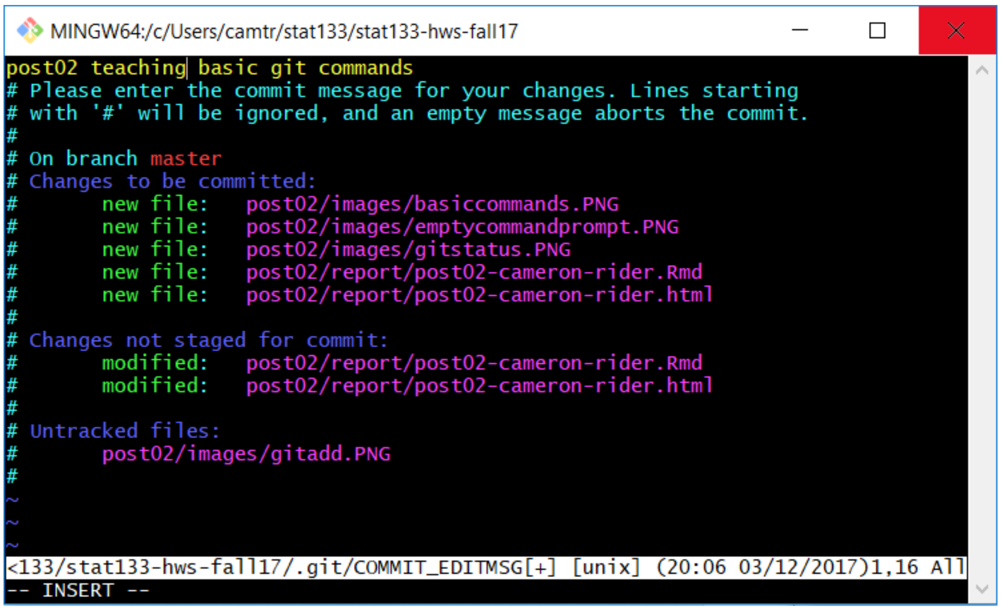
git commit

Now, to set these changes in stone so we can get back to them later if we so desire, we have to "commit" them to the git repository. As you probably figured, the command is **git commit**. If we type that in, we should see something like this:



```
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#       new file:   post02/images/basiccommands.PNG
#       new file:   post02/images/emptycommandprompt.PNG
#       new file:   post02/images/gitstatus.PNG
#       new file:   post02/report/post02-cameron-rider.Rmd
#       new file:   post02/report/post02-cameron-rider.html
#
# Changes not staged for commit:
#       modified:   post02/report/post02-cameron-rider.Rmd
#       modified:   post02/report/post02-cameron-rider.html
#
# Untracked files:
#       post02/images/gitadd.PNG
#
~
~
~
<at133/stat133-hws-fall17/.git/COMMIT_EDITMSG [unix]  (20:06 03/12/2017)1,0-1 All
<tr/stat133/stat133-hws-fall17/.git/COMMIT_EDITMSG" [unix] 19L, 622C
```

Git now expects you to put in a message designating what your commit is all about. This helps you keep track of what your different commits were for at a later time, so even if you do feel the temptation to write "AHHH GIT YOU'RE SO ANNOYING LEAVE ME ALONE", it's probably best to simply give a quick status update instead. Here's a quick example:



```
post02 teaching basic git commands
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
# Changes to be committed:
#       new file:   post02/images/basiccommands.PNG
#       new file:   post02/images/emptycommandprompt.PNG
#       new file:   post02/images/gitstatus.PNG
#       new file:   post02/report/post02-cameron-rider.Rmd
#       new file:   post02/report/post02-cameron-rider.html
#
# Changes not staged for commit:
#       modified:   post02/report/post02-cameron-rider.Rmd
#       modified:   post02/report/post02-cameron-rider.html
#
# Untracked files:
#       post02/images/gitadd.PNG
#
~
~
~
<133/stat133-hws-fall17/.git/COMMIT_EDITMSG[+] [unix]  (20:06 03/12/2017)1,16 All
-- INSERT --
```

Then exit the editor (or, alternatively, don't change anything and exit to abort the commit).

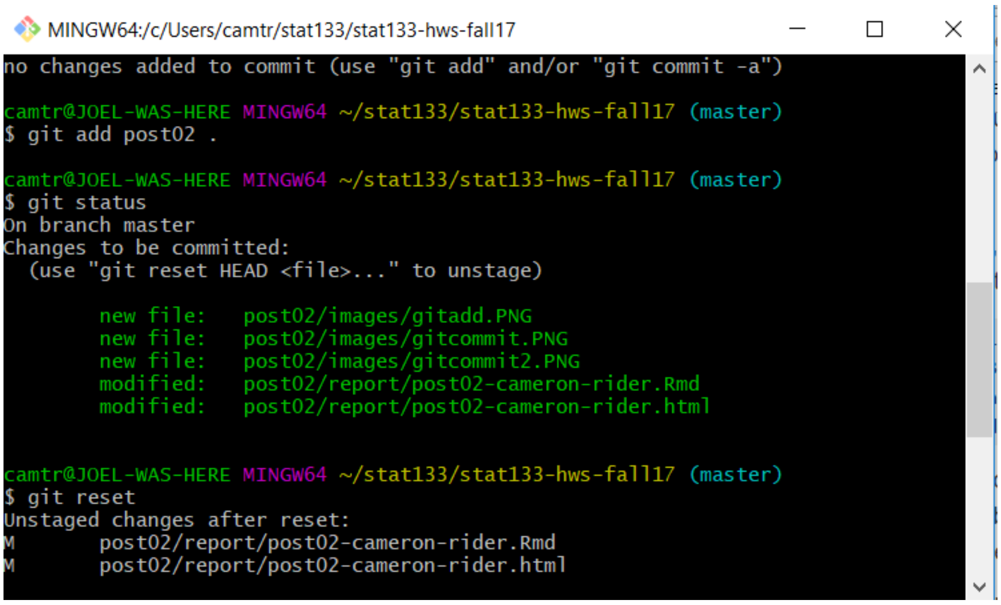
An even easier way to commit is to do it in one line. After adding your files, simply write:

git commit -m "*insert your commit message here*"

Don't forget the quotation marks! When you're done, you should see a confirmation message saying which files changed.

git reset

If you accidentally tell git to keep track of the wrong file, you can restart the process without having to worry about committing the wrong thing by using **git reset**.


MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17

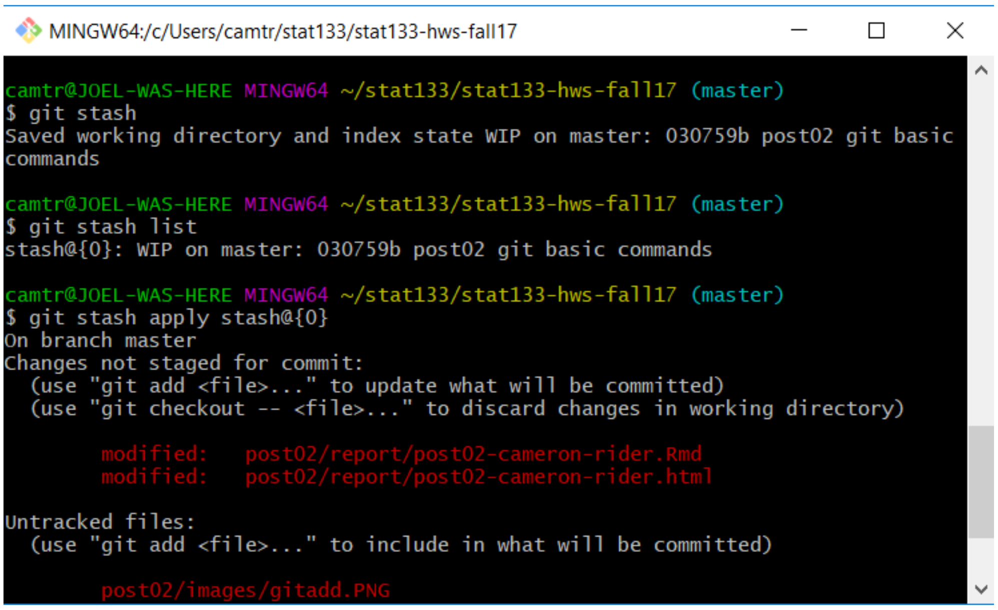
```
no changes added to commit (use "git add" and/or "git commit -a")

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git add post02 .

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   post02/images/gitadd.PNG
        new file:   post02/images/gitcommit.PNG
        new file:   post02/images/gitcommit2.PNG
        modified:   post02/report/post02-cameron-rider.Rmd
        modified:   post02/report/post02-cameron-rider.html


camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git reset
Unstaged changes after reset:
M       post02/report/post02-cameron-rider.Rmd
M       post02/report/post02-cameron-rider.html
```

git stash

say you're still half-finished with some code but want to save it for later for *some reason* (sounds silly, I know, but it'll come in handy in about 2 minutes). For that, we use **git stash**. This grabs all the work we currently have on our computer that hasn't been committed in the repository (make sure you save your work before stashing!) and then reverts our repository back to the last commit we did.

To see what stashes we have, we can use **git stash list**.

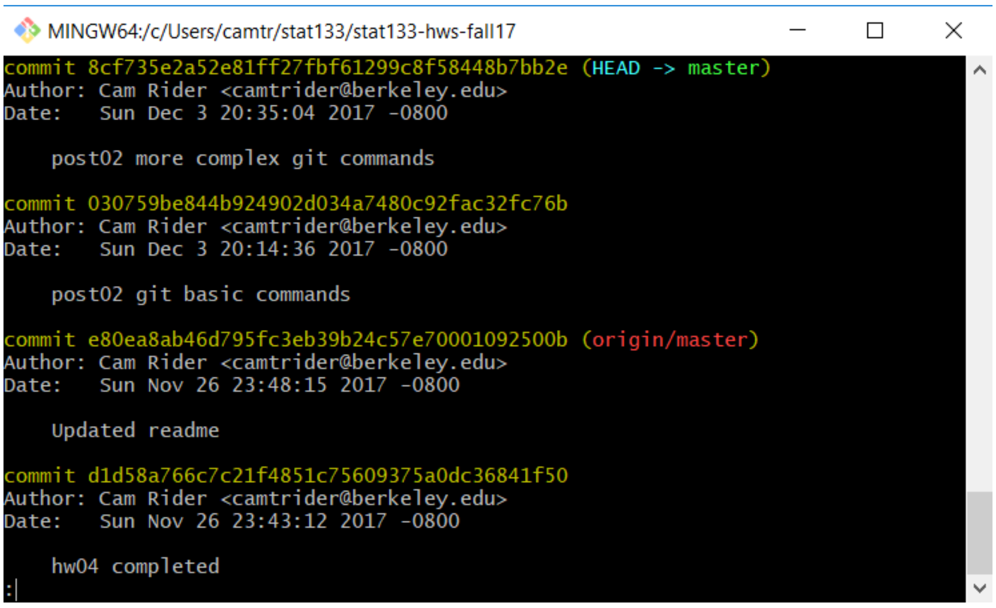
To get back to any stash, we use **git stash apply stash${*insert number here*}**.
If we only have one stash, **git stash apply** will automatically grab that one.


MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17

```
camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git stash
Saved working directory and index state WIP on master: 030759b post02 git basic
commands

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git stash list
stash@{0}: WIP on master: 030759b post02 git basic commands

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git stash apply stash@{0}
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   post02/report/post02-cameron-rider.Rmd
        modified:   post02/report/post02-cameron-rider.html

Untracked files:
  (use "git add <file>..." to include in what will be committed)

        post02/images/gitadd.PNG
```

git log

Now comes some of the real magic of git! There's no point to committing all those past files if we have no idea how to access them, right? Now we can! Let's look at them with **git log**:



```
MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17
commit 8cf735e2a52e81ff27fbf61299c8f58448b7bb2e (HEAD -> master)
Author: Cam Rider <camtrider@berkeley.edu>
Date:   Sun Dec 3 20:35:04 2017 -0800

    post02 more complex git commands

commit 030759be844b924902d034a7480c92fac32fc76b
Author: Cam Rider <camtrider@berkeley.edu>
Date:   Sun Dec 3 20:14:36 2017 -0800

    post02 git basic commands

commit e80ea8ab46d795fc3eb39b24c57e70001092500b (origin/master)
Author: Cam Rider <camtrider@berkeley.edu>
Date:   Sun Nov 26 23:48:15 2017 -0800

    Updated readme

commit d1d58a766c7c21f4851c75609375a0dc36841f50
Author: Cam Rider <camtrider@berkeley.edu>
Date:   Sun Nov 26 23:43:12 2017 -0800

    hw04 completed
:
```

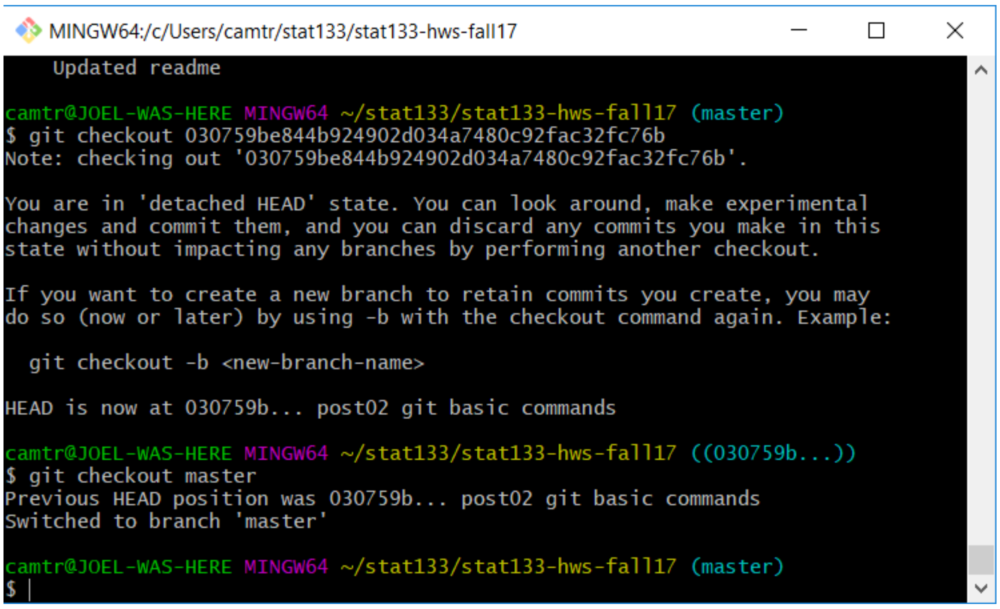
Now we get a whole list of all our commits! These are ordered with the most recent at the top. If we want to see older commits or scroll through the list, we can use the arrow keys to do so. To exit out of the log and get back to your command line, press Q.

git checkout

Now with git log to see all our previous commits, we can check them out using checkout! You may have noticed each commit had a very large number listed after it - this is each commit's commit number. First, ake sure you commit or stash everything you don't want to get erased (you cannot get your missing files once you've done this so PLEASE commit or stash them). Then, grab the number you desire and type:

git checkout *insert commit number here*

You should get a somewhat cryptic message like this:



```
MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17
    Updated readme

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git checkout 030759be844b924902d034a7480c92fac32fc76b
Note: checking out '030759be844b924902d034a7480c92fac32fc76b'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at 030759b... post02 git basic commands

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 ((030759b...))
$ git checkout master
Previous HEAD position was 030759b... post02 git basic commands
Switched to branch 'master'

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ |
```

If you go back into your files and look around, you should see that everything has magically reverted back to the commit you designated! Alright, so maybe it isn't that magical. But it's cool right! It's very useful if you have a program that once worked but is now broken or if you just want to access an older version of your files. If you modify your files while in checkout, it won't overwrite your most modern stuff: remember, your new stuff is still safe from when you committed/stashed it. In fact, you can commit changes you make to the file while in checkout!

If you're ever having trouble keeping track which commit you're in, notice that the commit number is listed after the file path in your command line. If this says "master", that means you're at your master branch (essentially, where you started off before using git checkout). To return to master, simply type **git checkout master**.

Going Online

Now that we have a pretty good idea of how git keeps track of stuff, we can start to see how git bridges the gap between collaborators. You may at some point have noticed (after committing) some variation of this line:

3 files changed, 53 insertions(+), 32 deletions(-)

That's right - git doesn't just take note of what changed and then copy the whole thing over again each time you commit. Rather, it notices exactly what parts of each file you edited and keeps track of those edits. Each time you make a commit with a modified file, it grabs those edits along with the file from the previous commit and puts them together, culminating into the file you uploaded.

Now, that may not seem like a big difference when you're the only one changing each file, but it makes a huge difference if multiple people are working on a project or even a single file! You can individually work on something separately, and when the time comes to smash your work together into something that (hopefully) pieces together into one beautiful result, it's as easy as a couple commands with git. There's no more worrying about overwriting other peoples code, copy-pasting lines to another file and accidentally messing up, etc - you can get rid of as many sources of error as possible, which is extremely valuable when something as small as a single parenthesis can take down your whole program. Hopefully now you can see why git, and by proxy websites like GitHub, are so valued among the community of developers.

Speaking of GitHub: let's make a GitHub repository and connect it to our repository so we can do all the collaboration we've dreamed of doing for so long! The first step is to create a GitHub account and set up a repository (it's pretty self explanatory and the website helps you along). Once you've finished that, get the url of your github repository (for example, mine is https://github.com/camtrider/stat133-hws-fall17) and then navigate to your repository folder on your computer. To connect your repository to the GitHub one, type in:

git remote add *local nickname for your online repository* *repository url*

And you're done! Of course, nothing has changed in the online repository - you still have to move everything you've committed into the online one. Not that that takes long either!

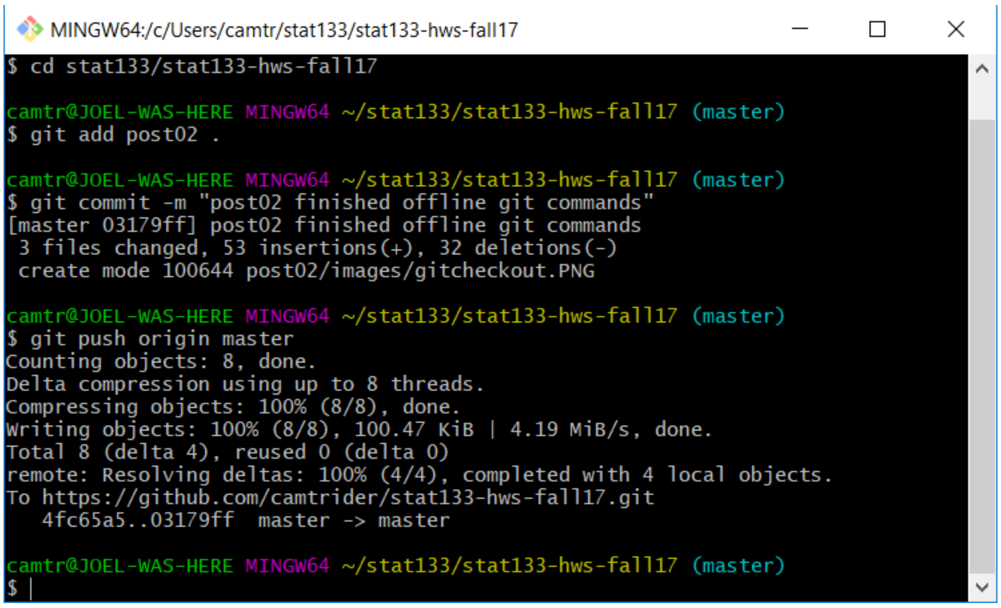
Alternatively, if you have an remote repository you'd like to move locally onto your computer, you can use the command **git clone *repository url***. This has the same effect as remoting into your remote repository, just in reverse: all the files from the remote repository will be copied onto your computer, creating a new git repository locally.

git push

Back to something a little basic: now that I have a remote repository, how do I put stuff in it? Simply put, you have to *push* your commits into the remote repository. The command works like this:

git push *nickname of the remote repository* *name of branch you'd like to push from*

The nickname of the remote repository should be the one you set before, while the branch should be the branch you'd like to take your changes from (unless you've been moving around with git checkout, this will be your master branch - also known as master). It'll look something like this:



MINGW64:/c/Users/camtr/stat133/stat133-hws-fall17

```
$ cd stat133/stat133-hws-fall17

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git add post02 .

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git commit -m "post02 finished offline git commands"
[master 03179ff] post02 finished offline git commands
 3 files changed, 53 insertions(+), 32 deletions(-)
 create mode 100644 post02/images/gitcheckout.PNG

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ git push origin master
Counting objects: 8, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 100.47 KiB | 4.19 MiB/s, done.
Total 8 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To https://github.com/camtrider/stat133-hws-fall17.git
   4fc65a5..03179ff  master -> master

camtr@JOEL-WAS-HERE MINGW64 ~/stat133/stat133-hws-fall17 (master)
$ |
```

In my case, "origin" is the nickname for my remote repository, and I am located in the master branch of my git repository (which is also displayed after my file location.)

git fetch

Just as we can push stuff to our remote repository, we can also grab new things from it. This can be done using **git fetch *nickname of the remote repository***. This will create a new branch (it won't change your master branch, so you're not done yet!) under the name *nickname of the remote repository*/master. Keep that in mind for the next command…

git merge

Now we have the power to merge two branches together! Merge effectively takes the changes from one branch and then puts them into the files on your current (HEAD) branch, listed next to your file directory on your command line. For example, say you're in the master branch, and someone has a hotfix for your program in a remote repository. Fetch their repository and then merge the branch of your choosing with the command:

git merge *branch name*

Check your code! All of their changes should be in your files now, as long as you stay on your branch. It is possible to get an error message here: if you see something along the lines of CONFLICT: Merge conflict in *file name*, it means you both edited the same file. One of you will most likely have to delete your changes from the most recent commit you both worked off of to get everything to merge properly. This is in place simply to make sure you don't override your own code, and to alert you if you've both modified the same thing.

This ends my quick guide to some of the commands in Git! This may be second nature to a lot of developers - however, to those who are just getting started, it can be hard to understand what's so great about git and how it operates without having to resort to reading through the entire Pro Git book. I tried to be helpful and concise just to allow readers to get an easy start with Git - there are far better sources online that can really go in-depth into how git operates. I actually used many of those to help write this guide, so please check them out down below:

Pro Git, an extremely helpful and straightforward guide to everything git. Includes inline code, graphics, and pretty much everything you could possibly need to get started. Seems very beginner friendly despite the fact that it's, well, a book solely about git.

Git Documentation, a full list of all the different commands in git, straight on the git homepage itself. It doesn’t get much more reliable than that. Each page contains a full description of the command, what the command can take as inputs, and usage examples.

Git Cheat Sheet, an even faster way to read through the git commands and what they do. Doesn’t really provide explanations aside from a one-liner for each command but definitely handy for referencing imporant commands. Also includes git installation links!

Git - The Simple Guide, a *very colorful* and quick overview of the commands I go over in this post. Also contains some helpful images that depict some of the nuances of git that would’ve stood out a bit TOO much in this post.

Try Git, an interactive git course that allows you to try out how git works before downloading it. Very straightforward and simple, only takes 15 minutes to get through.

Become a Git Guru, a much more indepth guide that takes a little more patience to get into but will reward you for your time. Goes into the concepts behind each set of commands and what git allows to do in full.

Think Like a Git, a fun guide to git that quickly dives into the deep end. Super nice if you want to take your git understanding to the next level.