Git Console Commands

Updated: Nov 1st, 2020

Legend:

- '$' indicates a command in git bash. Do not include them when writing the command.
- Underlined commands indicate that the command is common and part of typical git usage.
- Blue writing indicates a command in PowerShell.
- Words surrounded by [square brackets] indicate that something needs to be inserted into that spot. Do not include them when writing the command.

Resources:

- You will likely need either a GitHub or BitBucket account.
- Git Command Line Tutorial: https://www.youtube.com/watch?v=HVsySz-h9r
- There are a variety of step-by-step examples of common git use cases in the examples section.
    - **Examples assume you've installed git to work with git bash**, not your IDE (expect minor differences).

Background:

Git is quite unintuitive to learn initially. However, once you are comfortable with the basics, it becomes an incredibly useful and powerful tool. From my experience, it can only truly be understood through practice. If you are new to git, try not to be overwhelmed by the initial learning curve. Instead, push through, watch video tutorials, and practice it for yourself.

Git is a tool used to provide version control tracking to software projects. When using git, developers implement changes on their local repository and then "push" their changes to the online repository. These changes can then be "pulled" by other developers into their local repositories. Git is considered decentralized because a copy of the project repository is located on each computer in addition to the copy hosted online. As such, work can be performed by each developer offline and then merged into the shared online repository.

Your local repository consists of 3 basic components maintained by git:

1. Working Directory (Contains working files AKA uncommitted changes)
2. Index (Staging area for files that are selected for committal)
3. HEAD (Current working tree of published commits)

Commits are the foundation of using git. Version history is tracked using trees of commits (AKA formally published updates). When you edit a software file and press save, that change is not recorded by version control until it has been committed. In order to commit changes (i.e. publish changes into version control history) a few general steps must be followed:

1. Local changes must be made in the working directory.
2. Desired changes must be staged (i.e. Selected for committal).
3. Staged changes must be committed (locally).
4. Locally committed changes must be "pushed" to the online repository.

Since git is such a robust tool, there is much more to learn. The following contains descriptions for typical git commands as well as examples of common use cases. If any particular commands are confusing, check the examples for context.

Cloning Remote Repositories:

If there is an online git repository that you want to duplicate as a local repository on your computer, cloning is used. Using this command, a project folder containing the contents of the online git repository is copied to your computer. The git-link can be found on the web page of the online repository (look for a clone button).

```
$ git clone [git-link]                      // Clones an existing remote repository to current path.
$ git clone [git-link] [desired file path]  // Clones an existing remote repository to local machine.
```

Setup:

If you have a local repository that you want to track using git version control, *git init* is used. It creates a hidden ".git" folder that contains project history. If this folder is deleted, your history is lost (locally).

```
$ git init                                  // Initializes local git repository.
```

Configuring Remote Repositories:

After you have initialized a git project using *git init*, you'll likely want to link your local repository with an online (AKA remote or origin) repository hosted on GitHub or BitBucket. This is done by syncing an "origin" (i.e. online repo) to correspond with your local repository. The git-link used to link the local and online repos can be found on the web page of a newly created online repository. You'll need to login to GitHub or BitBucket and create an online repo for this step.

```
$ git remote add origin [git-link]          // Links remote repository with local repository.
$ git remote remove origin                  // Removes the remote repository.
$ git remote -v                             // Lists remote repositories.
```

Storing Credentials:

If you have not used git before on a certain computer, you will be unable to perform commits until you have set up your credentials. Every commit is associated with a name and email. As such, they must be established before a commit can be made.

```
$ git config --global user.name "[name]"    // Stores username that will show in central repository.
$ git config --global user.email "[email]"  // Stores email that will show in central repository.
$ git config --global credential.helper store // Stores git login used to commit to central repository.
                                            // Note: Must have already logged in after committing (unverified).
```

Staging:

Before a commit can be made, you have to tell git exactly what files you want published. This is done by staging files from the *Working Directory* into the *Index*. Files can be added/removed to the *Index* in bulk, by name, or by extension. You can use *git status* to check which files have been staged for committal.

```
$ git add [fileName]                        // Adds named file(s) to index/staging area.
$ git add .                                 // Adds all files to index/staging area.
$ git add *.html                            // Adds all .html files to index/staging area.
$ git reset [filename]                      // Removes file from index/staging area.
$ git reset                                 // Removes all files from index/staging area.
```

Commits:

After all relevant files have been staged, they can be committed (i.e. published) to official version control history. Unstaged files are not reflected in git history. Keep in mind that **commits are done locally**. Local commits must be PUSHED to the online repository (i.e. the remote repository must be updated to reflect your latest commits) before they are visible to others. It is good practice to commit related changes together with detailed descriptions.

```
$ git commit                              // Prompts for detailed description and then commits.
$ git commit -m "[comment]"               // Commits with input comment as commit title.
$ git log                                 // Shows a list of commits w/ unique hash numbers.
```

Updating Remote Repositories:

Once you have committed local changes to a repository, you'll likely want your online (AKA remote or origin) repository to reflect these changes. This is done by pushing (sending updates to the remote repo) and pulling (receiving updates from the remote repo). Initially, the connections between your local repo and the remote repo are undefined. Therefore, you must define the locations that you want to push to and pull from. This is done by specifying the location at the end of a push/pull command (i.e. adding *origin [branchName],* see below). If a connection between branches is already established, the typical *git push* and *git pull* commands can be used. When working with others, **pull** the latest updates **before pushing** your latest commits in order to avoid/address merge conflicts.

```
$ git pull origin [branchName]            // Pulls latest from specific branch of remote repository.
$ git push origin [branchName]            // Push to specific branch of remote repository.
$ git pull                                // Pulls from the current branch of remote repository.
$ git push                                // Push to the current branch of remote repository.
$ git push -u origin [branchName]         // The -u allows us to associate local w/ remote branches. This
$ git pull -u origin [branchName]         // way, git push/pull can always be used without specifying location.
$ git diff                                // Shows changes from the last commit.
```

Branches:

When working with version control, you may want to update working code with significant, and potentially code-breaking, changes. Instead of modifying the master branch (i.e. the working code that others may depend on), branching is used to make a copy of the current project. New features can be implemented and debugged on this branch/version of the code and then merged back into the master branch once it is complete and bug free. A branch is more than just a local copy. Branches are reflected in the online repo and can therefore be collaborated on by a group of developers.

```
$ git checkout -b [branchName]            // Creates and checks out new branch with input name.
$ git checkout [branchName]               // Changes working branch.
$ git branch -d [branchName]              // Deletes branch locally.
$ git push origin --delete [branchName]   // Deletes branch on remote repository.
$ git branch                              // Lists all the local branches, including current.
$ git branch -a                           // Lists all the local and remote branches.
$ git fetch --prune                       // Removes 'origin' branches from your local repo if they have been
                                          // deleted from the online repo. Local versions of those branches remain.
```

Merges:

Once changes have been completed on an alternate branch, you may want to update the master branch to reflect these changes. This can be done by either merging or rebasing the branches together. In a merge, all the commits on a branch are combined into a single "merge" commit that is added to the master branch. Merging can be seamless but, in some cases, you may have to deal with merge conflicts. Merge conflicts can occur when two developers change the same lines of code and both try to push their changes to the online repo. When this happens, the online repo can't decide which change to publish and which change to overwrite. In these cases, developers need to either manually overwrite one of the changes or rework the code to reflect both changes. To avoid merge conflicts, regularly pull the latest updates into your working branch. For example, before merging a branch into your local master, run *git pull origin master* to ensure that your local master contains the most recent updates.

```
$ git merge [branchName]                // Merges another branch into the active branch. If merging to master,
                                        // make sure you are currently checked out to the master branch.
$ git add [fileName]                    // After editing conflicted files, mark files as merged.
$ git diff [sourceBranch] [targetBranch] // Preview changes before merging.
$ git branch --merged                   // Shows the branches that have been merged.
```

Rebases:

Like merges, rebasing is typically performed to add the commits of an alternate branch to the master branch. However, instead of cramming everything into a "merge" commit, rebasing simply severs the bade of the alternate branch and adds it on top of the master branch. In other words, the commits of the branch are essentially being duplicated as new commits on top of the master branch. This approach typically results in a much straighter, cleaner git tree that is easier to interpret. However, rebasing can be dangerous when working on public/shared branches and must be used with caution. Here is a good resource to help you understand the theory: https://www.youtube.com/watch?v=f1wnYdLEpgI

```
$ git rebase [branchName]               // Rebases working branch on top of named branch
```

Status:

Since you're sending git commands through the literal black box of a terminal, it can be difficult to visualize where you are and your current progress on a particular task. Git status is a good catch-all if you ever find yourself asking "Where am I and what is happening?" Common use cases include checking your current working branch, checking the status of staged/unstaged files, and checking the progress of a merge.

```
$ git status                            // Check status of working tree.
```

Undo commits:

There will be times where you fly too close to the sun and royally screw up your local repository. Thankfully, there's a way out of this hole. You can reset your branch to match the origin or revert back to a previous commit. This will erase all changes.

```
$ git reset --hard origin/[branchName]  // Resets your local branch to reflect the origin repo.
$ git reset --hard [commitSHA]          // Resets your local branch to reflect a previous commit.
                                        // The SHA of a commit is an 8-digit hash associated with the commit.
```

Git Ignore:

Git ignore files are used to ensure that only necessary files are being uploaded to the repo hosting website (i.e. Github or BitBucket). Without them, you are likely needlessly bloating your online repo with local or temp files. Standard gitignore files for various contexts can be found on the internet. For example, Java has a standard gitignore file that prevents unnecessary items, such as ".class" files from being uploaded to the remote repo and wasting space. If there is a standard gitignore file available, include it in your project folder (ensure that you didn't download it as a text file). Otherwise, you can make your own. Ideally, a .gitignore file is introduced to the project folder before the initial commit of a project.

```
$ touch .gitignore                        // Creates a blank .gitignore file
new-item .gitignore                       // PowerShell: Creates a blank .gitignore file
```

Files can be added to the .gitignore file in one of the following formats:

- filename.txt
- /foldername or foldername/
- *.txt (all text files)

Console Navigation:

You can only use git commands while traversing a git repository. If you open the terminal from your desktop, you may need to navigate to your project folder.

```
$ ls                                      // Lists items in current directory
ls -Force                                 // PowerShell: Lists all items (including hidden)
$ cd                                      // Changes directory
```

Remove Git Tracking:

There may be a scenario where you want to remove all git version control tracking from a project. In that case, you can use the following command. You can also just delete the hidden .git folder for the same result.

```
$ rm -rf git                              // Removes git version control tracking
```

Other:

```
$ git --version                           // Indicates current version of git
$ touch [filename.filetype]               // Creates blank file of specified type
new-item [filename.filetype]              // PowerShell: Creates a blank file of specified type
$ git config --list                       // Shows the git config details
$ git help [gitAction]                    // Shows help for the specified git action
```

Git Console Examples

Note: **Examples assume you've installed git to work with git bash, <u>not your IDE</u>**.
If you linked your IDE, git will sometimes want you to perform certain actions through your IDE rather than the terminal. You will likely not have to enter "insert mode" before inputting text. Instead, your IDE will open up a file for you to manipulate, save, and close.

Connect Project to Github Repository Example (Basic):

1. Have pre-existing files to be backed on github
2. Add a ".gitignore" file if necessary
3. Navigate to project folder containing all relevant files using git bash or equivalent terminal
4. $ git init   // Initialize repository for local git version control tracking
5. $ git remote add origin [git-link]   // Connect local repository to online repository
6. $ git add **.**   // Add all files to the staging area
7. $ git commit -m "Initial commit"   // Commit staged files locally
8. $ git push origin master   // Push commit to online repo

Clone Git Project From Github Repository Example (Basic):

1. Have clone link of github repository
2. Navigate to location where project folder is to be cloned using git bash or equivalent terminal
3. $ git clone [git-link]   // Clone (copy and link) online repo to local repo

Clone Git Project From Github Repository Alternative Example (Basic):

1. Have clone link of github repository
2. Navigate to folder where files are to be cloned using git bash or equivalent terminal
3. $ git init   // Initiate git project
4. $ git remote add origin [git-link]   // Add remote to git project (connect local to online github repo)
5. $ git pull origin master   // Pull content of online repo to local folder

Commit Substantial Changes to Project Example (Basic):

1. Complete desired changes in existing git project
2. Navigate to project folder using git bash or equivalent terminal
3. $ git add **.**   // Add files to staging area (index)
4. $ git commit   // Commit staged files
5. If unable to write, press insert to enter "insert mode" <u>(ignore if using IDE to input text)</u>
6. Write descriptive title for the commit on the first line
7. Two lines down from the title (leave a line of whitespace between), write a detailed description for the commit
8. Press escape and enter ":wq" to leave insert mode and accept changes <u>(or save/close if using IDE to input text)</u>
9. $ git push   // Push local changes to remote

Append Minor Changes to Previous Commit Example (Intermediate):

1. Complete desired changes in existing git project
2. Navigate to project folder using git bash or equivalent terminal
3. $ git add **.** // Add files to staging area (index)
4. $ git commit --amend // Append changes to previous commit
5. Edit commit message if necessary (see steps 5-7 above)
6. Press escape and enter ":wq" to leave insert mode and accept changes <u>(or save/close if using IDE to input text)</u>
7. **If working with others, pull all recent changes before force pushing or else you will overwrite their work**

1. $ git push -f // Force push local changes to remote. Push must be forced because you
// locally changed the content of a commit that currently exists in the
// remote. The discontinuity prevents a push unless forced.

Create/Merge Branch Example (Basic):

Note:

- (b-name → branch name)
- Useful command: "$ git branch -a" // Lists all branches (* indicates the current working branch)
// Can be used to double check working branch or if branch was
// created/deleted successfully

1. Navigate to master branch.
2. $ git branch [b-name] // Creates new branch
3. $ git checkout [b-name] // Changes local working directory to named branch
4. Make changes to files as usual
5. Commit changes locally as usual
6. $ git push -u origin [b-name] // Pushes and associates local branch with remote branch
// After this command is used for the first time, can use
// "$ git push" and "$ git pull" for future changes to branch
7. $ git checkout master // Changes local working directory back to master branch
8. $ git pull origin master // Pulls changes from remote master branch (if any)
9. $ git merge [b-name] // Merges branch into local master
10. Resolve merge conflicts if applicable
11. $ git push origin master // Pushes changes to master remote repository
12. $ git branch -d [b-name] // Deletes local branch
13. $ git push origin --delete [b-name] // Deletes remote branch

Git Rebase Example (Advanced):

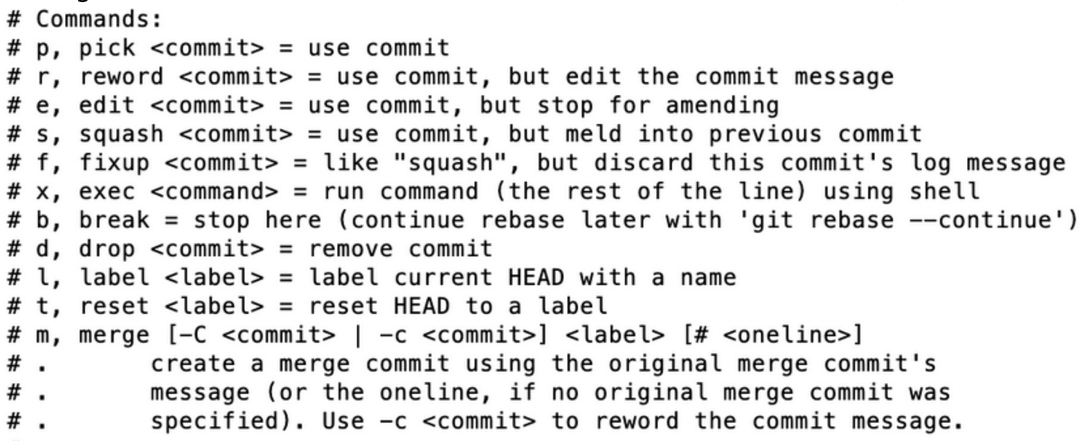
Note that this example involves rebasing a feature branch to the master branch. However, a branch can be rebased on any other branch. Git rebasing can be dangerous when working on public projects and should be used with caution. See this for background: https://www.youtube.com/watch?v=f1wnYdLEpgI

1. Complete and commit desired changes in existing git feature branch
2. $ git checkout master // Check out master branch
3. $ git pull // Pull master (Ensures your local repo is up to date with the most current // version of master. VERY IMPORTANT)
4. $ git checkout [b-name] // Recheck out feature branch
5. $ git rebase master // Rebase feature branch commits on top of latest master branch
6. Resolve merge conflicts if applicable
7. $ git checkout master // Recheck out master branch
8. $ git rebase [b-name] // Rebase master on feature branch (This merges the branches into one)
9. $ git push -f // Force push changes to origin. A force push is necessary here because // you have rewritten git history (i.e. rearranged the git tree)

Git Interactive Rebase Example (Advanced):

Interactive rebasing is a fancy way of saying "manipulating commit history". Interactive rebasing is commonly used to reorder/edit/erase previous commits, squash multiple commits into one, or update documentation.

1. $ git rebase -i [b-name]~4 // Interactive rebase going back 4 commits on the named branch
2. Manipulate tree
   a. If unable to write, press insert to enter insert mode (ignore if using IDE to input text)
   b. Reorder lines around to reorder commits
   c. Delete lines to delete a commit
   d. Change commands at the front of lines for desired effect (see command list)



```
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
```

3. Press escape and enter ":wq" to leave insert mode and accept changes (or save/close if using IDE to input text)
4. If the rebase is in progress and does not continue on its own, type "git rebase --continue". If you're not sure if the rebase is still in progress, type "git status" for details.
5. Rename changed commits/comments as usual if necessary
6. $ git push -f // Force push local changes to remote. Push must be forced because you // locally changed the content of a commit that currently exists in the // remote. The discontinuity prevents a push unless forced.