**Practical Git Commands Kickstart**

Author: Crystal Tenn

Published Web Blog Article: <https://blogs.msdn.microsoft.com/premier_developer/2017/05/22/practical-git-commands-kickstart/>

Git command line is extremely useful for managing your resources – it is the only way to utilize all Git commands possible as most GUI’s only have a set of available commands. Although many guides exist to explain Git command line, many are brief introductions or extremely comprehensive. I will share a practical guide of commands I needed to use to get through my day, no more, no less as part of a kick start to get you going with Git. I will list commands by group in tables and will give an explanation below each table in italics for why/when you need these commands if you are new to Git.

|  |  |
| --- | --- |
| **Project/Directory Setup Commands** | **Use/Explanation** |
| git init | New repository from current directory |
| git init <directory> | New repository to a specific directory |
| git clone /path/to/repository | Connecting to an existing local repository |
| git clone username@host:/path/to/repository | Connecting to an existing remote repository |

*You need initialize a local file directory to be a Git repository for a new project (your local file directory will be associated with a certain online repository). Initializing will create a .git folder in the directory and this is where Git will record different versions of the project and the .git folder will contain all of the metadata and changes to the project (rest of the project will not be altered while you are working).*

|  |  |
| --- | --- |
| **Adding/Staging/Unstaging Files Commands** | **Use/Explanation** |
| git add filename.txt | Stages a specific file to commit |
| git add -A | Stages all |
| git add . | Stages new and modified, without deleted. ***\*This add command is most commonly used*** |
| git add -u | Stages modified and deleted, without new |
| git add \* | Stages all files in current directory except files whose name begin with a dot. Wildcard interpreted as part of git. |
| git reset -- filename.txt | Unstages a file by name, the changes are still there that you made but it will not be committed until you add it back |
| git reset | Unstages all files (again changes are still there and now nothing is committed until you add it in) |
| git status | Checking file status (see which files are staged/unstaged for the commit) |

*Staging and unstaging do not affect the changes of the files or delete and files, they are for adding/removing files to track for a commit or stash. You push your commits to make a change. In Git, you stage(add) the files you want to commit and unstage (reset) files you do not want to commit. Note ‘--’ in a command specifies we are talking about a file, not a branch (in case some file names are the same as a branch name).*

|  |  |
| --- | --- |
| **Commit/Reset Commands** | **Use/Explanation** |
| git commit -m “Your message here to describe what you are committing to your branch.” | Commit changes with a message |
| git checkout <file> | Abandon/Undo changes to one file |
| git reset --hard HEAD~1 | Removing the most recent one commit (HEAD~1 means the commit before the head) |
| git log | Find a commit id |
| git reset --hard <sha1-commit-id> | Removing a very specific commit id |
| git reset --hard | Removing all commits/changes and reset back to original branch pulled |

*Committing is not permanent, do not be afraid to commit. You don’t have to push your commit. You can roll back this commit easily if you want to erase all of it and go back to the original code in the branch. You must commit or stash everything before you can change branches, pull new commits in, etc (see final table for stashing info).*

|  |  |
| --- | --- |
| **Push/Pull Commands** | **Use/Explanation** |
| git push | Pushing changes from current branch to the same current branch |
| git push originally-cloned-branch new-branch | Pushing changes from an originally cloned branch to a new branch |
| git fetch | Fetching commits and remote branch changes, does not put new changes into your code until you pull though |
| git pull | Pulls all the latest commits to your current branch from the server |

*You must run a* ***git fetch*** *to see new commits or any new branches added to your repository.* ***Git pull*** *does a* ***git fetch + git merge****. You can* ***git fetch*** *anytime to update remote tracking branches or see any latest commits from others. If you already pushed a commit and need to revert it, you can do a force push to get rid of it but honestly it is safer for you to create a new branch and fix your changes as others are working in the same code (pushing is a little permanent, make sure you test everything out before you push your changes to the server where other developers will see your work).*

|  |  |
| --- | --- |
| **Switching/Creating Branches** | **Use/Explanation** |
| git checkout -b my-new-branch-name | Create a new branch based on your current branch, and switch to the new branch. *Note the -b means that you need to create a new branch* |
| git checkout -b new-branch existing-branch | Create a new branch based on a specific branch |
| git checkout existing-branch-name | Switching to another existing branch |

*Make sure you commit or stash your changes before trying to change branches. Note: you are able to swap between different branches with one code same base—all of the changes that appear are applied based on the contents of the .git folder on top of the original files.*

|  |  |
| --- | --- |
| **Merging Branches Commands** | **Use/Explanation** |
| git merge branchname | Merges *branchname* into your current branch. Make sure to pull latest into both your current branch and the branch to merge in |
| git reset --merge . | Abandon a merge with merge conflicts, goes back to state you were in before you attempted to merge. Can use this command if Git v1.6.1+ |
| git reset --hard HEAD | If Git version is less than 1.6.1 then use this to abandon a merge |

*You merge one branch into another one, for example if you are working in a feature branch and want to put your work into develop.. or if you want to merge stable develop changes into QA. Careful with merge conflicts and to test after fixing all of them before committing and finally pushing to the server.*

|  |  |
| --- | --- |
| **Dealing with Stashes Commands** | **Use/Explanation** |
| git stash | Stash all tracked files without a message/description to index 0 |
| git stash save “my description here” | Stash all tracked files that have not yet been committed |
| git stash save -u “my description here” | Stash all files that have not yet been committed *including* untracked files |
| git stash pop  -or-  git stash apply | All changes to stashed files will be applied to the current workspace (unstashing). You can reapply to same branch or apply changes to a new branch. By default gets stash@{0}. **Important: git stash pop throws away the stash after applying it, whereas git stash apply leaves it in the stash list for possible later reuse.** |
| git stash pop stash@{2}  -or-  git stash apply stash@{2} | Specifies which index to apply/unstash to your current workspace |
| git stash list | List the current stashes, you can see the index of each one to delete a certain one |
| git stash clear | Clear/Delete all the current stashes |
| git stash drop stash@{0} | Replace {0} with the index of the stash you want to drop |

*Use git add/reset to determine which files you want to stash. You can stash changes as a “save-state” so you can do work in another branch, all changes will be saved in your local Git repo and nothing from the stash will go to the server. You can easily unstash your changes onto the same or another branch. You can apply one stash to multiple other branches. Sometimes you may make a feature branch and make a lot of foundational domain changes.. then suddenly you realize another developer needs your changes to work. You can stash your changes and unstash them on top of the other branch so you are both working in the same branch with shared foundational code. New git stashes by default are always created to stash@{0}, and older ones’ will have the index pushed to higher index numbers.*