Git Real

About
Learn more advanced Git by practicing the concepts of Git version control. Increase your Git knowledge by learning more advanced systems within Git.

Introduction (Level 1)

Version control is an invaluable asset that is used every day. It works by keeping track of who makes changes to a file, what changes were made, and when they were made. This is useful for when there are multiple users working on the same file, because you want to know who did what.
There exists a central repository, for which local repositories are connected to. The central repository is where the files a re accessed from, where other users can access them. Every user has a local version of the central repository, where they can make changes, and stage commits to push back to the central repository.

Show Help `> git help`	This will show the way the "git" command can be used, as well some of the most commonly used git commands.
Show Command Help `> git help <command>`	This will show the info for a specific passed git command. Namely it shows what it does, and how you can use it.
Set Global Username `> git config -global user.name <username>`	This will set your username across all your local repositories, so you can be identified as the one who committed/pushed changes.
Set Global Email `> git config --global user.email <email>`	This will set your email across all your local repositories, so you can be identified as the one who committed/pushed changes.
Initialize Current Directory as Git Repo `> git init`	This will initialize the directory you are currently in as a git repository, which is useful when working with an existing project.

Staging & Remotes (Level 2)

Continuing on from the previous topic, Introduction, the stage in Git is the intermediary between your workspace and the remote. It is where you will prepare the changed that you have made before they move on. You will be able to check, mark, and even undo changes in the staging area until you are ready to proceed.

Show Unstaged Differences `> git diff`	This will show the unstaged differences since the last commit. It is useful for finding a change when you don't know or remember where it is. It is important to note that added files will not show up when running git diff.
Show Staged Differences `> git diff --staged`	By adding the --staged flag, the git diff will now only show the differences of staged files. This can be useful for making sure the changes you want are going to make it into the commit and push.
Unstage File from HEAD `> git reset HEAD <file>`	This command will unstage a staged file from the HEAD. The HEAD refers to the latest commit on the current branch. This will not remove the modifications made to the file, it only unstages it.
Remove Changes `> git checkout -- <file>`	This will get rid of all the changes to a file since the last commit. This is useful for going back to a safe version, free of unwanted changes.
Add & Commit All `> git commit -a -m <message>`	This is a shortcut command to add and commit. Note that the -a flag will add all tracked files in the repository, and will not add any untracked files before committing.
Undoing a Commit (Soft)	This will, effectively undo a commit that you made, and move the files back into the

`> git reset --soft HEAD^`	staging area. Note that the --soft flag is what is moving the files back to the staging area, while the ^ after HEAD is moving to the commit before the current HEAD.
Adding to a Commit `> git commit --amend -m <message>`	This add a file to a commit, instead of unstaging all the files and recommitting with a new message. Note that you do need to git add <file> before you can amend to the commit.
Undo a Commit (Hard) `> git reset --hard HEAD^`	Unlike the soft reset previously mentioned, the --hard flag will undo the last commit and changes, effectively destroying the last commit. This is useful for when you make a mistake you can't recover from, you hard reset to the last safe commit.

Now that we have made changes and committed to our local repository, we want to share for others to see. We want to be able to push the commits out, and pull in new changes for us to see. Its quite possible that we're working on a team, so we need t o be able to see the changes that others have made. What we'll be pushing to and pulling from is known as a remote repository. It is important to note that Git doesn't do access control. That is to say, that Git doesn't restrict or allow access to what yo u push, or what you can pull. You can look at GitHub or BitBucket for Hosted Remote Repositories, or you can try Self-Managed Remote Repositories with Gitosis and Gitorious.

Adding a Remote `> git remote add <name> <address>`	This will add a new remote to a repository. A remote is a known address that your local repository knows about and has access to.
List Remotes `> git remote -v`	This will show a list of remote that the repository knows about. This can be useful for knowing if you need to push/pull/fetch from a specific remote.
Pushing to Remote `> git push -u <remote> <local>`	This will push from your local branch to the specified remote branch. The -u flag will make it so next time you pus, you don't need to reenter the remote and local branch names.
Pull from Remote `> git pull`	This will pull changes from the remote branch into your local branch. This is important for keeping up with changes that others have made.
Remove a Remote `> git remote rm <name>`	This will remove a known remote from your repository. This is useful for keeping your known remote list clean and up to date.

Cloning & Branching (Level 3)

When you want to pull down the changes of a repository you are already following, you'll use the pull command. But to initial ly bring in a repository, you'll want to use git clone. Cloning will make a local repository on your machine that you'll later b e able to stage to, push from, and pull into. Cloning is mostly used to bring down and look at work that another person has started.

Clone a Repository `> git clone <address> <local-name>`	Cloning a repository can be seen as the first pull to your local repository. Note that the <local-name> is optional, and will only rename the local folder the repository is in.

Being able to effectively branch is crucial to understanding Git. First, a branch can be thought of as a divergence point in your repository's history. Everything up to the branch is included in the branch, but whatever you do in the branch doesn't show u p in the master branch. That's not to say you couldn't bring the branch back in, but we'll get to that later. A branch can be thou ght of a place to test your code without if affecting everything else. You could work on a feature or an idea in a branch, and it if doesn't work out, it won't hurt the master branch.

Creating a Branch `> git branch <name>`	This will create a branch in your repository with whatever name you give it.
List Branches `> git branch`	This will show a list of branches in your repository, as well as indicate the branch you are currently working in.
Switching to a Branch `> git checkout -b <branch>`	This will switch you out of whatever branch you are in, and move you into the branch you select. Note that the -b flag will both create and checkout the new branch.

As stated earlier, making a branch will allow us to work without changing our master branch. And I mentioned that we could bring out branch back into master, this is called merging. When you merge, you take whatever you have in one branch, and bring it into another branch. Merging is not the same as removing a branch, it just brings the contents of one branch into another.

Merging Branches `> git merge <branch>`	This will bring the content of one branch into another branch. Note that you need to be checked out on the branch you want to bring the <branch> content into.
Branch Clean Up `> git branch -d <branch>`	This will delete the specified branch.

When you make a new branch, commit in it, then merge it back into its parent (without having made commits in the parent) it is known as a fast forward merge. Since there have been no changes and commits in the parent branch, it simply has to fast forward to the point of the merge.
If you work and commit in the parent branch before merging back into it, Git will need to do a recursive merge. It might prompt you before making the merge, but that is mostly due the numerous differences between the branch and its parent.

Collaboration Basics (Level 4)

Knowing the basics of Git are undoubtedly important if you'll be using it regularly. Being able to stage, commit, push, pull, clone, branch, and merge are what makes Git Git. However, using all these aspects of Git by yourself and collaborating with Git are completely different things. This section will introduce to you the basic etiquettes of working together with others on the same Git repository.

Let's say Alice is working on a Git repository and calls in Bob to help. Bob will clone down Alice's repository so he has a local version to work on. Alice then makes a commit and pushes it to the remote. Then Bob makes a commit and tries to push to the remote, but it fails. The reason that Bob cannot push his commit, is that his local version of the repository does not include the push that Alice just made, and he cannot write over Alice's commit. Bob will need to first pull from the remote before he can push his commit to the remote.

Fetching Changes `> git fetch`	The fetch command is actually the first part of a pull. Fetch will go to the remote repository and bring down all the code across all branches to the local repository. However, it will not update the local repository, it will simply say that there are differences between remote and local repositories.
Merging Changes `> git merge origin/<branch>`	This is the second part of the pull command. It merges changes from the remote branch (origin/<branch>) into your local branch. Do keep in mind both of these commands are simply the behind the scenes when you make a pull.

When Bob makes a pull to update his local repository, could be a "merge commit". Essentially, it a commit that is made when you make a pull to update your local repository with the remote repository. Bob will need to push after the merge commit for his local repository is the same as the remote repository. There is the opinion that merge commits can make your log messy with so many commits just to make sure everyone is on the same page. There is a way around that, and we'll look at it in a later section.

A few days after Bob learned about pulling before pushing, Alice made a change to the "README.md" file, committed and pushed. Moments later, Bob made a different change to the same "README.md" file, made his commit and pulled, and ran into a "merge conflict". A merge commit happens when changes in a file try to overwrite changes in the same file. Bob will need to go in and resolve the merge conflict, by specifying what version to keep and which to discard. Bob can manually edit the file, or use one of many merge conflict tools available online.

Branching (Level 5)

This section will cover remote branches and tags. A branch is useful for working on code that is separate from your main branch. But there will come a time where you want someone else to see what you're working on in your branch, but are not ready to merge your branch. In comes the remote branch, a branch that is separate from remote master that can accessed the same way your main branch can be. It's also good practice to have a backup of your branch, even though it is separate from the master branch.

Create a Remote Branch	This will push your local branch <branch> to a remote branch "origin/branch", and

`> git push origin <branch>`	start tracking it at the same time.
List All Remote Branches `> git branch -r`	This will list all remote branches. This is useful for when you make a pull, but haven't checked out on the remote branch yet. You just need to checkout on it to start tracking and have a local branch.
Remote Show `> git remote show origin`	This will show a list of remote branches for a repository and tell you whether they are tracked or not on your local repository. You will also see the local branches the remote branches merge with, and the local branches that are configured to push. Further, remote show will tell you which local branches are outdated.
Delete Remote Branch `> git push origin :<branch>`	This will delete a remote branch. Note that this does not delete your local branch, you will need to do that in addition to this. Note that Git will tell you if there are unmerged commits in a branch before you delete it.

Lets say Alice deletes her local "cats" branch as well as the remote "origin/cats" branch. If Bob tries to push from his local "cats" to the now deleted "origin/cats" branch, nothing will happen. He can run the remote show command to see that the "origin/cats" branch is now stale.

Pruning Origin `> git remote prune origin`	This will go into the remote and prune all the stale branches. This is useful for keeping your remote clean of deleted branches.

A tag is a reference to a commit, and is mostly used for release versioning.

List All Tags `> git tag`	This will list all the tags that have been used.
Checkout a Tag `> git checkout <tag>`	This will checkout your repository at the specific commit that the tag was added to.
Add a New Tag `> git tag -a <tag> -m <message>`	This will attach a specific tag and message to a commit to be made.
Push Tags `> git push --tags`	This will tell the git push command to push tags to a remote. Note that not pushing the tags will keep them local.

Rebase Belong to Us (Level 6)

In the previous section merge commits were introduced. You need to merge your remote and local branch, or vice versa, you have a merge commit. Merge commits can quickly fill up your log, and make it look messy. Thankfully there is an alternative to a merge commit, a rebase.

Going back to a previous situation, Alice has made a commit and pushed it to the origin. Bob does not have this commit on his local repository, meaning he is out of date. And instead of doing a pull and push, Bob will do a fetch and rebase. As a reminder, a fetch goes to the origin and gets the changes, but does not merge them with the local.

Rebase `> git rebase`	Rebase is going to do three things. First, it will move all the changes to the local master which are not in origin/master to a temporary area. Second, it will run all the origin/master commits, to match the local with the origin. Lastly, it will run the commits that were moved to temporary storage, one at a time.

It is important to consider when doing a rebase, you might have a conflict with files from the origin and the local. Since git rebase adds the local commits one at a time, it will simply prompt you to resolve each conflict as it arises, then continue the rebase. You are able to skip applying a local commit during the rebase process if you so wish, then just commit it normally after rebase ends, resulting in no merge commits.

History and Configuration (Level 7)

In previous sections, we used to the git log command to see a list of our commits. The git log will usually show the SHA, which is theoretically a unique identifier for the specific commit, as well as the author of the commit, the date timestamp of the commit, and the message attached to the commit. Generally, a git log is not the easiest thing to read, but there are things we can do to change that.

Colorizing the Log `> git config --global color.ui true`	This will colorize your commit SHA hash in a Git log.
One Line Log `> git log --prety=online`	This will display your git logs as the SHA has and the commit message on one line.
Log Format `> git log --prety=format:"<format>"`	This will let you format your git log as you want. You format can be replaced with a combination of the following placeholders: %ad (author date), %an (author name), %h (SHA hash), %s (subject), and %d (ref names).
Log Patch `> git log --oneline -p`	This will set git log to show the removed lines, as well as the inserted lines, in other words the actual changes that were made during a commit.
Log Stats `> git log --oneline --stat`	This is set your git logs to show the number of files that were changed, and the number of insertions and deletions for each file.
Log Graph `> git log --oneline --graph`	This will set your git logs to show a graph representation of the branches and commits made.
Log Date Ranges `> git log --since=<since> --until=<until>`	This will show the commits since or until the specified inputs. Note that you can use either since or until or both. Keep in mind that <since> and <until> need to be in <number>.minute.ago, <number>.day.ago, <number>.hour.ago, <number>.month.ago, or YYYY-MM-DD format.

Much like the various ways you can customize the git log command, you can utilize the git diff command. You can use it to compare between the current version and any known commit, between the current commit and previous versions of the HEAD, or the current version and a version specified by a date. You can also compare two commits using similar specifications as the git diff. For more detailed information, remember to use the git help <command>.

There are times when you have files in your repository that you don't want to add to a commit for whatever reason or another. And it would be troublesome to specify not to add those ever time you need to commit, so you can instead exclude them. In your .git directory, you can go to .git/info/exclude, and specify in that file any files or directories you want to exclude from being committed. You can use patterns like "file.txt" to exclude a specific file, or "*.mp3" to exclude all .mp3 files, or "develop/" to exclude everything in the "develop" directory, or "test/*.txt" to exclude every text file in the "test" directory.

Similarly to the exclude file, there is a .gitignore file, that will simply ignore files or patterns that are specified in it in the directory.

Removing Files `> git rm <file>`	This command will remove a file from your local repository, and after committing and pushing will remove the file from the origin.
Untracking a File `> git rm --cached <file>`	This command will tell Git to stop paying attention to a file, without deleting it from your local repository.
Default Editor `> git config --global core.editor <editor>`	This will set your default editor to the editor of your choice.
Merge Tool Default `> git config --global merge.tool <tool>`	This will set your default merge conflict tool to the tool of your choice.
Local Config `> git config --list`	This will list out the local configurations that are set.

There is going to come a point, probably sooner than you think, that you will tire of writing the same git commands over and over again. Having to fully write out the pattern you want to use for a git log, or even as simple as git status takes time to write.

Fortunately, in most CLI (Command Line Interfaces) there exists aliases, which can effectively make a shorthand git command to call the full version when used.

Aliases `> git config alias.<name> <command>`	You can set an alias to reduce to time it takes to write out any command. Ex. git config alias.st status will set the command "git st" to run the "git status" command.