**Git notes**

https://git-scm.com/book/en/v2/

<https://www.atlassian.com/git/tutorials/>

Git has three main states that your files can reside in: modified, staged and commited. Committed means that the data is safely stored in your local database. Modified means that you have changed the file but have not committed it to your database yet. Staged means that you have marked a modified file in its current version to go into your next commit snapshot.

**Create a git repository**

You can get a Git project using two main approaches. The first takes an existing project or directory and imports it into Git. The second clones an existing Git repository from another server.

First approach:

$ **git init**

$ git add \*.c

$ git add LICENSE

$ git commit -m 'initial project version'

**git clone --branch [branch\_name] [URL]:** clone a specific branch at the remote url, instead of where the remote HEAD is pointing to (usually the master branch)

Remember that each file in your working directory can be in one of two states: tracked or untracked. Tracked files are files that were in the last snapshot; they can be unmodified, modified, or staged. Untracked files are everything else – any files in your working directory that were not in your last snapshot and are not in your staging area. When you first clone a repository, all of your files will be tracked and unmodified because Git just checked them out and you haven’t edited anything.

**git config --list**

**git config user.name**

**git help config**

**git status**

**git checkout -- file.txt**

**git add** : to stage files, to add to repository

**git add** is a multipurpose command – you use it to begin tracking new files, to stage files, and to do other things like marking merge-conflicted files as resolved. It may be helpful to think of it more as “add this content to the next commit” rather than “add this file to the project”.

**git add <directory>:** stage all changes in <directory>

To see what you’ve changed but not yet staged, type **git diff** with no other arguments.

That command compares what is in your working directory with what is in your staging area. The result tells you the changes you’ve made that you haven’t yet staged.

If you want to see what you’ve staged that will go into your next commit, you can use **git diff --staged**. This command compares your staged changes to your last commit.

**git diff --base <filename>:** View the conflicts against the base file

**git diff <sourcebranch> <targetbranch>:** preview changes, before merging

Committing Your Changes: **git commit -m “message”**

Anything that is still unstaged – any files you have created or modified that you haven’t run git add on since you edited them – won’t go into this commit. They will stay as modified files on your disk.

Commit records the snapshot you set up in your staging area. Anything you didn’t stage is still sitting there modified; you can do another commit to add it to your history. Every time you perform a commit, you’re recording a snapshot of your project that you can revert to or compare to later.

Adding the -a option to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the git add part: **git commit -a -m 'skipped stage'**

Remove files

To remove a file from Git, you have to remove it from your tracked files (more accurately, remove it from your staging area) and then commit. The **git rm** command does that, and also removes the file from your working directory so you don’t see it as an untracked file the next time around. The next time you commit, the file will be gone and no longer tracked

If you modified the file and added it to the staging area already, you must force the removal with the -f option: **git rm -f**

**git rm \\*~**: This command removes all files whose names end with a ~.

**git rm --cached filename.txt**

keep the file in your working tree but remove it from your staging area. In other words, you may want to keep the file on your hard drive but not have Git track it anymore.

**git mv file\_from file\_to**: rename/move files

**git log**: view commit history

git log -p **-2**

-p, which shows the difference introduced in each commit. You can also use -2, which limits the output to only the last two entries

git log –stat

if you want to see some abbreviated stats for each commit, you can use the --stat option

git log –oneline

**git log --oneline --decorate --graph –all**

**git log –oneline origin/master**: see commits in origin/master branch

git log –pretty=oneline

git log --pretty=format:"%h - %an, %ar : %s"

git log --pretty=format:"%h %s" –graph

If you want to see which commits modifying test files in the Git source code history were committed by Junio Hamano in the month of October 2008 and are not merge commits:

git log --pretty="%h - %s" --author=gitster –since="2008-10-01" --before="2008-11-01" --no-merges -- t/

**git tag 1.0.0 <commitID>**: use tagging to mark a significant changeset, such as a release.

If you commit and then realize you forgot to stage the changes in a file you wanted to add to this commit, you can do something like this:

$ git commit -m 'initial commit'

$ git add forgotten\_file

$ **git commit –amend**

You end up with a single commit – the second commit replaces the results of the first.

**git commit –amend**: It lets you combine staged changes with the previous commit instead of committing it as an entirely new snapshot. It can also be used to simply edit the previous commit message without changing its snapshot. DO NOT amend public commit!

git commit –amend **--no-edit:** allow you to make the amendment without changing its commit message.

**git reset HEAD file.txt:** unstage a staged file

**git checkout – file.txt**: unmodify a modified File

Remote

**git remote -v**: showing remote

Add an existing remote repository to local working directory:

**git** **remote** **add** leti [git@github.ibm.com:chunling/leti.git](https://github.com/paulboone/ticgit)

Fetch and pull from remote

**git fetch <remote>:** fetch all branches from the remote. It also downloads all the required commits and files from remote.

**Git fetch <remote> <branch>**: only fetch the specified branch

The command goes out to that remote project and pulls down all the data from that remote project that you don’t have yet. *After you do this, you should have references to all the branches from that remote, which you can merge in or inspect at any time*.

If you clone a repository, the command automatically adds that remote repository under the name “origin”. So, **git fetch origin** fetches any new work that has been pushed to that server since you cloned (or last fetched from) it. It’s important to note that the git fetch command only downloads the data to your local repository – it doesn’t automatically merge it with any of your work.

**git clone** command automatically sets up your local master branch to track the remote master branch (or whatever the default branch is called) on the server you cloned from. Running **git pull** generally fetches data from the server you originally cloned from and automatically tries to merge it into the code you’re currently working on.

**git push origin master**: push it upstream. To push your master branch to your origin server (again, cloning generally sets up both of those names for you automatically. Here master is your local master branch!!).

This command works only if you cloned from a server to which you have write access and if nobody has pushed in the meantime. If you and someone else clone at the same time and they push upstream and then you push upstream, your push will rightly be rejected. You’ll have to fetch their work first and incorporate it into yours before you’ll be allowed to push.

**git push --all origin**: Push all branches to your remote repository

See more information about a remote

**git branch -r**: show remote branches

**git remote show** leti: inspect a remote. For example: **git remote show origin**.

**git remote remove** or **git remote rm:** remove a remote. E.g. git remote remove leti

**git remote rename oldname newname:** rename a remote. E.g.: git remote rm leti

The **git checkout** command serves three distinct functions: checking out files, checking out commits, and checking out branches.

**git checkout <commit>**

Update all files in the working directory to match the specified commit. You can use either a commit hash or a tag as the <commit> argument. Checking out an old commit is a read-only operation. *Nothing* you do will be saved in your repository. To continue developing, you need to get back to the “current” state of your project: git checkout master

**git checkout <commit> <file>**

Checking out an old file does affect the current state of your repository. It serves as a way to revert back to an old version of an individual file.

If you only wanted to see the hello.py file from the old commit: git checkout a1e8fb5 hello.py

If you don’t want to keep the old version, you can check out the most recent version with the following: git checkout **HEAD** hello.py

**git revert <commit>**

Generate a new commit that undoes all the changes introduced in <commit>, then apply it to the current branch.

**git revert HEAD**: undo the last commit

**git reset file**: unstage a file, and leave working directory unchanged

**git reset**: unstage all files in the staging area, and leave working directory unchanged.

**git reset –hard**: Reset the staging area and the working directory to match the most recent commit.

**git reset <commit>:** Move the current branch tip/pointer backward to <commit>, reset the staging area to match, but leave the working directory alone. All changes made since <commit> will reside in the working directory, which lets you re-commit the project history using cleaner, more atomic

snapshots.

**git reset –hard <commit>:** Move the current branch tip/pointer backward to <commit> and reset both the staging area and the working directory to match. This obliterates (destroys utterly) not only the uncommitted changes, but all commits after <commit>, as well.

The **git reset HEAD~2** command moves the current branch backward by two commits, effectively removing the two snapshots we just created from the project history.

**git reset --hard HEAD:** your work space will be back to the last committed state. Any uncommitted work will be lost.

Never use git reset <commit> when any snapshots after <commit> have been pushed to a public repository!!!

Drop all your local changes and commits, fetch the latest history from the server and point your local master branch at it:

git fetch origin  
**git reset --hard origin/master**

**Branches**

Git doesn’t store data as a series of change sets or differences, but instead as a series of snapshots!

When you make a commit, Git stores a commit object that contains a pointer to the snapshot of the content you staged. This object also contains the author’s name and email, the message that you typed, and pointers to the commit or commits that directly came before this commit (its parent or parents).

When you make a commit, Git repository contains: blobs for the contents of staged files (one blob for each staged file), one tree that specifies which file names are stored as which blobs, and one commit object with the pointer to the tree, the pointer to parent commits and all the commit metadata.

A branch in Git is simply a lightweight movable pointer to one of these commits. The default branch name in Git is master. The “master” branch in Git is not a special branch. It is exactly like any other branch.

Create branch

Doing so creates a new pointer for you to move around. Let’s say you create a new branch called “testing”: **git branch** testing. This creates a new pointer to the same commit you’re currently on.

How does Git know what branch you’re currently on? It keeps a special pointer called HEAD. It is a pointer to the local branch you’re currently on. The **git branch** command only created a new branch – it didn’t switch to that branch.

Because a branch in Git is actually a simple file that contains 40 character SHA-1 checksum of the commit it points to, branches are cheap to create and destroy. Creating a new branch is as quick and simple as writing 41 bytes to a file.

Switch branches

**git checkout**. Let’s switch to the new testing branch: git checkout testing

This moves HEAD to point to the testing branch.

Switching branches changes files in your working directory. when you switch branches, Git resets your working directory to look like it did the last time you committed on that branch. It adds, removes, and modifies files automatically to make sure your working copy is what the branch looked like on your last commit to it.

To create a branch and switch to it at the same time, you can run **git** **checkout -b** [branch]

git checkout -b iss53: Switched to a new branch "iss53"

This is shorthand for: git branch iss53; git checkout iss53

Delete a branch: **git branch -d** [branch]

Branch merge: check out the branch you wish to merge into and then run the **git merge**.

git checkout master

git merge leti

When automatic merge fail, run git status to check merge conflicts. Anything that has merge conflicts and hasn’t been resolved is listed as unmerged.

After you’ve resolved each of these sections in each conflicted file, run git add on each file.

To use a graphical tool to resolve these issues, you can run **git mergetool**

Recommend kDiff3

To get a simple listing of your current branches: **git branch**

To see the last commit on each branch, you can run **git branch -v**

To see which branches are already merged into the branch you’re on, run **git branch –merged**

Note: branches on this list without the \* in front of them are generally fine to delete with git branch -d

To see all the branches that contain work you haven’t yet merged in, run **git branch –no-merged**

Abort a merge: **git merge –abort**

Merge ignoring whitespace with -Xignore-all-space or -Xignore-space-change option.

The first option ignores whitespace completely when comparing lines, the second treats sequences of one or more whitespace characters as equivalent.

**git merge -Xignore-space-change [branch]**

**Remote branches**

You can get a full list of remote references explicitly with **git ls-remote [remote]**, or **git remote show [remote]** for remote branches as well as more information.

For example: git ls-remote origin; git remote show origin

Git’s clone command automatically names the remote repository **origin** for you, pulls down all its data (all remote branches, master, and non-master), creates a pointer to where its master branch is, and names it **origin/master** ***locally.*** Git also gives you your own **local** **master** **branch** starting at the same place as origin’s master branch, so you have something to work from.

“origin” is the default name for a remote when you run git clone.

If you do some work on your local master branch, and, in the meantime, someone else pushes to remote repository and updates its master branch, then your histories move forward differently. As long as you stay out of contact with your origin server, your origin/master pointer doesn’t move.

To synchronize your work, you run a **git fetch origin** command. This command looks up which server “origin” is, fetches any data from it that you don’t yet have, and updates your local database, moving your origin/master pointer to its new, more up-to-date position.

Pushing: **git push [remote\_name] [local\_branch\_name]**

It pushes the local repo branch under [local\_branch\_name] to the remote repo at [remote\_name]

**git push origin serverfix**

Take my serverfix local branch and push it to update the remote’s serverfix branch

**git push origin serverfix:awesomebranch**

Push your local serverfix branch to the awesomebranch branch on the remote project. If there isn’t “awesomebranch” on the server, remote repository will create one.

when you do a fetch that brings down new remote-tracking branches, you don’t automatically have local, editable copies of them. You don’t have a new serverfix branch – you only have an origin/serverfix pointer that you can’t modify.

If you want your own serverfix branch that you can work on, run: **git checkout -b serverfix origin/serverfix**

Tracking branches

Checking out a local branch from a remote-tracking branch automatically creates a “tracking branch” (and the branch it tracks is called an “upstream branch”). Tracking branches are local branches that have a direct relationship to a remote branch.

When you clone a repository, it generally automatically creates a master branch that tracks origin/master. However, you can set up other tracking branches if you wish, running

**git checkout -b [branch] [remotename]/[branch]**, or

**git checkout --track origin/[branch]**

**git checkout -b sf origin/serverfix**: your local branch sf will automatically pull from origin/serverfix.

If you already have a local branch and want to set it to a remote branch you just pulled down, or want to change the upstream branch you’re tracking, use the **-u** or **--set-upstream**-to option to git branch to explicitly set it at any time.

**git branch -u origin/serverfix**: Branch set up to track remote branch serverfix from origin.

When you have a tracking branch set up, you can reference its upstream branch with the **@{upstream}** or **@{u}** shorthand. E.g: git merge @{u}

Inc conclusion, there are 3 ways to have a tracking branch set up:

(1) git clone (2) git checkout (3) set it explicitly by -u or -set-upstream-

**git branch -vv**: see what tracking branches you have set up

**git pull** = git fetch + git merge

**git pull –rebase <remote>:** instead of using git merge to integrate the remote branch with the local change, use git rebase

Delete remote branches

To delete your serverfix branch from the server, run: **git push origin --delete serverfix**

Rebase

Rebasing replays changes from one line of work onto another in the order they were introduced, whereas merging takes the endpoints and merges them together.

With the rebase command, you can take all the changes that were committed on one branch and replay them on another one. It makes a clean history.

**git rebase <base>**

Rebase the current branch onto <base>, which can be any kind of commit reference (an ID, a branch name, a tag, or a relative reference to HEAD).

git checkout experiment

**git rebase master**

git checkout master

git merge experiment

It works by going to the common ancestor of the two branches (the one you’re on and the one you’re rebasing onto), getting the diff introduced by each commit of the branch you’re on, saving those diffs to temporary files, resetting the current branch to the same commit as the branch you are rebasing onto, and finally applying each change in turn.

There could be rebase merge conflict. If so, then need to resolve the conflicts for each commit and run **git rebase –continue**.

Often, you’ll do this to make sure your commits apply cleanly on a remote branch – perhaps in a project to which you’re trying to contribute but that you don’t maintain. In this case, you’d do your work in a branch and then rebase your work onto origin/master when you were ready to submit your patches to the main project. That way, the maintainer doesn’t have to do any integration work – just a fast-forward or a clean apply.

**git rebase -i <base>**

Running git rebase with the -i flag begins an interactive rebasing session. Instead of blindly moving all the commits to the new base, interactive rebasing gives you the opportunity to alter individual commits in the process. This lets you clean up history by removing, splitting, and altering an existing series of commits.

*It lets developers commit a “messy” history while they’re focused on writing code, then go back and clean it up after the fact.*

Example of a push flow with rebase

Git checkout master

Git fetch origin master

Git rebase -i origin/master (squash commits, fix up commit messages etc.)

Git push origin master

Changing multiple commit messages

If you want to change the last three commit messages, run **git rebase -i HEAD~3**

Stashing

**git stash** temporarily shelves (or stashes) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on.

It is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.

By default Git won’t stash changes made to untracked or ignored files.

Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

**git stash save “message’:** annotate stash with a message description. The message shows up in git stash list

To see which stashes you’ve stored, run: **git stash list**

To apply a stash: **git stash apply**

To apply one of the older stashes, specify it by naming it, like**: git stash apply stash@{2}**

To remove a stash, run **git stash drop** with the name of the stash to remove.

git stash drop stash@{0}

**git stash show**: view the summary of a stash

**git stash show -p**: view the full diff of a stash

**git stash drop stash@{1}:** delete a stash

**git stash clear**: remove all stashes

**git stash pop:** apply the most recent stash and then immediately drop it from your stack.

The changes to your files were reapplied, but the file you staged before wasn’t restaged. Run

**git stash apply –index** to reapply the staged changes.

**git stash –keep-index**: stash modified files, i.e.: don’t stash anything that you’ve already staged.

**git stash -u**: stash untracked files as well.

If the changes on your branch diverge from the changes in your stash, you may run into conflicts when popping or applying your stash. Instead, you can use git stash branch to create a new branch to apply your stashed changes to:

**git stash branch <branch-name>:** create a branch out of stash.

**git stash branch add-stylesheet stash@{1}**

This checks out a new branch “add-stylesheet” based on the commit that you created your stash from, and then pops your stashed changes onto it.

The **git clean** command removes untracked files from your working directory.

**git clean -d -n**: check what untracked files would be cleaned.

**git clean -d -f**: remove anything that is untracked form working directory.

Run: git reset --hard and git clean -f commands if you’ve made some embarrassing developments in your local repository and want to burn the evidence.

**GitHub flow**

Github is designed around a particular work, centered on Pull Requests.

1. Fork the project
2. Create a topic branch from master.
3. Make some commits to the topic branch to improve the project.
4. Push this topic branch to your GitHub project.
5. Open a Pull Request on GitHub.
6. Discuss, and optionally continue committing.
7. The project owner merges or closes the Pull Request.

Pull request

Pull requests are a mechanism for a developer to notify team members that they have completed a feature. Once their feature branch is ready, the developer files a pull request. This lets everybody involved know that they need to review the code and merge it into the master branch. It is also a dedicated forum for discussing the proposed feature.

When you file a pull request, all you’re doing is ***requesting*** that another developer (e.g., the project maintainer) ***pulls*** a branch from your repository into their repository.

Find remote branch creators:

**git for-each-ref --format='%(committerdate) %09 %(authorname) %09 %(refname)'**