### **GIT BASICS**

If you create a new file in local , it is an untracked file. When you add the newly created file to the staging area , git starts tracking the newly created file.

Git ls-files to show all the files git is tracking.

Git commit --amend will allow us to allow us to amend our commit message for the current commit.

Git cherry-pick <commit\_id> will take a specific commit from one branch and apply it on top of the current branch from where you are running the cherry-pick command. Beware , that merge conflicts might occur in cherry-pick.

### **GIT DIFF**

Git diff checks the difference between the staging area(where we add) and the local area.

Git diff HEAD checks the difference between local and last commit.

Git diff --staging HEAD checks the difference between staging area and last commit

Git diff commitID1 commitID2 checks the difference between two commits

Git diff master origin/master checks the difference between HEAD of local repo and HEAD of remote repo.

### **GIT BRANCHING**

Git branch shows all local branches

Git branch -a shows all local and remote branches.

Git branch -m oldbranchname newbranchname

Git branch -d <branch name to delete> ( You cannot be on the same branch you would want to delete)

Git checkout -b <new branch name> to first newly create a branch and then switch to it.

To delete remote branches in git -  **git push origin --delete test**

### **GIT MERGING**

Fast Forward Merge - When you branch out from mainline , and make some commits to branch.

Then you go to mainline and do git merge <branch name> , it is a fast-forward merge , if there are no further commits in the mainline from the time the branch to be merged was cut. Git acts like the branch was never created and all the commits were as if made on mainline itself. NO NEW COMMITS ARE MADE.

Mainline : A - B - C - D

Branch : \

E - F

Now after merging the branch ,

Mainline : A - B - C - D - E -F

You see this fast forward was possible as no commits were being made after the branch was cut out.

However , if we also had made some changes in the mainline branch and then doing merge of new branch , a NEW MERGE COMMIT WILL BE GENERATED and it is RECURSIVE MERGE.

While doing git merge , if a merge conflict happens , resolve the conflict and then commit , it will be merged then with a new commit.

### **GIT REBASE**

In a git rebase ,

Master : A -B - C - D - E

\

Feature: F1-F2

Now in the feature branch if we apply git rebase master

Master : A -B - C - D - E

\

Feature: F1`-F2`

So now if we go to master and do git checkout <feature\_branch> , we get a fast-forward merge.

F1` and F2` are different commits with the same changes.

So basically , changes in the feature branch are rewinded , then the changes of the master branch(after the feature branch was branched out) are applied and then on top of that , changes of feature branch that was rewinded are now applied.

Now rebase conflict might happen , if there is some conflicting changes between master and feature branch.

You can either do git rebase --abort or resolve the conflicts.

Steps to resolve rebase conflict :

1. Resolve the merge conflict manually or using a merge tool

2. Git add the files which you modified to resolve the merge conflict. (and then check if u need a commit)

3. Git rebase --continue.

Also you can do a git pull --rebase origin master - to rebase your local mainline changes on top of new changes made in remote. No new COMMITS are generated.

Git pull without rebasing merges remote changes with your local changes and a new merge commit is created.

### GIT FETCH

Git fetch syncs the metadata with remote and local. Like how many new commits were made in remote. It is a non destructive command. It does not fetches any new file.

### GIT PUSH

**git push command push commits made on a local branch to a remote repository.**

**The git push command basically takes two arguments:**

* **A remote name, for example, origin**
* **A branch name, for example, master**

**Syntax:**

**git push <REMOTENAME> <BRANCHNAME>**

When we create a new branch directly in remote , say RemoteBranchA , and we do git branch -a in local , we won’t be able to see RemoteBranchA in local.

Now do git fetch or a git pull.

Git branch -a -> you would be able to see RemoteBranchA

But if you do git branch -> you would not see RemoteBranchA .

Now do git checkout/switch RemoteBranchA in local and you would be switched to RemoteBranchA in local. And also if you now type git branch it shows both the branches.

git pull

From https://github.com/IamRiddhi/GitTutorial

\* [new branch] 1st\_branch -> origin/1st\_branch

Already up to date.

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git branch

\* main

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git branch -a

\* main

remotes/origin/1st\_branch

remotes/origin/HEAD -> origin/main

remotes/origin/main

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git fetch

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git branch -a

\* main

remotes/origin/1st\_branch

remotes/origin/HEAD -> origin/main

remotes/origin/main

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git branch

\* main

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git checkout 1st\_branch

Switched to a new branch '1st\_branch'

Branch '1st\_branch' set up to track remote branch '1st\_branch' from 'origin'.

C:\Users\Riddhi Dutta\Desktop\Git Shits\GitTutorial>git branch

\* 1st\_branch

main

### **GIT STASH**

Git stash is used for removing any changes from local. (u can later get back those changes while typing git stash apply). This is mainly done , if you were in the middle of unfinished work and then u had to make a quickfix on a bug. So u stash your current changes , which you can’t commit now and then work on the bug fix , commit and then git stash apply to get back your unfinished work you were working on previously.

By default git stash uses git stash save.

Git stash list - to get a list of stashes.

Git stash drop command drops the last stash from the list.

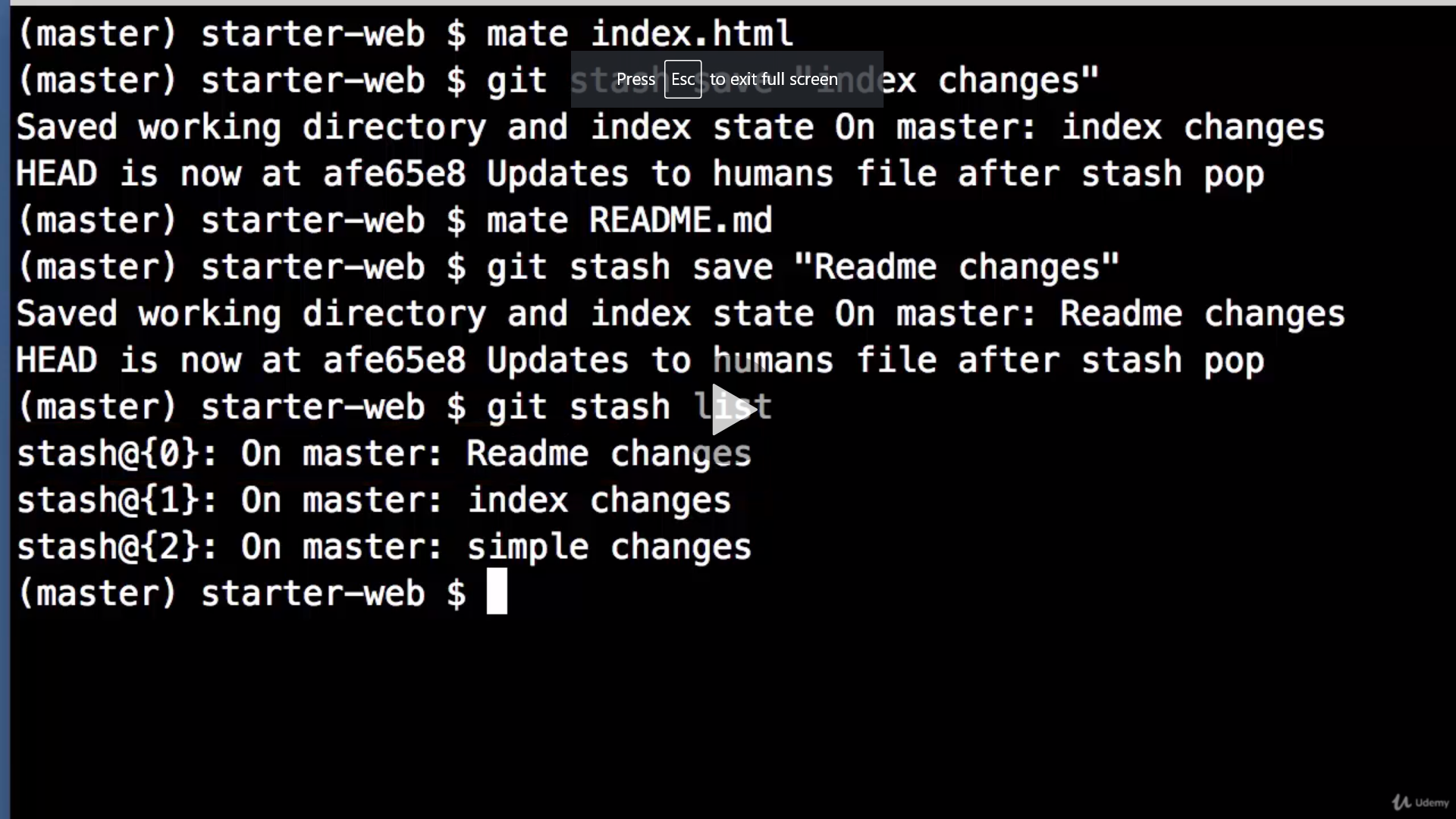
Git stash only comes into picture with tracked files.

To include untracked files for stash as well use , git stash -u.

Use git stash pop which is git stash apply + git stash drop.

For multiple stashes , you can type git stash save “stashing message” - this stashing message will help you to differentiate between multiple stashes.

Git stash works like a stack. Last stash will be on index 0 or top of the list.



Git stash show <stash\_ref> - to show changes of a particular stash.

Eg git stash show stash@{2}

To apply a particular stash and not the last one(for last one do git stash pop) ,

Git stash apply <stash\_ref>

Also you need to drop that stash after applying,

Git stash drop <stash\_ref>

Git stash clear - deletes all stashes from the list.

Git stash stashes staging area and unstaged area tracked files.

Git stash -u stashes staging , unstaged area tracked and untracked files.

Suppose you have made some changes in a mainline , and then you realise , no i want to put all these changes on a feature branch.

Do git stash -u. It clears ur working directory in mainline.

Then do git stash branch <new\_branch\_name> <stash ref>(by default stash\_ref will be 0th index)

As a result you checkout to this new branch , and then the stash is dropped from the list.

When you do git stash pop , the files are not added to staging area even if they were taken from staging area. They are added to local.

### **GIT TAG**

Git tag <tagname> to create a new tag on the current commit. (this is a lightweight tag)

We use tag to specify milestones or adding information to our commits.

Also you can do git tag <tagname> -m <tagging message>

Git tag --list to show the list of tags.

Git show <tag\_name> to show a particular tag.

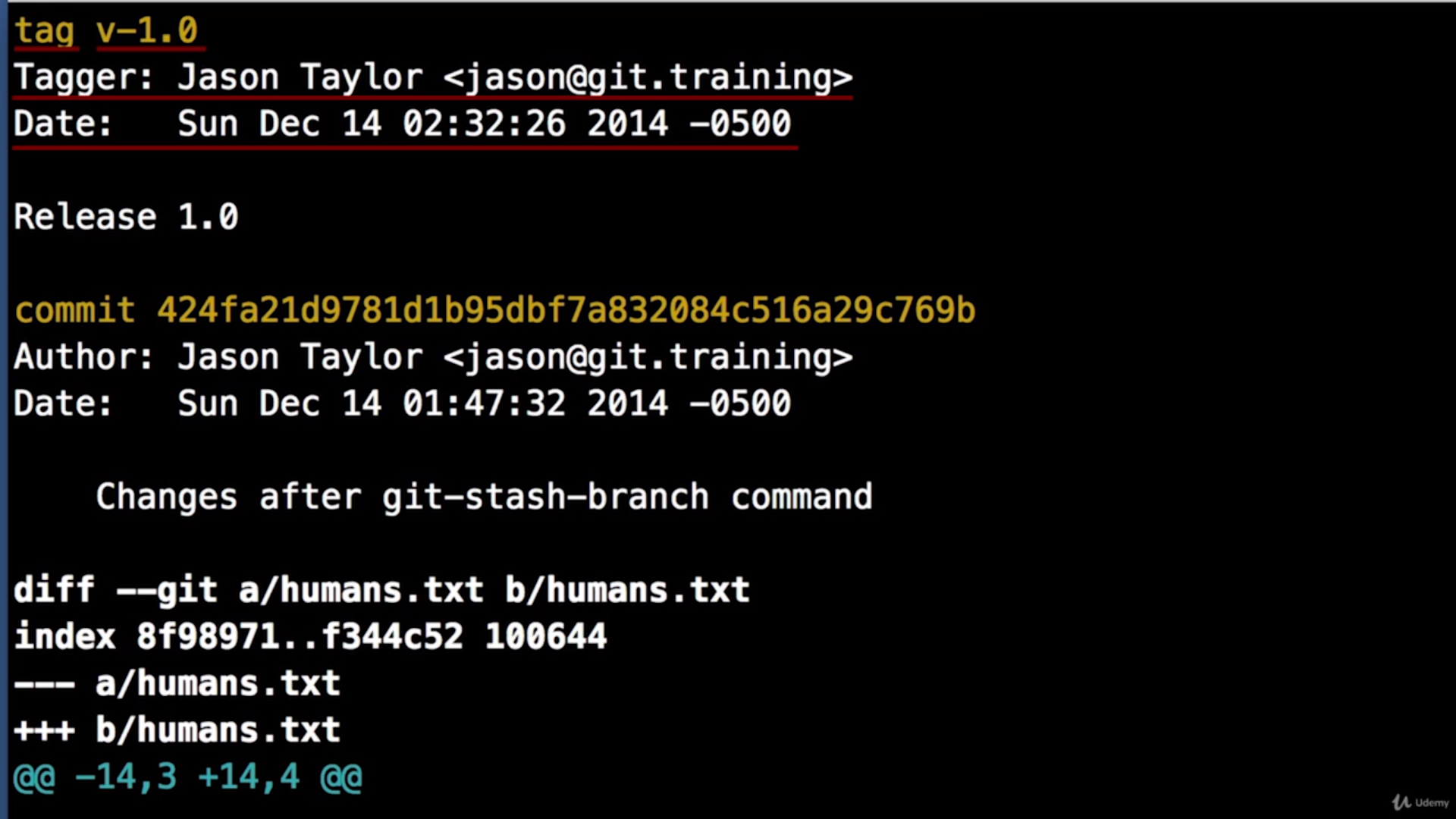
Git tag --delete <tag name> to delete a particular tag

To create annotator tag (with extra information , usually used for version releases)

Type git tag -a <tag\_name>

And then type in the tag message

Annotator tag has more information than a lightweight tag. It forces you for a tagging message whereas lightweight tag won’t force you for a message.



For a lightweight tag if you haven’t provided any message using -m , the information for git show <tag\_name> will start from commit id.

Git tag tags your current commit id.

But , Sometimes you forget to tag.

So in order to tag a specific commit,

Git tag <tagname> <commit-id> (lightweight tag)

OR

Git tag <tagname> <commit-id> -m <tagging message> (lightweight tag with a message)

OR

git tag -a <tagname> <commit-id> (for annotator tag)

Sometimes you might associate a tag to a wrong commit. One way of correcting this would be to delete the tag and then create a new tag with the correct commit. Another way would be to

Git tag <tag\_name> -f <correct\_commit\_id> (lightweight tag)

OR

Git tag <tagname> -f <correct\_commit\_id> -m <tagging message> (lightweight tag with a message)

OR

git tag -a <tagname> -f <correct\_commit\_id> (for annotator tag)

To push one particular tag to remote,

Git push origin <tag\_name>.

Also if the commit for that particular tag was not pushed to remote , with tag the commit will also be pushed to remote , but the commit would be rendered invisible.

Hence all always do git push origin && git push origin --tags

To push all tags in local to remote,

Git push origin --tags.

After we have accidentally pushed one tag to remote,

Type git push origin :<tag\_name\_to\_be\_deleted\_from\_remote>

This tag will be deleted from remote but would still be there on our local.

### **GIT RESET**

(USED TO REVERT COMMITS)

Git reset HEAD^^1. To go back 1 commit.

Or git reset HEAD@{n} to go back n commits.

There is also git reset -- soft -> brings the changes till the commit(that was just resetted) to staging area ready to be committed

(by default git reset is mixed - unstages changes till the commit(that was just resetted). Basically after doing git reset --mixed commit\_id/HEAD^^ and then git status , we would see that the changes till the resetted commit are being asked to be staged.

There is also git reset --hard -> effects both staging area and working directory(local). All the changes till the commit would be removed from working directory(local).

Git reset <commit\_id> - to reset to particular commit.

Git log shows the current commit u r in. if u had done reset , the previous commit goes away.

But if you do a git reflog , you get to see everything. Also checkout and pull logs.

TIME TRAVEL :

Let’s say you commit one change and then do a reset --[hard/mixed/soft] to the previous commit. Now using git reflog , get the commit id (which you won’t get in git log as it is gone after doing git reset) and then type git reset --[hard/mixed/soft (whichever flavor you had used to reset previously)] <commit\_id> to go back to the newly committed thing which you resetted.

But remember git reflog only stores data for the last 30-60 days.