**GIT LAB MANUAL**

**Experiment – 1:**

**Main Git commands.**

* + - 1. Init
      2. Status
      3. Add
      4. Commit
      5. Log
      6. Config
* **git init:**

$ git init

The git init command is used to create a new blank repository. The git init command creates a .git subdirectory in the current working directory. This newly created subdirectory contains all of the necessary metadata.

  To create a blank repository, open command line on your desired directory and run the init command as follows:

****

* **git add:**

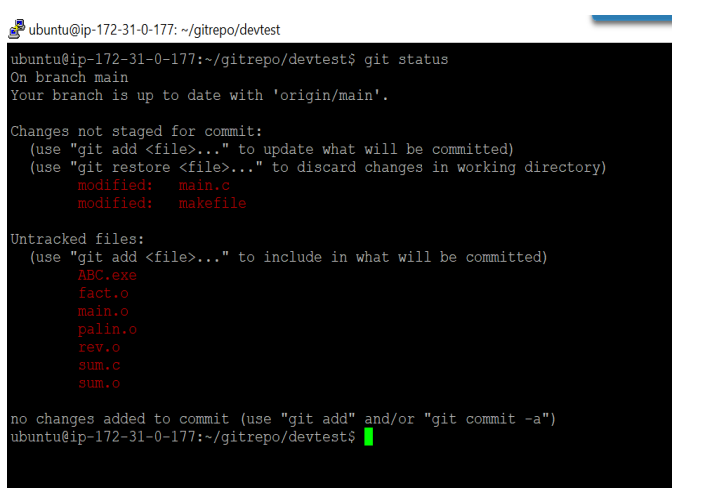
                          git add <filename>

The git add command adds a change in the working directory to the staging area. It tells Git that you want to include updates to a particular file in the next commit.

* **git status:**

git status

The git status command displays the state of the working directory and the staging area. It lets you see which changes have been staged, which haven't, and which files aren't being tracked by Git.

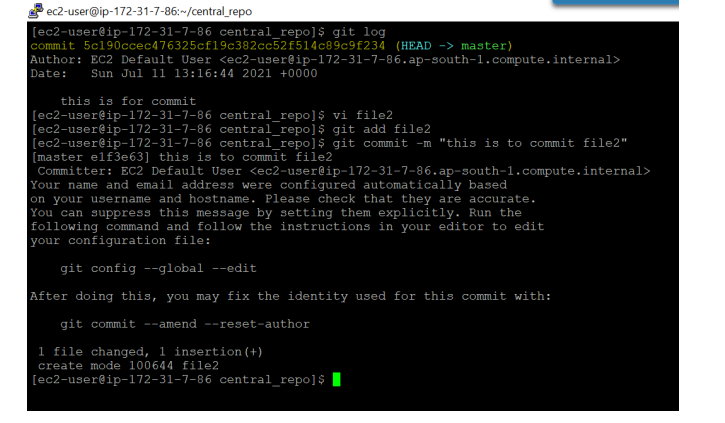
****

* **Git commit:**

git commit -m "Initial commit"

This creates a new commit with the given message. A commit is like a save or snapshot of your entire project. You can now push, or upload, it to a remote repository, and later you can jump back to it if necessary.

IT WILL ADD FILE TO GIT LOCAL REPOSITORY ONLY WHEN YOU RUN GIT ADD AND GIT COMMIT COMMANDS.

****

* **Git log:**

The git log command displays a record of the commits in a Git repository. By default, the git log command displays a commit id, the commit message, and other commit metadata.

****

* **Git config:**

We can use this command to configure git like user name, mail id etc

git config --global user.email "ABC@gmail.com"

git config --global user.name "ABC"

\*\*\*Note: global means these configurations are applicable for all repositories created by git. If we are not using global then it is applicable only for current repository.

$ git config --list To list out all git configurations

$ git config user.name - To display user name

$ git config user.email - To display user email

We can change user name and mail id with the same commands

git config --global user.email "ABC@gmail.com"

git config --global user.name "ABC"

**Experiment – 2:**

**Creating and Managing Branches**

**a)**  Creating a New Branch – feature branch

**b)** Merging “Main” and “feature” Branches and understand git merge conflict.

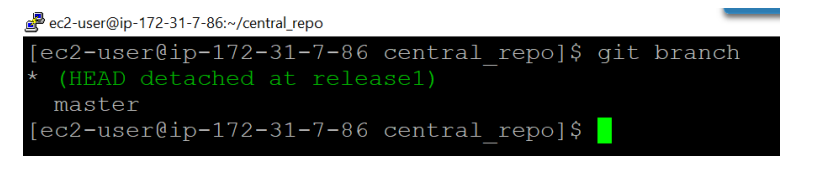
**c)** Write a command to identifyif the branch is already merged into master.

**a) Creating a New Branch – feature branch**

Branch - A branch represents an independent line of development. When you are building new features, we have to check its compatibility with the existing features Hence, we use Master branch. It is a default branch. The default branch name in Git is master or main. As you start making commits, you're given a master branch that points to the last commit you made. Every time you commit, the master branch pointer moves forward automatically. Note.

The “master” branch in Git is not a special branch. Let’s consider we have B1 as a code, B2 as different set of code, later we merge B1 and B2 into single branch. For parallel development, branch is used. Two people or two teams when they work on the same source code to development different features and integrate these changes by merging.

**DEFAULT BRANCH IS MASTER.**

****

There are 3 different ways to create a branch

1. Create a new branch git branch

2. Create a new branch same as existing branch git branch

3. Create a new branch using tag git branch

To create a branch, use the command

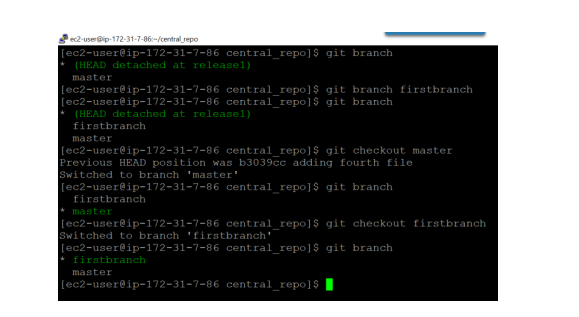
Git branch Git branch branch1

Branch will be created from Master.

\*on the particular name indicates that am on that branch.

Branch1

\*Master Means now am on Master branch

****

1. **Write a command to identify if the branch is already merged into master.**

git branch -merged It lists the branches that have been merged into the current branch.

git branch -no-merged It lists the branches that have not been merged.

**Experiment - 3**

**Creating and Managing Branches**

**a)** How to Ignoring unwanted Files and Directories in git.

**b)** Check the difference between git checkout [branch name] and git checkout -b [branch name].

**c)** Write a command to pick a commit from one branch and apply it to another.

**a) How to Ignoring unwanted Files and Directories in git.**

It is very common requirement that we are not required to store everything in the repository. We have to store only source code files like .java files etc.

Not required to store ◊README.txt

Not required to store◊log files

We can request git, not to consider a particular file or directory. We have to provide these files and directories information inside a special file .gitignore

  .gitignore File: We have to create this file in working directory.

# Don't track abc.txt file abc.txt

  # Don't track all .txt files \*.txt

# Don't track logs directory logs/

#Don't track any hidden file .\*

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)

$ touch a.txt b.txt Customer.java

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)

  $ mkdir logs

lenovo@DESKTOP-ECE8V3R MINGW64 /d/gitprojects/project8 (master)

$ touch logs/server.log logs/access.log

$ git status

On branch master Untracked files: (use "git add ..." to include in what will be committed) Customer.java a.txt b.txt logs/

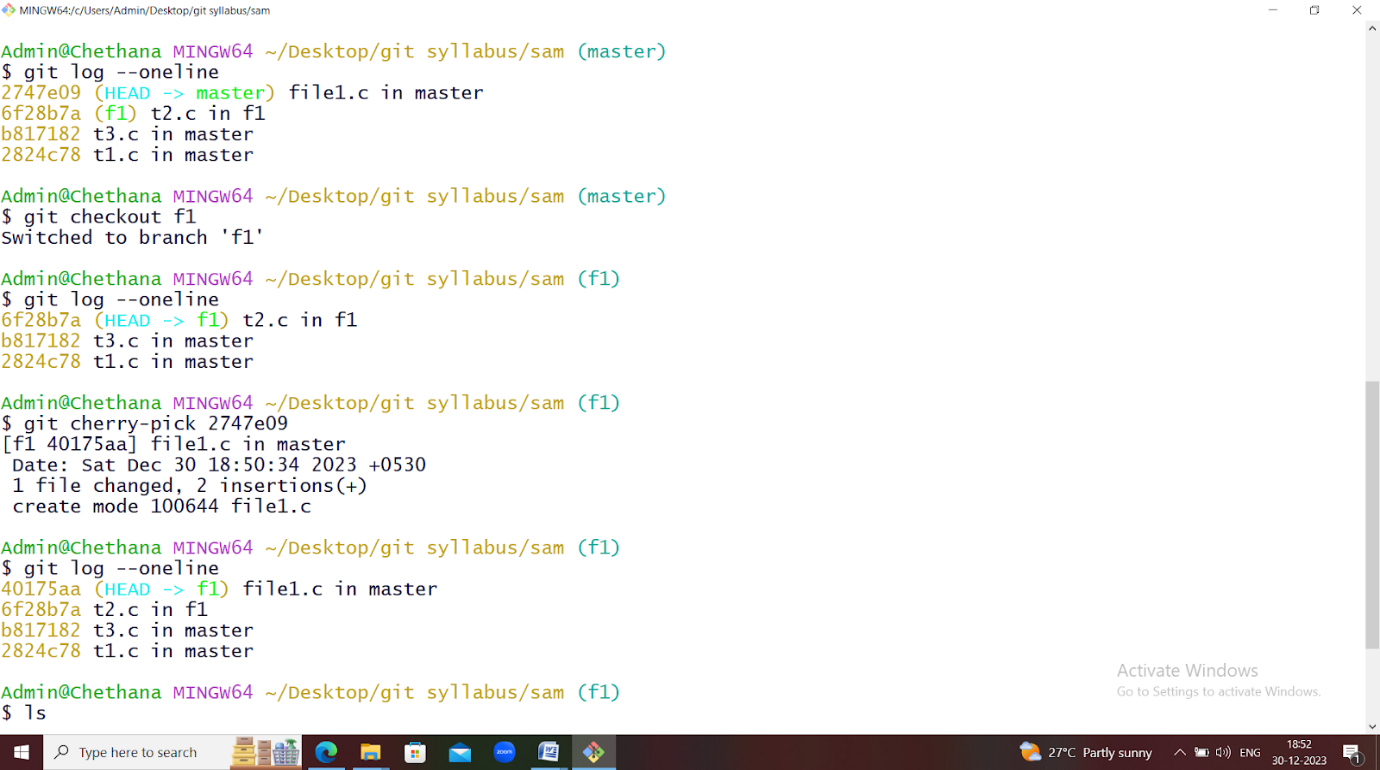
1. **check the difference between git checkout [branch name] and git checkout -b [branch name].**

* git checkout [branch name]: To switch to an existing branch.
* git checkout -b [branch name]: To create a new branch and switch to it:

1. **Write a command to pick a commit from one branch and apply it to another.**

Cherry-picking in Git stands for applying some commit from one branch into another branch.

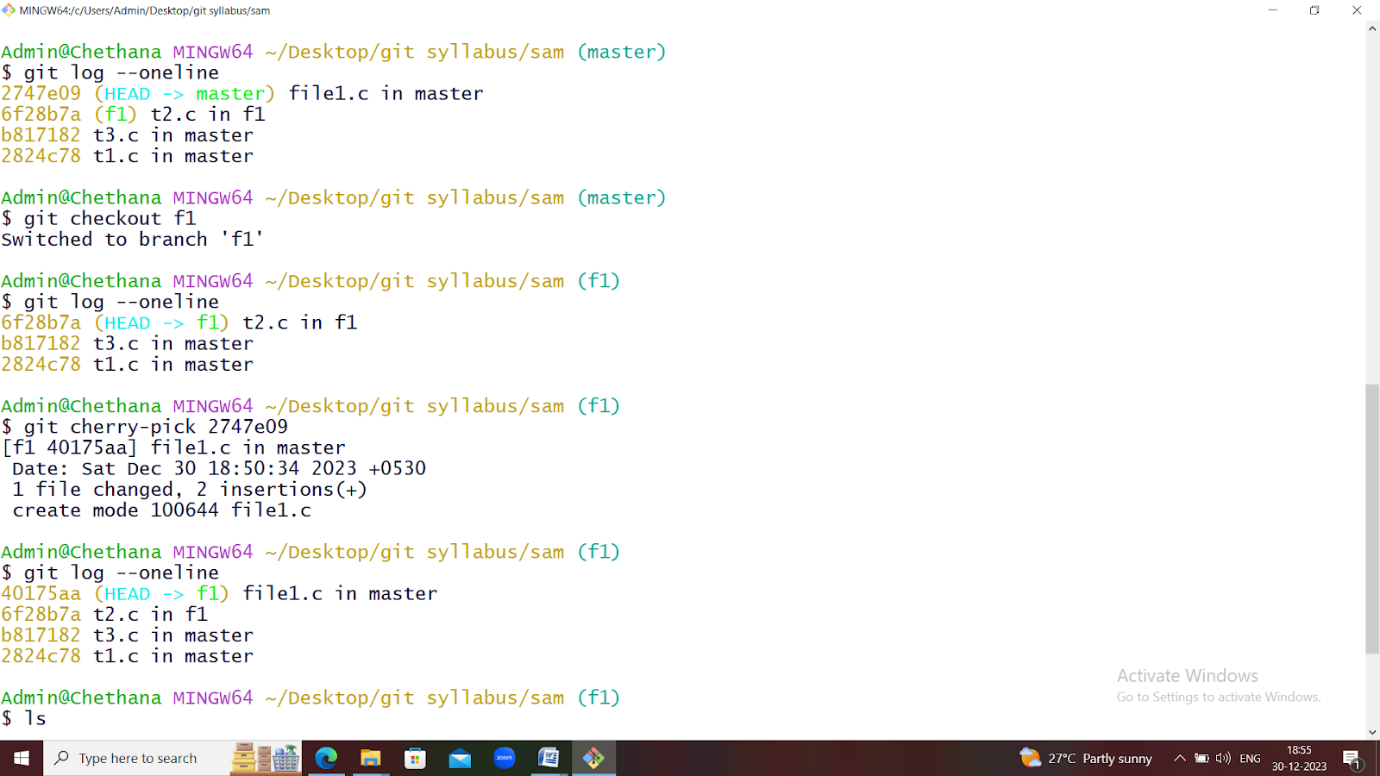
git cherry-pick commit\_ID



For example, am in master branch, which has 4 commits. Am taking commit id of file1.c creation with commit –id 2747e09

I will go to feature-branch  f1, there are only 3 commits in this branch, now I will run the following command

                    Git cherry-pick  2747e09(commit id of creating file1.c)



Now you can see, 3 commits is changed to 4 commits. And File file1.c  is created in f1. To merge more than one commit,

             git cherry-pick  commit1  commit2  commit3.

**Experiment – 4:**

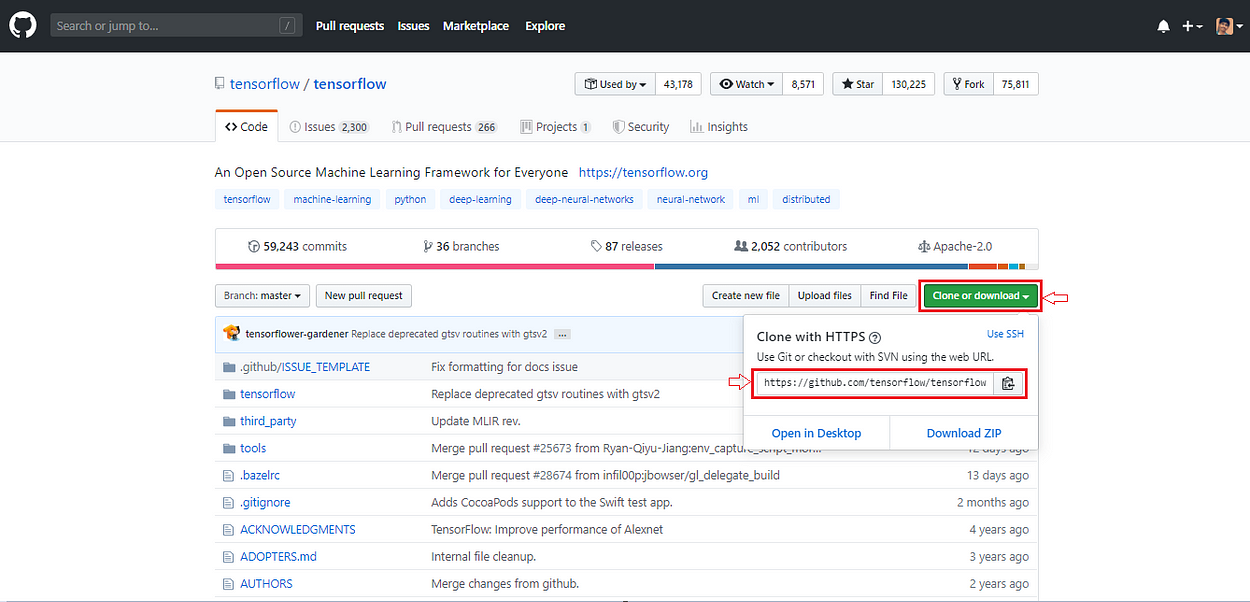
**Collaboration and Remote Repositories**

Clone a remote Git repository to your local machine.

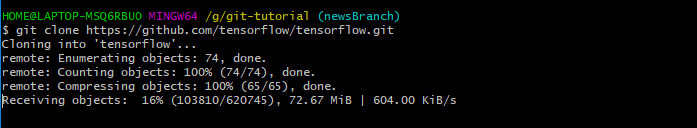
**GIT CLONE**

Git clone is a Git command-line utility that is used to target an existing repository and create a clone, or copy of the target repository. The command for cloning the repository is:

**git clone   <***repository-link***>**



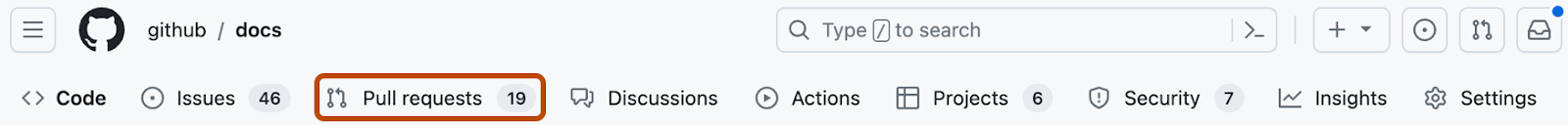
In the above image, it’s a TensorFlow repository and we want to clone it means to download it in our system. We have  to use the link to clone this repository.



Cloning the repo

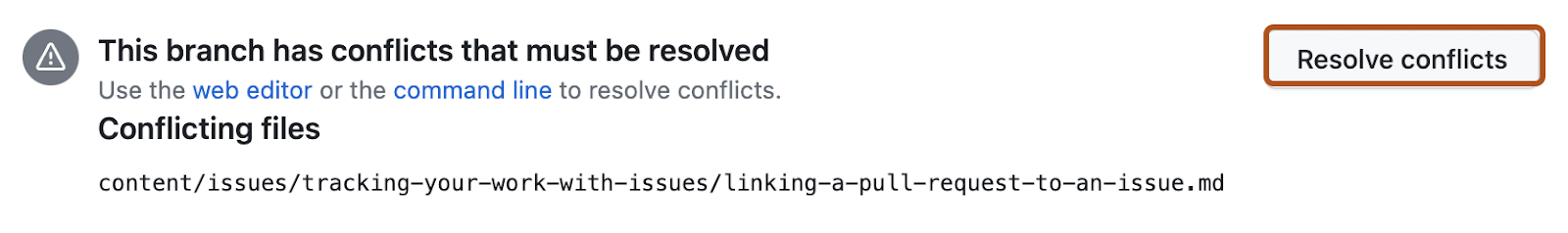
* Resolve simple merge conflicts that involve competing line changes on GitHub.

1. Under your repository name, click Pull **requests**.



2. In the "Pull Requests" list, click the pull request with a merge conflict that you'd like to resolve.

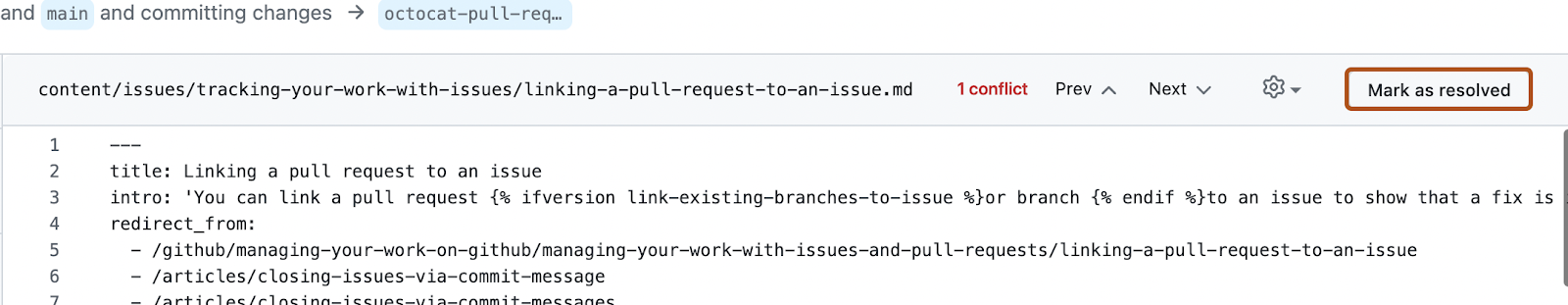
3. Near the bottom of your pull request, click **Resolve conflicts**.



**4.** Decide if you want to keep only your branch's changes, keep only the other branch's changes, or make a brand new change, which may incorporate changes from both branches. Delete the conflict markers <<<<<<<, =======, >>>>>>> and make the changes you want in the final merge.

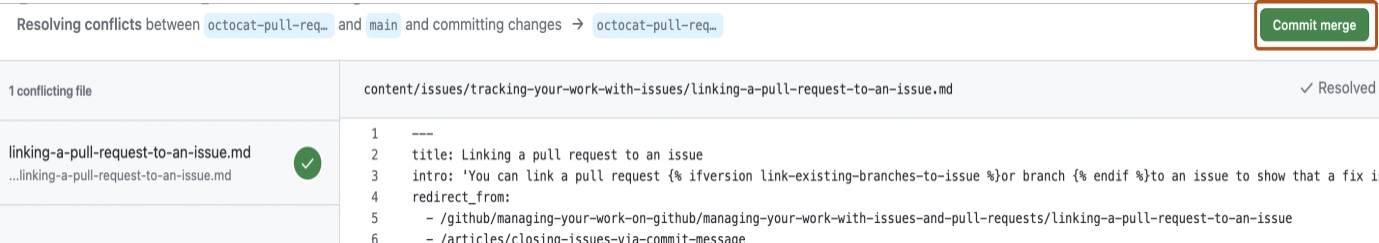
**5.** If you have more than one merge conflict in your file, scroll down to the next set of conflict markers and repeat steps four and five to resolve your merge conflict.

**6.** Once you've resolved all the conflicts in the file, click **Mark as resolved**.



**7.** If you have more than one file with a conflict, select the next file you want to edit on the left side of the page under "conflicting files" and repeat steps four through seven until you've resolved all of your pull requests merge conflicts.

**8.** Once you've resolved all your merge conflicts, click **Commit merge**. This merges the entire base branch into your head branch.



**9.** If prompted, review the branch that you are committing to.

If the head branch is the default branch of the repository, you can choose either to update this branch with the changes you made to resolve the conflict, or to create a new branch and use this as the head branch of the pull request.

**10.**To merge your pull request, click **Merge pull request**. For more information about other pull request merge options, see "[Merging a pull request](https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/incorporating-changes-from-a-pull-request/merging-a-pull-request)."

**Experiment – 5:**

**Undo Operations on Git.**

Execute a command “git revert” and “git rebase” then find the difference.

**Git Revert**

* git revert is used to undo a commit safely.
* It does NOT delete or modify commit history.
* It is used when you want to undo a commit without removing history.

Example:

ABC ----> C\* (MISTAKE)

**C\*** = wrong commit

ABC ----> C\* ----> REVERT(C)

**REVERT(C)** = new commit that cancels the wrong commit

ABC ----> C\* ----> REVERT(C) ----> D

**D** = your correct new commit

Commands for git revert

echo "first line" > newfile.txt

git add newfile.txt

git commit -m "commit 1"

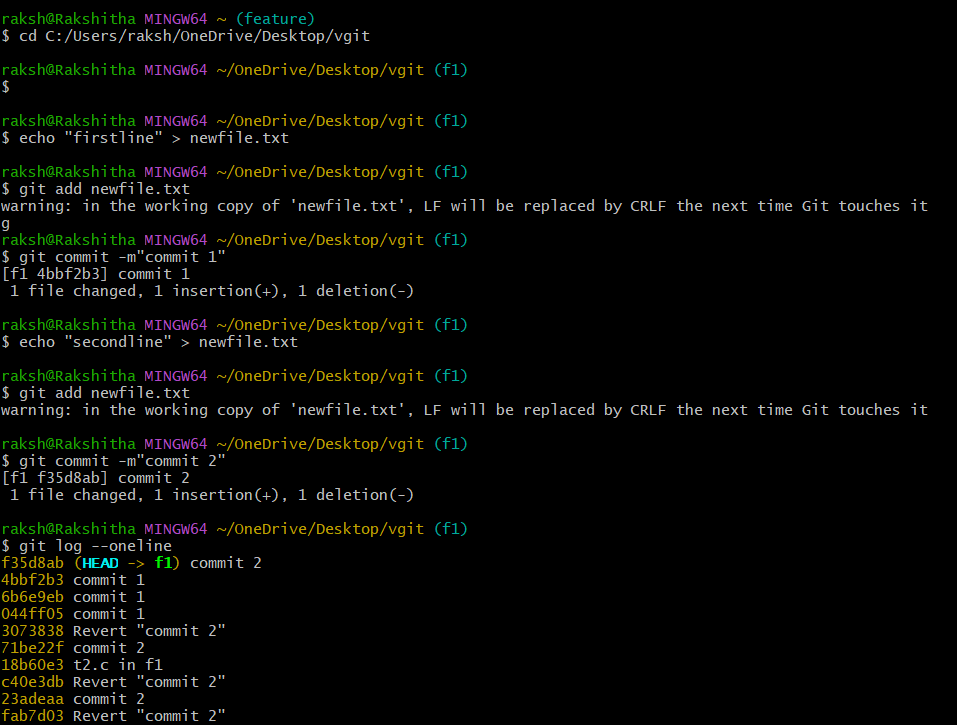
echo "second line" > newfile.txt

git add newfile.txt

git commit -m "commit 2"

git log --oneline

git revert HEAD



**Git Merge**

* git merge is used to combine changes from two branches.
* It creates a new merge commit.
* It keeps the complete history of both branches.

Example :

Before merge:

MASTER: A---B---C

FEATURE: D---E

After merge:

MASTER: A---B-- C ---M(merge commit )

FEATURE: D---E

* **D, E** = new commits on another branch
* **MERGE-COMMIT** = brings **D + E** together
* **No history is deleted** (safe)

Commands for git merge

git init

echo “line in t1.c” > t1.c

git add t1.c

git commit -m “t1.c in master”

git branch

git checkout master

echo “line in t3.c “ > t3.c

git add t3.c

git commit -m “t3.c in master”

git log --oneline

git checkout f1

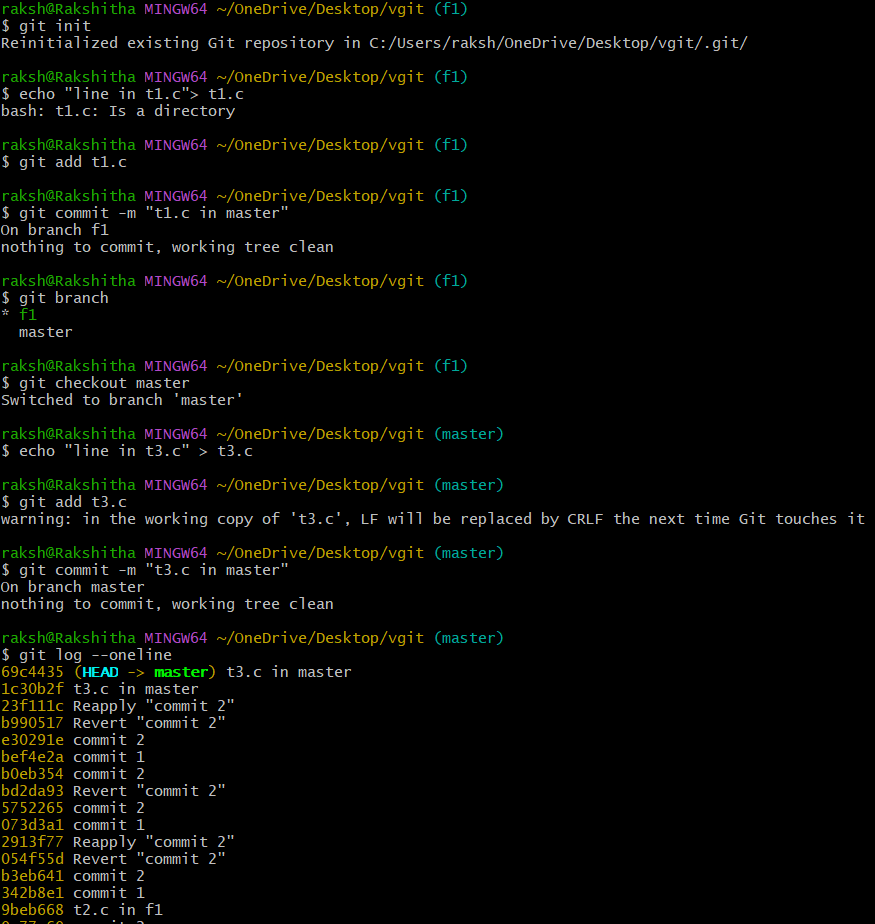
echo “line in t2.c” > t2.c

git add t2.c

git commit -m “t2.c in f1”

git log --oneline

git merge f1



**Git Rebase**

* git rebase moves or “replays” your commits on top of another branch.
* It rewrites commit history.
* Makes history linear and clean.

Example:

Before rebase:

MASTER : A--- B--- C

FEATURE: D--E

After rebase:

MASTER : A--- B--- C--- D---E

Rebase moves **D and E** and puts them after **C**, making the history straight.

Commands for git rebase

git checkout -b f1

vi t2.c

git add .

git commit -m “t2.c in f1”

git log --oneline

git checkout master

vi t3.c

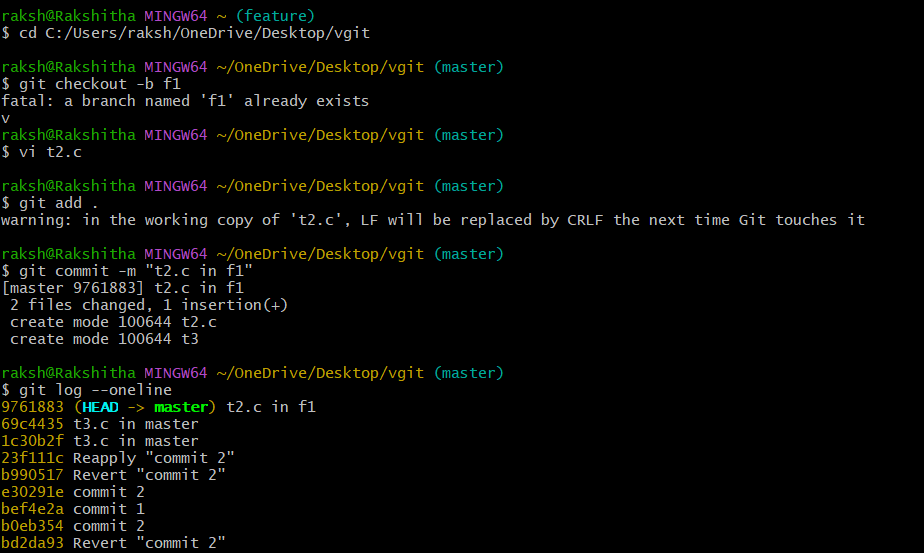
git add .

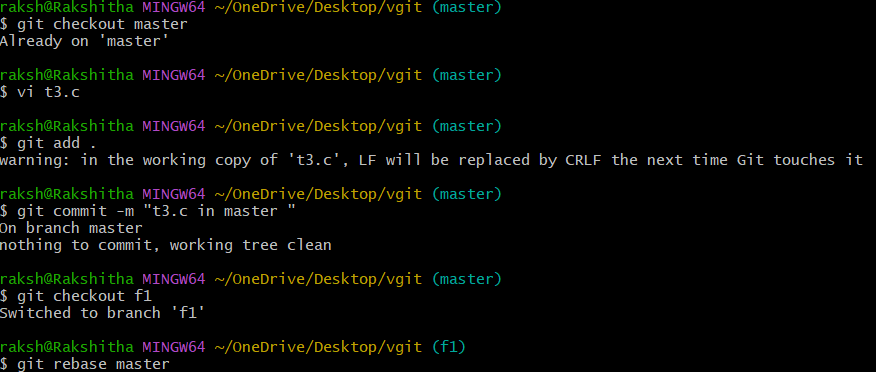
git commit -m “t3.c in master”

git log --oneline

git checkout f1

git rebase master





**Git Revert vs Merge vs Rebase**

| **GIT REVERT** | **GIT MERGE** | **GIT REBASE** | |
| --- | --- | --- | --- |
| Undoes a commit by creating a new opposite commit. | C Combines two b branches together. | Moves commits from one branch on top of another (rewrites history). | |
| Does not change old history. | Does not rewrite his history. | R Rewrites commithistory. | |
| Very safe; used to undo  changes. | S Safe; normal way of comb combining work. | | on Risky shared branches. |

**Experiment – 6:**

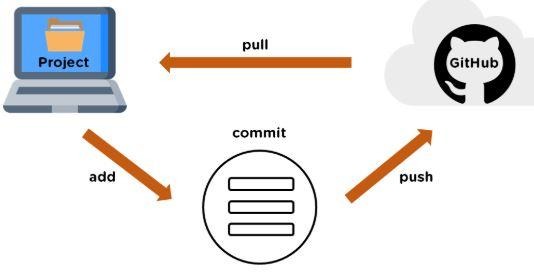
**Operations on GitHub Repository.**

**a)** Write a Command to push the files to master branch in remote repo.

**b)** Command to push files from local to particular branch in remote repo.

**c)** Command push new branch and its data to remote repository.

**a) Write a Command to push the files to master branch in remote repo**

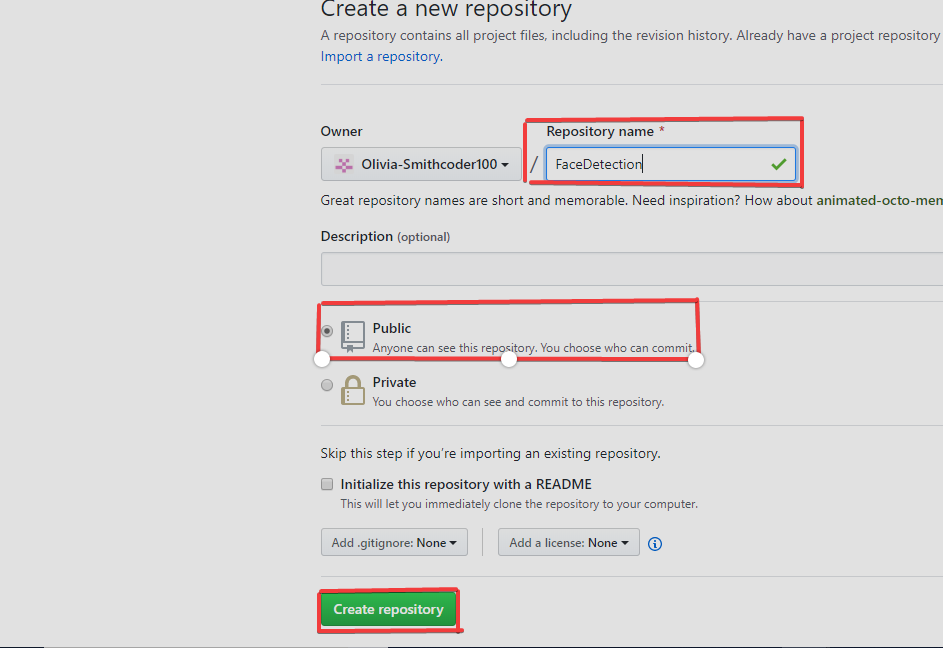


**Git PUSH**

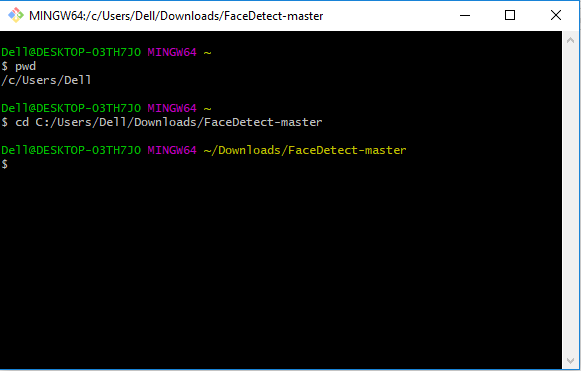
The git push command is used to transfer or push the commit, which is made on a local branch in your computer to a remote repository like GitHub. The command used for pushing to GitHub is given below.

                                         git push 'remote\_name' 'branch\_name'

**1. Creating a new repository**



**2. Open your Git Bash**



**3. Initialize the git repository**

**4. Add the file to the new local repository**

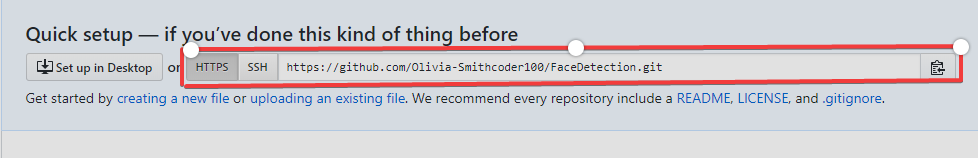
* Use git add . in your bash to add all the files to the given folder.
* Use git status in your bash to view all the files which are going to be staged to the first commit.



**5. Commit the files staged in your local repository by writing a commit message**



**6. Copy your remote repository's URL from GitHub**



**7. Add the URL copied, which is your remote repository to where your local content from your repository is pushed**

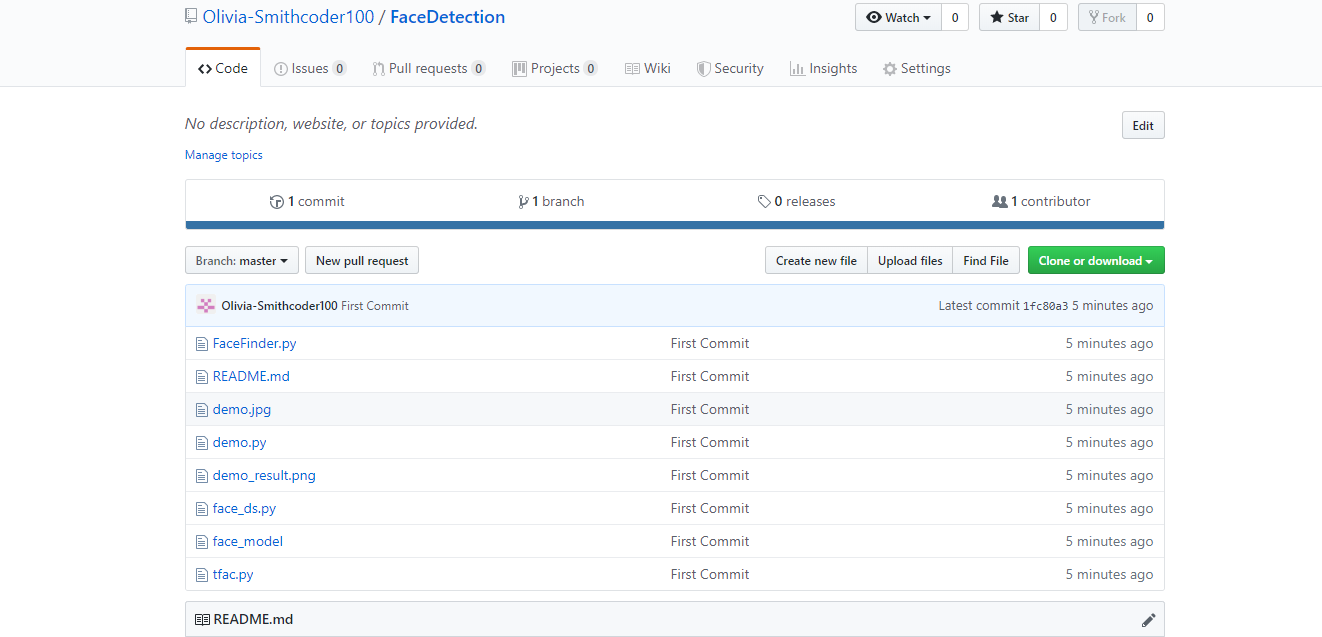
              git remote add origin 'your\_url\_name'

**8. Push the code in your local repository to GitHub**

                 git push -u origin master            is used for pushing local content to GitHub.

**9. View your files in your repository hosted on GitHub**

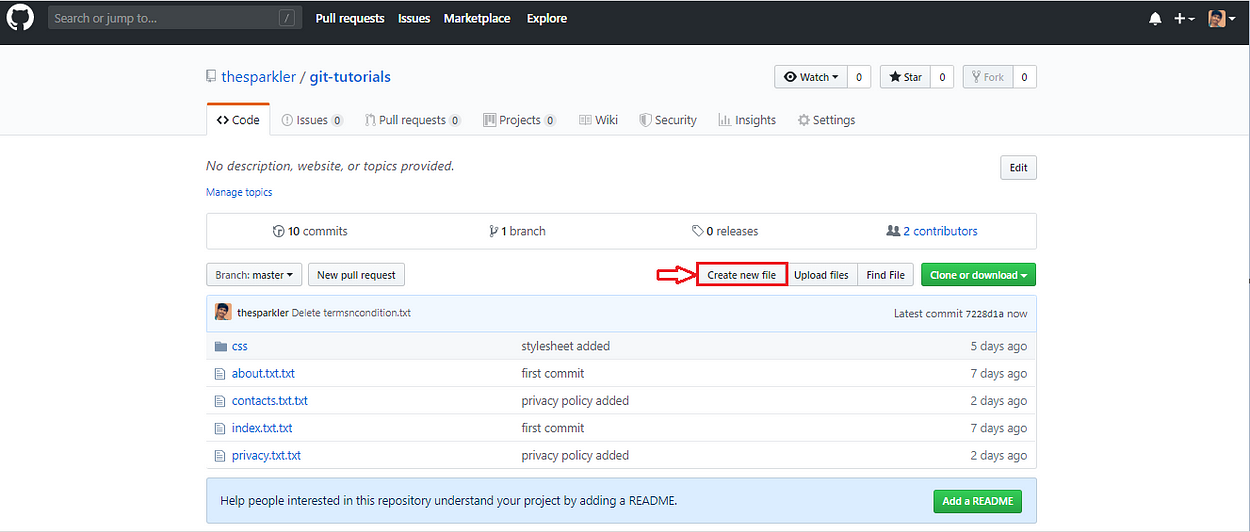
                    You can finally see the file hosted on GitHub.

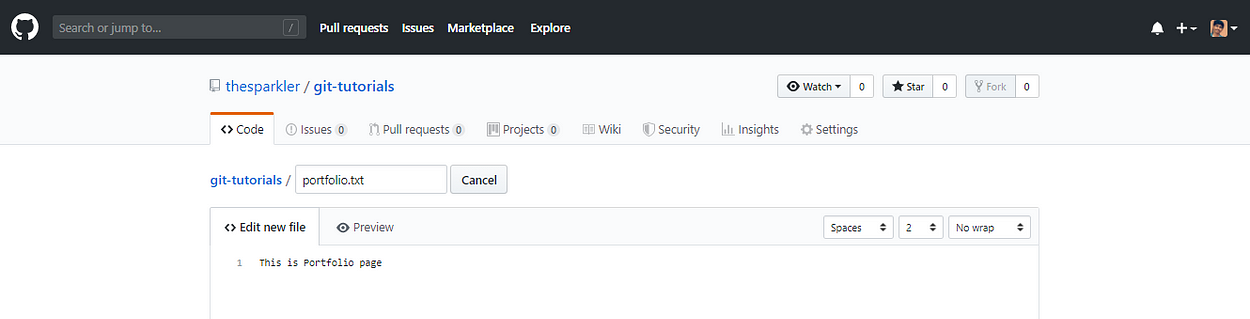


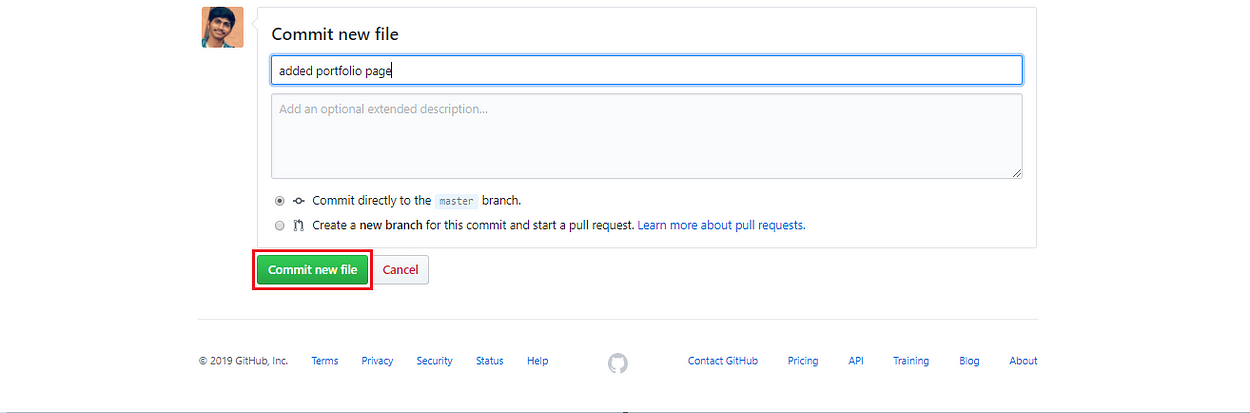
**b)**   **Command to push files from local to particular branch in remote repo.**

**Git Pull:**

The git pull command is used to fetch and download content from a remote repository and immediately update the local repository to match the content. Merging remote upstream changes into your local repository is a common task in Git-based collaboration workflows.



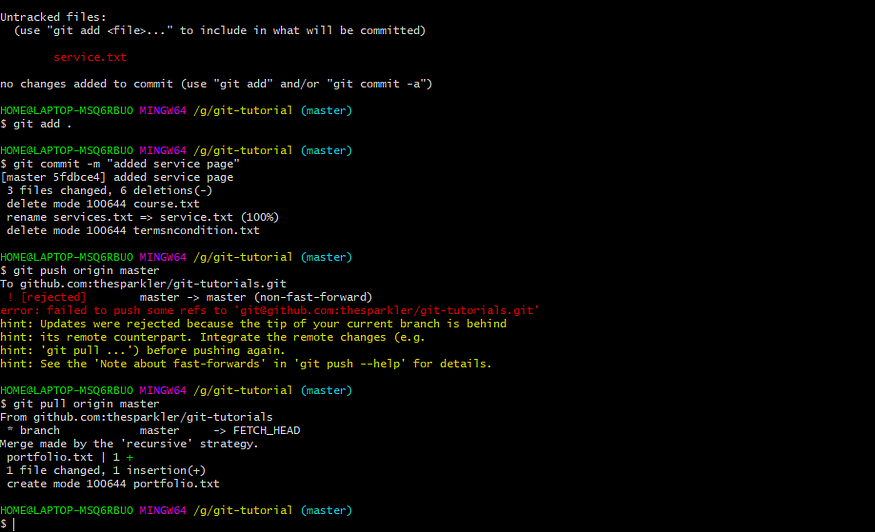




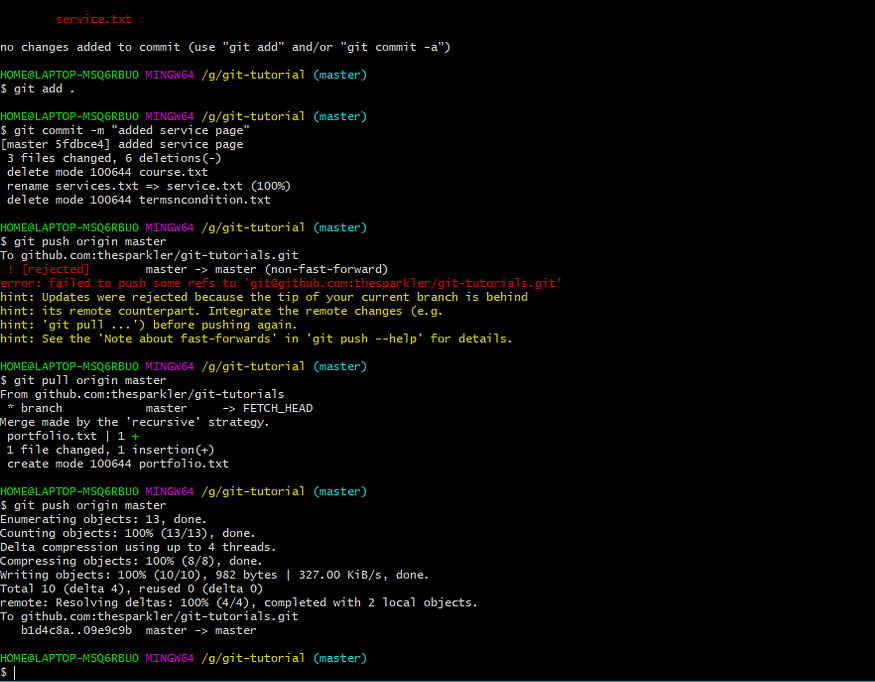
Added commit message and click in commit new file to save changes

Create another file in the local repository (in your system).

After creating the **“*service.txt”*** file you will add the file to staging area using **git add *“****service.txt****”***command after that you will commit that file using **git commit -m***“added service page”*command and after that, you will push that file on remote repository using git push origin mastercommand.



In the above image, you see that we got an error while performing git pushthe command. Now, to solve this error new needs to run git pull origin master command.



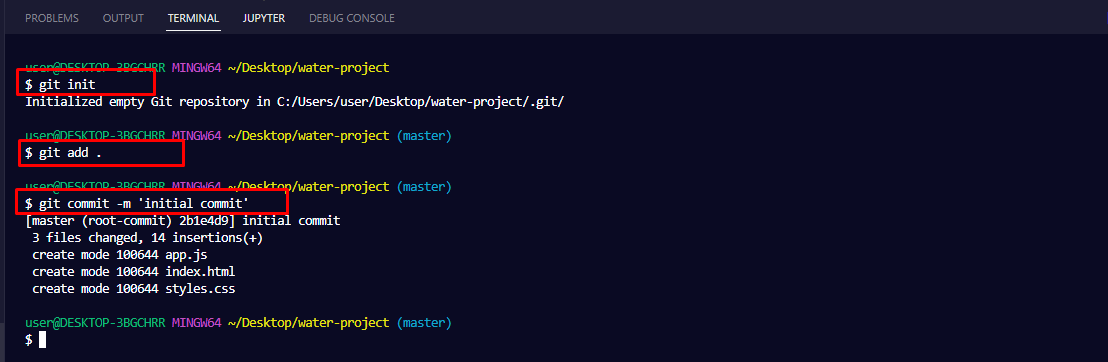
In the above image, you see that your file is successfully pushed to the master branch because we use git pull origin master the command to up-to-date our local repository from a remote repository.

**c)   Command push new branch and its data to remote repository**

* **Push the Main Branch to Remote**

 To push the main branch to remote, first executed these commands in local repo:

* git init for initializing a local repository
* git add . to add all your files that the local repository
* git commit -m ‘commit message’ to save the changes you made to those files



To push the main repo, you first have to add the remote server to Git by running

                                      git remote add <url>.

To confirm the remote has been added, run

git remote -v:



To finally push the repo, run

                                      git push -u origin <branch-name>



* **Push a New Branch to Remote**

       Create a new branch,

                                            git branch branch-name**.**

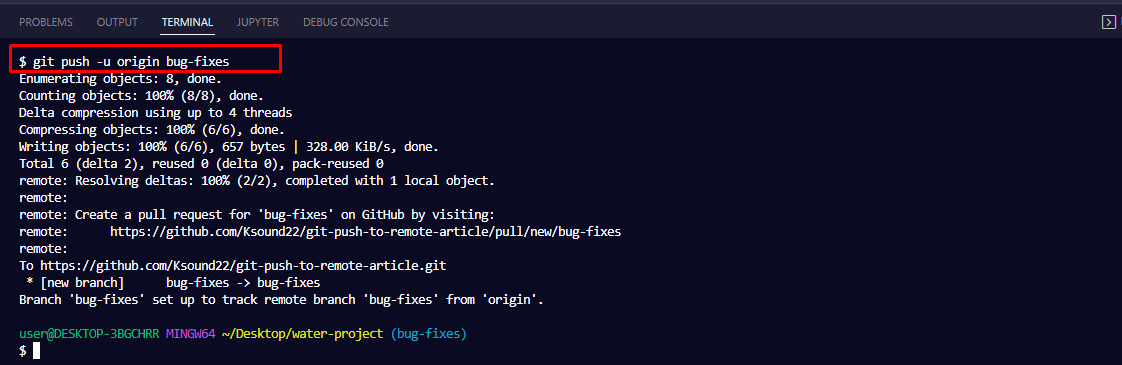
And to switch to that branch so you can work there, you have to run

git switch branch name**or**git checkout branch-name**.**

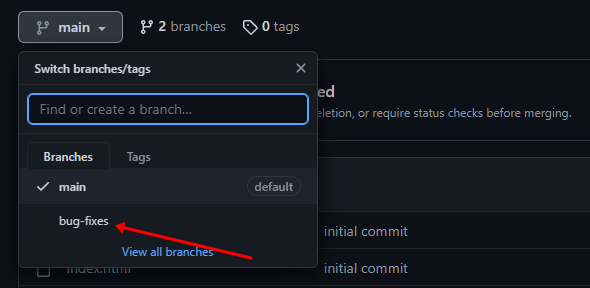
To push the branch to the remote server, run

git push –u origin <branch name>**.**

 In my case, the name of that branch is bug-fixes. So, I have to run git push -u origin bug-fixes:



To confirm that the branch has been pushed head over to GitHub and click the branches drop-down. You should see the branch there:



**Experiment – 7:**

**Git Tags and Releases**

* Write the command to create a Git tag name as "version1.0" for a commit in your local repo, and then delete that tag name.

**Git-Reset command**

* Execute “git reset --soft”, “git reset –mixed”, “git reset --hard” commands in your local repository.

**Git Tags and Releases**

* **Write the command to create a Git tag name as "version1.0" for a commit in your local repo, and then delete that tag name.**

git tag is used to **mark a specific commit** as important — usually to identify **releases or versions** like v1.0, v2.5, etc.

Tags are like **labels** given to commits so you can easily return to them later.

Two types of tags:

1. **Lightweight tag** → simple name pointing to a commit
2. **Annotated tag** → includes metadata (author, date, message)

**1. Create Lightweight Tag**

git tag version1.0

Tags the current commit.

Tagging a specific commit:

git tag version1.0 3c2a1b0

**View tags:**

git tag

**2. Create Annotated Tag (Preferred for releases)**

git tag -a version1.0 -m "First stable release"

**Tag a specific commit:**

git tag -a version1.0 -m "First stable release" 3c2a1b0

**3. Push Tags to Remote**

By default, tags are not pushed automatically.

**Push one tag:**

git push origin version1.0

**Push all tags:**

git push origin --tags

**4. Delete Tag**

**Local delete:**

git tag -d version1.0

**Remote delete:**

git push origin --delete version1.0

**5. Checkout or Switch to a Tag (Detached HEAD state)**

git checkout version1.0

(You can look around but shouldn't commit here.)

**Git-Reset command**

* **Execute “git reset --soft”, “git reset –mixed”, “git reset --hard” commands in your local repository.**

git reset command enables to undo or "reset" code changes that you previously made.

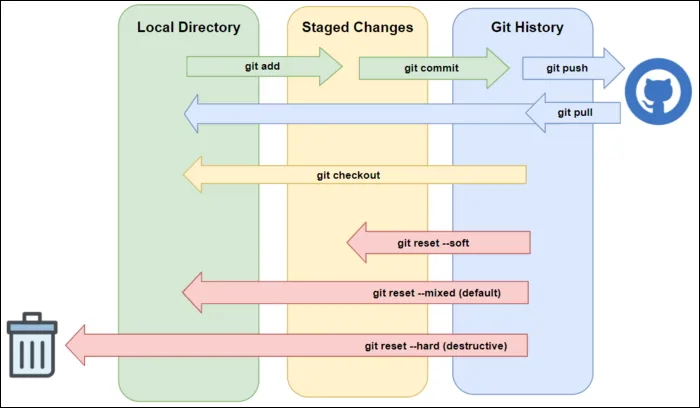
git reset syntax, usage, and modes

The basic syntax for git reset is as follows:

git reset [<mode>] [<commit>]

Git reset offers three main modes (or options) that determine how it behaves. They are --mixed, --soft, and --hard. Here's a brief description of each mode:

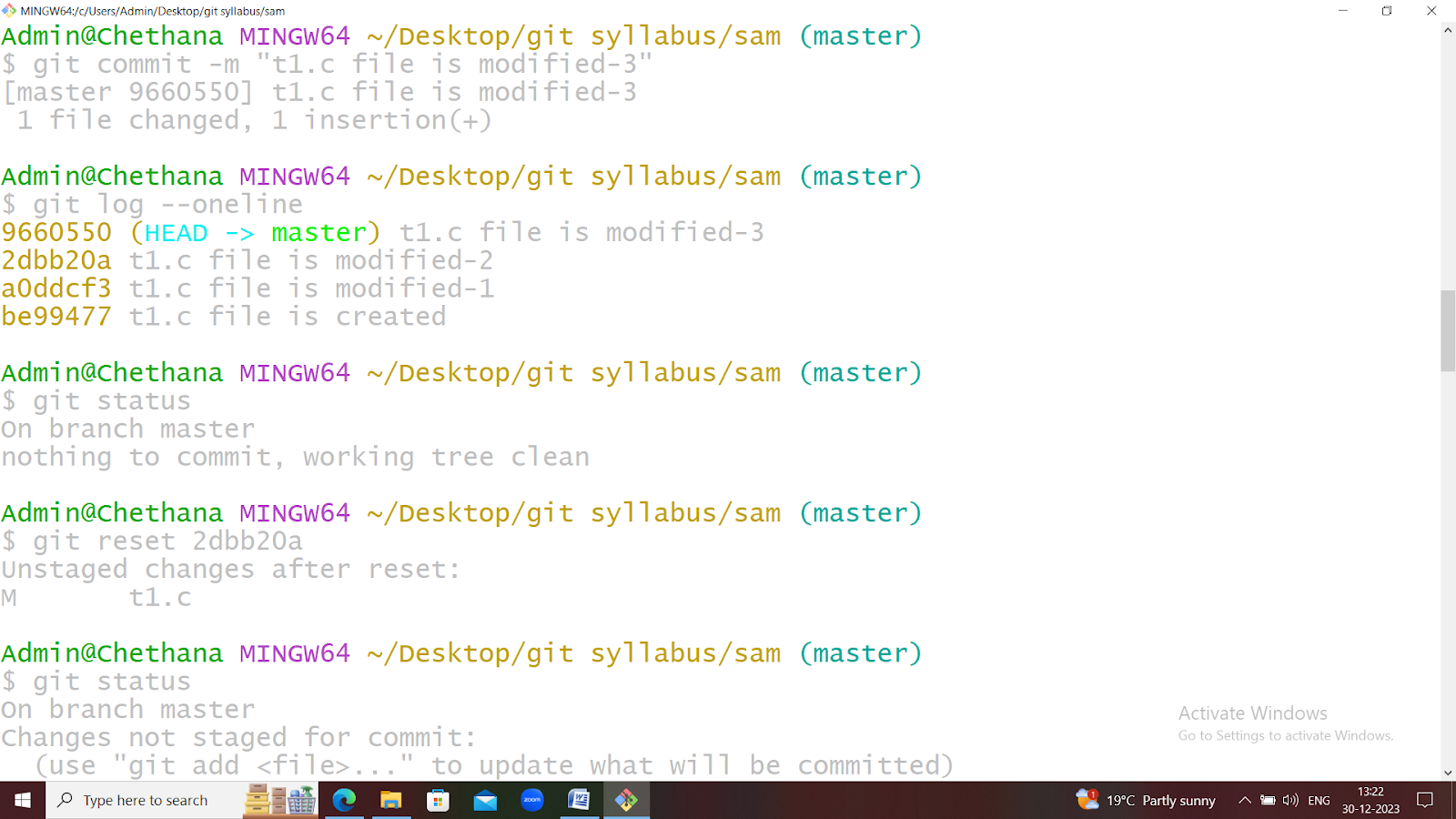
* git reset --mixed: The default option for git reset. Updates the current branch tip to the specified commit and unstages any changes by moving them from the staging area back to the working tree.
* git reset --soft: Known as a soft reset, this updates the current branch tip to the specified commit and makes no other changes.
* git reset --hard: Known as a hard reset, this updates the current branch tip to the specified commit, unstages any changes, and also deletes any changes from the working directory.



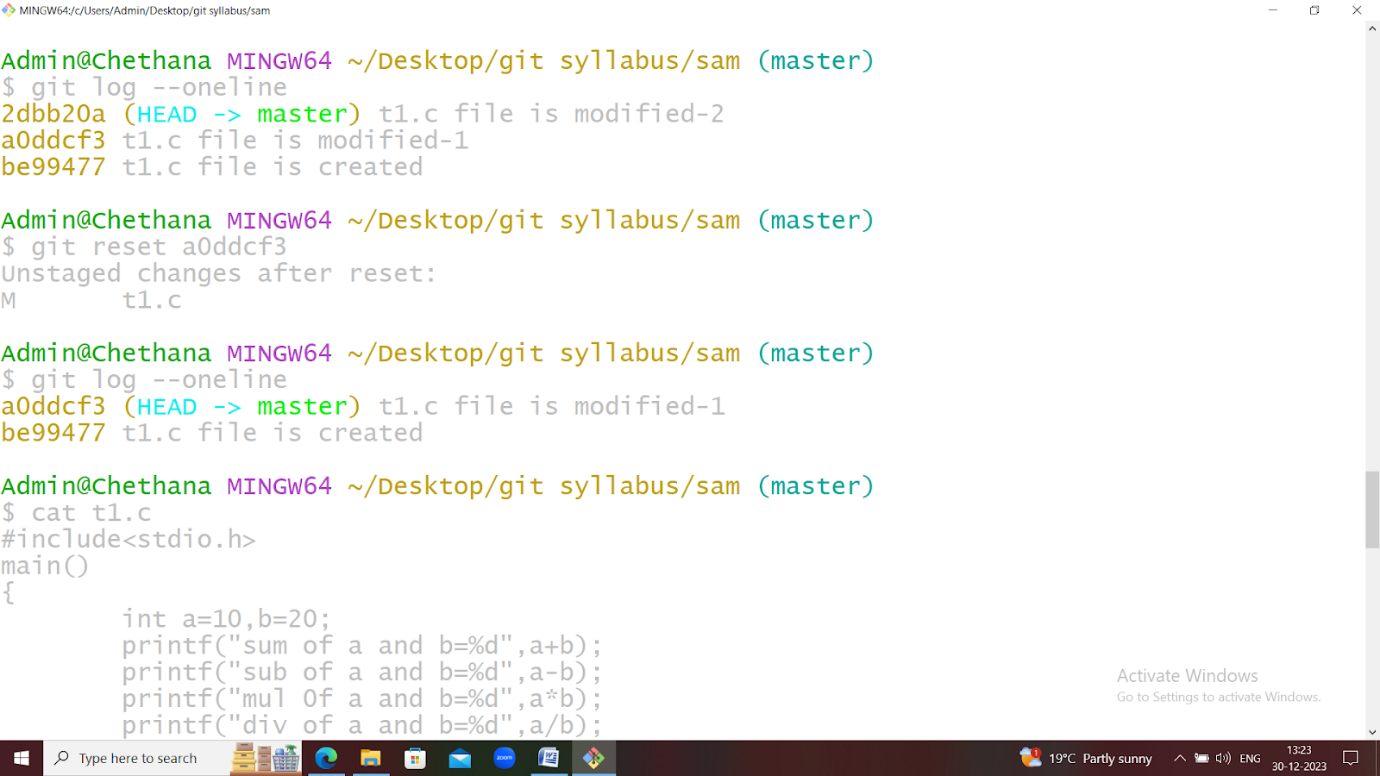
1. Git reset -- mixed <commit-id>:

* Git reset with the --mixed option will undo all the changes between HEAD and the specified commit, but will preserve your changes in the Working Directory, as unstaged changes.  If you perform a Git reset and do not supply an option of --soft, --mixed, or --hard, Git will use --mixed by default.

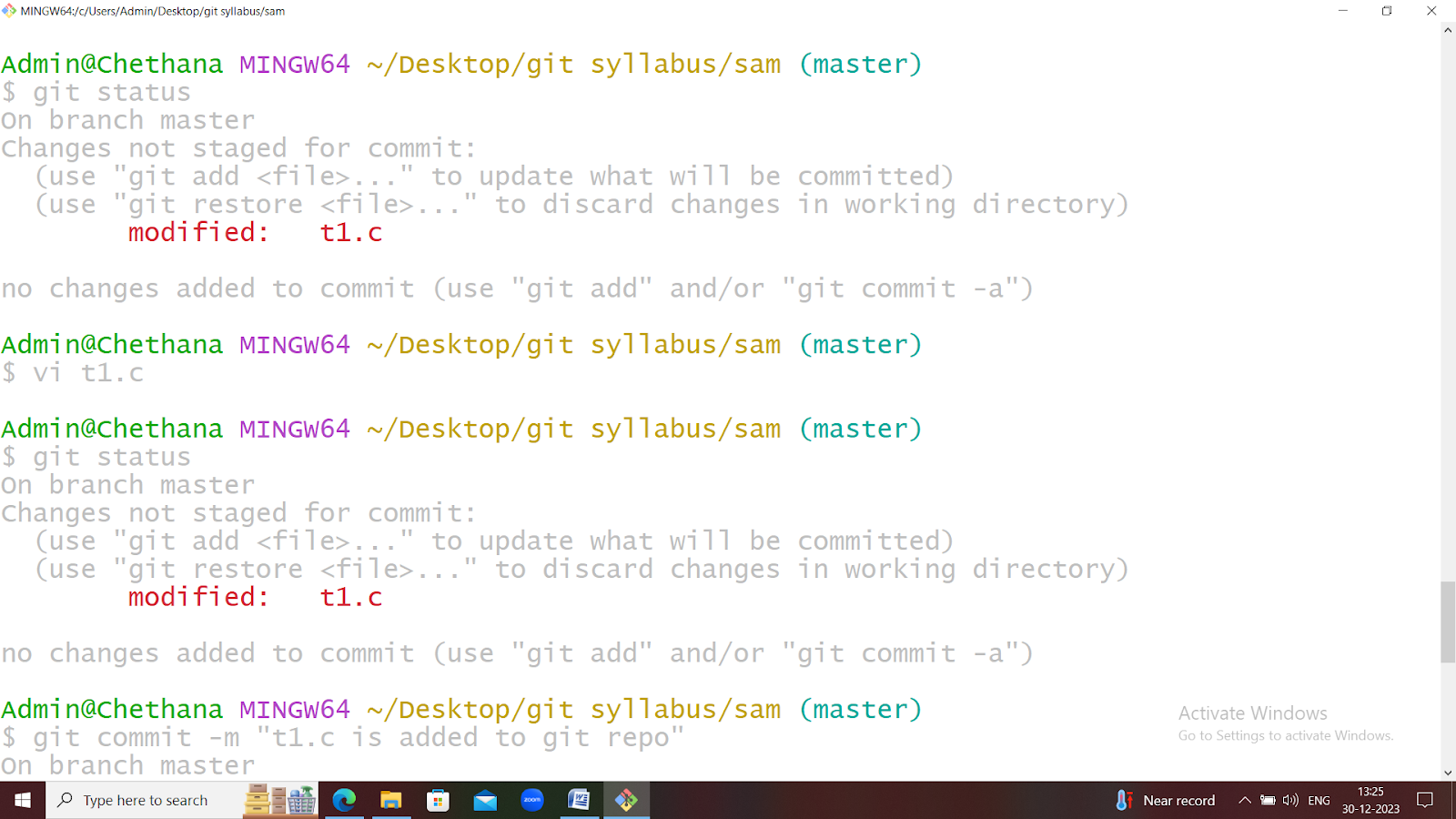
                                     --mixed *(default)*: uncommit + unstage changes, changes are left in *working tree*.



* It is the default mode
* To discard commits in the local repo and in staging area

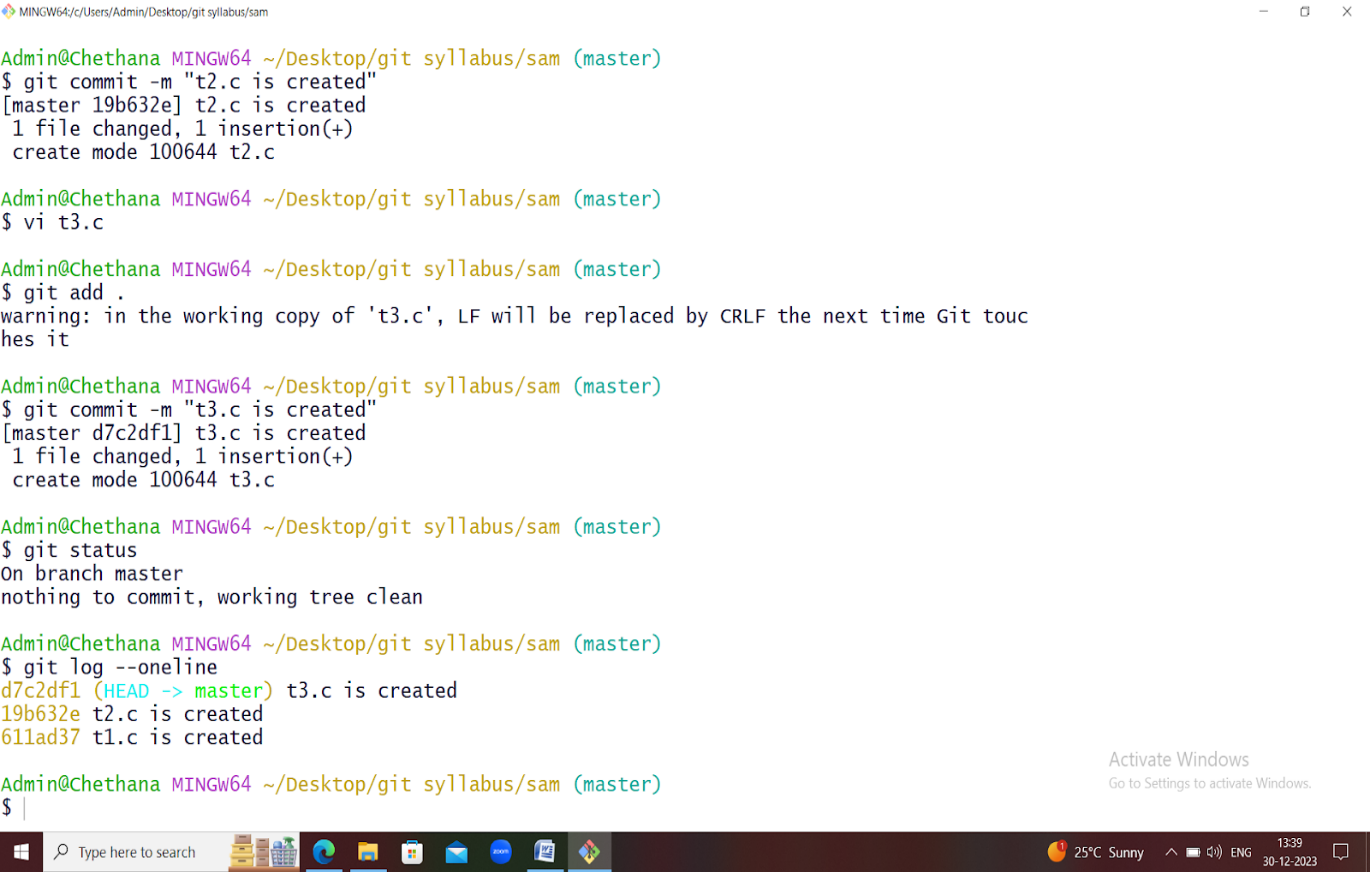


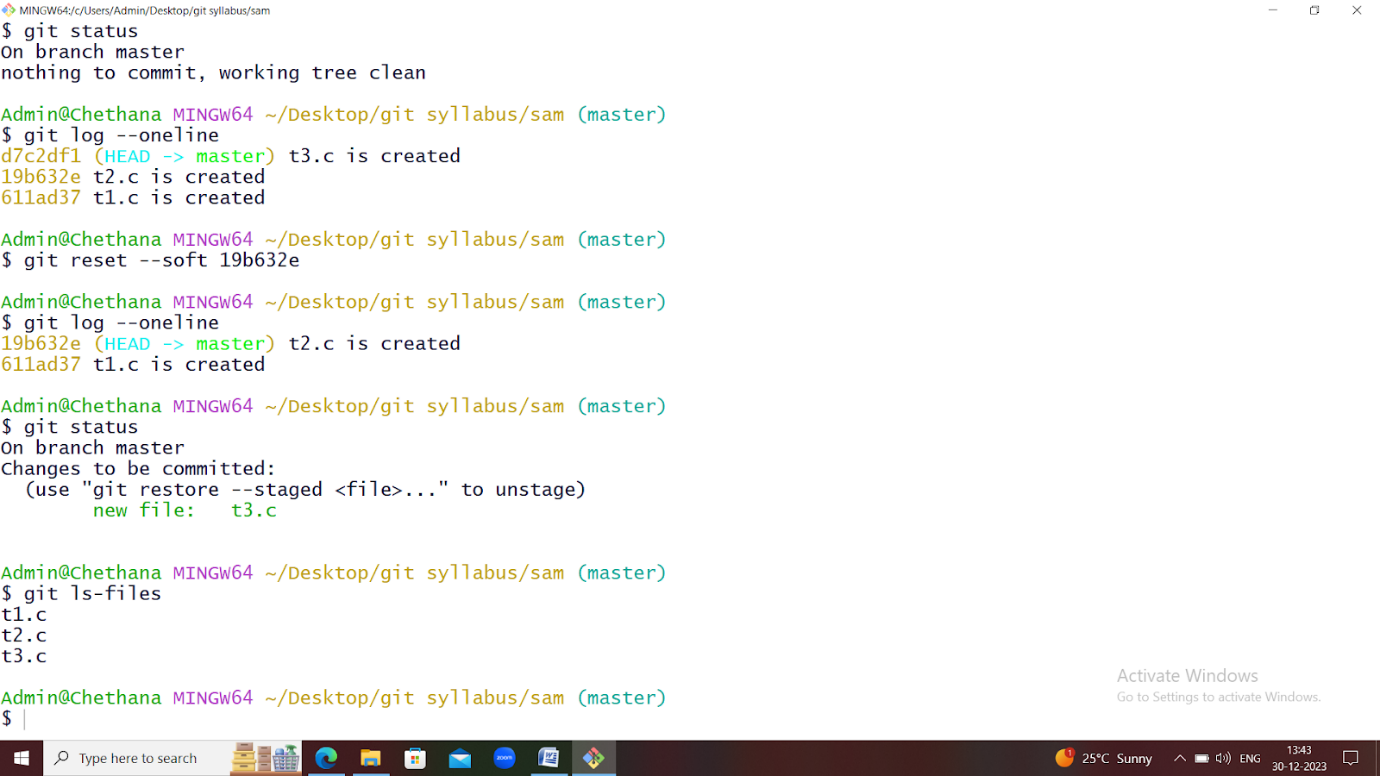
* It will not touch working directory



2. git reset --soft:

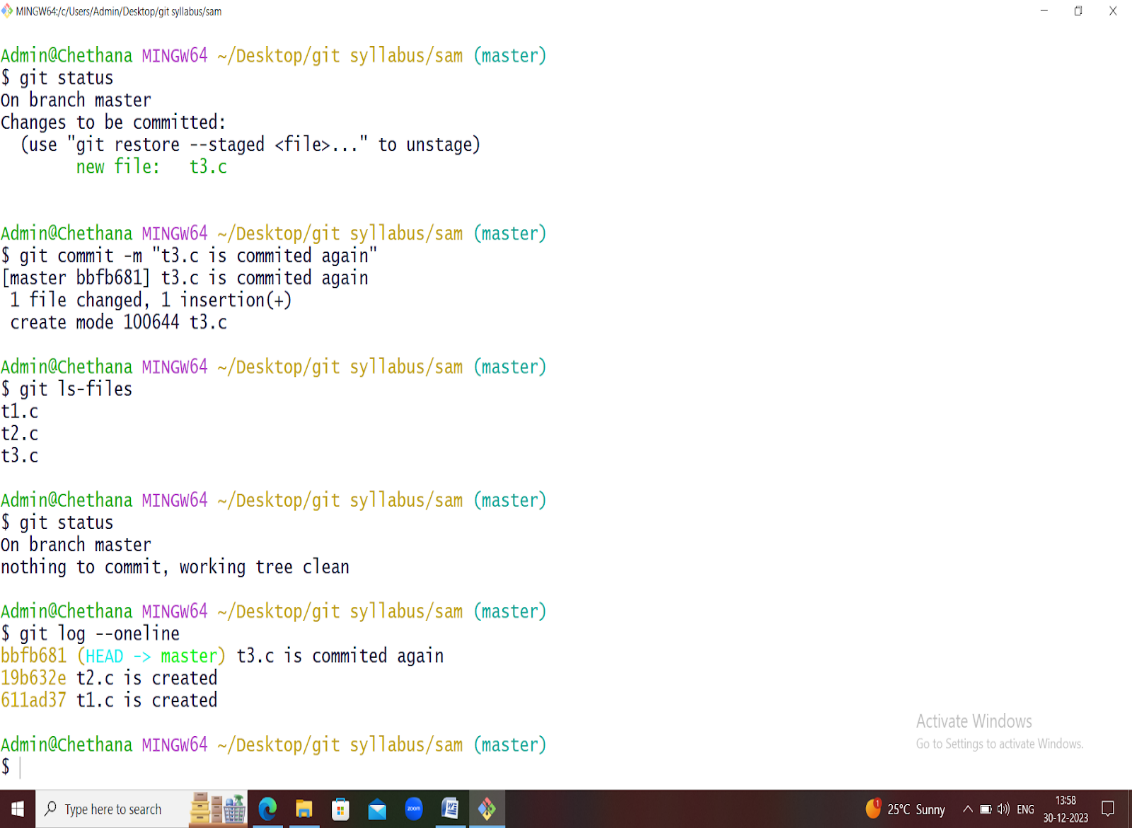
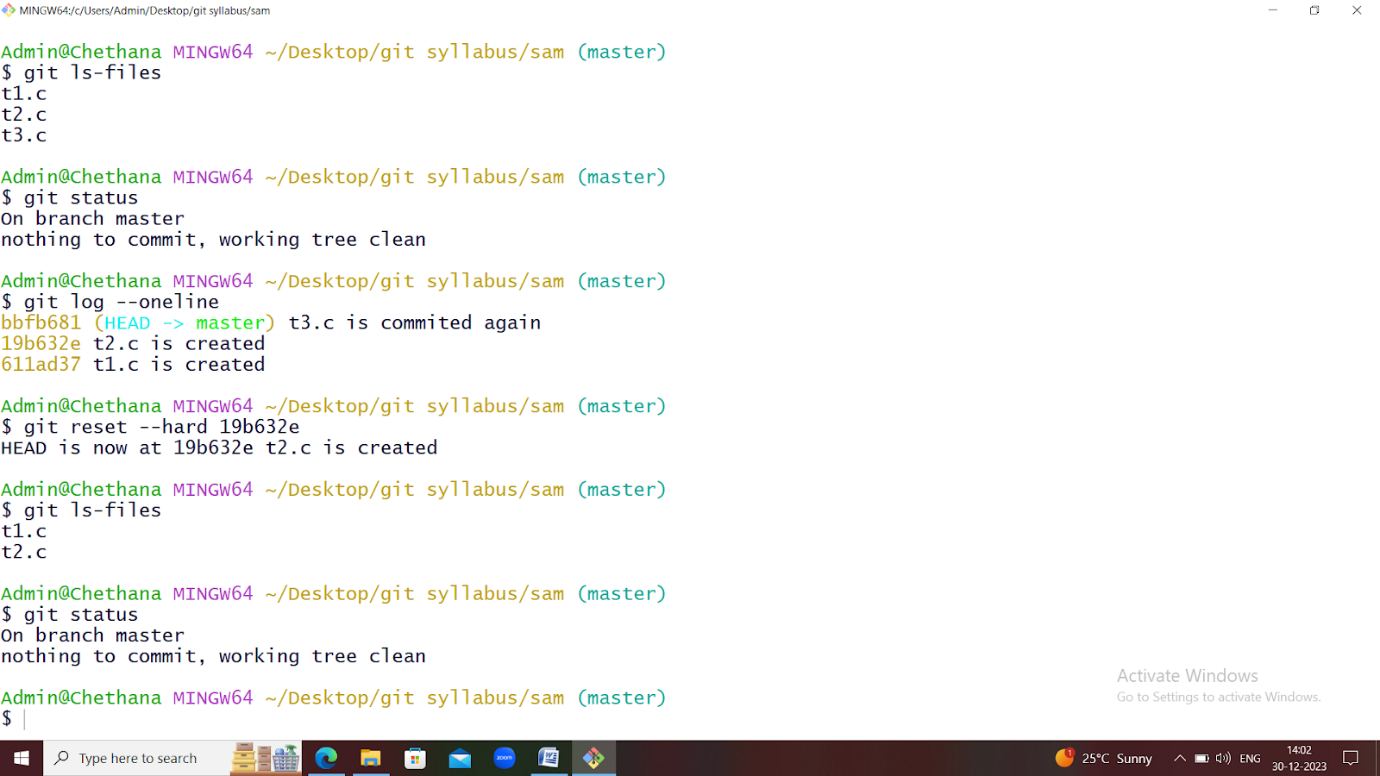
             It is exactly same as - -mixed option but changes are available in working directory as well as in the staging area.

It won’t touch staging and working director



3. git  reset  - -hard <commit-id>

* --hard will remove the changes from everywhere (local, staging and working directory).
* It is impossible to revert back and hence while using hard reset we have to take special care.

** **

**Experiment – 8:**

**Advanced Git Operations**

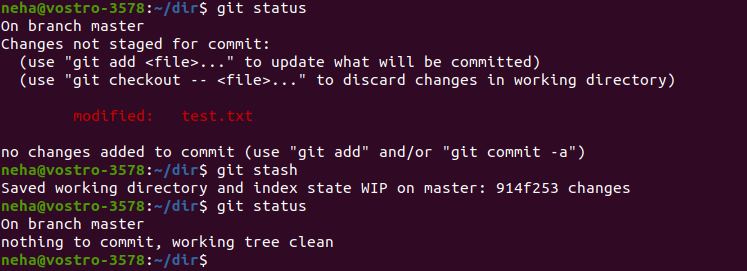
* Write a command that takes your uncommitted changes, saves them away for later use, and then revert them from your working copy.
* Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

**Write a command that takes your uncommitted changes saves them away for later use, and then reverts them from your working copy.**

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

**Stashing Work**

**1. git stash**



**2. git stash list**

To create multiple stashes and view them using the **‘git stash list’** command. Each stash entry is listed with its name (e.g. **stash@{1}**), the name of the branch that was current when the entry was made, and a short description of the commit the entry was based on.

                              git stash list

git stash list

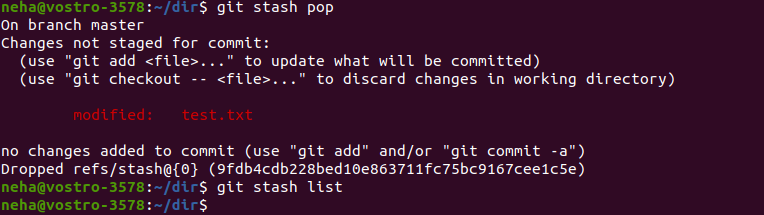
To provide more contexts to the stash we create the stash using the following command:

                             git stash save "message"

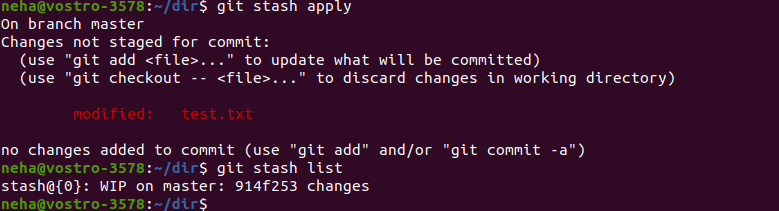
**3. Git Stash Apply And POP(Git Stash Apply & POP)**

To reapply the previously stashed changes with the **‘git stash pop’** or **‘git stash apply’** commands. The only difference between both the commands is that **‘git stash pop’** removes the changes from the stash and reapplies the changes in the working copy while **‘git stash apply’** only reapplies the changes in the working copy without removing the changes from the stash. In simple words, **“pop”** removes the state from the stash list while **“apply”** does not remove the state from the stash list.

* **git stash pop**



* **git stash apply**

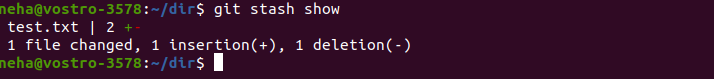


  By default **‘git stash pop’** or **‘git stash apply’** will reapply the most recently created stash: stash@{0} To choose which stash to apply, you can pass the identifier as the last argument (**For eg.**:- git stash pop stash@{2}).

**4. Git Stash Show**

**git stash show** command is used to display the summary of operations performed on the stash.

* **git stash show**



**5. Cleaning Up Your Stash (Git Stash Clear)**

To delete any particular stash (**For ex:**– stash@{1}), use **‘git stash drop stash@{1}’**. By default, this command will drop **stash@{0}** if no argument is provided (**git stash drop**). To delete all stashes at once, use the **‘git stash clear’** command.

