1. Installation of GIT and GIT Hub repository to understand working directory, staging area and local repository.

Installing on windows

- go to https://git-scm.com/download/win and the download will start automatically.
- Once the installer has started, follow the instructions as provided in the Git Setup wizard screen until the installation is complete.
- Open the windows command prompt
- Type git version to verify Git was installed.

Install Git on Linux:

Debian/Ubuntu

- Git packages are available using **apt**.
- Navigate to your command prompt shell and run the following command to check software is up-to-date: **sudo apt-get update**.
- To install Git, run the following command: **sudo apt-get install git-all**.
- Once the command output has completed, you can verify the installation by typing: **git version**.

Creating github account:-

➢ open www.github.com

➢ create an account by sign up.

➢ Click join free plan.

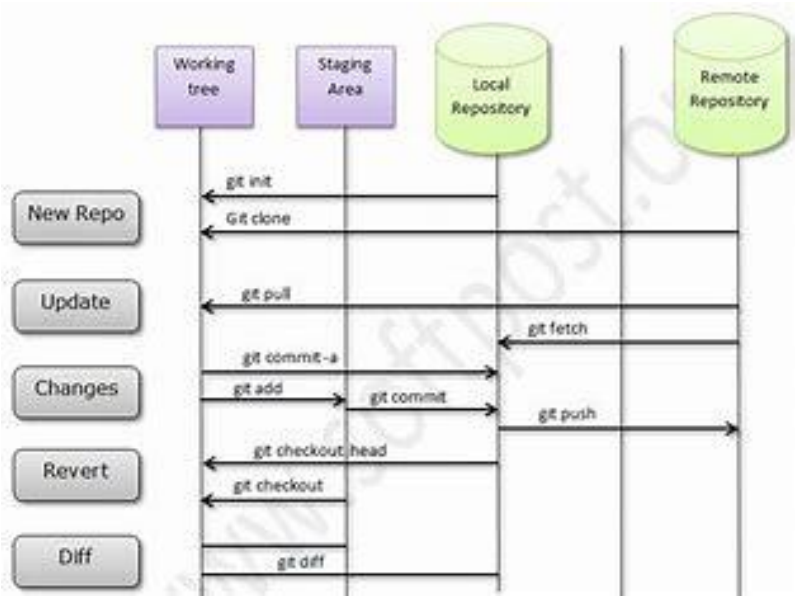
➢ Now click on complete setup.

➢ Now verify your email from gmail.

➢ Sign out then sign in again.

In GIT Architecture we have 4 stages

1.Work Space
2.Staging Area
3.Local repo / git repo
4.Central repo



1.Work Space : it is a place where we edit,modify project related files all the files in workspace are visible to all directories

2.Staging Area : on Git Add files are moved from work space to staging area where changes are saved

3.3.Local repo / git repo : on Git Commit,files will be added to local/git repo & then we track the file versions and Commit ID are created here

4.Central repo : on Git Push,files are moved to central repo.

2. **Main Git commands.**
    1. Init
    2. Add
    3. Commit
    4. Status
    5. Log
    6. config

- 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 init

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$ git init
Initialized empty Git repository in C:/Users/HiMaNshU/Desktop/.git/

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop (master)
$
```

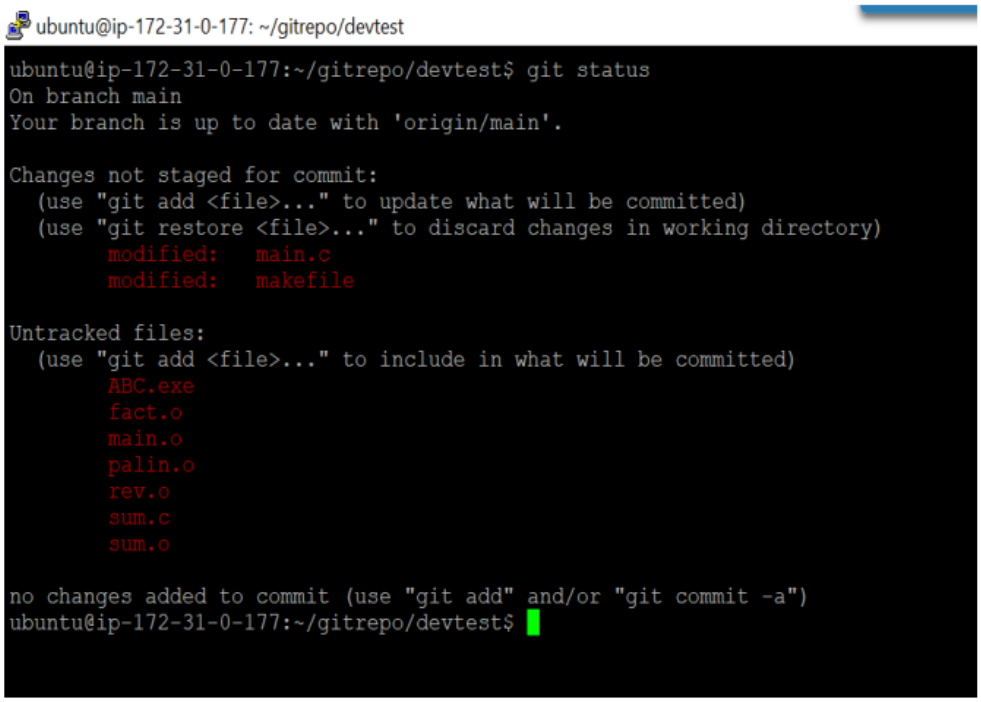
- git add :

  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:

  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.

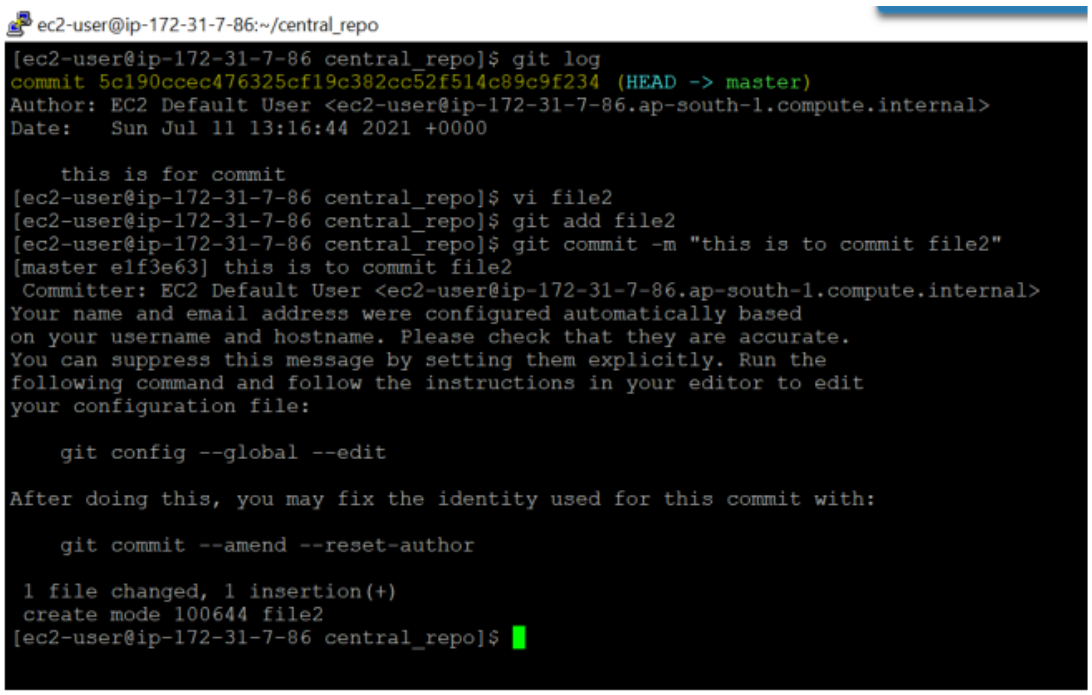


- Git commit:

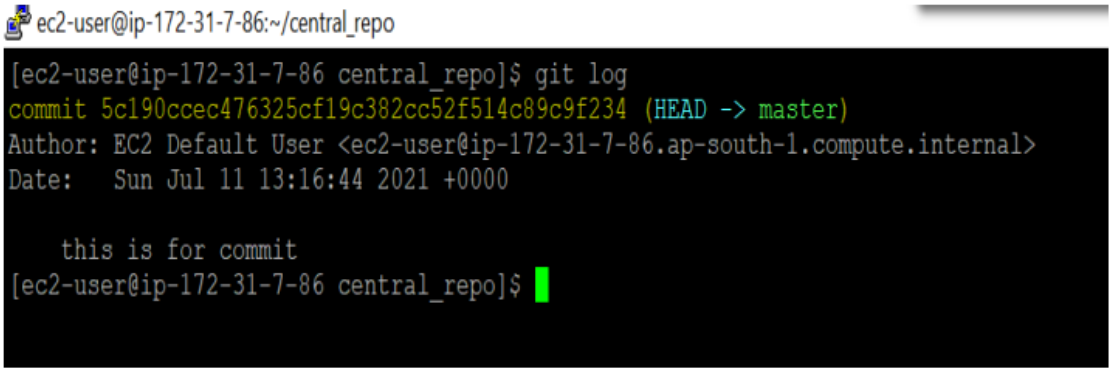
  git commit -m "Initial commit"

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"

3. Creating and Managing Branches

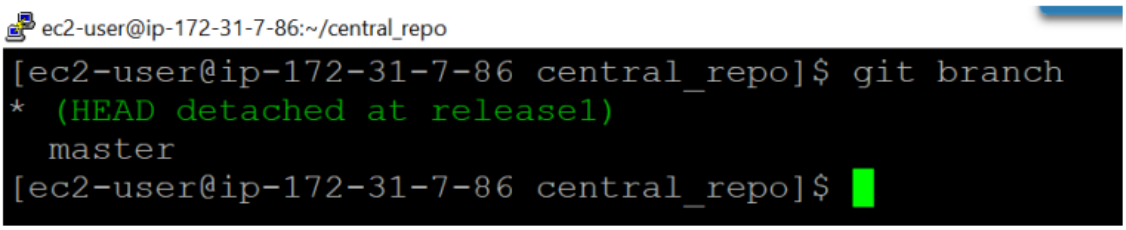
- Creating a New Branch – feature branch
- Merging "Main" and "feature" Branches and understand git merge conflict.
- Write a command to identify if the branch is already merged into master.

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

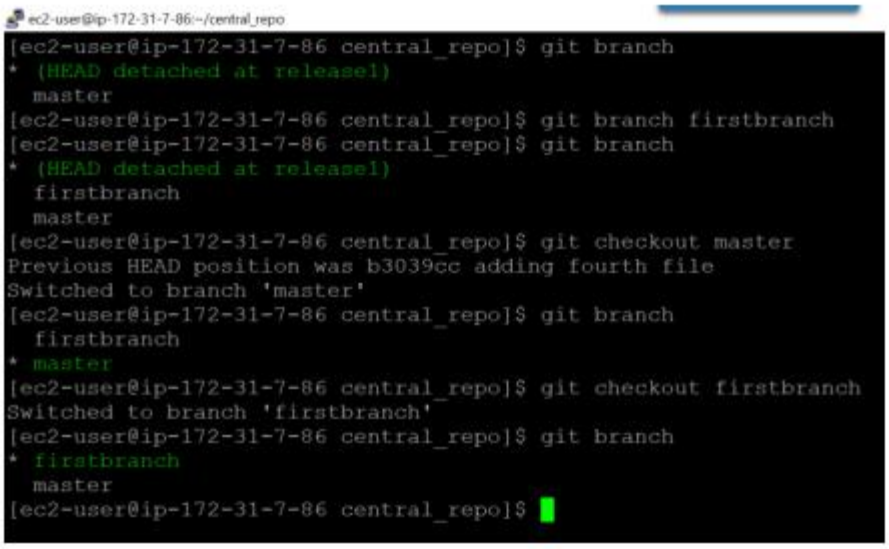
Git branch Git branch branch1

Branch will be created from Master.

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

Branch1

*Master Means now am on Master branch



- 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.

4. Creating and Managing Branches

- **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/

- 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:

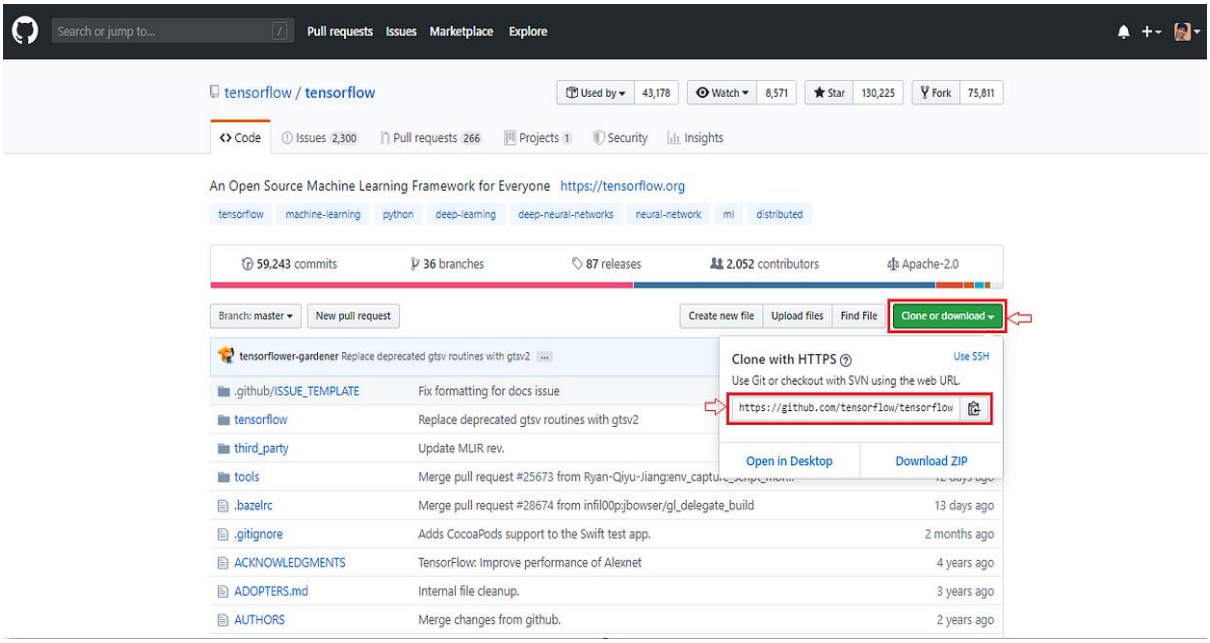
5. Collaboration and Remote Repositories

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>*



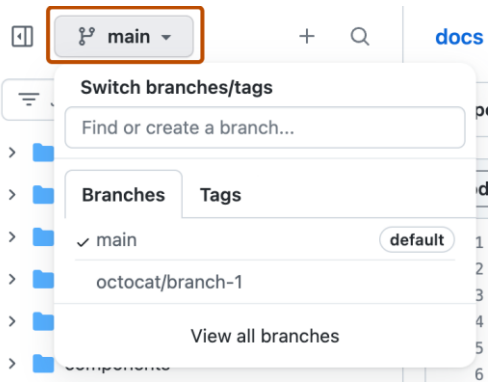
In the above image, it's a TensorFlow repository and we want to clone it means to download it in our system. We have to use the link to clone this repository.

```
HOME@LAPTOP-MSQ6RBUO MINGW64 /g/git-tutorial (newsBranch)
$ git clone https://github.com/tensorflow/tensorflow.git
Cloning into 'tensorflow'...
remote: Enumerating objects: 74, done.
remote: Counting objects: 100% (74/74), done.
remote: Compressing objects: 100% (65/65), done.
Receiving objects:  16% (103810/620745), 72.67 MiB | 604.00 KiB/s
```

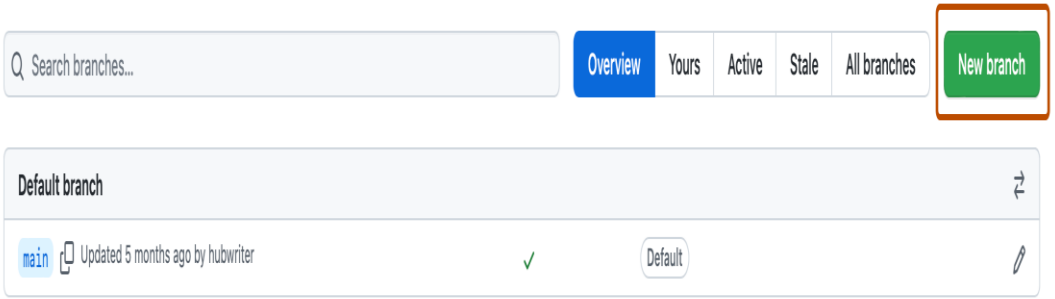
Cloning the repo

Creating a branch via the branches

1. On GitHub, navigate to the main page of the repository.
2. From the file tree view on the left, select the branch dropdown menu, then click **View all branches**. You can also find the branch dropdown menu at the top of the integrated file editor.



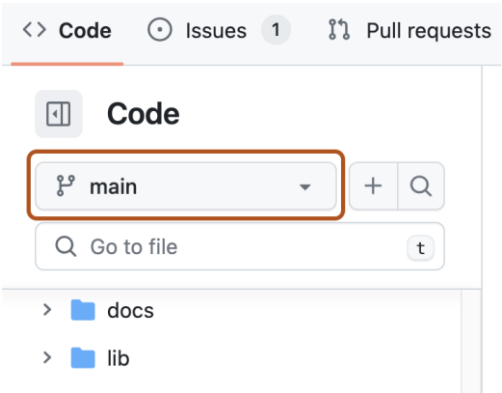
3. Click **New branch**.



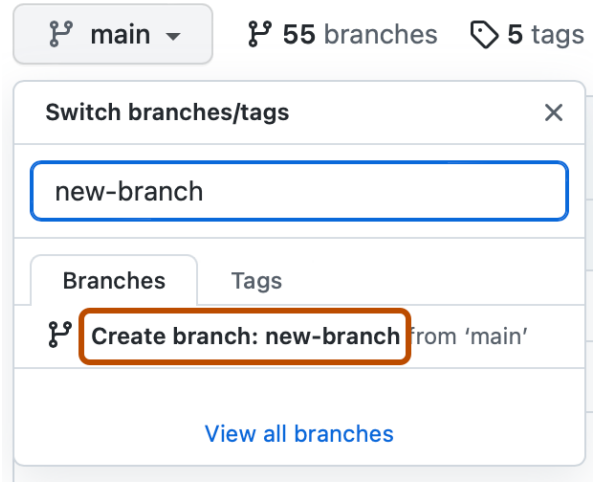
4. Under "Branch name", type a name for the branch.
5. Click **Create branch**.

Creating a branch using the branch dropdown

1. On GitHub, navigate to the main page of the repository.
2. Select the branch dropdown menu, in the file tree view or at the top of the integrated file editor.



3. Optionally, if you want to create the new branch from a branch other than the default branch of the repository, click another branch, then select the branch dropdown menu again.
4. In the "Find or create a branch..." text field, type a unique name for your new branch, then click **Create branch**.



- Resolve simple merge conflicts that involve competing line changes on GitHub

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

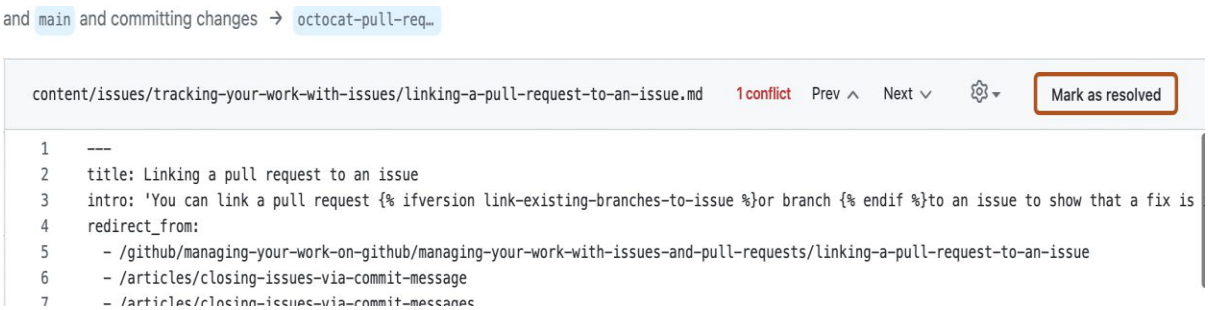
2. In the "Pull Requests" list, click the pull request with a merge conflict that you'd like to resolve.

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.

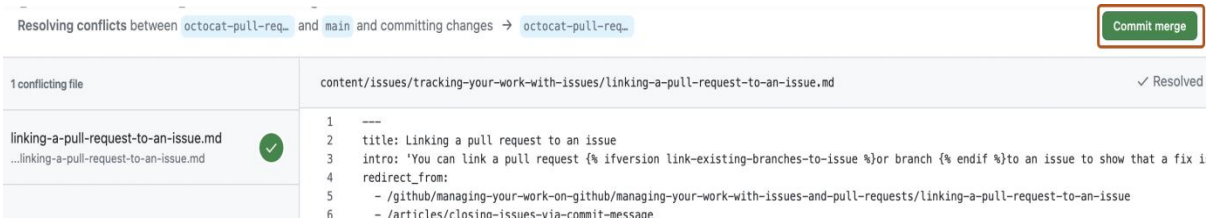
5. If you have more than one merge conflict in your file, scroll down to the next set of conflict markers and repeat steps four and five to resolve your merge conflict.

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."

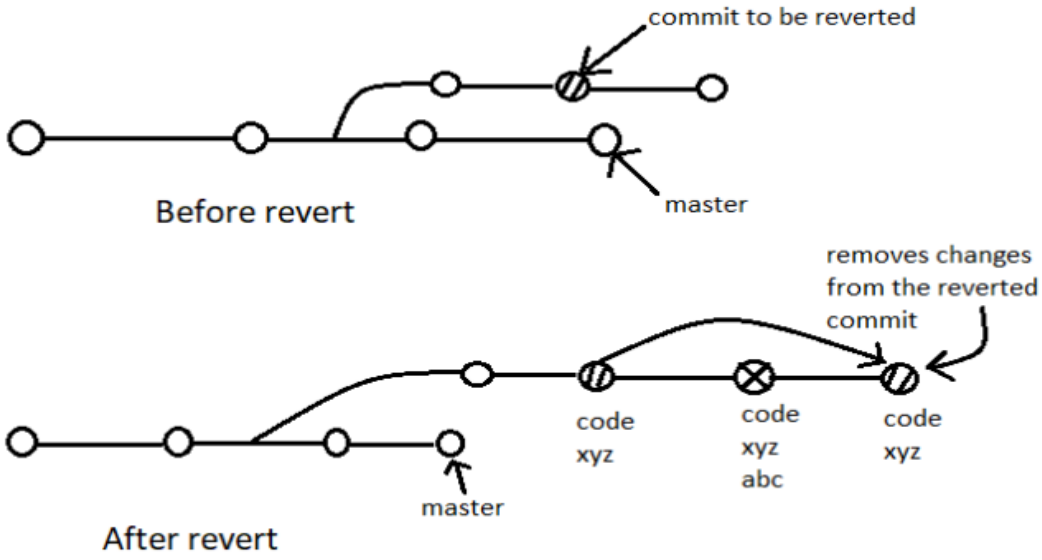
6. Undo Operations on Git.

- **Execute a command "git revert" and "git rebase" then find the difference.**

git revert:-

➢ Revert command helps you undo an existing commit.

➢ It doesn't delete any data in the process instead rather git creates new commit with the included files reverted to their previous state. So, your version control history moves forward while the state of your file moves backward.



➢ when you revert a commit, a commit id is assigned to reverted commit.

Commands for git revert:-

git status

```
root@ubuntuserver:/home/abdlqdr/quadirgit# git status
On branch master
nothing to commit, working tree clean
```

cat >newfile

```
root@ubuntuserver:/home/abdlqdr/quadirgit# cat >newfile
guftgu tune sikhai hai ke mai gunga tha
```

git add .

git commit –m “code”

```
root@ubuntuserver:/home/abdlqdr/quadirgit# git add .
root@ubuntuserver:/home/abdlqdr/quadirgit# git commit -m "checking revert"
[master ff72eb1] checking revert
 1 file changed, 1 insertion(+)
 create mode 100644 newfile
```

git log –oneline

```
root@ubuntuserver:/home/abdlqdr/quadirgit# git log --oneline
ff72eb1 (HEAD -> master) checking revert
f9b75e4 (origin/master) code file complete
f939e20 code file stashing test
2d3e81c second code done
293ec7f first code done
```

git revert <commit id>

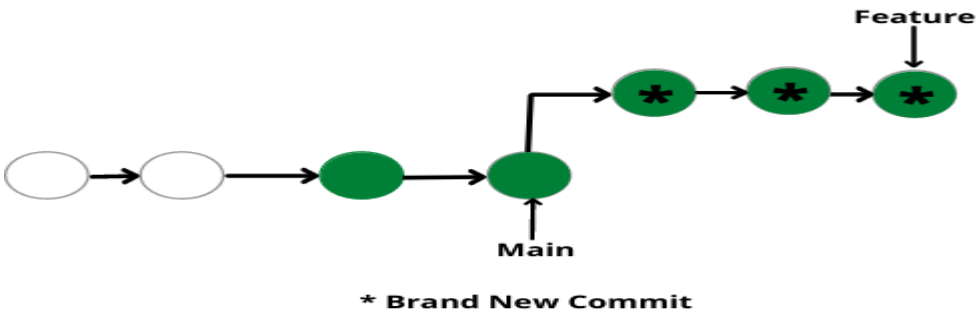
```
root@ubuntuserver:/home/abdlqdr/quadirgit# git revert  c720124c89cf496a
[master 2ed04fc] Revert "checking revert update" plese consider this code.
 1 file changed, 1 deletion(-)
```

Git rebase:

- The alternative to git merge is the git rebase option. In this, we rebase the entire feature branch to merge it with the main branch. Follow the following commands to perform merge commit:-

git rebase main

- Git rebase actually rebases the feature branch and merges it with the main branch. In simple words, it moves the entire feature branch to the tip of the main branch. The pictorial representation looks a bit like this:-



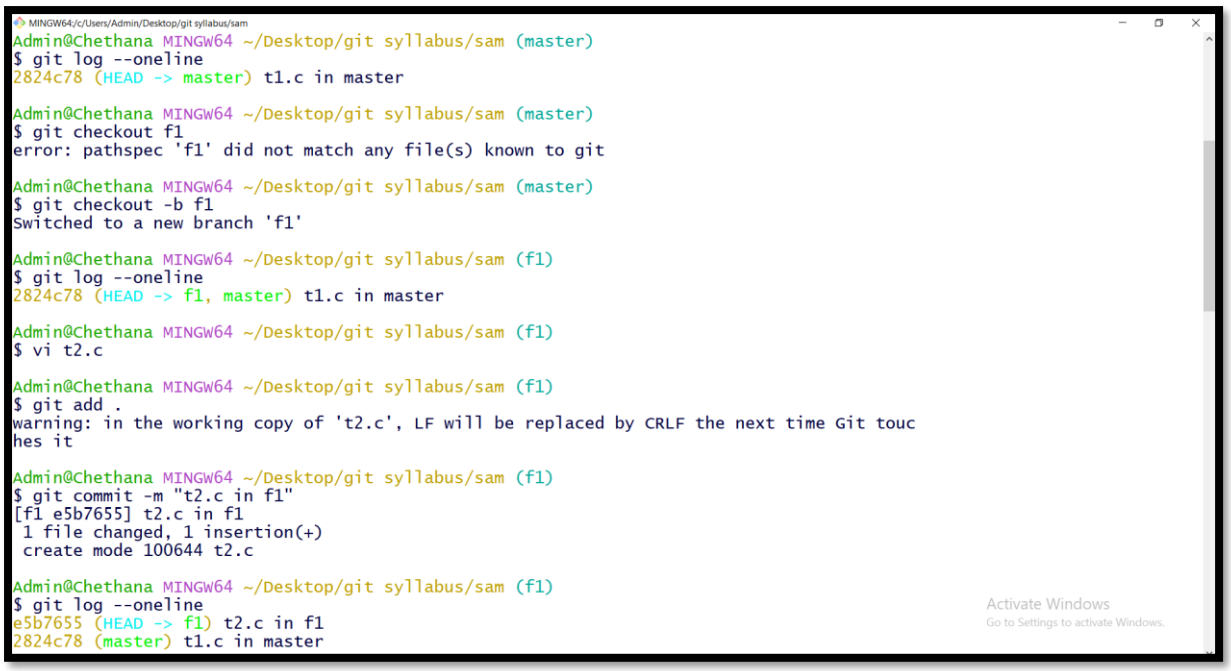
Advantage

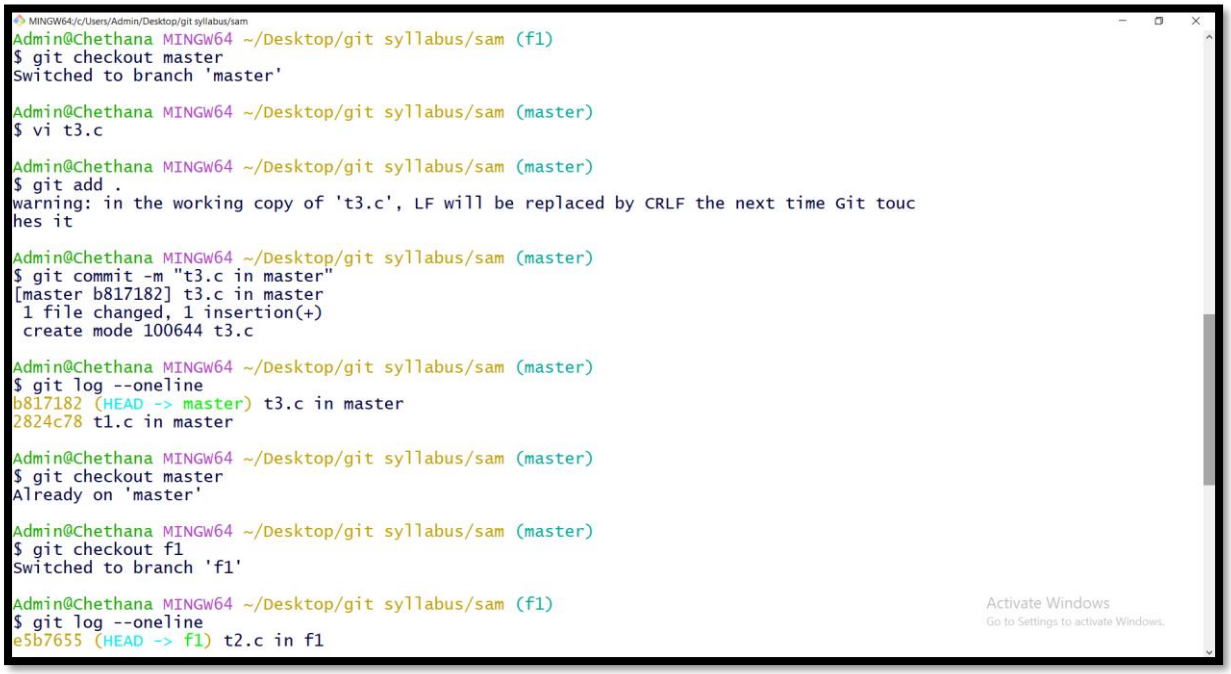
The major benefit of using git rebase is it provides a cleaner merge history. It works linearly, removing the unnecessary merge commits, unlike git merge. It makes it easier to move along the log history and understand the changes.

Git merge :

```
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git commit -m "t3.c in master"
[master 9a132ae] t3.c in master
 1 file changed, 1 insertion(+)
 create mode 100644 t3.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
9a132ae (HEAD -> master) t3.c in master
d303db4 t1.c in master

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline f1
fac778e (f1) t2.c in f1
d303db4 t1.c in master

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git merge f1
Merge made by the 'ort' strategy.
 t2.c | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 t2.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
2e60d0b (HEAD -> master) Merge branch 'f1'
9a132ae t3.c in master
fac778e (f1) t2.c in f1
d303db4 t1.c in master

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$
```

Git rebase:

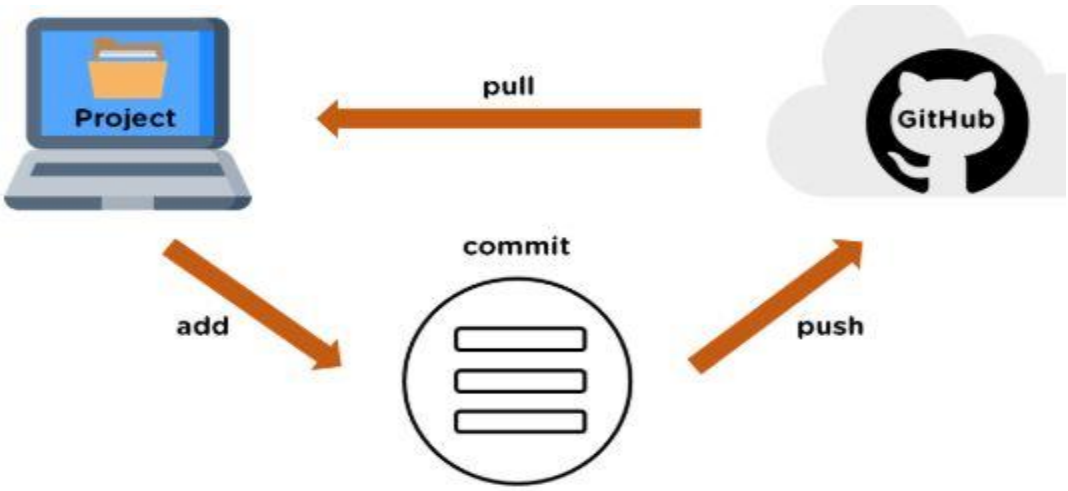

MINGW64:/c/Users/Admin/Desktop/git syllabus/sam
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
2824c78 (HEAD -> master) t1.c in master
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git checkout f1
error: pathspec 'f1' did not match any file(s) known to git
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git checkout -b f1
Switched to a new branch 'f1'
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git log --oneline
2824c78 (HEAD -> f1, master) t1.c in master
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ vi t2.c
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git add .
warning: in the working copy of 't2.c', LF will be replaced by CRLF the next time Git touc
hes it
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git commit -m "t2.c in f1"
[f1 e5b7655] t2.c in f1
1 file changed, 1 insertion(+)
create mode 100644 t2.c
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git log --oneline
e5b7655 (HEAD -> f1) t2.c in f1
2824c78 (master) t1.c in master
Activate Windows
Go to Settings to activate Windows.


MINGW64:/c/Users/Admin/Desktop/git syllabus/sam
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git checkout master
Switched to branch 'master'
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ vi t3.c
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git add .
warning: in the working copy of 't3.c', LF will be replaced by CRLF the next time Git touc
hes it
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git commit -m "t3.c in master"
[master b817182] t3.c in master
1 file changed, 1 insertion(+)
create mode 100644 t3.c
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
b817182 (HEAD -> master) t3.c in master
2824c78 t1.c in master
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git checkout master
Already on 'master'
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git checkout f1
Switched to branch 'f1'
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git log --oneline
e5b7655 (HEAD -> f1) t2.c in f1
Activate Windows
Go to Settings to activate Windows.

MINGW64:/c/Users/Admin/Desktop/git syllabus/sam
$ git log --oneline
e5b7655 (HEAD -> f1) t2.c in f1
2824c78 t1.c in master
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git rebase master
Successfully rebased and updated refs/heads/f1.
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git log --oneline
6f28b7a (HEAD -> f1) t2.c in f1
b817182 (master) t3.c in master
2824c78 t1.c in master
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (f1)
$ git checkout master
Switched to branch 'master'
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git merge f1
Updating b817182..6f28b7a
Fast-forward
t2.c | 1 +
1 file changed, 1 insertion(+)
create mode 100644 t2.c
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
6f28b7a (HEAD -> master, f1) t2.c in f1
b817182 t3.c in master
2824c78 t1.c in master
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$
Activate Windows
Go to Settings to activate Windows.

7. Operations on GitHub Repository.

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

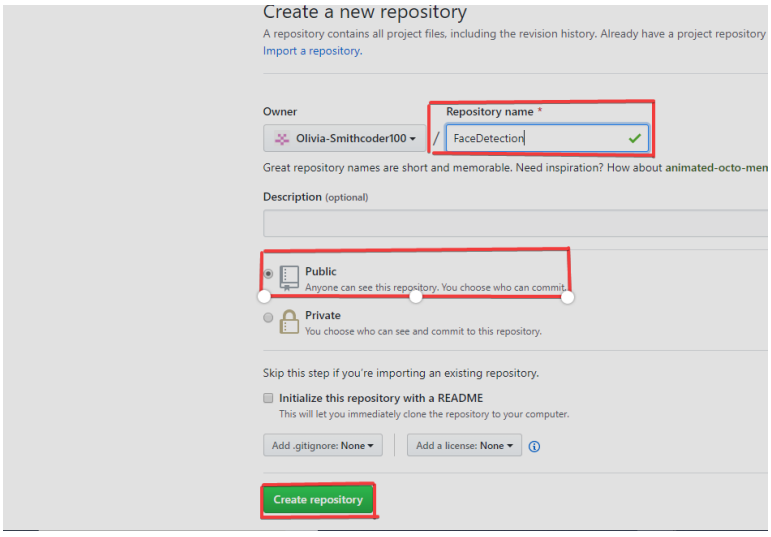


Git PUSH

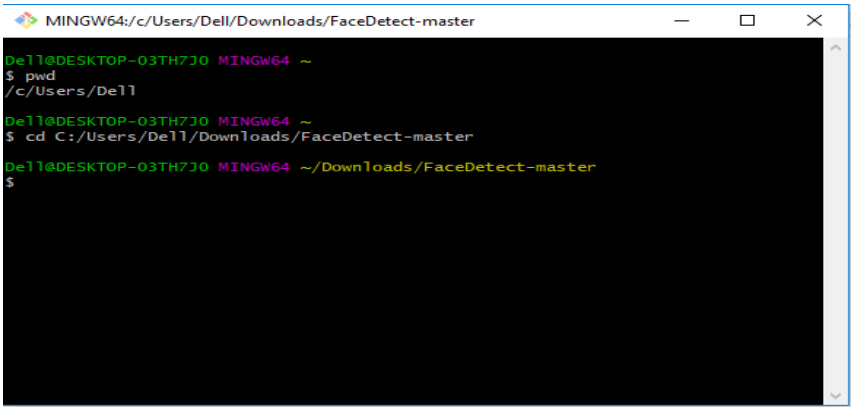
The git push command is used to transfer or push the commit, which is made on a local branch in your computer to a remote repository like GitHub. The command used for pushing to GitHub is given below.

git push 'remote_name' 'branch_name'

1. Creating a new repository



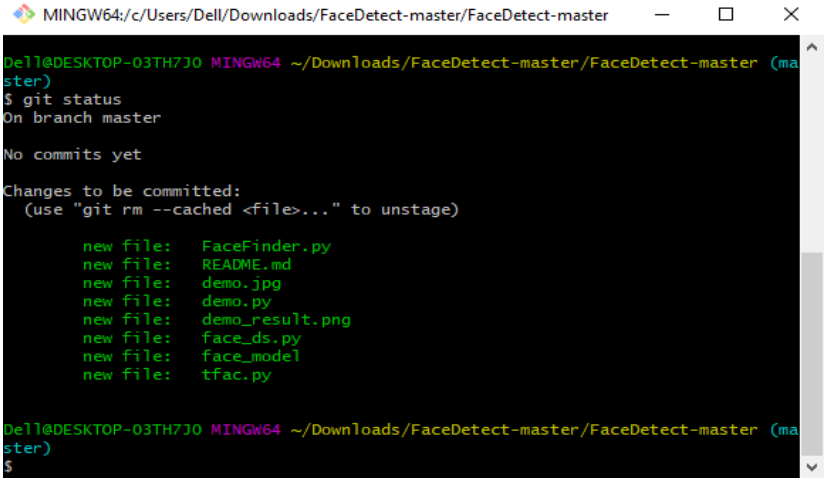
2. Open your Git Bash



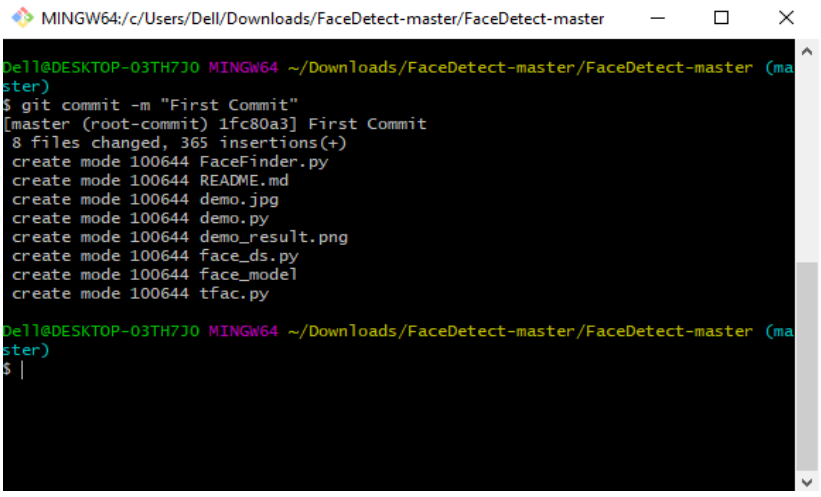
3. Initialize the git repository

4. Add the file to the new local repository

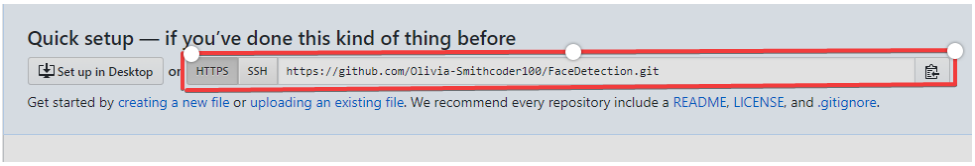
- Use git add . in your bash to add all the files to the given folder.
- Use git status in your bash to view all the files which are going to be staged to the first commit.



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

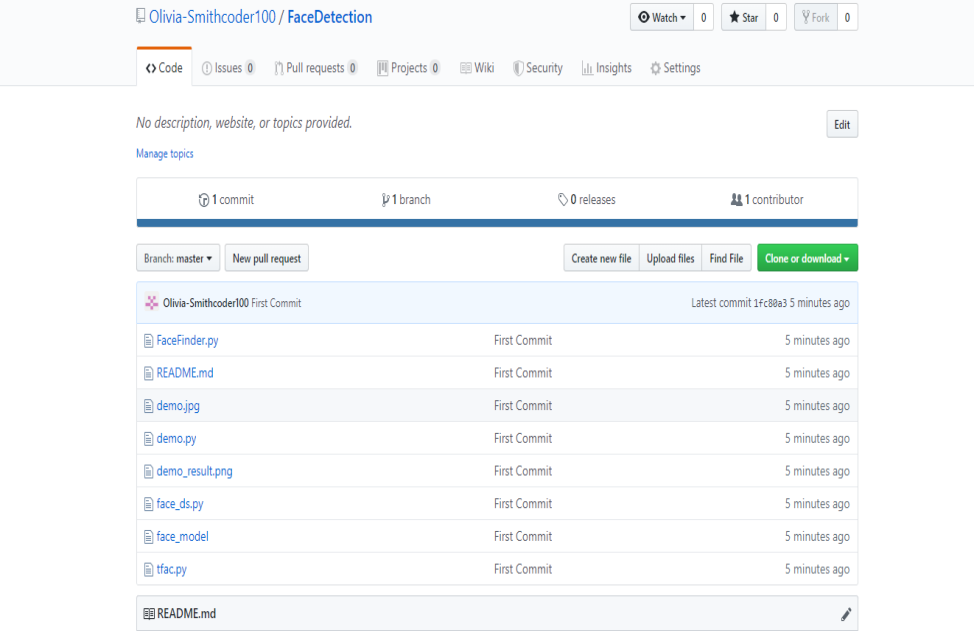
```
git remote add origin 'your_url_name'
```

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

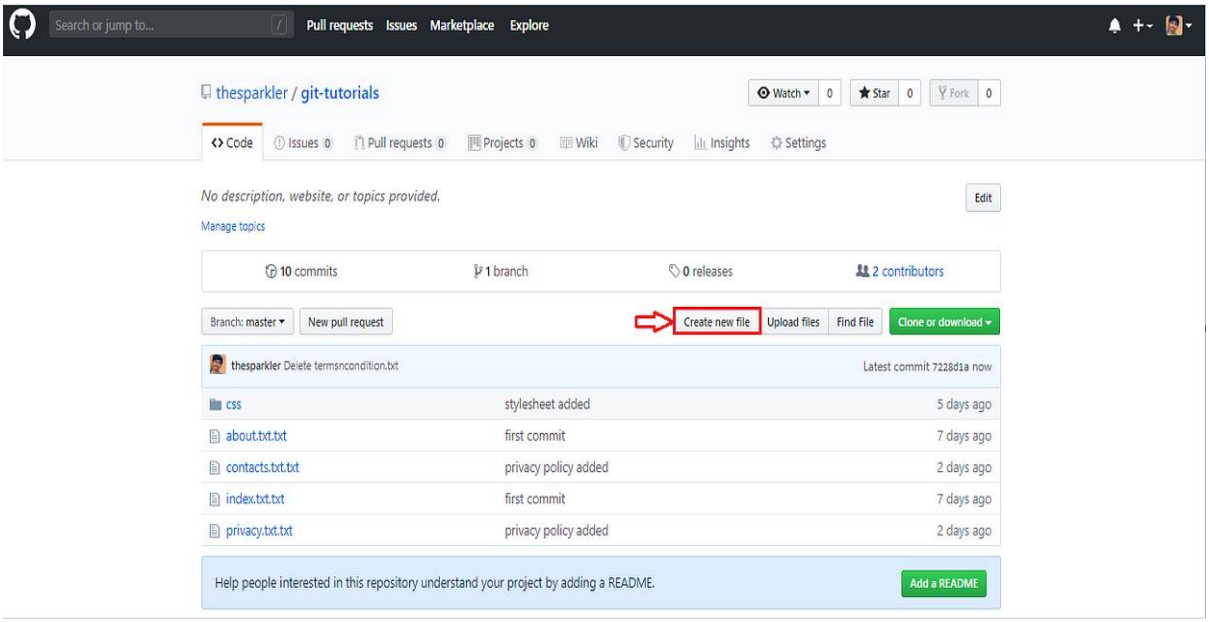
You can finally see the file hosted on GitHub.

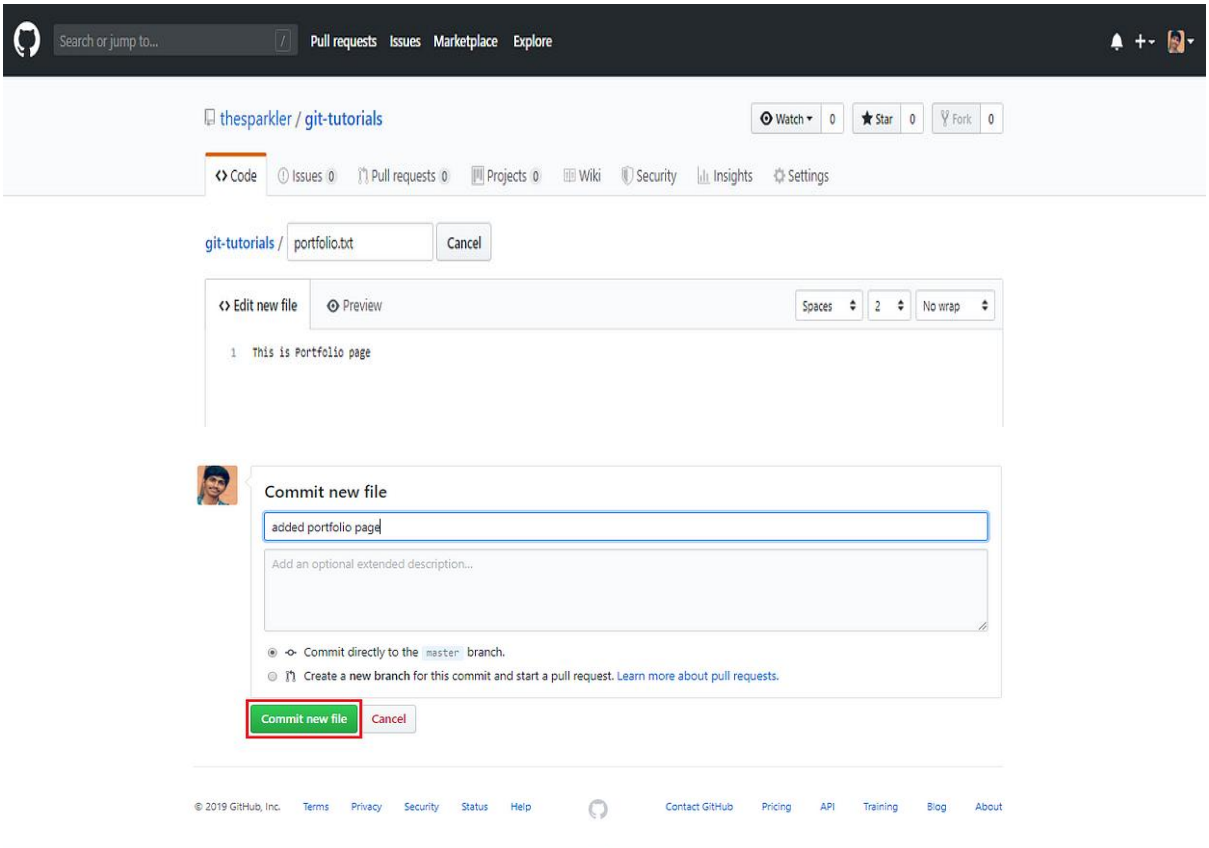


b) Command to push files from local to particular branch in remote repo

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.



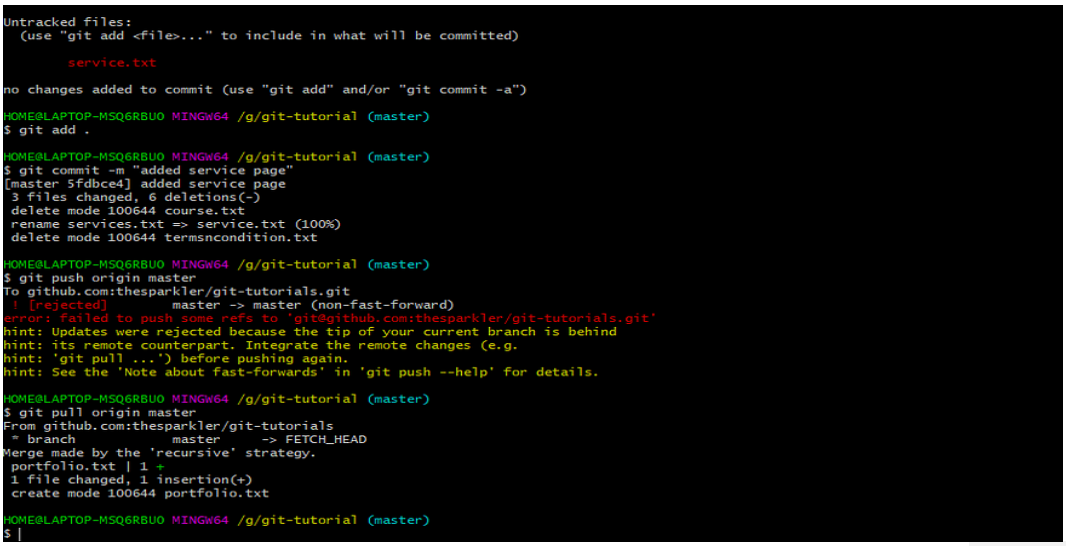


Added commit message and click in commit new file to save changes

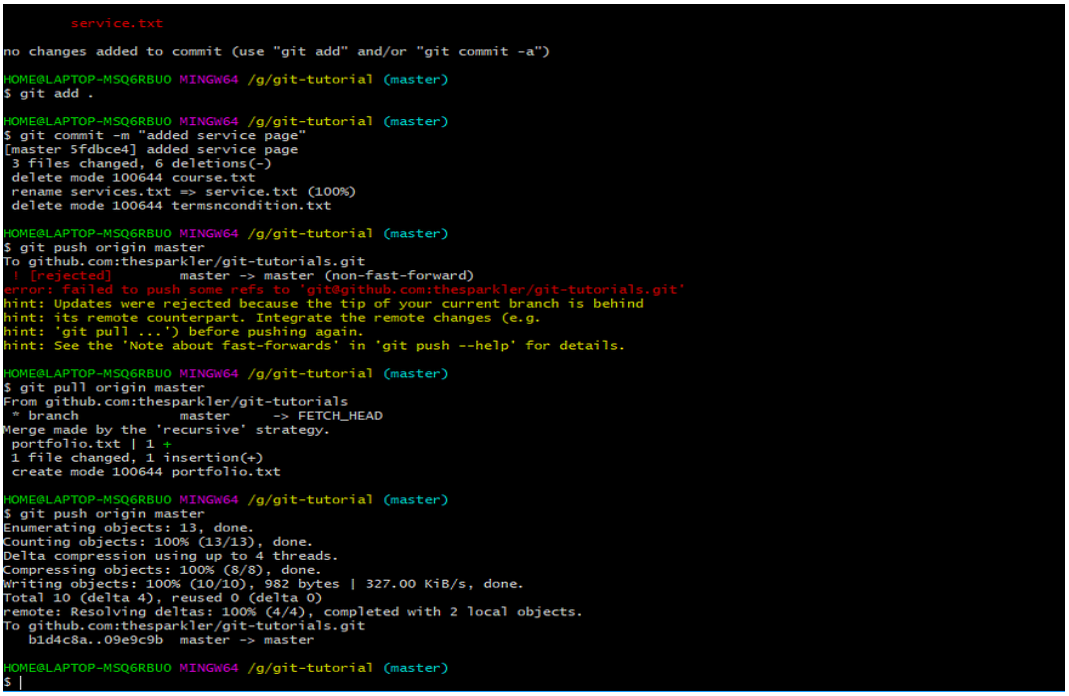
Create another file in the local repository (in your system).

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 master command.



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



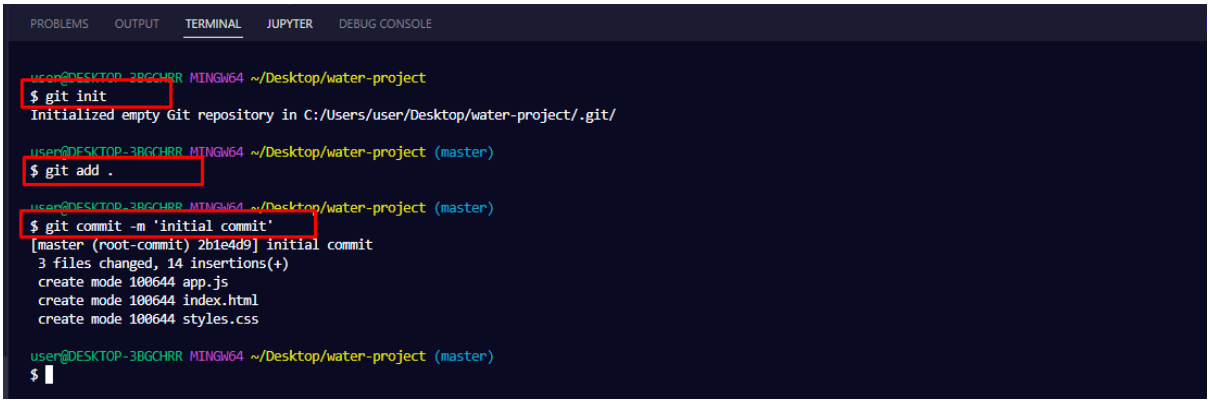
In the above image, you see that your file is successfully pushed to the master branch because we use git pull origin master the command to up-to-date our local repository from a remote repository.

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

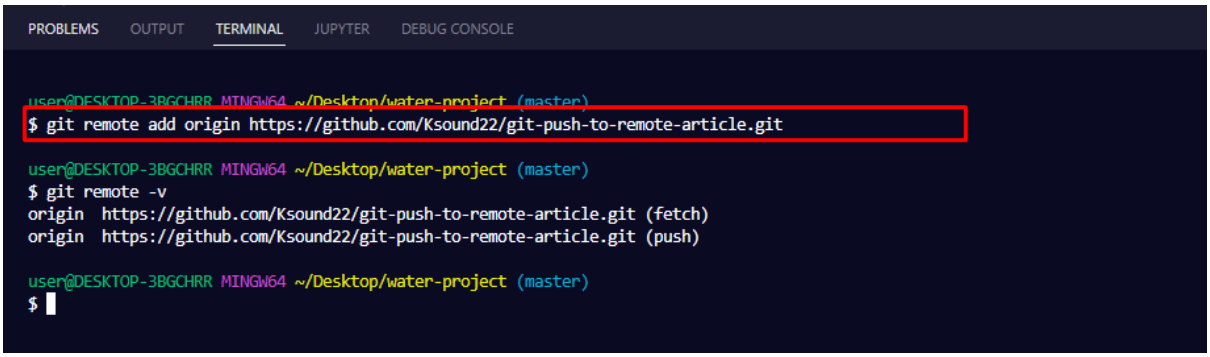


To push the main repo, you first have to add the remote server to Git by running

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**.**

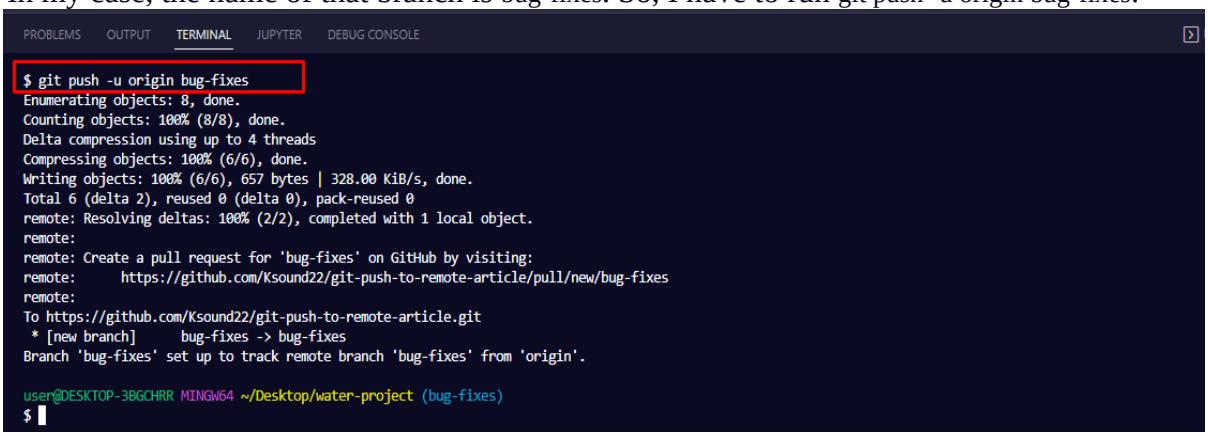
  And to switch to that branch so you can work there, you have to run

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

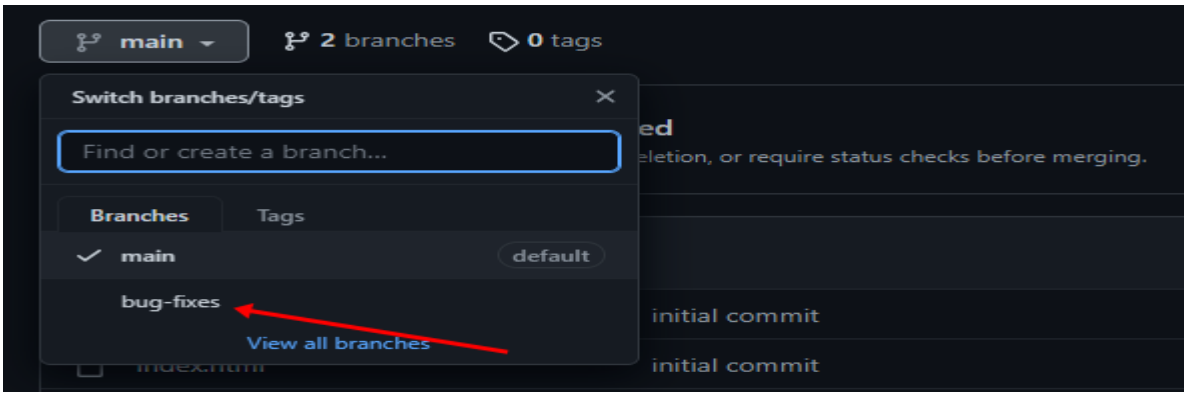
To push the branch to the remote server, run

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:



8. 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.

Tags are ref's that point to specific points in Git history. Tagging is generally used to capture a point in history that is used for a marked version release (i.e. v1. 0.1).

Tag is the name given to a set of versions of files and dirs. It indicates milestones of a project. we cannot modify tag... we can only delete it

1. Command to list the tags

git tag

2. Command to create a tag

git tag <tag_name>

```
Admin@Chethana MINGW64 ~/Documents (master)
$ git tag

Admin@Chethana MINGW64 ~/Documents (master)
$ git tag version 1.0
fatal: Failed to resolve '1.0' as a valid ref.

Admin@Chethana MINGW64 ~/Documents (master)
$ git tag version1.0

Admin@Chethana MINGW64 ~/Documents (master)
$ git tag
version1.0
```

ONCE YOU CREATE A TAG, YOU CANNOT CHANGE ANYTHING IN THAT TAG, ANYWAY YOU CAN DELETE THE TAG.

2. To create a tag with some commit:

Git tag tag-name commit-identifier

This will create a local tag with the commit-identifier of the branch you are on.

3. COMMAND TO DELETE THE TAG

Git tag –d <tag_name>

9. Advanced Git Operations

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

```
neha@vostro-3578:~/dir$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
neha@vostro-3578:~/dir$ git stash
Saved working directory and index state WIP on master: 914f253 changes
neha@vostro-3578:~/dir$ git status
On branch master
nothing to commit, working tree clean
neha@vostro-3578:~/dir$
```

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

```
neha@vostro-3578:~/dir$ git stash list
stash@{0}: WIP on master: 914f253 changes
neha@vostro-3578:~/dir$
```

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**

```
neha@vostro-3578:~/dir$ git stash pop
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (9fdb4cdb228bed10e863711fc75bc9167cee1c5e)
neha@vostro-3578:~/dir$ git stash list
neha@vostro-3578:~/dir$
```

- **git stash apply**

```
neha@vostro-3578:~/dir$ git stash apply
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
neha@vostro-3578:~/dir$ git stash list
stash@{0}: WIP on master: 914f253 changes
neha@vostro-3578:~/dir$
```

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

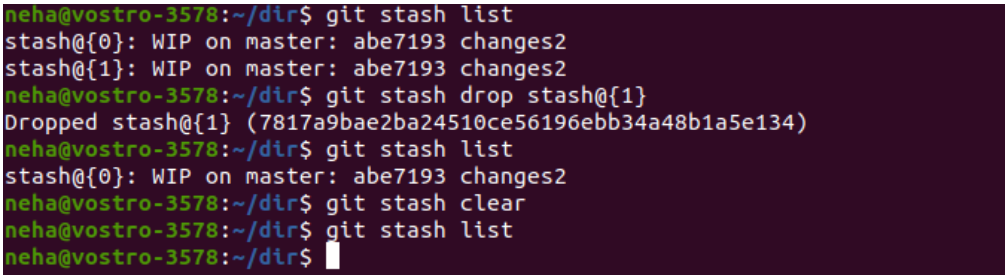
git stash show command is used to display the summary of operations performed on the stash.

- **git stash show**

```
neha@vostro-3578:~/dir$ git stash show
test.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
neha@vostro-3578:~/dir$
```

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.



10. 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.

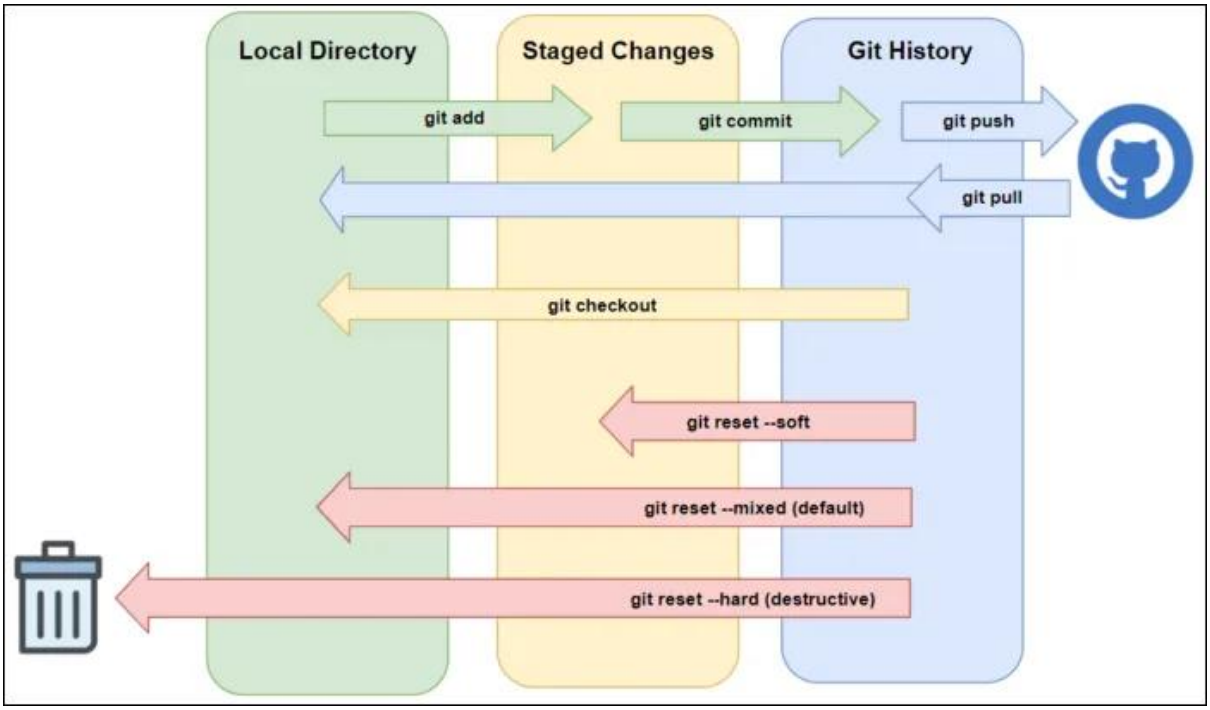
git reset syntax, usage, and modes

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*.

```
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
9660550 (HEAD -> master) t1.c file is modified-3
2dbb20a t1.c file is modified-2
a0ddcf3 t1.c file is modified-1
be99477 t1.c file is created

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
nothing to commit, working tree clean

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git reset 2dbb20a
Unstaged changes after reset:
M       t1.c
```

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

```
MINGW64:/c/Users/Admin/Desktop/git syllabus/sam

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
2dbb20a (HEAD -> master) t1.c file is modified-2
a0ddcf3 t1.c file is modified-1
be99477 t1.c file is created

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git reset a0ddcf3
Unstaged changes after reset:
M       t1.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
a0ddcf3 (HEAD -> master) t1.c file is modified-1
be99477 t1.c file is created
```

- It will not touch working directory

```
MINGW64:/c/Users/Admin/Desktop/git syllabus/sam

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   t1.c

no changes added to commit (use "git add" and/or "git commit -a")
```

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

```
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git commit -m "t2.c is created"
[master 19b632e] t2.c is created
 1 file changed, 1 insertion(+)
 create mode 100644 t2.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ vi t3.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git add .
warning: in the working copy of 't3.c', LF will be replaced by CRLF the next time Git touc
hes it

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git commit -m "t3.c is created"
[master d7c2df1] t3.c is created
 1 file changed, 1 insertion(+)
 create mode 100644 t3.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
nothing to commit, working tree clean

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
d7c2df1 (HEAD -> master) t3.c is created
19b632e t2.c is created
611ad37 t1.c is created
```

```
MINGW64:/c/Users/Admin/Desktop/git syllabus/sam
$ git status
On branch master
nothing to commit, working tree clean

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
d7c2df1 (HEAD -> master) t3.c is created
19b632e t2.c is created
611ad37 t1.c is created

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git reset --soft 19b632e

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
19b632e (HEAD -> master) t2.c is created
611ad37 t1.c is created

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   t3.c


Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git ls-files
t1.c
t2.c
t3.c
```

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.

```
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git ls-files
t1.c
t2.c
t3.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
nothing to commit, working tree clean

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
bbfb681 (HEAD -> master) t3.c is commited again
19b632e t2.c is created
611ad37 t1.c is created
```

Activate Windows

```
Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git ls-files
t1.c
t2.c
t3.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
nothing to commit, working tree clean

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git log --oneline
bbfb681 (HEAD -> master) t3.c is commited again
19b632e t2.c is created
611ad37 t1.c is created

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git reset --hard 19b632e
HEAD is now at 19b632e t2.c is created

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git ls-files
t1.c
t2.c

Admin@Chethana MINGW64 ~/Desktop/git syllabus/sam (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Activate Windows

11. **Advanced Git Operations**

- 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 goto 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.

12. **Analysing and Changing Git History**

- Given a commit ID, how would you use Git to view the details of that specific commit, including the author, date, and commit message?

To view the details of a specific commit, you use the command git show with the first few characters of the commits hash. For example, the command git show b817182 produces this: