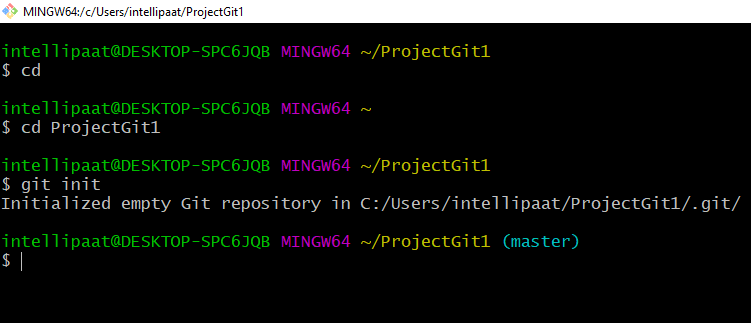
Git Commands:

1. git init

Usage: git init [repository name]

We have to navigate to our project directory and type the command git init to initialize a Git repository for our local project folder. Git will create a

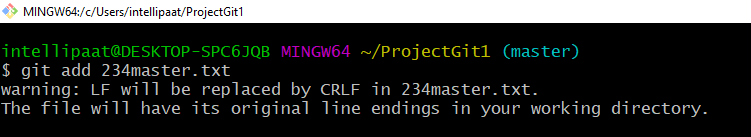
hidden. git directory and use it to keep its files organized in other subdirectories.



2. git add

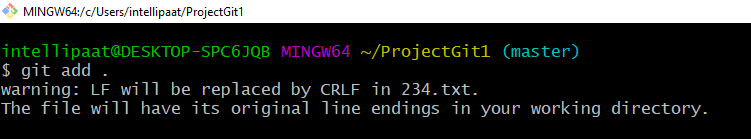
Usage (i): git add [file(s) name]

This will add the specified file(s) into the Git repository, the staging area, where they are already being tracked by Git and now ready to be committed.



Usage (ii): git add . or git add \*

This will take all our files into the Git repository, i.e., into the staging area.



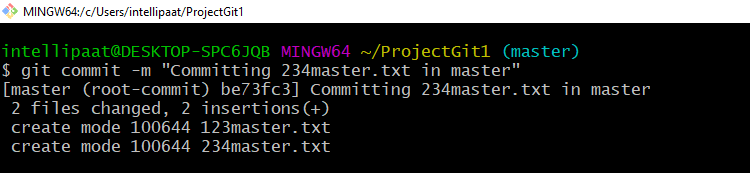
We can use this command as git add -A as well.

Note: We will have to commit our files after we add them to the staging area.

**3. git commit**

**Usage: git commit -m “message”**

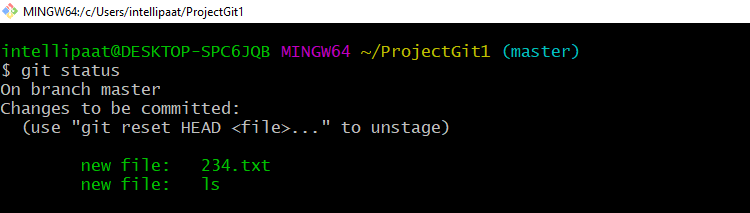
This command records or snapshots files permanently in the version history. All the files, which are there in the directory right now, are being saved in the Git file system.



**4. git status**

**Usage: git status**

This command will show the modified status of an existing file and the file addition status of a new file, if any, that has to be committed.

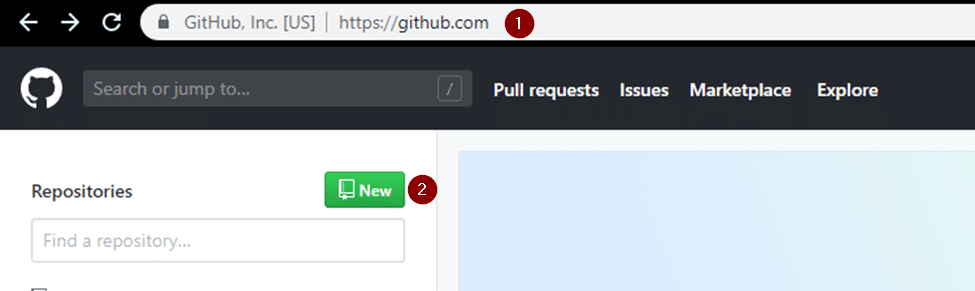


**5. git remote**

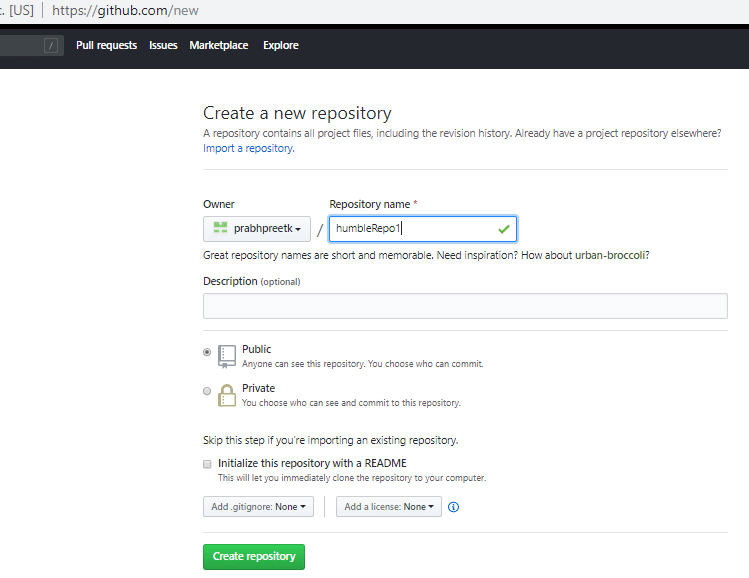
**Usage: git remote add origin “[URL]”**

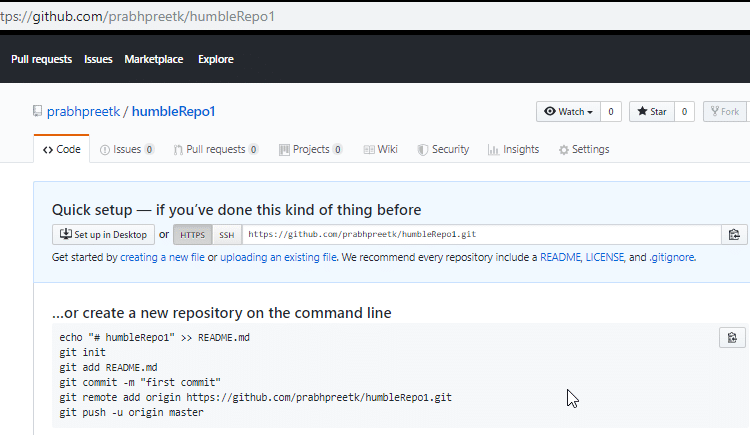
Once everything is ready on our local system, we can start pushing our code to the remote (central) repository of the project. For that, follow the below steps:

**Step 1:**  
**(1)**Login to the **GitHub account** if the account already exists (If not, sign up on github.com)  
**(2)** Click on**New**



**Step 2:**Now, we have to create a new repository. Provide a **name**to our**repository**, select the **privacy**of the repository as**Public**, and then click on **Create repository**

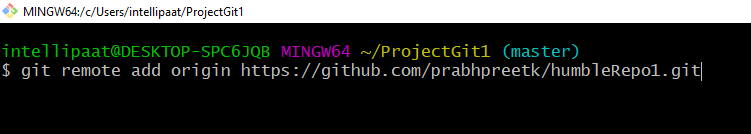




**Step 3:** Click on the Copy icon on the right side of the URL box of the Github repository to copy the link and paste it as shown below:

git remote add origin “URL”

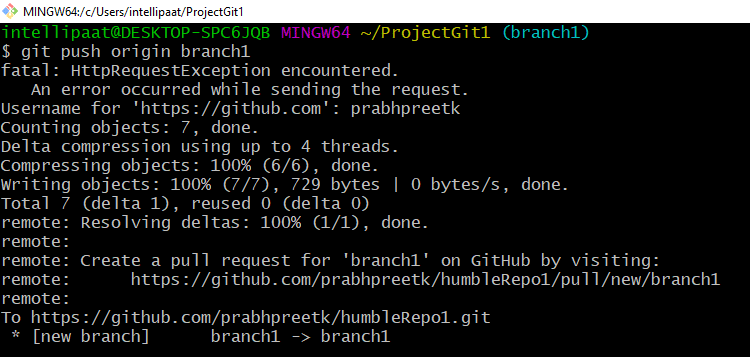
Now, we are ready to operate the remote commands in our repository that we have just created.



**6. git push**

**Usage: git push origin [branch name]**

Suppose, we have made some changes in the file and want to push the changes to our remote repository on a particular branch. By using the command ‘git push,’ the local repository’s files can be synced with the remote repository on GitHub.

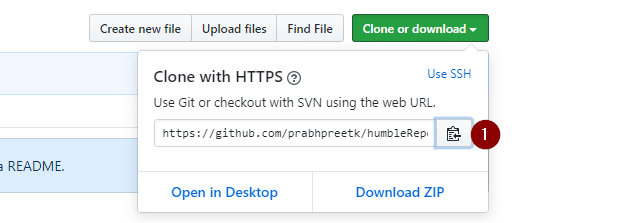


**7. git clone**

**Usage: git clone [URL]**

Suppose, we want to work on a file that is on a remote GitHub repository as another developer. How can we do that? We can work on this file by clicking on **Clone or** **Download**and copying the link and pasting it on the terminal with the git clone command. This will import the files of a project from the remote repository to our local system.

**(1)** The below screenshot shows from where to copy the link:



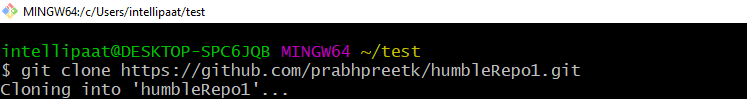
To create a local folder, we have to use the following command:

mkdir [directory- name]

cd [directory- name]

git clone [URL]

Now, paste the copied link along with the git clone command as shown below:



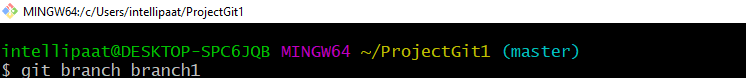
**Note**: Here, we don’t have to use the **git remote add origin** command because we have already cloned the remote repository in the local directory. Now, if we push any new file, it knows where it has to go.

**Read on:**[**What is GitLab?**](https://intellipaat.com/blog/what-is-gitlab/)

**8. git branch**

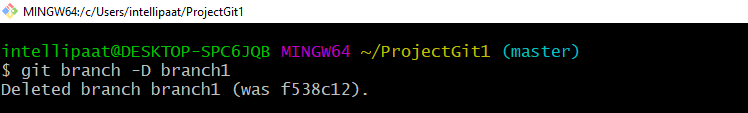
**Usage (i): git branch [name-of-the-branch]**

When multiple developers are collaborating on a project or repository, branches become essential for managing different workspaces. Using this command, we can create a new branch (for example, ‘branch1’). This allows developers to work independently on their respective branches, making changes and commits without affecting the main branch or other branches.



**Usage (ii): git branch -D [name-of-the-branch]**

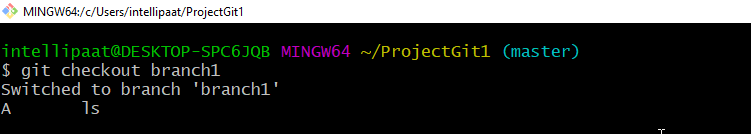
Likewise, to delete a branch, we utilize the “git branch -D” command. This enables us to remove a specific branch (e.g., ‘name-of-the-branch’) that is no longer needed, cleaning up the repository and reducing clutter.



**9. git checkout**

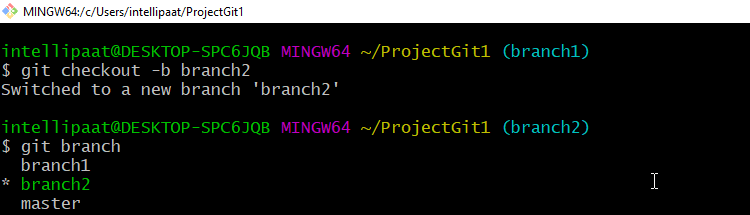
**Usage (i): git checkout [name-of-the-new-branch]**

This command allows us to switch to an existing branch within our repository. It facilitates navigating to the desired branch, enabling us to add new files, make changes, and commit those files within that specific branch.



Usage (ii): git checkout -b [name-of-the-new-branch]

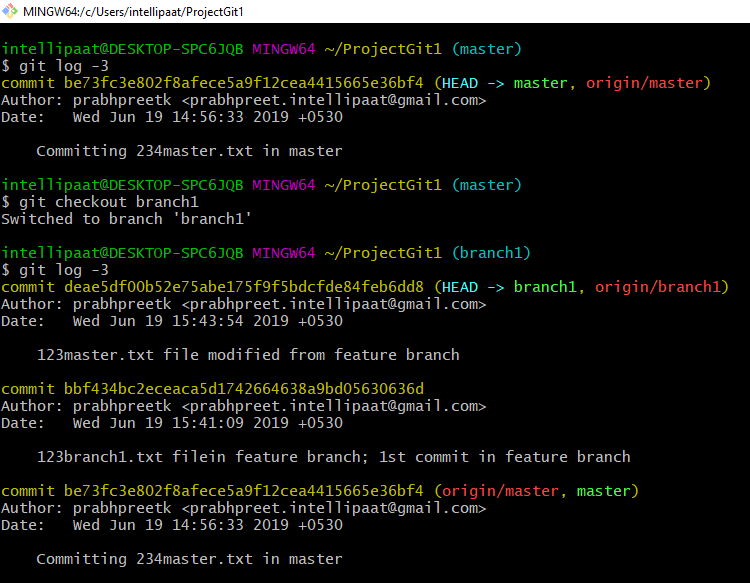
This command serves a dual function. Firstly, it creates a new branch with the given name (for example, ‘branch2’). Secondly, it immediately switches our working environment to that newly created branch. This allows us to seamlessly begin working within the newly created branch, making it convenient to add files, make modifications, and commit changes exclusively within that branch.



**10. git log**

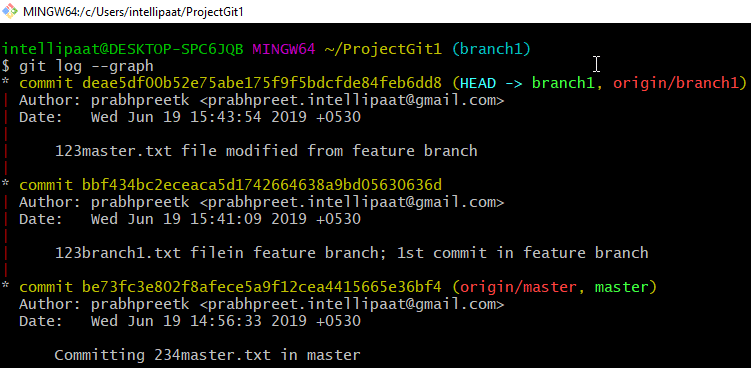
**Usage (i): git log**

The “git log” command is handy when we want to examine the detailed log of every commit in our repository. By executing this command, we can view the log specific to the branch we are currently in. Additionally, we can use “git log -3” to display the last three logs.



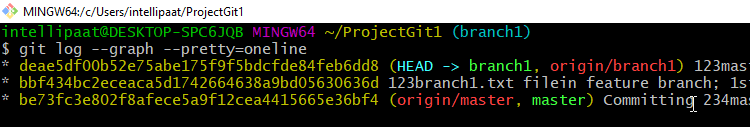
**Usage (ii): git log –graph**

For a visual representation of the commit history, we can utilize “git log –graph”. This option presents the commit-graph, showcasing the branching and merging of commits



**Usage (iii): git log –graph –pretty=oneline**

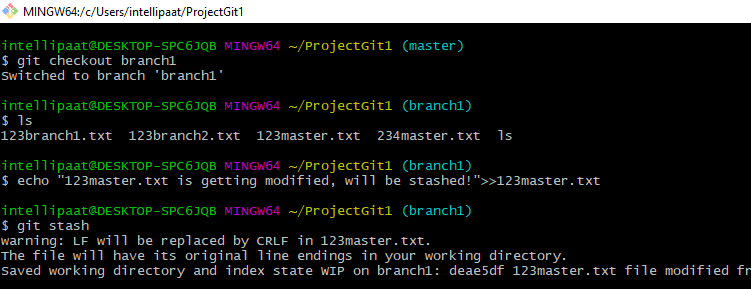
To further customize the output, we can use “git log –graph –pretty=oneline”. This format displays the commit graph along with a concise one-line description for each commit.



**11. git stash**

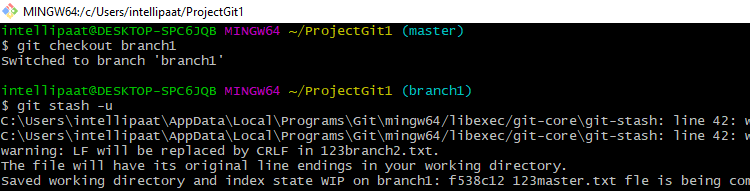
**Usage (i): git stash**

This command can be used when we want to save our work without staging or committing the code to our Git repository and want to switch between branches.



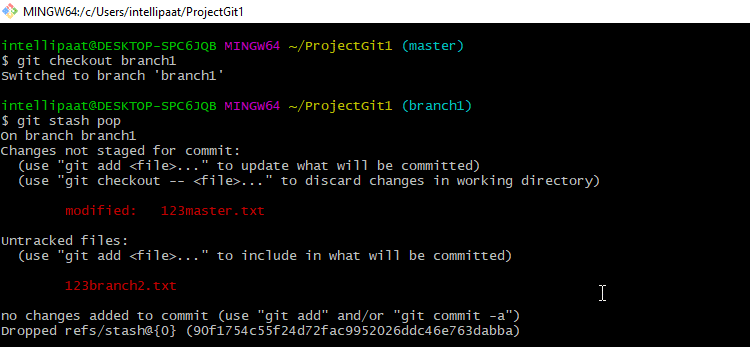
**Usage (ii): git stash -u**

This command is used when we want to stash the untracked files.



**Usage (iii): git stash pop**

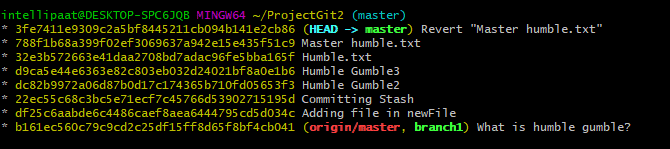
This command is used when we are back on our branch and want to retrieve the code.



**12. git revert**

**Usage: git revert [commit id]**

The git revert command can be considered as an ‘undo’ command. However, it does not work as the traditional ‘undo’ operation. It figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

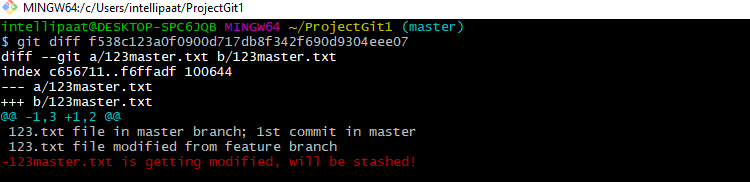


**13. git diff**

**Usage: git diff [commit-id-of-version-x] [commit-id-of-version-y]**

Diffing is a function that takes two input datasets and outputs the changes between them. The git diff command is a multi-use Git command which, when executed, runs a diff function on Git data sources. These data sources can be commits, branches, files, and more. The git diff command is often used along with the git status and git log commands to analyze the current state of our Git repository. We use **git log** to get the details of commit IDs.

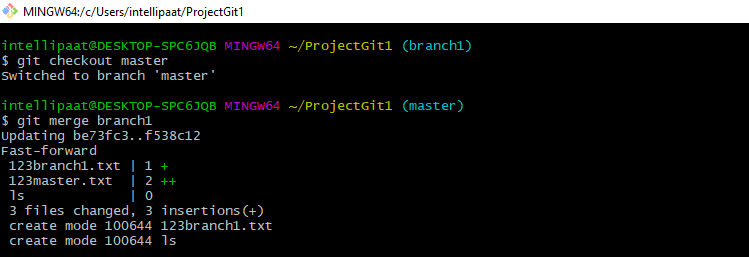
Let’s compare the working directory with the index as shown below:



**14. git merge**

**Usage: git merge [another-file-name]**

This command will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The git merge command takes two commit pointers, usually the branch tips, and finds a common base commit between them. Once it finds a common base commit, it will create a commit sequence.

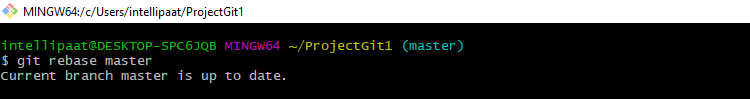


**15. git rebase**

**Usage: git rebase [base]**

[Rebase](https://intellipaat.com/blog/git-rebase-vs-merge/) is the process of moving and combining a sequence of commits to a new base commit. Rebasing is changing the base of our branch from one commit to another, making it appear as if we’ve created our branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It’s very important to understand that even though the branch looks the same, it is composed of entirely new commits.

The git rebase command performs an automatic git checkout before doing anything else. Otherwise, it remains on the current branch.



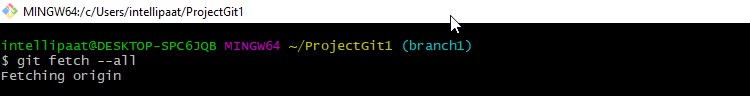
Consider a situation where we have branched off from the master and have created a feature branch, but the master branch is still having more commits. We want to get the updated version of the master branch in our feature branch, keeping our branch’s history clean, so that it appears as if we are working on the latest version of the master branch.

**Note**: We don’t rebase public history. We should never rebase commits once they are pushed to a public repository. Why because the rebase would replace the old commits with the new ones, and it would appear that a part of our project history got abruptly vanished.

**16. git fetch**

**Usage: git fetch**

When we use the command git fetch, Git gathers any commit from the target branch that does not exist in our current branch and stores it in our local repository. However, it does not merge it with our current branch.

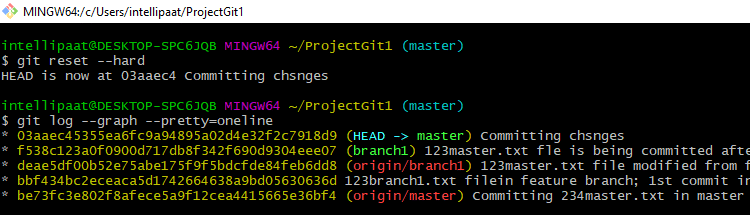


In situations where we want to keep our repository up to date but are concerned that updating our files might lead to issues, a specific technique comes to the rescue. To integrate the commits into our master branch, we use the merge feature. This feature actively retrieves all the branches from the repository and then proceeds to download all the required commits and files from another repository. It ensures that our repository remains current while mitigating the risk of potentially breaking our ongoing work.

**17. git reset**

**Usage: git reset –hard [SOME-COMMIT]**

We use this command to **return** the *entire* working tree to the last committed state.



This will discard commits in a private branch or throw away the uncommitted changes!