

1. Setting Up and Basic Commands

Step to initialize a new Git repository, create a new file, add it to the staging area, and commit the changes.

Open the terminal or command prompt and navigate to the directory where you want to initialize the Git repository.

To initialize a new Git repository, use the following command: **git init** This will create a new, empty Git repository in the current directory.

Now, let's create a new file in the repository. we can use any text editor you prefer. For example, if you want to create a file named "example.txt", you can use the command: **touch example.txt** This command creates an empty file named "example.txt" in the current directory. or **vi exm.txt**

After creating the file, we need to add it to the staging area. The staging area is where we specify which changes we want to include in the next commit. Use the following command to add the file to the staging area: **git add example.txt** This command **git add "example.txt"** to the staging area.

Finally, we can commit the changes with an appropriate commit message. A commit is like a snapshot of the current state of the repository. Use the following command to commit the changes: **git commit -m "Add example.txt file"** Replace "Add example.txt file" with a meaningful commit message that explains the purpose of the commit.

GIT COMMANDS:

- **git --version**

This command to check the version

- **mkdir foldername** eg: **mkdir demo**
- **cd desktop**
- **cd demo**
- **git init**
- **git status**
- **git add demo.txt**
- **git commit -m "committing text file"**
- **git config user.name " "**
- **git config user.email " "**
- **git config --global user.name " " //To link with git hub account ,git hub username & password**
- **git config --global user.email " "**

Usage: **git config --global user.name "name"**

Usage: **git config --global user.email "email address"**

This command sets the author name and email address respectively to be used with your commits.

```
student@student1:~$ mkdir 1sv22is004
student@student1:~$ cd 1sv22is004
student@student1:~/1sv22is004$ git --version
git version 2.34.1
student@student1:~/1sv22is004$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/student/1sv22is004/.git/
student@student1:~/1sv22is004$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
student@student1:~/1sv22is004$ vi demo1.txt
student@student1:~/1sv22is004$ git add demo1.txt
student@student1:~/1sv22is004$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   demo1.txt

student@student1:~/1sv22is004$ git commit -m "DEMO FILE 1"
[master (root-commit) 1717e75] DEMO FILE 1
 1 file changed, 1 insertion(+)
 create mode 100644 demo1.txt

student@student1:~/1sv22is004$ git config --global user.name"avinashparagoudar"
student@student1:~/1sv22is004$ git config --global user.email"avinashparagoudar469@gmail.com"
student@student1:~/1sv22is004$
```

2. Creating and Managing Branches

Create a new branch named "feature-branch". Switch to the "master" branch. Merge the "feature-branch" into "master".

Commands:

- Create a new branch named "feature-branch":

git branch feature-branch //This will create new branch called feature-branch

git checkout feature-branch

This command creates a new branch named "feature-branch" and switches to it.

git checkout master

- Switch back to the "master" branch:

This command switches back to the "master" branch.

- Merge the "feature-branch" into "master":

git merge feature-branch

This command will merge the changes from "feature-branch" into the "master" branch.

After completing these steps, your "feature-branch" changes will be integrated into the "master" branch.

git status //it will check the status

git push origin master

```
student@student1:~/1sv221s004$ git branch feature-branch
student@student1:~/1sv221s004$ git checkout feature-branch
Switched to branch 'feature-branch'
student@student1:~/1sv221s004$ git add .

student@student1:~/1sv221s004$ git merge feature-branch
Updating 84e40af..6bb0cf6
Fast-forward
 demo5.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 demo5.txt
student@student1:~/1sv221s004$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
       new-one/

nothing added to commit but untracked files present (use "git add" to track)
student@student1:~/1sv221s004$
```

3. Creating and Managing Branches

Commands to stash your changes, switch branches, and then apply the stashed changes

Stash your changes: **git stash** This command will temporarily save your changes, allowing you to switch branches without committing them. Git will revert your working directory and staging area to the last commit, giving you a clean state to switch branches.

Switch branches: **git checkout <branch-name>** Replace **<branch-name>** with the name of the branch you want to switch to. This command allows you to move to a different branch in your Git repository.

Apply the stashed changes: **git stash apply** This command reapplies the most recent stash you created. It will restore your previously stashed changes, allowing you to continue working on them.

If you have multiple stashes, you can apply a specific stash by using its index. First, list the stashes using the command **git stash list**. Then, apply a specific stash by index using the command: **git stash apply stash@{<index>}** Replace **<index>** with the index number of the stash you want to apply.

```
student@student1:~/1sv221s004$ git push origin v1.0
Username for 'https://github.com': avinashparagoudar
Password for 'https://avinashparagoudar@github.com':
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 2 threads
Compressing objects: 100% (1/1), done.
Writing objects: 100% (2/2), 286 bytes | 286.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/avinashparagoudar/hii.git
 * [new tag]          v1.0 -> v1.0
```

Additionally, if you want to remove the stash after applying it, you can use the command

git stash drop followed by the stash's index or **git stash drop stash@{<index>}**.

It's worth noting that stashing is useful when you want to switch branches without committing your changes. It allows you to work on multiple branches while keeping your changes separate and easily applicable when needed.

Commands:

git branch feature-branch2

git status

vi sample.txt

`git add sample.txt`

`git checkout feature-branch2`

`git log --oneline`

`git stash`

`git stash list`

`git stash show stash@{1}`

`git stash apply`

`git stash pop`

`git stash list`

```
student@student1:~/1sv221s004$ git branch feature-branch2
student@student1:~/1sv221s004$ git status
On branch master
nothing to commit, working tree clean
student@student1:~/1sv221s004$ vi sample.txt
student@student1:~/1sv221s004$ git add .
student@student1:~/1sv221s004$ git add sample.txt
student@student1:~/1sv221s004$ git checkout feature-branch2
A       sample.txt
Switched to branch 'feature-branch2'
student@student1:~/1sv221s004$ git log --oneline
84e40af (HEAD -> feature-branch2, tag: v1.1, tag: v1.0, master) Revert "DEMO FILE 1"
f57b5bf (origin/master) DEMO FILE 1
8556b4a (tmp) done
student@student1:~/1sv221s004$ git stash
Saved working directory and index state WIP on feature-branch2: 84e40af Revert "DEMO FILE 1"
student@student1:~/1sv221s004$ git stash list
stash@{0}: WIP on feature-branch2: 84e40af Revert "DEMO FILE 1"
student@student1:~/1sv221s004$ git stash apply
On branch feature-branch2
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   sample.txt
```

4. Collaboration and Remote Repositories

Clone a remote Git repository to your local machine

The process of cloning a remote Git repository to your local machine:

Open the terminal or command prompt on your local machine and navigate to the directory where you want to clone the remote repository.

Obtain the URL of the remote Git repository you want to clone. This can usually be found on the repository's webpage or by asking the repository owner. The URL will typically end with **.git**.

To clone the remote repository, use the following command: **git clone <repository-url>** Replace **<repository-url>** with the URL of the remote repository you obtained in the previous step. This

command will create a copy of the remote repository on your local machine.

Git will start downloading the remote repository's contents to your local machine. Once the cloning process is complete, you will have a local copy of the entire repository, including its commit history and branches.

You have successfully cloned a remote Git repository to your local machine. The cloned repository will be stored in a new directory with the same name as the remote repository.

Now, you can start working with the cloned repository on your local machine. You can make changes, create branches, commit your work, and push your changes back to the remote repository when you're ready to share or collaborate with others.

Cloning a remote repository is an essential step in collaborating on Git projects. It allows you to have a local copy of the project, work independently on your machine, and easily synchronize your changes with the remote repository.

command: `git clone <repository-url>`

```
student@student1:~/1sv221s004$ git clone "https://github.com/avinashparagoudar/new-one.git"
Cloning into 'new-one'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
student@student1:~/1sv221s004$
```

5. Collaboration and Remote Repositories

Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Step1: Ensure that you are in the working directory of your local repository. Run `git fetch origin` to fetch the latest changes from the remote repository. Replace origin with the name of your remote repository if it is different. Check out the branch you want to rebase onto the updated remote branch. For example, if you want to rebase your feature-branch onto the updated master branch, run `git checkout feature-branch`.

Step2: Run `git rebase origin/master`, where origin/master is the remote branch you want to rebase onto. This command replays your commits on top of the updated remote branch. If there are any conflicts during the rebase process, Git will pause the rebase and display the conflicting files. You need to resolve the conflicts manually by editing the conflicting files. After resolving the conflicts, use `git add <file>` for each resolved file to stage them for the rebase. Once all conflicts have been resolved and files have been staged, continue the rebase process by running `git rebase --continue`. This will apply the next commit on top of the updated remote branch. If there are any further conflicts, repeat steps 5-7 until the rebase is successfully completed. After the rebase is complete, you may need to force push your updated branch to the remote repository if you have already pushed the previous commits. Use `git push -f origin feature-branch` to force push the updated branch. Be cautious with this step, as it rewrites the history and can cause issues for other collaborators. The local branch is now rebased onto the updated remote branch.

Commands step-by-step:

git fetch origin

git checkout feature-branch

git rebase master/origin feature-branch(Resolve any conflicts if prompted)

git add <resolved-file> (Stage each resolved file) **git rebase --continue** (Repeat steps 4-5 if necessary)

git push -f origin feature-branch (Force push the updated branch to the remote repository)

Note: Replace origin with the name of your remote repository if it is different, and change featurebranch and master with the actual branch names you are working with. Be cautious with the force push (git push -f), as it rewrites history and can cause issues for other collaborators.

```
student@student1:~/1sv22is004$ vi demo4.txt
student@student1:~/1sv22is004$ git add demo4.txt
student@student1:~/1sv22is004$ git commit -m "DEMO 4"
[feature-branch2 b57bdf4] DEMO 4
2 files changed, 2 insertions(+)
create mode 100644 demo4.txt
create mode 100644 sample.txt
student@student1:~/1sv22is004$ git rebase master feature-branch
warning: skipped previously applied commit 1717e75
hint: use --reapply-cherry-picks to include skipped commits
hint: Disable this message with "git config advice.skippedCherryPicks false"
Successfully rebased and updated refs/heads/feature-branch.
student@student1:~/1sv22is004$
```

6. Collaboration and Remote Repositories

Write the command to merge "feature-branch" into "master" while providing a custom commit message for the merge

First, ensure that you are currently on the "master" branch. You can switch to the "master" branch using the following command: **git checkout master**

Next, initiate the merge of the "feature-branch" into "master" by using the following command: **git merge feature-branch** This command will merge the changes from the "feature-branch" into the "master" branch.

At this point, Git will try to automatically merge the changes. If there are any conflicts (i.e., conflicting changes in the same files), Git will notify you and prompt you to resolve the conflicts manually. You can use a text editor or a specialized merge tool to address the conflicts.

Once you have resolved the conflicts, you can proceed with the merge. Git will create a new commit to represent the merge. To provide a custom commit message for the merge, use the following command:

git commit -m "Custom commit message for merging feature-branch into master" Replace "Custom commit message for merging feature-branch into master" with your desired commit message. Make sure to provide a meaningful message that accurately describes the purpose of the merge.

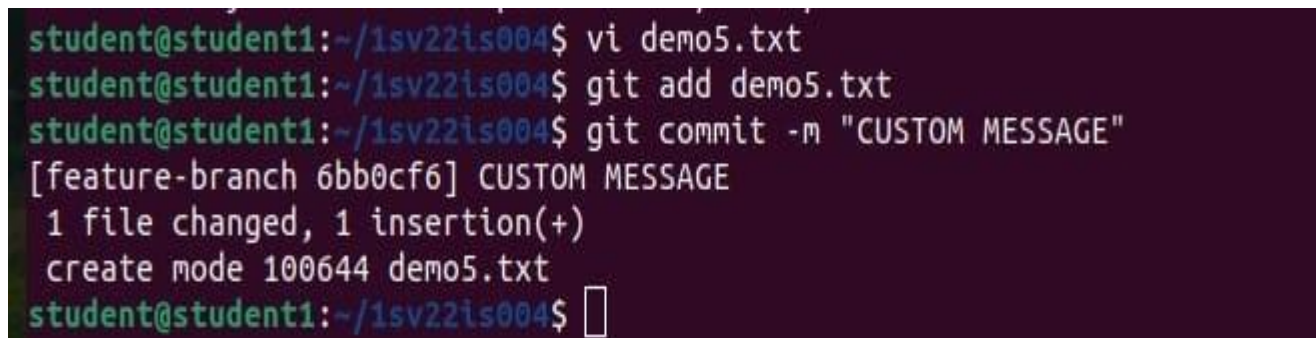
After entering the command, Git will create the merge commit with the provided custom commit message. This commit represents the merge of the changes from the "feature-branch" into the "master" branch.

You have successfully merged the "feature-branch" into the "master" branch while providing a custom commit message for the merge.

Merging branches is a crucial step in collaboration, as it brings together different sets of changes from various branches into a single branch, such as "master." It allows different team members to work on separate features or bug fixes and later integrate them into the main branch.

Commands:

git commit -m "custom commit message"



```
student@student1:~/1sv22is004$ vi demo5.txt
student@student1:~/1sv22is004$ git add demo5.txt
student@student1:~/1sv22is004$ git commit -m "CUSTOM MESSAGE"
[feature-branch 6bb0cf6] CUSTOM MESSAGE
1 file changed, 1 insertion(+)
create mode 100644 demo5.txt
student@student1:~/1sv22is004$
```

7. Git Tags and Releases

Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

First, ensure that you are in the desired Git repository directory on your local machine.

Identify the commit that you want to tag. You can find the commit hash by using the **git log** command, which displays the commit history.

To create a lightweight Git tag named "v1.0" for the identified commit, use the following command:

git tag v1.0 <commit-hash> Replace <commit-hash> with the specific commit hash that you want to tag. For example, if the commit hash is abc123, the command would be: **git tag v1.0 abc123** This command creates a lightweight tag with the name "v1.0" for the specified commit.

You have successfully created a lightweight Git tag named "v1.0" for the specific commit in your local repository.

Git tags are useful for marking specific points in history, such as releases or important milestones.

Lightweight tags are simply pointers to specific commits, containing no additional metadata.

It's important to note that lightweight tags are local to your repository and are not automatically pushed to a remote repository. If you want to share the tags with others, you will need to explicitly push them to the remote repository using the **git push** command.

To push the "v1.0" tag to a remote repository, you can use the following command: **git push origin v1.0** Replace origin with the name of the remote repository you want to push the tag to.

Commands:

git checkout master

git tag v1.0

git tag //it will show v1.0

git tag -a v1.1 -m "tag for release ver 1.1"

git show v1.0

git tag -l "v1.*"

git push origin v1.0 //push tags to origin

```
student@student1:~/1sv221s004$ git tag v1.0
student@student1:~/1sv221s004$ git tag
v1.0
student@student1:~/1sv221s004$ git tag -a v1.1 -m "Tag for release ver 1.1"
student@student1:~/1sv221s004$ git show v1.0
commit 84e40af85b21bb5ec38581e6e17d7651c4b421d9 (HEAD -> master, tag: v1.1, tag: v1.0)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:58:01 2024 +0530

    Revert "DEMO FILE 1"

    This reverts commit 1717e759524b6c69d9549f58fe580127e7738545.

diff --git a/demo1.txt b/demo1.txt
deleted file mode 100644
index 47a950f..0000000
--- a/demo1.txt
+++ /dev/null
@@ -1 +0,0 @@
-hii
```

```
student@student1:~/1sv221s004$ git push origin v1.0
Username for 'https://github.com': avinashparagoudar
Password for 'https://avinashparagoudar@github.com':
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 2 threads
Compressing objects: 100% (1/1), done.
Writing objects: 100% (2/2), 286 bytes | 286.00 KiB/s, done.
Total 2 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/avinashparagoudar/hii.git
 * [new tag]          v1.0 -> v1.0
```

8. Advanced Git Operations

Write the command to cherry-pick a range of commits from "source-branch" to the current branch

First, ensure that you are currently on the branch where you want to apply the cherry-picked commits. Identify the range of commits you want to cherry-pick from the "source-branch". You can obtain the commit hashes or commit range using the git log command or other Git history visualization tools. To cherry-pick a range of commits from "source-branch" to the current branch, use the following command: **git cherry-pick <start-commit>..<end-commit>** Replace <start-commit> with the hash of the first commit in the range, and <end-commit> with the hash of the last commit in the range. For example, if you want to cherry-pick the commits with hashes abc123 to def456, the command would be: **git cherry-pick abc123..def456** This command applies the changes introduced by the specified range of commits to the current branch, effectively cherry-picking them.

Commands:

Create folder Git cherry picking

cd Git cherry picking

git init

vi alpha.txt

git add . | git commit -m "1st commit"

vi beta.txt

git add . | git commit -m "2st commit"

vi gamma.txt

git add . | git commit -m "3st commit"

git reflog //It shows all commit history

git add .

git status

git commit -m "all deleted"

git reflog

git cherry-pick id //----add commit history id eg:34946d4-----

git log // it show all commit history with commit id

```

student@student1:~$ mkdir Git_cherry_picking
student@student1:~$ cd Git_cherry_picking
student@student1:~/Git_cherry_picking$ git init
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /home/student/Git_cherry_picking/.git/
student@student1:~/Git_cherry_picking$ vi alpha.txt
student@student1:~/Git_cherry_picking$ git add alpha.txt
student@student1:~/Git_cherry_picking$ git commit -m "first commit"
[master (root-commit) eac2f67] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 alpha.txt
student@student1:~/Git_cherry_picking$ vi beta.txt
student@student1:~/Git_cherry_picking$ git add beta.txt
student@student1:~/Git_cherry_picking$ git commit -m "second commit"
[master cce0c33] second commit
 1 file changed, 1 insertion(+)
 create mode 100644 beta.txt
student@student1:~/Git_cherry_picking$ vi gamma.txt
student@student1:~/Git_cherry_picking$ git add gamma.txt
student@student1:~/Git_cherry_picking$ git commit -m "third commit"
[master 08c8647] third commit
 1 file changed, 1 insertion(+)
 create mode 100644 gamma.txt
student@student1:~/Git_cherry_picking$ git reflog
08c8647 (HEAD -> master) HEAD@{0}: commit: third commit
cce0c33 HEAD@{1}: commit: second commit
eac2f67 HEAD@{2}: commit (initial): first commit

```

```

student@student1:~/Git_cherry_picking$ git cherry-pick cce0c33
On branch master
You are currently cherry-picking commit cce0c33.
(all conflicts fixed: run "git cherry-pick --continue")
(use "git cherry-pick --skip" to skip this patch)
(use "git cherry-pick --abort" to cancel the cherry-pick operation)

nothing to commit, working tree clean
The previous cherry-pick is now empty, possibly due to conflict resolution.
If you wish to commit it anyway, use:

    git commit --allow-empty

otherwise, please use 'git cherry-pick --skip'

```

9. 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 in Git, including the author, date, and commit message, you can use the following command: **git show <commit ID>** Replace `**<commit ID>**` with the actual commit ID you want to view. This command will display the commit details, including the author's name, email, date, and the commit message.

To view the details of a specific commit in Git, you can use the **git show** command followed by the commit ID. Here's a step-by-step explanation of how to use Git to view the details of a commit: 1. Open your terminal or command prompt and navigate to the Git repository where the commit is located. 2. Obtain the commit ID of the specific commit you want to view. You can find the commit ID by using commands like **git log** or **git reflog**. The commit ID is a unique identifier for each commit in Git. 3. Once you have the commit ID, use the following command to view the details of that commit: **git show <commit ID>** Replace `**<commit ID>**` with the actual commit ID you want to view. 4. After executing the command, Git will display the details of the commit. This includes information such as the author's name, email, date of the commit, and the commit message. The commit message is a brief description of the changes made in that commit. It provides context and helps understand the purpose of the commit. By using the **git show** command with the appropriate commit ID, you can easily view the details of a specific commit in Git, including the author, date, and commit message.

git log // it show all commit history with commit id

git show commitId // paste commitid


```

student@student1:~/1sv221s004$ git log
commit 84e40af85b21bb5ec38581e6e17d7651c4b421d9 (HEAD -> master)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:58:01 2024 +0530

    Revert "DEMO FILE 1"

    This reverts commit 1717e759524b6c69d9549f58fe580127e7738545.

commit f57b5bf0582a00fb795cdd3b965eb2b9c7cc2f24 (origin/master)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:38:19 2024 +0530

    DEMO FILE 1

commit 8556b4a09a79b47f768ffd03a4b1bef773afce58 (tmp)
Author: avinashparagoudar <avinashparagoudar469@gmail.com>
Date: Sat Feb 3 12:56:42 2024 +0530

    done
student@student1:~/1sv221s004$ git show 8556b4a09a79b47f768ffd03a4b1bef773afce58
commit 8556b4a09a79b47f768ffd03a4b1bef773afce58 (tmp)
Author: avinashparagoudar <avinashparagoudar469@gmail.com>
Date: Sat Feb 3 12:56:42 2024 +0530

    done

diff --git a/e.txt b/e.txt
new file mode 100644
index 0000000..650a77a
--- /dev/null
+++ b/e.txt
@@ -0,0 +1 @@
+ahav

```

10. Analysing and Changing Git History

Write the command to list all commits made by the author "JohnDoe" between "2023-01-01" and "2023-12-31".

To list all commits made by the author "shamsiya parveen" between "2023-01-01" and "2023-12-31" in Git, you can use the following command:

git log --author="shamsiya parveen" --after="2023-01-01" --before="2023-12-31"

This command will display a list of commits made by "shamsiya parveen" within the specified date range.

git log: This is the command to view the commit history in Git.

--author="shamsiya parveen": This option filters the commit history based on the specified author name, in this case, "shamsiya parveen". Only commits made by this author will be displayed.

--after="2023-01-01": This option filters the commit history to show only the commits made after the specified date, which is "2023-01-01" in this case.

--before="2023-12-31": This option filters the commit history to show only the commits made before the specified date, which is "2023-12-31" in this case.

By combining all these options together, the command **git log --author="shamsiya parveen" --after="2023-01-01" --before="2023-12-31"** will display a list of commits made by "shamsiya parveen" between the dates "2023-01-01" and "2023-12-31".

command:

git log --author="shamsiya parveen " --after="2023-01-01" --before="2023-12-31"

```
student@student1:~/1sv22is004$ git log --author="avinashparagoudar" --after="2023-12-23" --before="2024-02-27"
commit 84e40af85b21bb5ec38581e6e17d7651c4b421d9 (HEAD -> master)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:58:01 2024 +0530

    Revert "DEMO FILE 1"

    This reverts commit 1717e759524b6c69d9549f58fe580127e7738545.

commit f57b5bf0582a00fb795cdd3b965eb2b9c7cc2f24 (origin/master)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:38:19 2024 +0530

    DEMO FILE 1

commit 8556b4a09a79b47f768ffd03a4b1bef773afce58 (tmp)
Author: avinashparagoudar <avinashparagoudar469@gmail.com>
Date: Sat Feb 3 12:56:42 2024 +0530

done
```

11. Analysing and Changing Git History

Write the command to display the last five commits in the repository's history.

To display the last five commits in the repository's history, you can use the following command:

git log -n 5

git log: This is the command to view the commit history in Git.

-n 5: This option limits the output to the specified number of commits, in this case, 5. It tells Git to display only the most recent 5 commits.

By running **git log -n 5**, you'll get a list of the last five commits made in the repository. Each commit will be displayed with its commit message, author, date, and a unique commit hash.

Command:

git log -n 5

```
student@student1:~/1sv221s004$ git log -n5
commit 84e40af85b21bb5ec38581e6e17d7651c4b421d9 (HEAD -> master)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:58:01 2024 +0530

    Revert "DEMO FILE 1"

    This reverts commit 1717e759524b6c69d9549f58fe580127e7738545.

commit f57b5bf0582a00fb795cdd3b965eb2b9c7cc2f24 (origin/master)
Author: avinashparagoudar469@gmail.com <avinashparagoudar469@gmail.com>
Date: Tue Feb 27 11:38:19 2024 +0530

    DEMO FILE 1

commit 8556b4a09a79b47f768ffd03a4b1bef773afce58 (tmp)
Author: avinashparagoudar <avinashparagoudar469@gmail.com>
Date: Sat Feb 3 12:56:42 2024 +0530

done
```

12. Analysing and Changing Git History

Write the command to undo the changes introduced by the commit with the ID "abc123".

To undo the changes introduced by the commit with the ID "abc123" in Git, you can use the **git revert** command. This command creates a new commit that undoes the changes made in the specified commit.

The command to undo the changes introduced by the commit with the ID "abc123" is:

textCopy code

git revert abc123

When you execute this command, Git will create a new commit that undoes the changes made in the "abc123" commit. The new commit will have a message indicating the revert and a unique commit ID. The git revert command is a safe way to undo changes since it does not rewrite the existing commit history. It adds a new commit that undoes the changes, which allows you to maintain a clear and accurate history of your project.

However, it's important to note that git revert is not the same as git reset. While git revert undoes a specific commit by creating a new commit, git reset allows you to move the branch pointer to a previous commit, effectively removing any commits after that point. **git reset** is a more powerful command and should be used with caution as it can permanently delete commits and history.

Commands:

Git revert “abc123”

Git log --oneline // it shows commit history in oneline

Git reset --hard head.1

```
student@student1:~/1sv221s004$ git revert 1717e75
[master 84e40af] Revert "DEMO FILE 1"
 1 file changed, 1 deletion(-)
 delete mode 100644 demo1.txt
student@student1:~/1sv221s004$ git log --oneline
84e40af (HEAD -> master) Revert "DEMO FILE 1"
f57b5bf (origin/master) DEMO FILE 1
8556b4a (tmp) done
student@student1:~/1sv221s004$ git reset --hard
HEAD is now at 84e40af Revert "DEMO FILE 1"
```