VISVESVARAYATECHNOLOGICALUNIVERSITY

"Jnana Sangama", Belagavi-560014, Karnataka

A PRACTICAL REPORT ON

"PROJECTMANAGEMENT WITHGIT"

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF

BACHELOR OF ENGINEERING

IN

INFORMATION SCIENCE & ENGINEERING

Submitted By

NAME: RAKESH.R

USN: 1SV22IS033

Under the guidance of

SHAMSIYA PARVEEN **B.E., MTech.,**

Assistant Professor, Dept. of ISE.

Department of Information Science and Engineering

SHRIDEVIINSTITUTEOFENGINEERINGANDTECHNOLOGY

(Affiliated To Visvesvaraya Technological University) Sira Road, Tumkuru – 572106, Karnataka.

2023-2024

Sri Shridevi Charitable Trust (R.)
SHRIDEVI INSTITUTE OF ENGINEERING & TECHNOLOGY
(Recognised by Govt. of Karnataka, Affiliated to VTU, Belagavi and Approved by AICTE, New Delhi)
Sira Road, Tumakuru - 572 106. Karnataka.
An ISO 9001:2015 Certified Institution
Phone: 0816-2212629 | Fax: 0816-2212628 | Email: info@shridevienginеering.org | Web: http://www.shridevienginеering.org

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

CERTIFICATE

This is to certify that, **PROJECT MANAGEMENT WITH GIT** Practical Report has been successfully carried out by **RAKESH.R [1SV22IS033]** in partial fulfilment for the Project Management With Git Lab of **Bachelor of Engineering** in **Information Science & Engineering** of the **Visvesvaraya Technological University, Belagavi** during the academic year **2023-24.** It is certified that all the corrections/suggestions indicated for internal assessments have been incorporated in the report. The Practical Report has been approved as it certifies the academic requirements in respect of Practical work prescribed for the Bachelor of Engineering Degree.

.

Signature of Guide

SHAMSIYA PARVEEN B.E., MTech.,

Assistant Professor, Dept. of ISE, SIET,

Tumkuru.

Signature of H.O.D

Dr. REKHA H MTech., Ph.D.,

Professor & HOD Dept. Of ISE, SIET, Tumkuru.

Sri Shridevi Charitable Trust (R.)
SHRIDEVINSTITUTE OF ENGINEERING & TECHNOLOGY
(Recognised by Govt. of Karnataka, Affiliated to VTU, Belagavi and Approved by AICTE, New Delhi)
Sira Road, Tumakuru - 572 106. Karnataka.
An ISO 9001:2015 Certified Institution
Phone: 0816-2212629 | Fax: 0816-2212628 | Email: info@shridevengineering.org | Web: http://www.shridevengineering.org

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

DECLARATION

I,**RAKESH.R ,USN [1SV22ISO33**], student of III semester B.E in Information Science & Engineering, at Shridevi Institute of Engineering & Technology, Tumkuru, hereby declare that, the Practical Report of **" PROJECT MANAGEMENT WITH GIT"** embodies the work carried out under the guidance of **SHAMSIYA PARVEEN, Assistant Professor Department of ISE, SIET**, Tumkuru as partial fulfilment of requirements for the Project Management With Git Lab in **Bachelor of Engineering in Information Science & Engineering of Visvesvaraya Technological University, Belagavi,** during the academic year **2023-24**. The Practical Report has been approved as it satisfies the academic requirements in respect to the Practical Report work

Place: Tumkur

_____
Student Name & Signature
RAKESH.R[1SV22IS033]

Date**:** _____

DEPARTMENT OF INFORMATION SCIENCE AND ENGINEERING

PARTICULARS OF THE EXPERIMENTS PERFORMED
CONTENTS

EXP NO	DATE	TITLE OF THEEXPT	CONDUCTION (20M)	VIVA (10M)	TOTAL (30M)	SIGN
1	24/11/23	Setting up and basic commands				
2	01/12/24	Creating and Managing branches				
3	09/12/23	Creating and Managing branches				
4	16/12/23	Collaboration and Remote Repositories				
5	23/12/23	Collaboration and Remote repositories				
6	06/01/24	Collaboration and Remote repositories				
7	20/01/24	Git Tags and Releases				
8	27/01/24	Advanced Git Operations				
9	03/02/24	Analysing and changing Git History				
10	12/02/04	Analysing and changing Git History				
11	24/02/24	Analysing and changing Git History				
12	02/03/24	Analysing and changing Git History				
AVERAGE						

SIGNATURE OF COURSE INSTRUCTOR

1. _____
2. _____

EXPERIMENT -01

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.

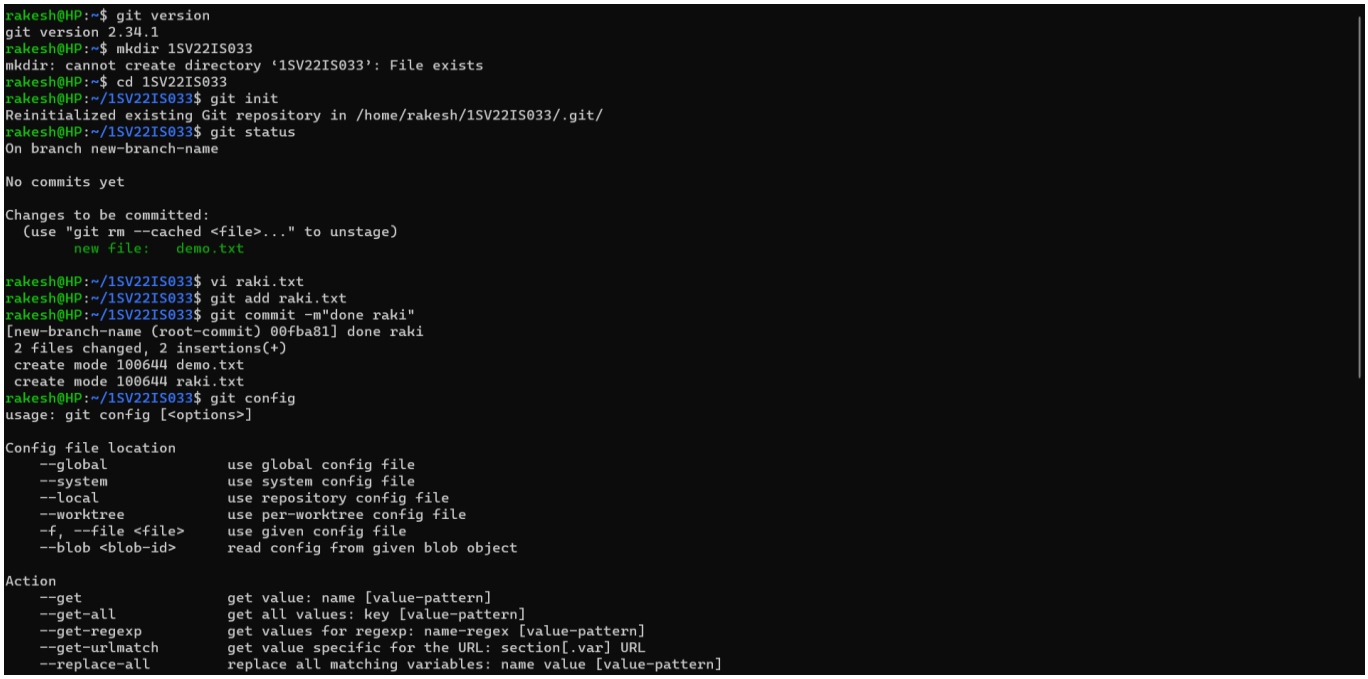
GIT COMMANDS:

- **git –version:** This command to check the version
- **$ mkdir 1SV22IS033:** mkdir (make directory), here new directory is created which is named as 1SV22IS033.

- **$ vi raki.txt:** this command is used to open a new file.
- **$ git –version:** this command is used to check whether git package is installed and also to know the version.
- **$ git init:** to initialize a new git repository int the current directory. When you run this command in a directory, It creates a new subdirectory named '. git' that contains all of the necessary metadata for the repository. This '. git' directory is where Git stores information about the repository's configuration, commits, branches and more. We can start adding the files, making commits, and managing our version-controlled project using Git.
- **$ git status:** It's a fundamental Git command used to display the state of working directory and the staging area. When you run 'git status', Git will show you
  - Which files are staged for commit in the staging area.
  - Which files are modified but not yet staged.
  - Which files are untracked
  - Information about the current branch, such as whether your branch is ahead or behind its remote counterpart.
  - This command is extremely useful for understanding what changes have been made and what actions need to be taken before pushing changes to a remote repository like GitHub. It helps us to manage our repository effectively and keep track of our progress.
- **$ git add raki.txt:** command is used to stage changes made to the specified file named raki.txt for the next commit in your Git repository. When you make changes to files in your working directory, Git initially considers them as modified but not yet staged for commit. By running git add filename.txt, you inform Git that you want to include the changes in filename.txt

in the next commit. This action moves the changes to the staging area, preparing them to be committed.

- **$ git commit –m “commit message”:** command is used to commit staged changes to your Git repository along with a commit message provided inline using the -m flag. After running the git commit command, Git will create a new commit with the staged changes and associate the provided commit message with it. This helps maintain a clear history of changes in your Git repository.



- **$ git config --global user.name “your username”:** command is used to set or update the global Git username configuration on your system. This command is typically used once to configure your username globally, so you don't have to specify it every time you make a commit.
  Replace "Your Username" with your actual Git username. For example: $ git config –global user.name “Rakesh. R”.
  By setting your username globally, Git will use this username for all repositories on your system unless overridden by a local

configuration specific to a particular repository. This helps identify who made each commit in the repository's history

- **$ git config - -global user. email “youremail@example.com”:** command is used to set or update the global Git email configuration on your system.
  This command is typically used once to configure your email address globally, so you don't have to specify it every time you make a commit.
  Replace "your_email@example.com" with your actual email address. For example: $ git config --global user. Email “rakisiet33@gmail.com”.
  By setting your email address globally, Git will use this email for all repositories on your system unless overridden by a local configuration specific to a particular repository.
  This helps associate your commits with your email address, providing contact information for collaborators and maintaining a clear history of changes.

EXPERIMENT-02

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 featurebranch** //This will create new branch called feature-branch
- **git checkout featurebranch**

//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.

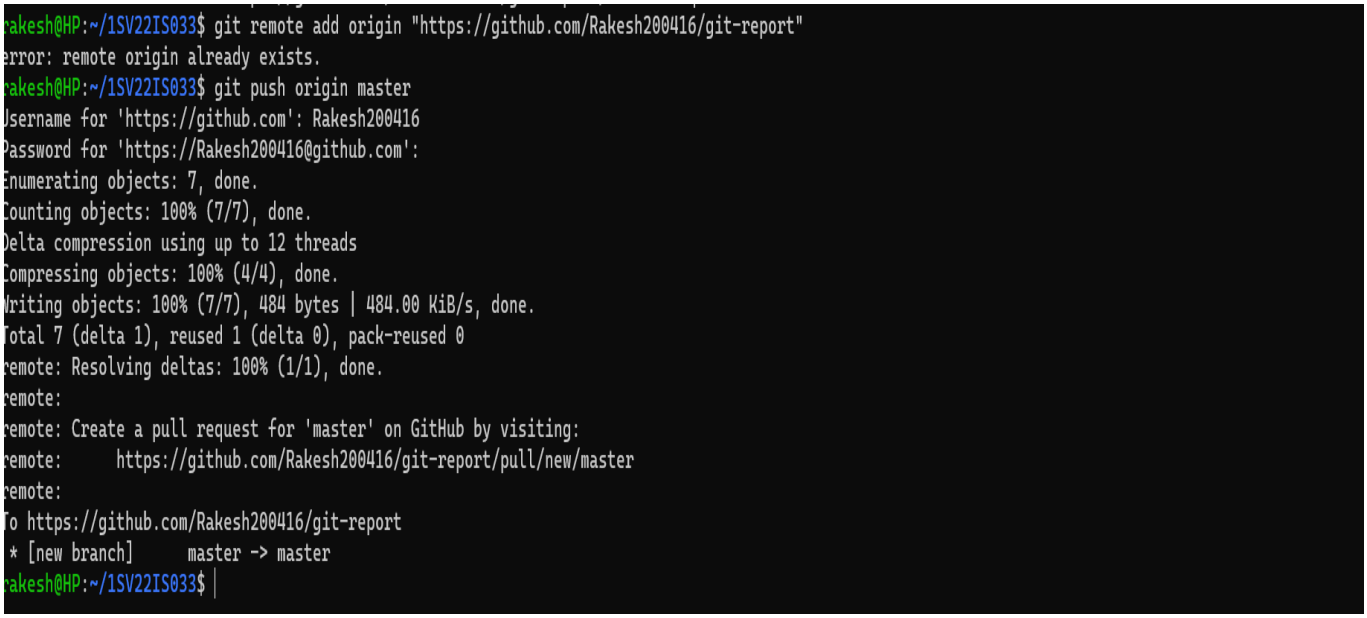
- **git status** //it will check the status
- **git push origin master**

- **$ git branch featurebranch:** command is used to create a new branch named featurebranch in your Git repository. After

running this command, you'll have a new branch based on your current branch's state. This command will create a new branch named feature-branch at your current commit. However, it won't switch you to that branch automatically. To start working on the new branch, you need to check it out using git checkout or git switch.

- **$ git checkout featurebranch:** This command will switch your working directory to the feature-branch branch.
- **$ git checkout master:** used to switch to the master branch in your Git repository. When you run this command, Git updates your working directory to reflect the state of the master branch. This means that any changes you make or files you create or modify will be based on the master branch. After running this command, you'll be on the master branch, and you can start working on it, making changes, creating
- **$ git merge feature-branch:** command is used to merge changes from the specified branch (in this case, feature-branch) into the current branch. Typically, you execute this command while you're on the branch where you want to merge the changes. This command will incorporate the changes from featurebranch into the branch you're currently on. After successfully merging feature-branch into master, you'll have all the changes from feature-branch incorporated into master, and you can continue working on master with the merged changes.

- **$ git push origin master:** used to push the commits from your local master branch to the remote repository named origin. This is a common command used to update the remote repository with the changes you've made locally. Here's a breakdown of what each part of the command does:
  - git push: This is the Git command used to push commits from your local repository to a remote repository.
  - origin: This refers to the name of the remote repository you're pushing to. In Git terminology, "origin" is a common name used to refer to the default remote repository.
  - master: This is the name of the local branch you're pushing. In many Git repositories, master is the default name for the main branch.



EXPERIMENT-03

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. 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.

- **$ git stash:** command is used to temporarily save changes in your working directory and staging area so that you can work on something else or switch branches without committing them. When you run git stash, Git will save your changes into a stack of stashes, leaving your working directory and staging area clean. You can then switch branches or perform other operations without worrying about the changes you've stashed.

- **$ git stash apply:** used to retrieve and reapply the most recent stash from the stash stack onto your current working directory. This command will reapply the changes from the stash onto your working directory without removing the stash from the stack.

- **$ git stash list:** used to display the list of stashes in your Git repository's stash stack. It shows all the stashes you've created, along with a reference for each stash. When you run this command, Git will list all the stashes you've created in the repository. Each stash will be listed along with a reference, typically in the format stash@{n}, where n is the index of the stash in the stash stack.

- **$ git stash pop**: is a Git command used to apply and remove the most recently stashed changes from the stash stack. It's a combination of two actions: applying the changes and then dropping the stash. The command is helpful when you want to reapply your saved changes and clean up the stash at the same time.

```
rakesh@HP:~/1SV22IS033$ git branch raki
fatal: A branch named 'raki' already exists.
rakesh@HP:~/1SV22IS033$ git status
On branch master
nothing to commit, working tree clean
rakesh@HP:~/1SV22IS033$ vi sample.txt
rakesh@HP:~/1SV22IS033$ git add sample.txt
rakesh@HP:~/1SV22IS033$ git checkout raki
A       sample.txt
Switched to branch 'raki'
rakesh@HP:~/1SV22IS033$ git stash
Saved working directory and index state WIP on raki: bf24c52 done raki2
rakesh@HP:~/1SV22IS033$ git stash list
stash@{0}: WIP on raki: bf24c52 done raki2
rakesh@HP:~/1SV22IS033$ git stash apply
On branch raki
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   sample.txt

rakesh@HP:~/1SV22IS033$ git stash pop
On branch raki
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   sample.txt

Dropped refs/stash@{0} (2713535ac0a7d967c306ae321b642bb4c8755fcc)
```

EXPERIMENT-04

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**.

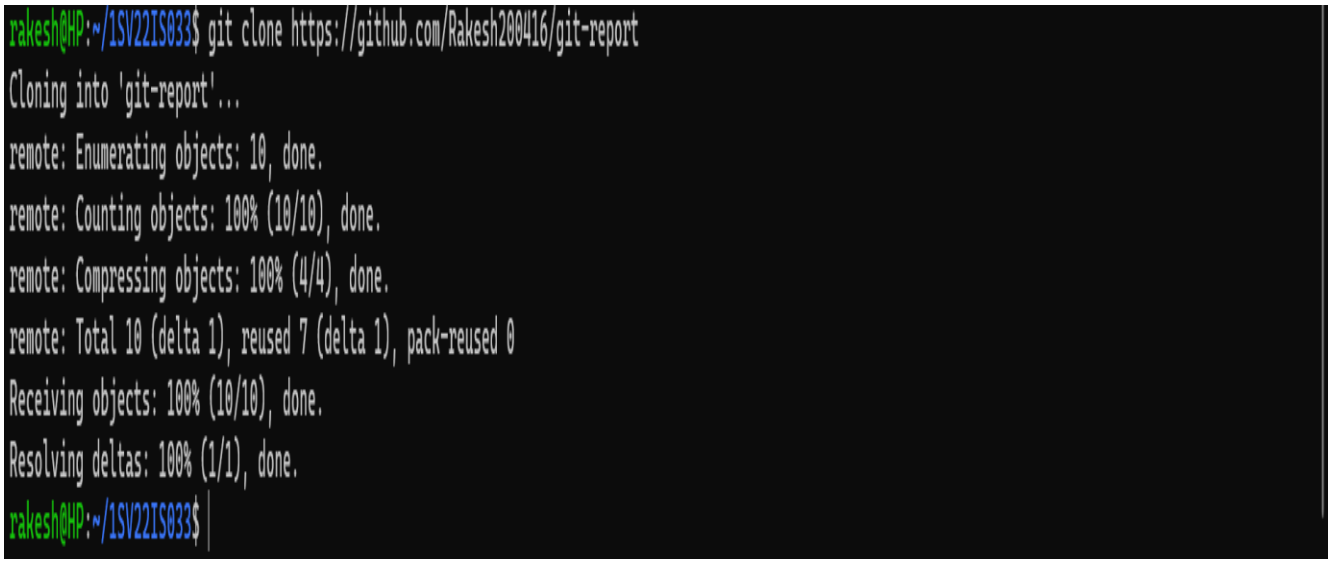
To clone the remote repository, use the following command: **git clone <repository-url> Replace <repository-url>** with the URL of the remote repository you obtained in the previous step. This command will create a copy of the remote repository on your local machine. Git will start downloading the remote 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.

- **$ git clone “repository URL”:** The git clone command is used to create a copy of an existing Git repository in a new directory. This is useful when you want to start working on a project that already exists in a remote repository, such as on GitHub or GitLab. Repository URL is the URL of the remote repository you want to clone. After cloning the repository, you'll have a complete copy of the project's history and files on your local machine. You can then make changes, create commits, and push them back to the remote repository as needed.



EXPERIMENT -05

5. Collaboration and Remote Repositories

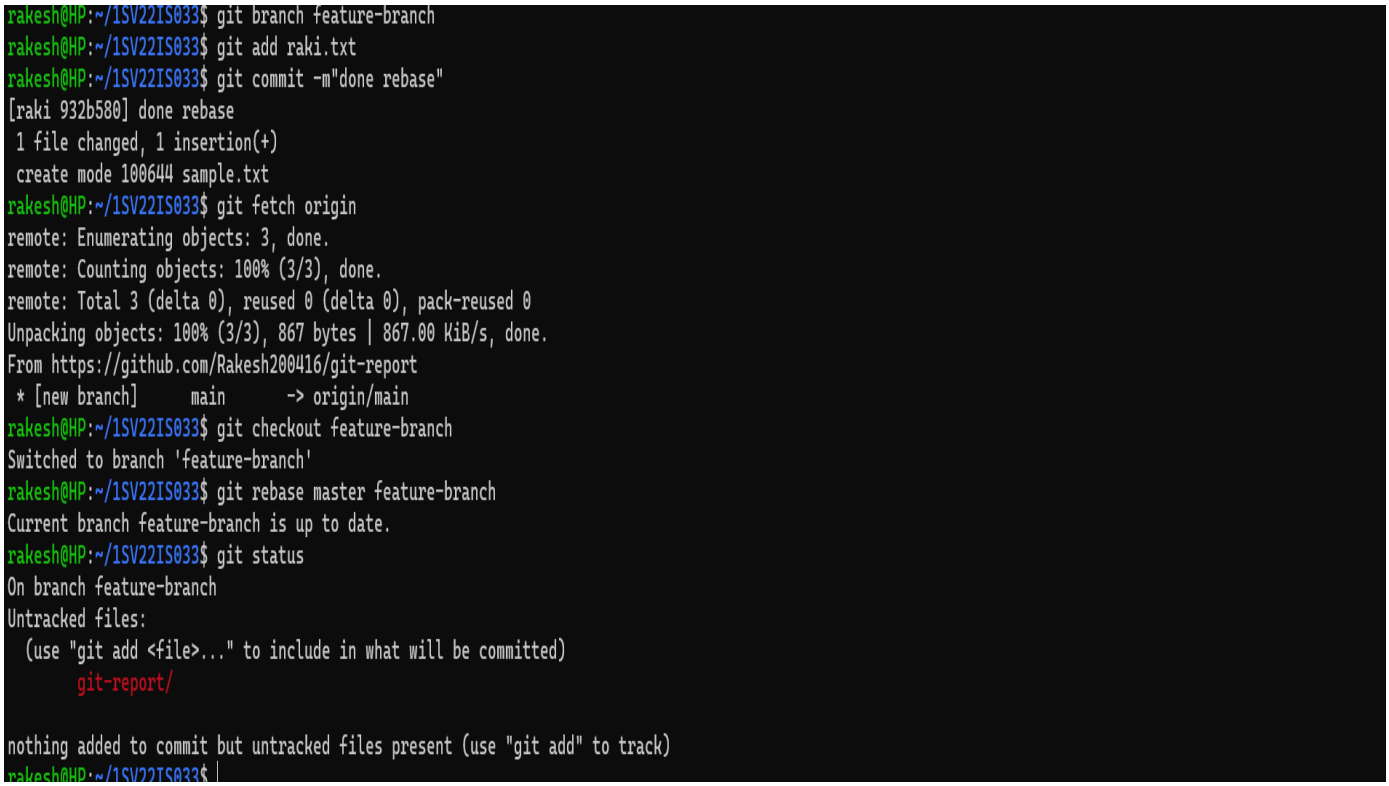
Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

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.

- **$git fetch origin:** is used to update your local repository with changes from the remote repository named "origin" without automatically merging them into your working directory.
- **$ git rebase master/origin feature-branch:** for rebasing, for apply git checkout master command to switch from feature branch to master branch and then apply git commit command and commit a message.

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.



EXPERIMENT -06

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.

- **$ git merge feature-branch:** command is used to merge changes from the specified branch (in this case, feature-branch) into the current branch. Typically, you execute this command while you're on the branch where you want to merge the changes. This command will incorporate the changes from featurebranch into the branch you're currently on. After successfully merging feature-branch into master, you'll have all the changes from feature-branch incorporated into master, and you can continue working on master with the merged changes.
- **$ git commit -m "custom commit message":** This command will commit a message that the custom commit message.

```
rakesh@HP:~/1SV22IS033$ git merge feature-branch
Already up to date.
rakesh@HP:~/1SV22IS033$ git commit -m"custom commit message rakesh"
On branch feature-branch
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        git-report/

nothing added to commit but untracked files present (use "git add" to track)
rakesh@HP:~/1SV22IS033$
```

EXPERIMENT -07

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.

- **$ git tag v1.0:** used to create a lightweight tag in your Git repository. Tags are used to mark specific points in history, such as releases or significant milestones.

After running this command, the tag v1.0 will be created at the current commit. This tag can then be used as a reference point in your repository's history.

- **$ git tag:** if you run the git tag command without any arguments, it will list all the tags in your Git repository. This command is useful for viewing the existing tags in your repository. Tags provide a way to mark specific commits in your repository's history, making it easier to reference them later.

They're commonly used to mark releases, so you can easily find the commit associated with a particular version of your software.

- **$ git tag –a v1.1 –m "tag to release":** This command creates an annotated tag named v1.1 with the message "tag to release". Annotated tags include additional metadata such as the tagger's name, email, and the date the tag was created. The message provides additional context or information about the tag.

After running this command, the tag v1.1 will be created at the current commit, and you can use it as a reference point in your repository's history.

- **$ git show v1.0:** The git show command is used to display information about commits, tags, or other objects in your Git repository.

When you run git show followed by a tag name, it will display information about the specified tag. When you run this command, Git will display detailed information about the tag v1.0, including the

commit it points to, the tagger information (if it's an annotated tag), and the commit message associated with the tagged commit. If v1.0 is an annotated tag, the output will also include any additional metadata and the tag message. If it's a lightweight tag, the output will be similar to git show for a commit. This command is useful for reviewing the details of a specific tag in your repository, such as when it was created and what changes it represents.

- **$ git tag –l "v1. *":** command is used to list all tags that match the specified pattern. In this case, the pattern "v1. *" is a regular expression pattern that matches tags starting with v1. followed by any characters (represented by *). When you run this command, Git will list all tags in your repository that match the pattern "v1. *". This means it will list tags like v1.0, v1.1, v1.2, etc., but not tags like v2.0 or release v1.0. This command is useful when you want to filter and list specific tags based on a pattern or criteria. It allows you to easily find tags that match a certain versioning pattern or naming convention in your repository
- **$git push origin v1.0:** command is used to push the local branch named "v1.0" to the remote repository called "origin." This is commonly done when you want to update the remote repository with the changes made in your local branch.

```
rakesh@HP:~/1SV22IS033$ git checkout master
Switched to branch 'master'
rakesh@HP:~/1SV22IS033$ git tag v1.0
rakesh@HP:~/1SV22IS033$ git tag
v1.0
rakesh@HP:~/1SV22IS033$ git tag -a v1.1 -m "tag for release"
rakesh@HP:~/1SV22IS033$ git tag
v1.0
v1.1
rakesh@HP:~/1SV22IS033$ git show v1.0
commit bf24c5213b290fa798d37f2c308ccdf788d10cb4 (HEAD -> master, tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 18:40:55 2024 +0530

    done raki2

diff --git a/raki2.txt b/raki2.txt
new file mode 100644
index 0000000..5b74664
--- /dev/null
+++ b/raki2.txt
@@ -0,0 +1 @@
+haiii
rakesh@HP:~/1SV22IS033$ git tag -l"v1.*"
error: switch `v' is incompatible with --list
rakesh@HP:~/1SV22IS033$ git push origin v1.0
Username for 'https://github.com': Rakesh200416
Password for 'https://Rakesh200416@github.com':
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Rakesh200416/git-report
 * [new tag]         v1.0 -> v1.0
rakesh@HP:~/1SV22IS033$
```

EXPERIMENT -08

8.ADVANCED GIT OPERATION0S:

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.

- Cherry picking is a very strong function that chooses any commit in the history and implements its feature to the current master.
- **$ git reflog:** command used to view the history of operations performed on a repository's Git references (like branches and HEAD). It stands for "reference log". The reference log keeps track of when the tips of branches and other references were updated in the repository. When you run git reflog, Git shows you a chronological list of recent actions, such as commits, checkouts, merges, and resets. This command is particularly

useful for recovering lost commits or branches, as it provides a record of recent changes even if they are not directly accessible via the current branch history or git log.

- **$ git cherry-pick commit id:** used to apply a specific commit (identified by its commit ID) onto the current branch. This is often used to selectively pick a commit from one branch and apply it to another branch. Replace with the actual commit hash of the commit you want to cherry-pick. Git will create a new commit on the current branch with the same changes as the original commit
- **$ git log:** command is used to display the commit history of the current branch. When executed, it shows a list of commits, starting with the most recent one. Each entry includes information such as the commit hash, author, date, and the commit message.

```
rakesh@HP:~/1SV22IS033$ vi alpha.txt
rakesh@HP:~/1SV22IS033$ git add alpha.txt
rakesh@HP:~/1SV22IS033$ git commit -m"1st raki"
[master 2a14744] 1st raki
 1 file changed, 1 insertion(+)
 create mode 100644 alpha.txt
rakesh@HP:~/1SV22IS033$ vi beta.txt
rakesh@HP:~/1SV22IS033$ git add beta.txt
rakesh@HP:~/1SV22IS033$ git commit -m"2nd raki"
[master bb8f72b] 2nd raki
 1 file changed, 1 insertion(+)
 create mode 100644 beta.txt
rakesh@HP:~/1SV22IS033$ vi gamma.txt
rakesh@HP:~/1SV22IS033$ git add gamma.txt
rakesh@HP:~/1SV22IS033$ git commit -m"3rd raki"
[master fa7247e] 3rd raki
 1 file changed, 1 insertion(+)
 create mode 100644 gamma.txt
rakesh@HP:~/1SV22IS033$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        git-report/

nothing added to commit but untracked files present (use "git add" to track)
rakesh@HP:~/1SV22IS033$ git commit -m"all deleted"
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        git-report/

nothing added to commit but untracked files present (use "git add" to track)
```

```
rakesh@HP:~/1SV22IS033$ git reflog
fa7247e (HEAD -> master) HEAD@{0}: commit: 3rd raki
bb8f72b HEAD@{1}: commit: 2nd raki
2a14744 HEAD@{2}: commit: 1st raki
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{3}: checkout: moving from feature-branch to master
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{4}: rebase: checkout feature-branch
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{5}: rebase: checkout feature-branch
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{6}: checkout: moving from raki to feature-branch
932b580 (raki) HEAD@{7}: commit: done rebase
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{8}: reset: moving to HEAD
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{9}: checkout: moving from master to raki
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{10}: checkout: moving from raki to master
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{11}: checkout: moving from master to raki
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{12}: merge featurebranch: Fast-forward
00fba81 (raki3, raki1, new-branch-name) HEAD@{13}: checkout: moving from featurebranch to master
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) HEAD@{14}: commit: done raki2
00fba81 (raki3, raki1, new-branch-name) HEAD@{15}: checkout: moving from master to featurebranch
00fba81 (raki3, raki1, new-branch-name) HEAD@{16}: checkout: moving from raki3 to master
00fba81 (raki3, raki1, new-branch-name) HEAD@{17}: checkout: moving from raki1 to raki3
00fba81 (raki3, raki1, new-branch-name) HEAD@{18}: checkout: moving from new-branch-name to raki1
00fba81 (raki3, raki1, new-branch-name) HEAD@{19}: commit (initial): done raki
rakesh@HP:~/1SV22IS033$ git cherry-pick fa7247e
On branch master
You are currently cherry-picking commit fa7247e.
  (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)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        git-report/

nothing added to commit but untracked files present (use "git add" to track)
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'
```

```
rakesh@HP:~/1SV22IS033$ git log
commit fa7247e9a282e7701b93c09759b792103b8e0888 (HEAD -> master)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 20:35:47 2024 +0530

    3rd raki

commit bb8f72b3db5eb1e029bb01410ed76cbd92b6a909
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 20:35:10 2024 +0530

    2nd raki

commit 2a147449ddfa156598e48226c1a2000faf2af4e5
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 20:34:21 2024 +0530

    1st raki

commit bf24c5213b290fa798d37f2c308ccdf788d10cb4 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 18:40:55 2024 +0530

    done raki2

commit 00fba815584e13fa6a73a58063fcccc7dc35032c (raki3, raki1, new-branch-name)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 18:14:52 2024 +0530

    done raki
```

EXPERIMENT-09

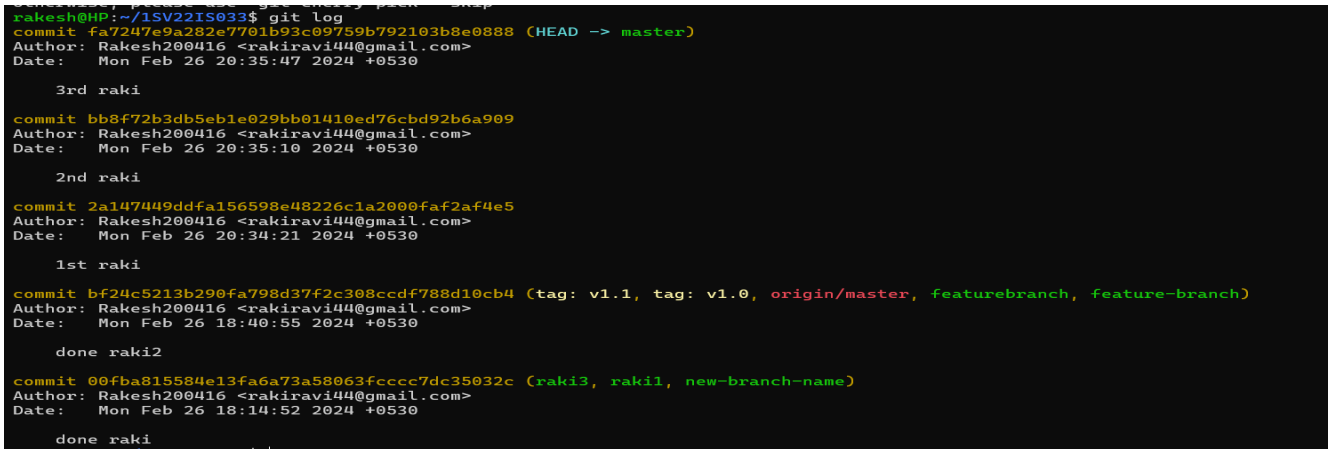
9. Analysing and Changing Git History:

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 :** The git log command is used to display the commit history of the current branch in your Git repository. By default, it shows the commits starting from the most recent one and goes backward. When you run this command, Git will display a list of commits in your repository, showing information such as the commit hash, author, date, and commit message for each commit.
- **$ git show "commit id" :** To show detailed information about a specific commit identified by its commit ID (or hash), you would use the git show command followed by the commit ID. Replace with the actual commit ID you want to display information about.
- This command will display detailed information about the commit with the specified commit ID, including the commit message, author, date, and the changes introduced by the commit.



EXPERIMENT -10

10. Analysing and Changing Git History:

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

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

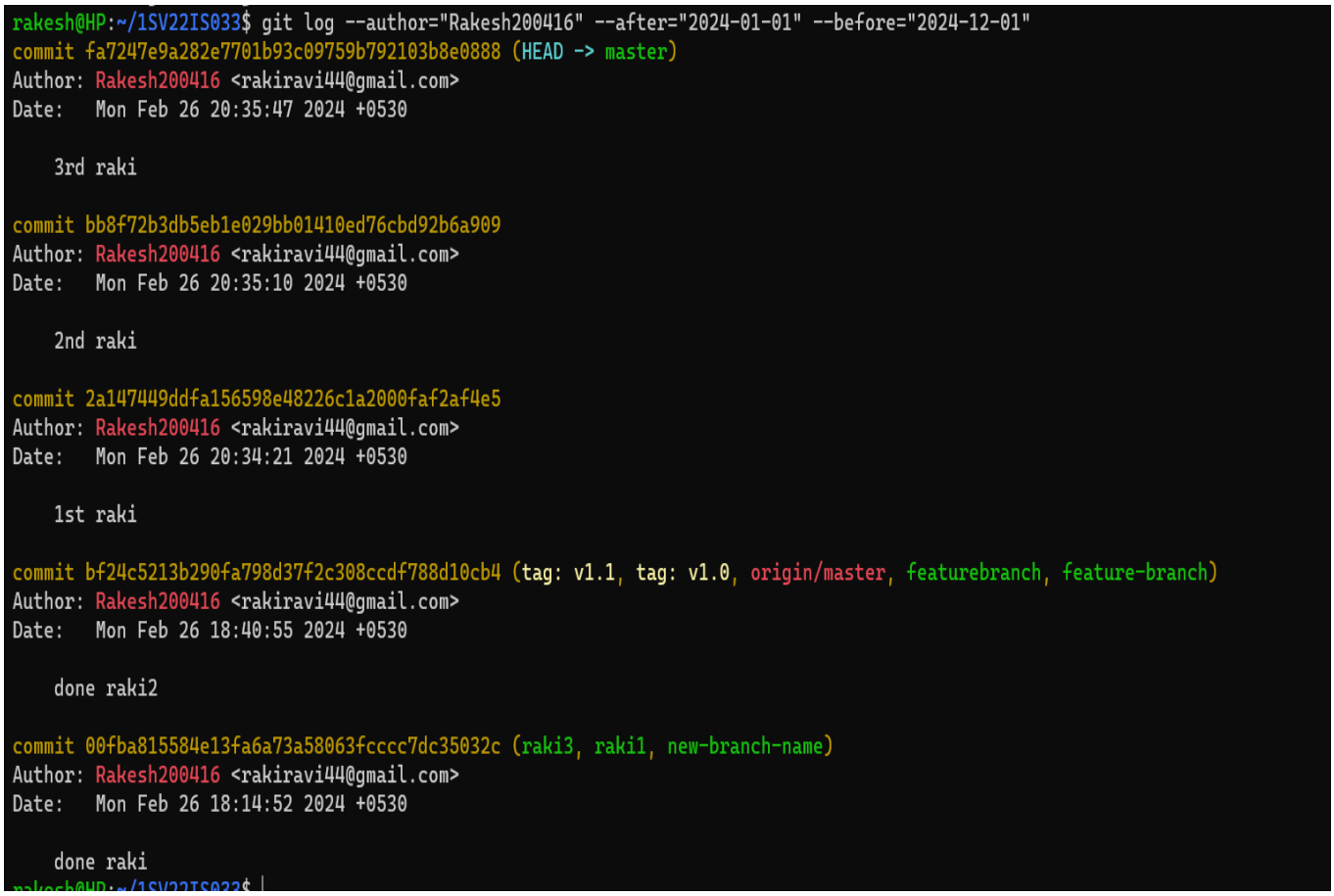
git log --author="Rakesh. R" --after="2023-01-01" --before="2023-12-31"

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

git log: This is the command to view the commit history in Git.
--author=" Rakesh.R ": This option filters the commit history based on the specified author name, in this case, " Rakesh.R ". 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=" Rakesh.R " --after="2023-01-01" --before="2023-12-31"** will display a list of commits made by " Rakesh.R " between the dates "2023-01-01" and "2023-12-31".

- **$ git log --author= “name” --after = “yyyy-mm-dd” --before = “yyyy-mm-dd”:** To filter the commit log by author and date range using the git log command, you can combine the --author, --after, and --before options. Replace "name" with the author's

name, "yyyy-mm-dd" with the desired dates, and adjust the date format accordingly. Remember to enclose the author's name in quotes if it contains spaces or special characters. If you want to search for commits by multiple authors, you can use --author multiple times, or you can use a regular expression to match authors' names.



EXPERIMENT -11

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: **it 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.

- **$ git log -n 5:** This command is used to display the last 5 commits in your repository's commit history. It limits the output to the specified number of commits, in this case, 5. When you run this command, Git will display the information for the last 5 commits in your repository, starting from the most recent commit and going backward in time. This command is useful when you want to quickly view the most recent commits in your repository, especially if you're only interested in a specific number of commits.

```
rakesh@HP:~/1SV22IS033$ git log -n 5
commit fa7247e9a282e7701b93c09759b792103b8e0888 (HEAD -> master)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 20:35:47 2024 +0530

    3rd raki

commit bb8f72b3db5eb1e029bb01410ed76cbd92b6a909
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 20:35:10 2024 +0530

    2nd raki

commit 2a147449ddfa156598e48226c1a2000faf2af4e5
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 20:34:21 2024 +0530

    1st raki

commit bf24c5213b290fa798d37f2c308ccdf788d10cb4 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 18:40:55 2024 +0530

    done raki2

commit 00fba815584e13fa6a73a58063fcccc7dc35032c (raki3, raki1, new-branch-name)
Author: Rakesh200416 <rakiravi44@gmail.com>
Date:   Mon Feb 26 18:14:52 2024 +0530

    done raki
rakesh@HP:~/1SV22IS033$
```

EXPERIMENT -12

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 theID"abc123"is:textCopycodegitrevertabc**123**
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.

- **$ git revert "commit id"**: The git revert command is used to create a new commit that undoes the changes made by a specific commit or range of commits.
- However, the -m option you've provided is used to specify the mainline parent number when reverting a merge commit,

which isn't applicable when reverting a regular commit. Replace with the commit ID of the commit you want to revert.

- However, it's important to note that -m is used for merge commits and doesn't apply to regular commits. After running git revert, Git will create a new commit that contains the changes to undo the specified commit. This approach allows you to keep a clean history while reverting changes in a controlled manner.
- **git log –oneline:** command displays a condensed version of the commit history, showing each commit as a single line with only its abbreviated commit hash and commit message. This is useful for a more concise overview of the commit history.
- **git reset --hard HEAD^:** command resets your working directory, staging area, and the current branch to the previous commit (one commit before the current HEAD). This effectively discards any changes made in the most recent commit.

```
rakesh@HP:~/1SV22IS033$ git revert "bf24c5213b290fa798d37f2c308ccdf788d10cb4"
[master 8ada0db] Revert "done raki2"
 Date: Mon Feb 26 20:35:47 2024 +0530
 1 file changed, 1 deletion(-)
 delete mode 100644 raki2.txt
```

```
rakesh@HP:~/1SV22IS033$ git log --oneline
8ada0db (HEAD -> master) Revert "done raki2"
fa7247e 3rd raki
bb8f72b 2nd raki
2a14744 1st raki
bf24c52 (tag: v1.1, tag: v1.0, origin/master, featurebranch, feature-branch) done raki2
00fba81 (raki3, raki1, new-branch-name) done raki
rakesh@HP:~/1SV22IS033$ git reset --hard HEAD
HEAD is now at 8ada0db Revert "done raki2"
rakesh@HP:~/1SV22IS033$
```