

Exp 1

Setting Up and Basic Commands

Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Initialize a new Git repository

- Open Git Bash: Open the Git Bash terminal.
- Navigate to the desired directory: Use the `cd` command followed by the directory path to navigate to the location where Git repository has to be created.

```
cd path/to/directory
```

- Initialize the Git repository:

```
git init
```

Create a new file

- Create a new file named filename.txt using the GitBash Terminal

```
touch filename.txt
```

Add the file to the staging area:

- Use the **git add** command to add the newly created file to the staging area. In the terminal, run the following command:

```
git add filename.txt
```

Commit the changes:

- Commit the changes to the repository with an appropriate commit message using the `git commit` command:

```
git commit -m "Initial commit: Add filename.txt"
```

Observation:

Upon completion of the steps, a Git repository will have been successfully initialized, a new file created, added to the staging area, and the changes committed with an appropriate commit message.

Result:

`git log --oneline` → screenshot of the result

Exp2

Creating and Managing Branches

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

- **Initialize a new Git repository**

```
git init
```

- **Create and commit a new file in the "feature-branch":**

```
git checkout -b feature-branch # Create and switch to the feature-branch
```

```
touch new_feature_file.txt # Create a new file
```

```
git add new_feature_file.txt # Stage the new file for commit
```

```
git commit -m "Add new feature file" # Commit the new file
```

- **Switch to the "master" branch and create and commit another file:**

```
git checkout master # Switch back to the master branch
```

```
git ls-files #Inspecting files in the master branch before merge
```

```
touch new_master_file.txt # Create another new file
```

```
git add new_master_file.txt # Stage the new file for commit
```

```
git commit -m "Add new master file" # Commit the new file
```

- **Merge the changes from the "feature-branch" into "master":**

```
git merge feature-branch # Merge the feature-branch into master
```

```
git ls-files #Inspecting files in the master branch after merge
```

Observation:

The experiment efficiently demonstrated Git branching and merging workflows, allowing for the isolation and integration of feature developments. Before the merge, the files present in each branch were inspected using `git ls-files`, revealing the file structure of both the "feature-branch" and "master" branches. Subsequently, after the merge process, another inspection using `git ls-files` showcased the combined file list, demonstrating successful integration of changes from the "feature-branch" into the "master" branch.

Results

screenshot- git ls-files in the master branch after and before the merge

Creating and Managing Branches

Write the commands to stash your changes, switch branches, and then apply the stashed changes.

Stashing is a powerful tool within Git that enables users to temporarily save changes without an immediate commitment. This functionality proves particularly beneficial when individuals need to switch branches, yet possess uncommitted changes in their working directory that they are unwilling to lose or commit at that moment.

- **Create a feature branch and switch to it:**

`git checkout -b feature-branch` #Creates a new branch, 'feature-branch' and switches to it.

- **Add a text file to the branch:**

`echo "This is a sample text file" > sample.txt`

`git add sample.txt` #adds a text file named sample.txt to the branch

- **Stash the changes:**

`git stash save "Stashing changes for now"` # stashes the changes made to sample.txt.

- **Switch to the master branch:**

`git checkout master` #switches back to the master branch.

- **Return to the feature branch:**

`git checkout feature-branch` #switches back to the feature-branch where changes were stashed.

- **Apply the stashed changes:**

`git stash apply` # applies the stashed changes (in this case, the changes made to sample.txt) to the current branch (feature-branch).

Observation:

Initially, a feature branch was created, changes were made by adding a text file, and then those changes were stashed. Upon switching to the master branch, it was observed that the changes made in the feature branch were not present, demonstrating the isolation of branches in Git. Upon returning to the feature branch and applying the stashed changes, the changes made earlier were successfully reapplied, indicating the effectiveness of Git's stash functionality in temporarily storing work. This experiment highlights the flexibility of Git for managing multiple branches and the usefulness of features like stashing for organizing work effectively. Additionally, it emphasizes the importance of understanding Git workflows for smooth collaboration and version control in software development projects.

Results: no need of results

Exp 4

Collaboration and Remote Repositories

Clone a remote Git repository to your local machine.

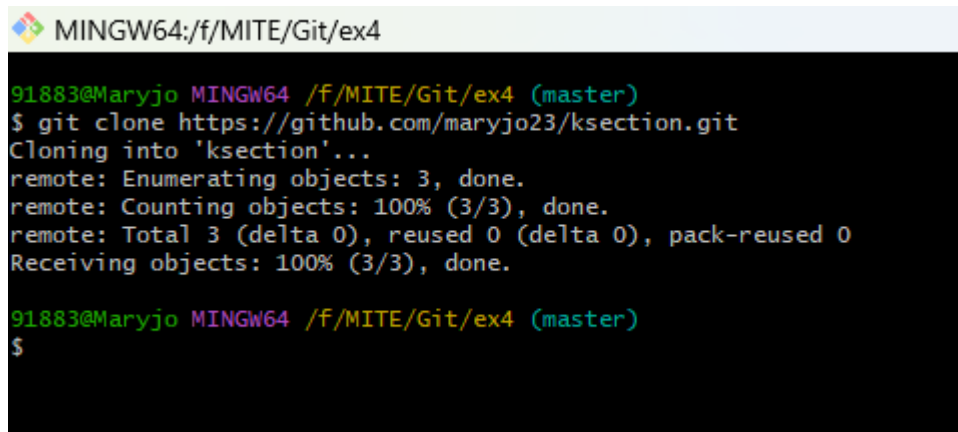
Cloning a remote Git repository to a local machine constitutes a fundamental operation in version control and collaboration workflows. When a repository is cloned, it generates a complete replica of the remote repository on the local system, encompassing all files, commit history, and branches. This process empowers individuals to work on the project locally, effect changes, and contribute to the codebase without impacting the original repository until they are prepared to push their changes.

```
git clone <repository-url> <directory-name>
```

Observation:

After executing the git clone command, the remote Git repository was successfully cloned to the local machine, providing the user with a complete copy of the project's files and commit history. This process established a direct link between the local and remote repositories, enabling the user to work on the project independently while maintaining synchronization with the central codebase. The cloned repository retained all branches and commit history, allowing the user to explore the project's development history and switch between branches as needed. Overall, the successful cloning operation facilitates collaboration and version control, empowering the user to contribute to the project's development seamlessly.

Result: (sample screenshot)

A screenshot of a terminal window with a dark background. The title bar at the top reads 'MINGW64:/f/MITE/Git/ex4'. The terminal shows a user prompt '91883@Maryjo MINGW64 /f/MITE/Git/ex4 (master)' followed by the command '\$ git clone https://github.com/maryjo23/ksection.git'. The output shows the cloning process: 'Cloning into 'ksection'...', 'remote: Enumerating objects: 3, done.', 'remote: Counting objects: 100% (3/3), done.', 'remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0', and 'Receiving objects: 100% (3/3), done.'. The terminal ends with the same user prompt and a '\$' prompt.

```
91883@Maryjo MINGW64 /f/MITE/Git/ex4 (master)
$ git clone https://github.com/maryjo23/ksection.git
Cloning into 'ksection'...
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.
91883@Maryjo MINGW64 /f/MITE/Git/ex4 (master)
$
```

Exp 5

Collaboration and Remote Repositories

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

The rebase command in Git functions by updating a branch with changes from another branch while preserving a linear and tidy history. Instead of generating cluttered merge commits, rebase applies an individual's commits on top of the other branch's modifications, presenting the work seamlessly.

integrated into that branch's history. This approach is beneficial for organizing project history in a straightforward manner. However, caution is warranted in its application, as conflicts may arise that require manual resolution. Overall, rebase serves as a valuable tool for enhancing the efficiency of Git workflows and maintaining a polished commit history.

- **Push a text file to the remote from the local:**

```
echo "Text content" > local_file.txt
git add local_file.txt
git commit -m "Add local_file.txt"
git remote add origin URL          # adding the remote repository
git push origin <branch-name>      # push commits to the remote repository
```

- **Add another text file directly to the remote repository:**

Add another file s via the repository's web interface with 'add file' option.

- **Fetch changes to the master branch and rebase:**

```
git checkout master          # Switch to the local master branch
git log --oneline            # commit history in one line
git fetch                    # Fetch the latest changes from the remote repository
git rebase origin/master     # Rebase the local master branch onto the updated remote master branch
git log --oneline            # commit history in one line
git fetch                    # No report implies there are no changes
```

Observation:

The provided sequence of Git commands involves updating the local master branch to reflect the latest changes from the remote repository's master branch and ensuring that any local modifications are properly integrated. Initially, the user switches to the local master branch and examines its commit history. Subsequently, they fetch the latest changes from the remote repository to synchronize the local repository's information with the central codebase. Then, a rebase operation is performed to incorporate the fetched changes from the remote master branch into the local branch, maintaining a linear commit history. After the rebase, the user reviews the updated commit history of the local master branch. Finally, another fetch operation is executed to check for any additional changes from the remote repository, with the absence of a report indicating no new changes fetched. Overall, these actions ensure that the local master branch is up to date with the remote repository, facilitating collaborative development and maintaining a coherent history of changes

Result:

Screenshot

6 Collaboration and Remote Repositories

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

The merge feature in Git allows developers to combine changes from different branches into a single branch, typically the main branch, such as master. When merging branches, Git analyses the changes in each branch and automatically integrates them into a new commit, creating a merge commit that joins the divergent histories. This merge commit represents the point where the two branches come together, incorporating the changes from both branches.

There are two main types of merges in Git: fast-forward merges and recursive merges.

Fast-forward merge: This type of merge occurs when the branch being merged has all commits directly ahead of the current branch. In this case, Git simply moves the pointer of the current branch forward to include the commits from the other branch, resulting in a linear history.

Recursive merge: If the branches being merged have diverged, Git performs a recursive merge. It analyses the changes made in each branch since they diverged from the common ancestor and combines them into a new merge commit. This process may involve resolving conflicts if changes overlap or conflict with each other.

Merging is a common operation in Git workflows, facilitating collaboration among team members working on different features or tasks in parallel. It allows developers to integrate their changes with the main codebase efficiently while maintaining a coherent history of the project. However, it's essential to handle merges carefully, resolve conflicts as they arise, and ensure that the resulting code remains functional and consistent with the project's objectives.

Commit changes in feature-branch

```
git checkout -b feature-branch          # Create and switch to a new branch named "feature-branch"
echo "Content for new file" > new_file.txt  # Create a new text file named "new_file.txt" and add content
git add new_file.txt                    # Stage the new file for commit
git commit -m "Add new_file.txt to feature-branch"  # Commit the new file with a descriptive message
```

Merge feature-branch with master

```
git checkout master                    # Switch back to the master branch
git merge --no-ff feature-branch -m "Merge feature-branch into master with a message"
                                     # Merge the feature-branch into master with a custom commit message with no fast forward (no-ff)
git log --oneline                     # Display the commit history in one line to verify the merge
```

Observation:

The provided sequence of Git commands begins by creating a new branch named "feature-branch" and switching to it. Within this branch, a new text file named "new_file.txt" is created and committed. Subsequently, the switching back to the master branch and merges the changes from "feature-branch" into master, specifying a custom commit message for the merge operation. Finally, the commit history is displayed in a condensed format to verify the successful merge. Overall, this series of commands demonstrates the process of creating a feature branch, making changes within it, and then integrating those changes back into the main branch (master) of the Git repository.

```

91883@Maryjo MINGW64 /f/MITE/Git/d (nb)
$ touch we.txt

91883@Maryjo MINGW64 /f/MITE/Git/d (nb)
$ git add we.txt

91883@Maryjo MINGW64 /f/MITE/Git/d (nb)
$ git commit -m "SdasdasD"
[nb ccda519] SdasdasD
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 we.txt

91883@Maryjo MINGW64 /f/MITE/Git/d (nb)
$ git branch
  master
* nb

91883@Maryjo MINGW64 /f/MITE/Git/d (nb)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

91883@Maryjo MINGW64 /f/MITE/Git/d (master)
$ git merge nb -m "merging nb"
Updating fd2a56f..ccda519
Fast-forward (no commit created; -m option ignored)
 we.txt | 0
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 we.txt

```

Results:

7 Git Tags and Releases

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

Git tag is a feature within the Git version control system that allows developers to mark specific points in a repository's history as significant milestones, releases, or points of interest. Tags provide a way to assign meaningful names or labels to specific commits, making it easier to reference and track important points in a project's development timeline. Tags are typically used to denote release versions, such as stable releases or major updates, but they can also be used for marking important commits, documenting release notes, or identifying specific points for reference. By creating tags, developers can organize and navigate the commit history of a repository more effectively, facilitating collaboration, version management, and software release processes.

```

git init                #Initialize a new Git Repository
git add new_file.txt    # Stage the new file for commit
git commit -m "Add new_file.txt" # Commit the new file with a descriptive message
git tag v1.0            # Tag the current commit as "v1.0"
git log --oneline       # Display the commit history in one line to verify the tag

```

Observation:

The provided sequence of Git commands initializes a new Git repository, adds a new file named "new_file.txt" to the repository, and commits it with a descriptive message. Subsequently, a lightweight tag named "v1.0" is created for the current commit, marking it as a significant milestone or release point. Finally, the commit history is displayed in a condensed format using the `git log --oneline` command to verify the

presence of the newly created tag. Overall, this sequence illustrates the process of setting up a new repository, adding content, committing changes, and applying a tag to signify a specific version or release.

Result: screenshot

Exp8

Advanced Git Operations: Write the command to cherry-pick a range of commits from "source-branch" to the current branch.

Git's cherry-pick command is a powerful tool used for selectively applying individual commits from one branch to another. This operation enables developers to extract specific changes from one branch and apply them onto another, allowing for fine-grained control over the integration of code changes between branches. When executing a cherry-pick, Git creates a new commit on the target branch, replicating the changes introduced by the selected commit(s) from the source branch. This process facilitates the incorporation of isolated features, bug fixes, or improvements from one branch into another, without merging the entire branch's history. While cherry-picking offers flexibility and precision in managing code changes, it's important to handle conflicts that may arise during the process and ensure that the cherry-picked commits integrate smoothly with the target branch's codebase. Overall, Git's cherry-pick operation enhances the versatility and efficiency of version control workflows by facilitating the selective transfer of commits between branches.

create and switch to the source-branch where the commits to be cherry-picked are located

```
git checkout -b source-branch
```

Create and add the first text file

```
touch file1.txt
```

```
echo "Content for file 1" > file1.txt
```

```
git add file1.txt
```

Commit the first text file

```
git commit -m "Add file1.txt"
```

Create and add the second text file

```
touch file2.txt
```

```
echo "Content for file 2" > file2.txt
```

```
git add file2.txt
```

Commit the second text file

```
git commit -m "Add file2.txt"
```

Create and add the third text file

```
touch file3.txt
```

```
echo "Content for file 3" > file3.txt
```



```
git add file3.txt
```

Commit the third text file

```
git commit -m "Add file3.txt"
```

Create and add the fourth text file

```
touch file4.txt
```

```
echo "Content for file 4" > file4.txt
```

```
git add file4.txt
```

Commit the fourth text file

```
git commit -m "Add file4.txt"
```

Switch to the master branch where the cherry-picked commits will be applied

```
git checkout master
```

Cherry-pick the latest 3 commits from the source-branch to the master branch

```
git cherry-pick <ID2>^..<ID4>
```

```
git log --oneline
```

Display the commit history in one line to verify the cherry-pick

Observation:

The provided sequence of Git commands starts by creating and switching to a new branch named "source-branch" where the commits to be cherry-picked will be located. Subsequently, four text files are created, each with content added and committed individually in this branch. Once the commits are made, the process switches back to the master branch, where the latest three commits from the "source-branch" are cherry-picked using their commit IDs. Finally, the git log --oneline command is used to display the commit history in a condensed format, verifying the successful cherry-picking of the specified commits. Overall, this sequence demonstrates the creation of a separate branch for specific commits, the addition and committing of files within that branch, and the cherry-picking of selected commits from one branch to another in Git.

Results:

Screenshot

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?

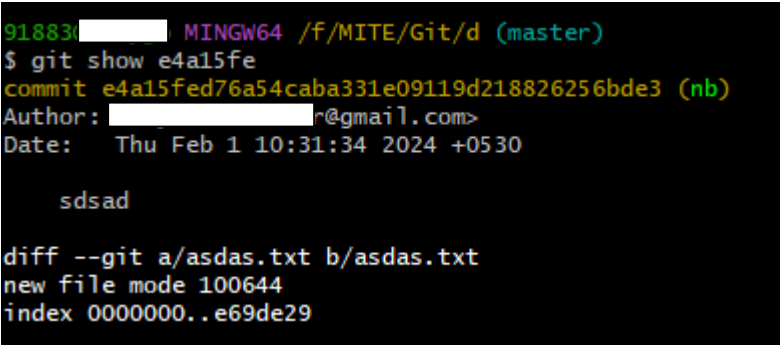
The git show command is a fundamental feature of the Git version control system, designed to provide detailed information about a specific commit within a repository. When invoked with a commit ID as an argument, git show retrieves and presents a comprehensive overview of the selected commit, including metadata such as the commit author, timestamp, and commit message. Additionally, it displays the actual changes introduced by the commit, allowing users to review the modifications made to files and directories.

This command is invaluable for inspecting the history of a project, understanding the context of individual changes, and facilitating collaboration among developers. Through its ability to offer granular insights into commits, git show serves as an essential tool for code review, debugging, and project management workflows within Git repositories.

`git show <commit-ID>`

Observation:

The `git show <commit-ID>` command in Git is used to display detailed information about a specific commit identified by its unique commit ID. When executed, this command presents a comprehensive view of the selected commit, including the commit message, author, timestamp, and the changes introduced by the commit. Additionally, it provides a unified diff of the changes made in the commit, showing the content differences between the commit and its parent commit. This command is particularly useful for reviewing the details of past commits, understanding the modifications introduced in each commit, and tracing the evolution of the project's history. By examining the output of `git show`, developers can gain insights into the context and rationale behind individual changes, aiding in code review, debugging, and collaboration efforts within a Git repository.



```
91883 [redacted] MINGW64 /f/MITE/Git/d (master)
$ git show e4a15fe
commit e4a15fed76a54caba331e09119d218826256bde3 (nb)
Author: [redacted]@gmail.com
Date: Thu Feb 1 10:31:34 2024 +0530

    sdsad

diff --git a/asdas.txt b/asdas.txt
new file mode 100644
index 0000000..e69de29
```

Results:

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."

`git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"`

The `git log --author="JohnDoe" --since="2023-01-01" --until="2023-12-31"` command provides a filtered view of the commit history within a Git repository, focusing specifically on commits authored by a user named "JohnDoe" and committed within a specified time range, from January 1, 2023, to December 31, 2023.

`--author="JohnDoe"`: This option filters the commit log to include only commits authored by the specified user, "JohnDoe". It restricts the output to show commits associated with this specific author.

`--since="2023-01-01"`: This option filters the commit log to include only commits made after January 1, 2023. It sets the starting point of the time range for which commits are displayed.

--until="2023-12-31": This option filters the commit log to include only commits made before December 31, 2023. It sets the ending point of the time range for which commits are displayed.

Together, these options allow users to narrow down the commit history to a specific timeframe and author, providing a focused view of the changes contributed by the specified user within the specified time period. This can be particularly useful for tracking the contributions of individual developers or reviewing changes made during specific project phases or time periods.

Observation :

After executing git log, retrieve the author ID, specify a required date range, and utilize the commit ID obtained from git log to execute git show

Result:

screenshot

11 Analysing and Changing Git History

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

```
git log -n 5
```

Observation:

The git log -n 5 command displays the commit history of the current branch, showing the latest 5 commits. This command limits the output to the most recent commits, allowing users to quickly review the recent changes made to the repository. Each commit is displayed with its unique commit ID, author, timestamp, and commit message. By specifying -n 5, Git restricts the output to the specified number of commits, in this case, the latest 5 commits. This command is commonly used to get a concise overview of recent activity in a Git repository and is particularly useful for tracking recent changes and identifying the contributors to the project.

Result:

Screenshot

12 Analysing and Changing Git History

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

Git revert is a vital command in Git that allows developers to undo specific commits by creating a new commit that inversely applies the changes introduced by the target commit. This operation effectively removes the modifications introduced by the specified commit while preserving the repository's history. Git revert is particularly useful in scenarios where a commit needs to be undone without altering the existing commit history, such as reverting faulty changes or addressing errors introduced by a specific commit. By creating a new commit that undoes the changes of the target commit, Git revert ensures a seamless and traceable reversal of modifications, enabling developers to maintain a consistent and reliable codebase.

```
git revert abc123
```

Observation:

The commit ID was retrieved from the Git log, and the revert operation was executed using the obtained commit ID. Subsequently, the Git log was printed to verify the successful execution of the revert process.

Result:

screenshot