55.Git and GitHub

Git:

- Git is a **distributed version control system (VCS)** used to track changes in source code during software development. It allows multiple developers to work on the same project without interfering with each other's work. Git keeps a record of changes, helps merge changes from different developers, and enables efficient collaboration.

GitHub:

- GitHub is a **web-based platform** that uses Git for version control. It provides hosting for software development and version control using Git. In addition to Git's core features, GitHub offers collaboration tools such as issue tracking, pull requests, and project management features. GitHub is widely used in open-source and private repositories.

Types of Repositories:

1. **Central Repository**:
    - A **central repository** is a single repository that is shared by all team members. It acts as the main hub where everyone pushes their changes and pulls updates. This type of setup is common in centralized version control systems, but in Git, it's still possible to have a central repo that acts as a common location for collaboration.
    - **Example**: GitHub repositories, GitLab repositories.
2. **Local Repository**:
    - A **local repository** is an individual's copy of the project stored on their local machine. Developers work locally, making changes and committing them without affecting the central repository until they are ready to push their changes. This is the default nature of Git's distributed model.
    - **Example**: A developer's local clone of a GitHub repository.
3. **Distributed Repository**:
    - Git is a **distributed version control system**, meaning each developer has a **complete copy of the repository** with its entire history. There is no central server required for version control, as each local repository is fully functional on its own. Developers can push and pull from others' repositories to sync changes.
    - **Example**: A developer cloning a repository from GitHub, working offline, and later syncing their changes with the central repository.

Git Features:

1. **Distributed**:
    - Git is **distributed**, meaning each developer has their own local copy of the entire repository, including its history. This allows developers to work offline and commit changes independently before syncing with others.
2. **Compatible**:
    - Git is **compatible** with many other systems and services. It can integrate with various tools, IDEs, and CI/CD platforms, making it flexible for different workflows.
3. **Non-Linear**:

- Git supports **non-linear development** by allowing multiple branches to be created simultaneously. Developers can work on different features or fixes in parallel without affecting each other's work, and later merge changes back into the main branch.

4. **Branching**:
   - Git makes branching extremely easy and lightweight. Developers can create **branches** to experiment with new features, fix bugs, or collaborate on isolated tasks. This enables parallel development with minimal risk of conflicts.
5. **Lightweight**:
   - Git is **lightweight** in terms of both resource consumption and operations. Branches, for example, are not expensive to create or switch between, as they are simply pointers to commits, making operations like merging or branching much faster.
6. **Speed**:
   - Git is known for its **speed**. Operations like commit, checkout, merge, and diff are highly optimized, even for large repositories, making Git an efficient choice for developers.
7. **Open Source**:
   - Git is **open source**, meaning it is free to use and its source code is available for anyone to contribute to or modify. This has led to widespread adoption and community-driven improvements.
8. **Reliable**:
   - Git is **reliable** in terms of data integrity. It uses checksums (SHA-1 hashes) to ensure that data is never lost or corrupted. Every commit has a unique hash, providing a safeguard against potential errors.
9. **Secure**:
   - Git has robust **security features**, such as cryptographic hashing, which ensures the integrity of the repository and the commits. Additionally, access control can be implemented with SSH keys or personal access tokens when interacting with platforms like GitHub.
10. **Economical**:
    - Git is **economical** in terms of storage. It uses efficient methods to store data, and because it's distributed, developers don't have to worry about continuously communicating with a central server to commit or retrieve changes. Additionally, GitHub and other platforms offer free repositories, especially for public projects.

Getting & Creating Projects

Command	Description
`git init`	Initialize a local Git repository
`git clone ssh://git@github.com/[username]/[repository-name].git`	Create a local copy of a remote repository

Basic Snapshotting

Command	Description
`git status`	Check status
`git add [file-name.txt]`	Add a file to the staging area
`git add -A`	Add all new and changed files to the staging area
`git commit -m "[commit message]"`	Commit changes
`git rm -r [file-name.txt]`	Remove a file (or folder)
`git remote -v`	View the remote repository of the currently working file or directory

Branching & Merging

Command	Description
`git branch`	List branches (the asterisk denotes the current branch)
`git branch -a`	List all branches (local and remote)
`git branch [branch name]`	Create a new branch
`git branch -d [branch name]`	Delete a branch
`git push origin --delete [branch name]`	Delete a remote branch
`git checkout -b [branch name]`	Create a new branch and switch to it
`git checkout -b [branch name] origin/[branch name]`	Clone a remote branch and switch to it

Command	Description
`git branch -m [old branch name] [new branch name]`	Rename a local branch
`git checkout [branch name]`	Switch to a branch
`git checkout -`	Switch to the branch last checked out
`git checkout -- [file-name.txt]`	Discard changes to a file
`git merge [branch name]`	Merge a branch into the active branch
`git merge [source branch] [target branch]`	Merge a branch into a target branch
`git stash`	Stash changes in a dirty working directory
`git stash clear`	Remove all stashed entries
`git stash pop`	Apply latest stash to working directory

Sharing & Updating Projects

Command	Description
`git push origin [branch name]`	Push a branch to your remote repository
`git push -u origin [branch name]`	Push changes to remote repository (and remember the branch)
`git push`	Push changes to remote repository (remembered branch)
`git push origin --delete [branch name]`	Delete a remote branch
`git pull`	Update local repository

Command	Description
	to the newest commit
`git pull origin [branch name]`	Pull changes from remote repository
`git remote add origin ssh://git@github.com/[username]/[repository-name].git`	Add a remote repository
`git remote set-url origin ssh://git@github.com/[username]/[repository-name].git`	Set a repository's origin branch to SSH

Inspection & Comparison

Command	Description
`git log`	View changes
`git log --summary`	View changes (detailed)
`git log --oneline`	View changes (briefly)
`git diff [source branch] [target branch]`	Preview changes before merging

SCM (Source Code Management) and Git

1. SCM (Source Code Management):

- **What is SCM?**
  - SCM refers to tools and practices used to manage changes in source code, track versions, and handle code revisions.
  - SCM systems help manage the history of code changes and collaborate among multiple developers working on a project.

Key Benefits of Using SCM:

- **Collaboration:** Multiple developers can work on the same project simultaneously without overwriting each other's work.
- **Version Control:** Tracks all code changes, allowing developers to go back to previous versions if needed.
- **Code Review:** Changes can be reviewed before merging into the main codebase.
- **Risk Mitigation:** Prevents loss of code and ensures that any mistakes or bugs can be traced back and corrected.
- **Improved Productivity:** Developers can work in parallel, switch between versions, and merge changes easily.

Git: A Distributed SCM Tool

1. Git Overview:

- **What is Git?**
  - Git is a **distributed version control system**. It helps track changes to code, manage versions, and collaborate on development.
  - Git is **incremental**, meaning it stores only the changes made (not the whole file each time).

2. Core Concepts in Git:

- **Commit:**
  - A commit is like a **snapshot** of your project at a specific point in time.
  - Each commit in Git records changes to the files and is assigned a unique ID (commit hash).
- **Incremental Versioning:**
  - Git tracks only changes (additions, deletions, modifications) rather than copying the entire file, making it more efficient.
- **Branching:**
  - Git allows developers to **branch** off the main codebase, make changes, and then merge them back. This lets developers work on different features simultaneously without interfering with each other's code.
  - **Feature Branching:** Each new feature is developed in a separate branch, keeping the main codebase stable.

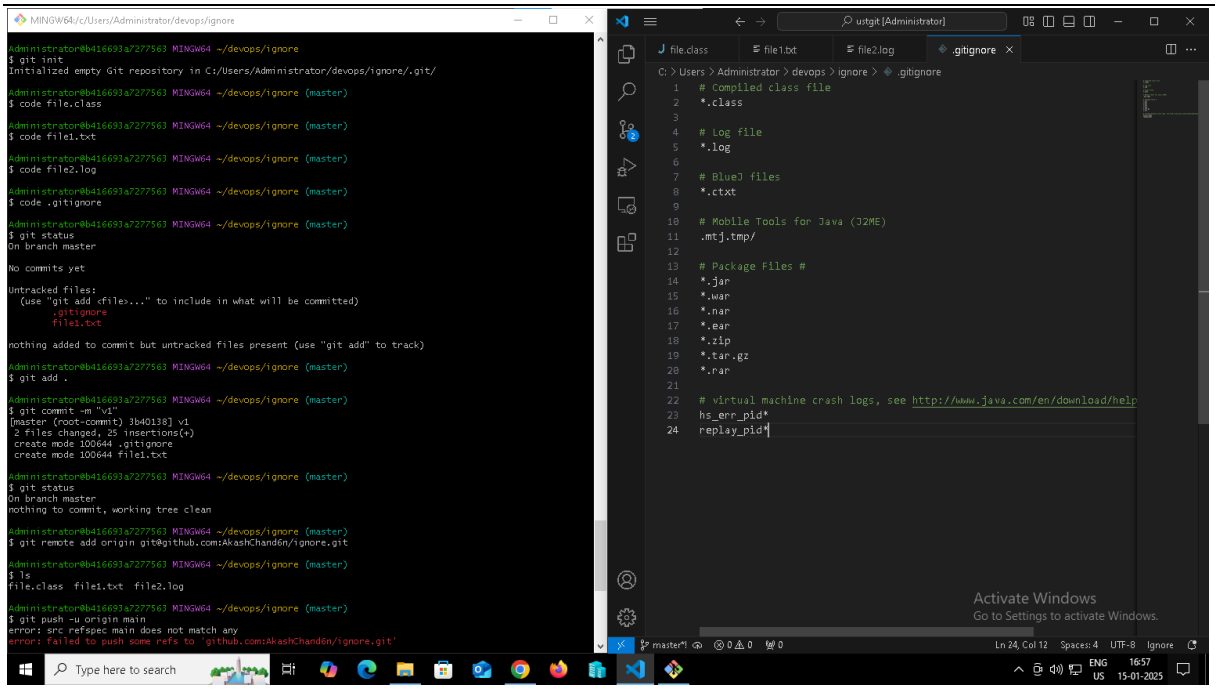
- **Repository:**
  - A Git repository is where your project's **version history** is stored. It holds all the commits, branches, and information about your project.
- **Version History:**
  - Git keeps a detailed **history** of all changes made to the codebase. This helps you track progress, go back to previous states, and find bugs introduced in past commits.

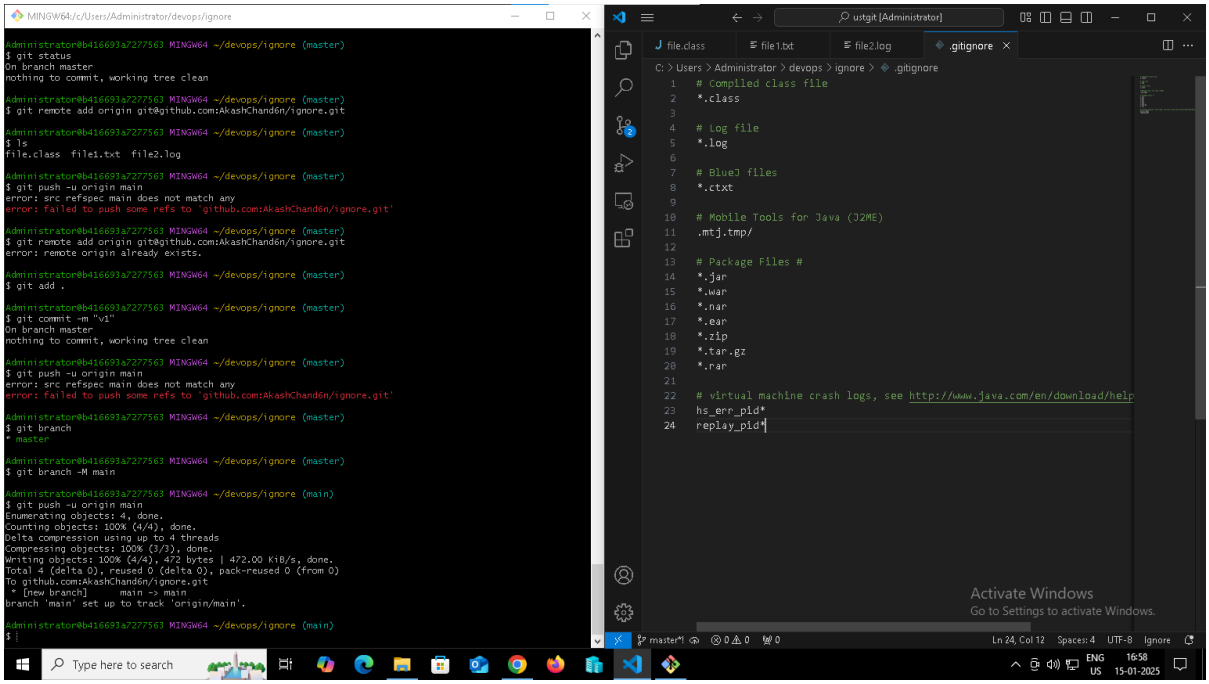
Benefits of Using Git:

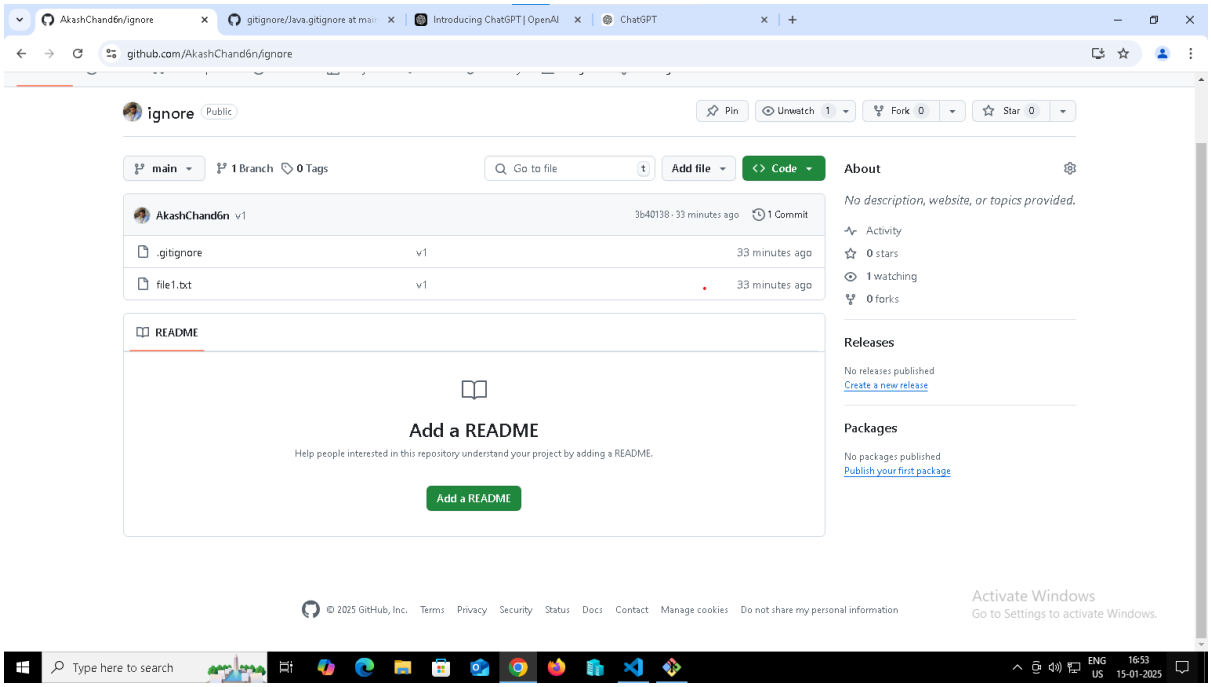
- **Collaboration:** Multiple developers can contribute to the same project without conflict.
- **Version Control:** Git keeps track of changes and allows reverting to previous versions of the code.
- **Code Review:** Changes can be reviewed before being merged into the main branch.
- **Risk Mitigation:** Mistakes are easier to identify and fix since all changes are recorded.
- **Improved Productivity:** Git allows for parallel development, easy switching between versions, and faster project management.

GIT Ignore

In Git, the .gitignore file is used to tell Git which files or directories to ignore in your repository. This is useful when you have files that you don't want to track or version control, such as build artifacts, log files, or personal configuration files.







1. git restore

The git restore command is used to restore files in your working directory to their state in a particular commit or branch. It's useful when you want to discard changes you've made in your working directory and revert files to the last committed state or another specific commit.

Usage:

- **Restore changes in a file** (to undo changes in the working directory):

  ```
  git restore <file>
  ```

- **Restore changes in all files** (undo changes in the entire working directory):

  ```
  git restore .
  ```

- **Restore files to a specific commit**:

  ```
  git restore --source <commit> <file>
  ```

- **Restore files to the staged area (index)**:

  ```
  git restore --staged <file>
  ```

- **Discard untracked files**:

  ```
  git restore --source=HEAD --staged --worktree -- <file>
  ```

---

2. git rebase

The git rebase command is used to move or combine a sequence of commits to a new base commit. It's typically used for:

- **Rewriting commit history**: Rebase allows you to change the order, squash, or edit commits.
- **Applying changes from one branch onto another**: You can rebase feature branches onto the latest commit of the main branch to incorporate new changes from the base branch.

Usage:

- **Rebase your current branch onto another branch**:

  ```
  git rebase <branch-name>
  ```

- **Interactive rebase (to edit commits)**:

  ```
  git rebase -i <commit-hash>
  ```

  This opens an editor where you can choose to reword, squash, or remove commits.

- **Abort a rebase** (if conflicts arise or if you want to cancel the rebase):

  ```
  git rebase --abort
  ```

- **Continue a rebase** (after resolving conflicts):

  ```
  git rebase --continue
  ```

---

3. git reset

The git reset command is used to undo commits and changes in the working directory. It can modify the index (staged changes), working directory (local changes), or both.

There are three primary modes for git reset:

- **Soft**: Moves the HEAD pointer to a previous commit but leaves your changes in the staging area.
- **Mixed** (default): Moves the HEAD pointer to a previous commit and resets the index, but leaves changes in the working directory.
- **Hard**: Moves the HEAD pointer to a previous commit and resets both the index and working directory (all changes are lost).

Usage:

- **Soft reset** (keep changes in the staging area):

  ```
  git reset --soft <commit-hash>
  ```

- **Mixed reset** (default, keep changes in the working directory but unstage them):

  ```
  git reset <commit-hash>
  ```

- **Hard reset** (discard all changes, reset to a specific commit):

  ```
  git reset --hard <commit-hash>
  ```

- **Reset to the previous commit**:

  ```
  git reset --hard HEAD
  ```

---

4. git revert

The git revert command is used to create a new commit that undoes the changes made in a specific commit. Unlike git reset, git revert does not alter the commit history; it adds a new commit that reverses the changes of a previous one.

Usage:

- **Revert a specific commit**:

  ```
  git revert <commit-hash>
  ```

- **Revert multiple commits**:

  ```
  git revert <commit-hash-1>..<commit-hash-2>
  ```

---

5. git stash

The git stash command is used to temporarily save changes that are not yet committed, so you can work on something else without committing your unfinished work. You can later apply these stashed changes back into your working directory.

Usage:

- **Stash changes** (including untracked files):

  ```
  git stash
  ```

- **Stash changes with a custom message**:

  ```
  git stash save "message describing the stash"
  ```

- **List stashes**:

  ```
  git stash list
  ```

- **Apply the most recent stash**:

  ```
  git stash apply
  ```

- **Apply a specific stash**:

  ```
  git stash apply stash@{1}
  ```

- **Pop the most recent stash** (apply and remove from the stash list):

  ```
  git stash pop
  ```

- **Drop a specific stash** (remove it from the list):

  ```
  git stash drop stash@{0}
  ```

- **Clear all stashes**:

  git stash clear

Summary of Key Differences:

Command	Purpose	Effects
git restore	Restore files to a particular commit or discard changes.	Affects working directory and/or staged files.
git rebase	Reapply commits on top of another branch or commit.	Modifies commit history by changing commit base.

Command	Purpose	Effects
git reset	Undo commits and changes.	Resets HEAD, index, and working directory (depending on mode).
git revert	Create a new commit that undoes changes of a specific commit.	Does not alter commit history; adds a new "revert" commit.
git stash	Temporarily save changes in progress.	Stashes changes to be reapplied later without committing.

When to Use Which Command?

- **Use git restore** when you want to discard local changes to files or reset them to a particular state.
- **Use git rebase** when you need to reapply commits on top of another branch (useful for keeping a clean commit history).
- **Use git reset** when you need to undo commits or unstage changes.
- **Use git revert** when you need to undo changes made by a commit but want to preserve the commit history (ideal for public branches).
- **Use git stash** when you need to temporarily save changes you're working on to switch tasks without committing them.

Tracking Branches

Checking out a local branch from a remote-tracking branch automatically creates what is called a "tracking branch" (and the branch it tracks is called an "upstream branch"). **Tracking branches are local branches that have a direct relationship to a remote branch. If you're on a tracking branch and type git pull, Git automatically knows which server to fetch from and which branch to merge in**.

`git checkout --track origin/branch ->` is used to create a local branch that tracks a remote branch.

`git push origin --delete branch ->` is used to delete a remote branch from the origin repository

`git branch -D branch ->` is used to **forcefully delete a local branch** in Git

Git LFS

Git LFS is an extension to Git that helps you manage large files (such as images, videos, datasets, etc.) by storing them outside the regular Git repository, while keeping lightweight references in the repository.

Initialize Git LFS

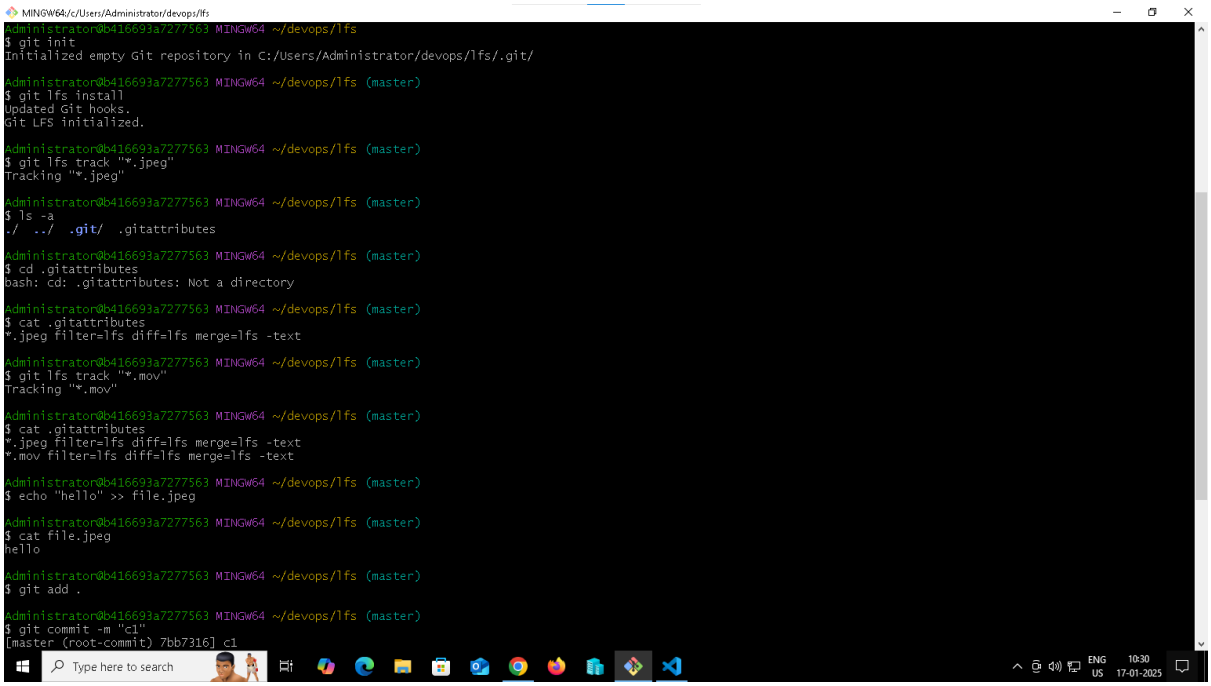
```
git lfs install
```

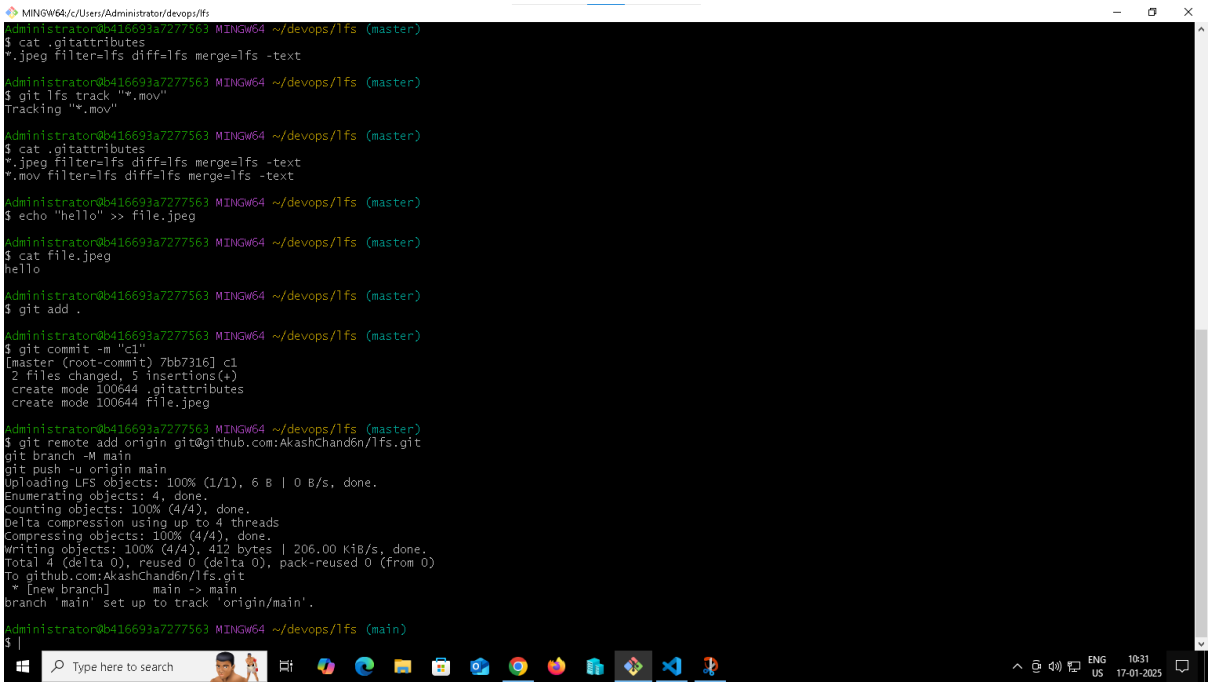
Track Large Files

```
git lfs track "*.png"
```

```
git lfs track "*.mp4"
```

This will create a .gitattributes file in your repository that tells Git which files to track with Git LFS.





Signed commit

1. **Generate a GPG Key**

   ```
   gpg --full-generate-key
   ```

2. **List Your GPG Keys**

   ```
   gpg --list-secret-keys --keyid-format LONG
   ```

3. **Configure Git to Use Your GPG Key**

   ```
   git config --global user.signingkey <KEY_ID>
   ```

4. **Tell Git to Sign All Commits**

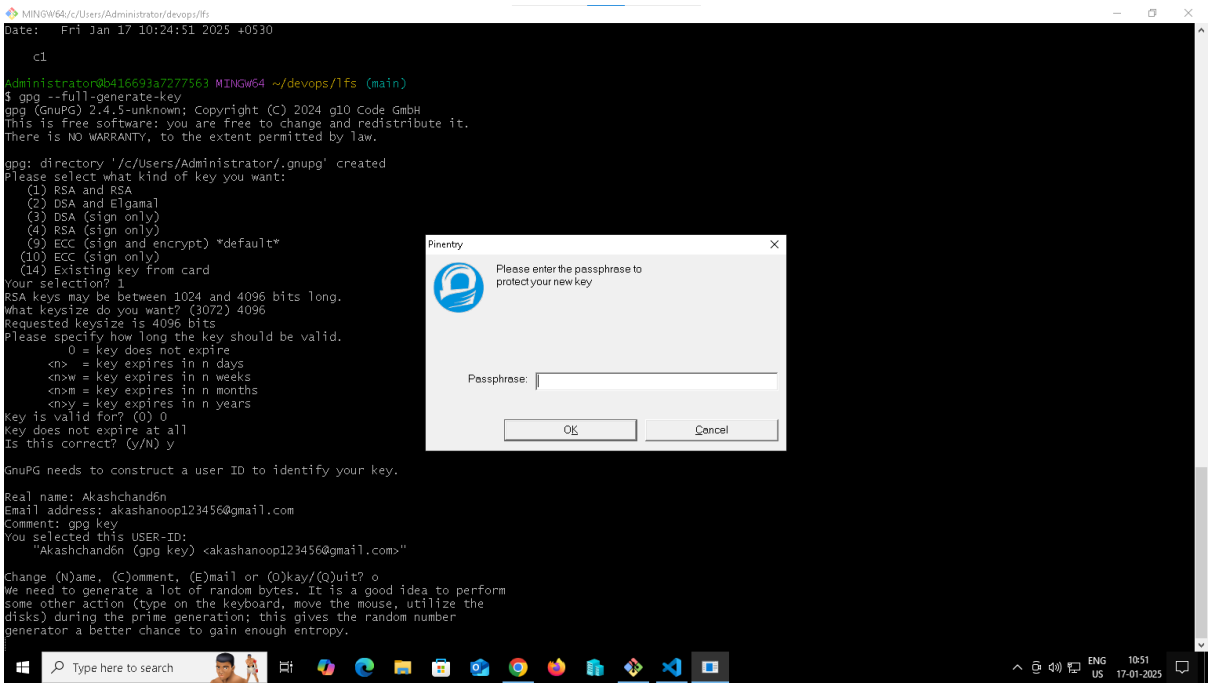
   ```
   git config --global commit.gpgSign true
   ```

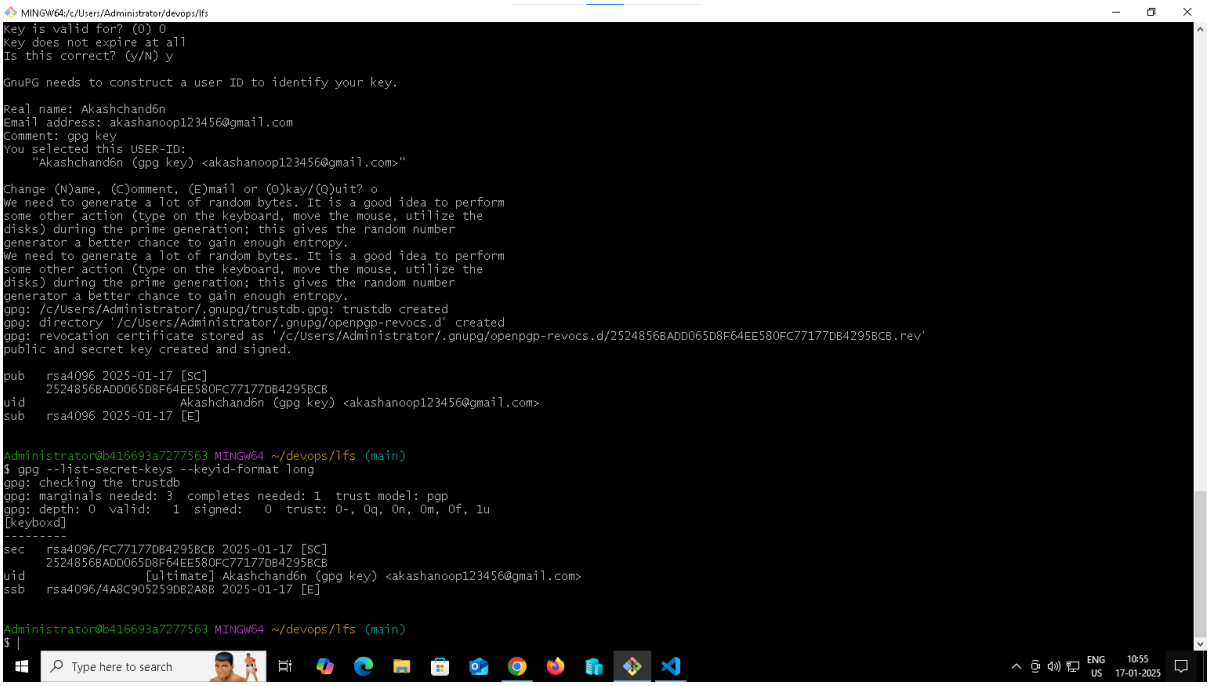
5. **Make a Signed Commit**

   ```
   git commit -m "My signed commit"
   ```

6. **Verify a Signed Commit**

   ```
   git log --show-signature
   ```




MINGW64:/c/Users/Administrator/devops/lfs
Key is valid for? (0) 0
Key does not expire at all
Is this correct? (y/N) y
GnuPG needs to construct a user ID to identify your key.
Real name: Akashchand6n
Email address: akashanoop123456@gmail.com
Comment: gpg key
You selected this USER-ID:
"Akashchand6n (gpg key) <akashanoop123456@gmail.com>"
Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? o
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilize the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.
gpg: /c/Users/Administrator/.gnupg/trustdb.gpg: trustdb created
gpg: directory '/c/Users/Administrator/.gnupg/openpgp-revocs.d' created
gpg: revocation certificate stored as '/c/Users/Administrator/.gnupg/openpgp-revocs.d/2524856BADD065D8F64EE580FC77177DB4295BCB.rev'
public and secret key created and signed.
pub rsa4096 2025-01-17 [SC]
2524856BADD065D8F64EE580FC77177DB4295BCB
uid Akashchand6n (gpg key) <akashanoop123456@gmail.com>
sub rsa4096 2025-01-17 [E]
Administrator@b416693a7277563 MINGW64 ~/devops/lfs (main)
$ gpg --list-secret-keys --keyid-format long
gpg: checking the trustdb
gpg: marginals needed: 3 completes needed: 1 trust model: pgp
gpg: depth: 0 valid: 1 signed: 0 trust: 0-, 0q, 0n, 0m, 0f, 1u
[keyboxd]
---
sec rsa4096/FC77177DB4295BCB 2025-01-17 [SC]
2524856BADD065D8F64EE580FC77177DB4295BCB
uid [ultimate] Akashchand6n (gpg key) <akashanoop123456@gmail.com>
ssb rsa4096/4A8C905259DB2A8B 2025-01-17 [E]
Administrator@b416693a7277563 MINGW64 ~/devops/lfs (main)
$
Type here to search
ENG
US
10:55
17-01-2025