

GITHUB ASSIGNMENT

What is Git?

Git is a version control system (VCS) used to manage changes to a project's code, track history, and collaborate efficiently among developers.

Version Control

Version control helps in **keeping track of changes** made to the code over time. It allows developers to:

- **Track changes:** Who made the change, what change was made, and when.
- **Undo changes:** If a bug is introduced, you can go back to an earlier version.
- **Collaboration:** Multiple developers can work on the same project without overwriting each other's work.

Why Maintain Versions?

Maintaining versions allows you to:

- **Track the history** of your code (who changed what and why).
- **Revert** to previous code versions if something breaks.
- **Collaborate** better, ensuring that multiple developers don't overwrite each other's work.

Central Code Repository

A **central code repository** is a place where the main code base is stored. Examples include GitHub, GitLab, or Bitbucket. It allows developers to **collaborate** on a project, ensuring everyone has access to the latest code.

Source Code Management (SCM)

SCM tools help you manage the source code effectively:

- **Local Repository:** Code is stored on a local machine.
 - **Advantages:** Faster performance, offline work, no latency, and no dependency on a server.
 - **Disadvantages:** Risk of losing data if the system crashes (because it's not backed up), single point of failure, and no collaboration.
- **Centralized Repository:** Code is stored on a central server.
 - **Advantages:** Easy backup and collaboration.
 - **Disadvantages:** Cannot work offline, slower operations, depends on the server's availability.

Distributed Repository

Git is a **distributed version control system (DVCS)**, meaning every developer has their own **local repository** (a full copy of the code), and there's also a **central repository** (for collaboration).

- **Advantages:**
 - Developers mostly work locally, making operations fast and efficient.
 - Changes are later pushed to a central repository for collaboration.
- **Disadvantages:**
 - If you don't push your changes regularly, it can lead to conflicts when syncing with others.

Git's **distributed** model allows each developer to have a complete version of the repository locally, and the majority of work happens in the local repo.

Git Commands

- **git commit:** This command saves changes you made locally in your repository (staged files).
- **git push:** This command sends your local changes to a **remote repository** (like GitHub) so others can access it.

Advantages of Git

1. **Distributed Version Control:**
 - Each developer has their own **local copy** of the entire repository, allowing them to work independently without needing a constant internet connection.
2. **Speed:**
 - Git is very **fast**. Since most operations happen locally, they are **incremental** and fast to execute.
3. **Branching:**
 - Git allows developers to create **branches** for new features or bug fixes, making it easy to **develop in parallel** without affecting the main codebase.
 - **Feature Branching:** You can work on a specific feature in isolation and later merge it with the main branch (master or main).

Branching in Git

- Branching allows developers to work on different features or versions of the code without affecting others.
- **Example:** Imagine you're adding a new feature. You create a branch, make changes there, and when it's ready, you merge it back into the main branch.

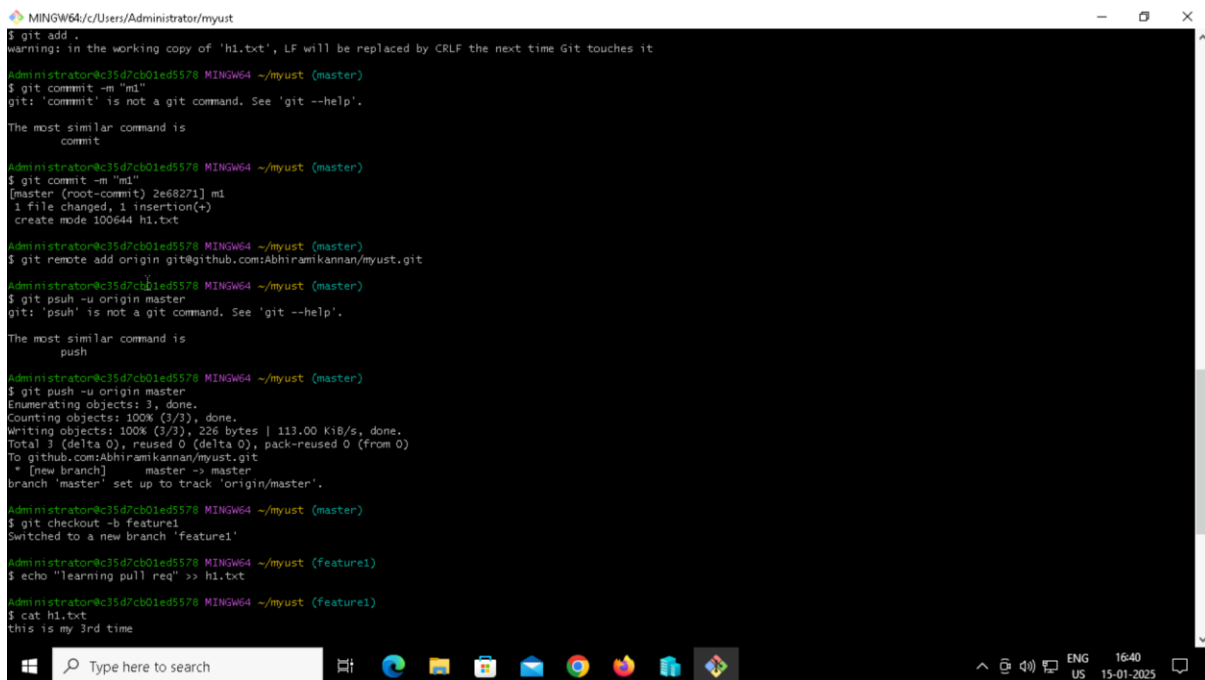
Git's Speed and Lightweight Nature

- Git uses **SHA-1 hashing** to identify changes, ensuring it's **fast** and **lightweight**.
- Git's **incremental model** means that only the changes are stored, rather than copying everything each time, improving performance.

Summary of Git Benefits:

- **Distributed:** Each developer has a full copy of the code.
- **Fast:** Most operations happen locally.
- **Efficient:** Only changes are saved, using SHA-1 hashing.
- **Branching:** Work on different features in parallel without interfering with the main code.

Git enables efficient collaboration and version management by combining both **local repositories** for speed and a **central repository** for collaboration.



```
Administrator@C35d7cb01ed5578 MINGW64 ~/myust
$ git add .
warning: in the working copy of 'h1.txt', LF will be replaced by CRLF the next time Git touches it
Administrator@C35d7cb01ed5578 MINGW64 ~/myust (master)
$ git commit -m "m1"
git: 'commit' is not a git command. See 'git --help'.

The most similar command is
commit

Administrator@C35d7cb01ed5578 MINGW64 ~/myust (master)
$ git commit -m "m1"
[master (root-commit) 2e68271] m1
1 file changed, 1 insertion(+)
create mode 100644 h1.txt

Administrator@C35d7cb01ed5578 MINGW64 ~/myust (master)
$ git remote add origin git@github.com:Abhiramkannan/myust.git
Administrator@C35d7cb01ed5578 MINGW64 ~/myust (master)
$ git push -u origin master
git: 'push' is not a git command. See 'git --help'.

The most similar command is
push

Administrator@C35d7cb01ed5578 MINGW64 ~/myust (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 226 bytes | 113.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:Abhiramkannan/myust.git
 * [new branch] master -> master
branch 'master' set up to track 'origin/master'.

Administrator@C35d7cb01ed5578 MINGW64 ~/myust (master)
$ git checkout -b feature1
Switched to a new branch 'feature1'

Administrator@C35d7cb01ed5578 MINGW64 ~/myust (feature1)
$ echo "learning pull req" >> h1.txt

Administrator@C35d7cb01ed5578 MINGW64 ~/myust (feature1)
$ cat h1.txt
this is my 3rd time
```

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.
 - **Example:**
If you're traveling from **Tvm** (Thiruvananthapuram) to **Kazhakootam**, you only note the changes (Tvm to Kazhakootam). On another trip, you record all stops (Tvm → Chakka → Infosys → Tvm) but avoid repeating the same information.
 - **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.
-

Installing Git:

For Ubuntu (Debian-based systems):

- To install Git:
- `sudo apt install git -y`

For Fedora (Red Hat-based systems):

- To install Git:
 - `sudo dnf install git -y`
-

Basic Git Commands:

1. `git init`: Initializes a new Git repository in your project directory.
 2. `git add .`: Stages all changes in your project for the next commit.
 3. `git commit -m "message"`: Commits your staged changes with a message describing the changes.
 4. `git push`: Pushes the local commits to the remote repository (e.g., GitHub).
 5. `git pull`: Pulls the latest changes from the remote repository.
 6. `git status`: Shows the current status of your files (modified, staged, untracked, etc.).
 7. `git log --oneline`: Shows a simplified commit history with each commit's hash and message.
 8. `git branch`: Lists the branches in your repository.
 9. `git checkout branch-name`: Switches to the specified branch.
 10. `git merge branch-name`: Merges changes from the specified branch into the current branch.
-

Conclusion:

- **Git** is an essential tool for **version control** and **collaboration** in software development.

- It tracks changes, helps developers work together, and allows efficient management of code with **branches**, **commits**, and **repositories**.
- **Installing Git** and getting started with the basic commands enables you to effectively manage code and keep a detailed history of all project changes.

GIT RESTORE

The git restore command is used to **undo changes** in your working directory or to **restore files** to a specific state, usually from the last commit or from a branch. It's primarily used to **discard local changes** or to restore deleted or modified files.

Sure! Let's break it down simply:

git restore Overview:

The git restore command is used to undo changes in your working directory or staging area. You can use it to either:

- **Unstage files** (remove files from the staging area without affecting the working directory)
- **Restore files** to their last committed state (discard changes).

git restore --staged

- **Use:** Removes changes from the staging area (unstages the file).
- **Example:** You've added a file to the staging area with git add, but now you don't want to commit it yet. You can unstage it with git restore --staged.
- git add file.txt
- git restore --staged file.txt

Explanation:

- After running git add, the file is staged, ready to be committed.
- If you change your mind and don't want to commit file.txt, you use git restore --staged file.txt to unstage it.
- The changes are still in the working directory, but it is no longer staged for commit.

git restore --worktree

- **Use:** Discards changes in the working directory (reverts the file to the last committed version).
- **Example:** You've made changes to a file, but you want to discard those changes and return the file to the state of the last commit.

- `git restore --worktree file.txt`

Explanation:

- If you have modified `file.txt` and want to undo the changes, you can use this command.
- This command will remove the changes from the working directory and restore the file to its last committed state.
- **Important:** This will **lose** your uncommitted changes to the file.

`git restore` (without options):

- **Use:** By default, `git restore` works like `git restore --worktree`.
- **Example:** Simply restore the file to the last committed state:
- `git restore file.txt`

Explanation:

- This command discards changes in the working directory and restores the file to the version in the last commit.

Summary of Usage:

1. **`git restore --staged <filename>`:**
 - Removes a file from the staging area (unstages it).
 - **Does not change the file in the working directory.**
2. **`git restore --worktree <filename>`:**
 - Discards local changes in the working directory.
 - **Reverts the file to the last committed version.**
3. **`git restore <filename>`** (same as `--worktree`):
 - Restores the file to the last committed version, removing local changes.

Branching and Merging are two key concepts in Git that allow you to work on different features or parts of your project without affecting the main codebase. Let's break down the commands for these operations:

Branching in Git

A **branch** in Git allows you to create a separate line of development. By default, you're working in the `main` or `master` branch, but you can create and switch to new branches.

1. Create a New Branch

To create a new branch:

```
git branch <branch_name>
```

Example:

```
git branch feature-branch
```

This will create a new branch called feature-branch but **won't switch** to it.

2. Switch to a Branch

To switch from the current branch to another branch:

```
git checkout <branch_name>
```

Example:

```
git checkout feature-branch
```

Alternatively, you can use the **git switch** command to switch branches (in newer versions of Git):

```
git switch <branch_name>
```

Example:

```
git switch feature-branch
```

3. Create and Switch to a Branch in One Command

You can create and immediately switch to a new branch with:

```
git checkout -b <branch_name>
```

Or with the newer git switch:

```
git switch -c <branch_name>
```

Example:

```
git checkout -b feature-branch
```

This creates feature-branch and switches to it.

4. List All Branches

To see a list of all branches in your repository:

```
git branch
```

- The currently active branch will be highlighted with an asterisk (*).
-

Merging in Git

Merging allows you to bring the changes from one branch into another.

1. Merge a Branch into the Current Branch

To merge another branch (e.g., feature-branch) into your current branch (e.g., main), use:

```
git merge <branch_name>
```

Example:

```
git merge feature-branch
```

This will merge the changes from feature-branch into your current branch.

2. Handle Merge Conflicts

Sometimes, changes made in both branches can't be automatically merged (conflicts). Git will mark these conflicts in the file, and you'll need to manually resolve them. Once resolved, use:

```
git add <resolved_file>
```

After resolving the conflicts, commit the merge:

```
git commit
```

Git may automatically create a merge commit message for you, or you can write your own.

3. Merge Using a Fast-Forward Merge

If there are no diverging changes between the branches, Git may perform a "fast-forward" merge. This means the branch pointer is simply moved forward, and no merge commit is created.

To ensure a fast-forward merge, use:

```
git merge --ff-only <branch_name>
```

If a fast-forward merge is not possible (because of diverging changes), Git will throw an error.

Other Useful Git Commands for Branching and Merging

1. Delete a Branch

To delete a branch that you no longer need:

- **Delete a local branch:**
- `git branch -d <branch_name>`

Example:

```
git branch -d feature-branch
```

(Use `-D` to force delete the branch even if it hasn't been fully merged.)

- **Delete a remote branch:**
- `git push origin --delete <branch_name>`

Example:

```
git push origin --delete feature-branch
```

2. View Merge History

To see the history of merges in your repository:

```
git log --merges
```

3. Rebase (Alternative to Merging)

Instead of merging, you can **rebase** a branch. Rebasing applies the changes from one branch onto another, creating a linear history.

To rebase a branch onto the current branch:

```
git rebase <branch_name>
```

Example:

```
git rebase feature-branch
```

Rebasing rewrites commit history, so it's used carefully, particularly when working in teams.

```
mkdir rebase
```

```
280 cd rebase/
```

```
281 git init
```

```
282 echo "M1" >> file.txt
```

```
283 git add .
```

```
284 git commit -m "M1"
```

```
285 echo "M2" >> file.txt
286 git add .
287 git commit -m "M2"
288 echo "M3" >> file.txt
289 git add .
290 git commit -m "M3"
291 git checkout -b feature
292 echo "F1" >> feature.txt
293 git add .
294 git commit -m "F1"
295 echo "F2" >> feature.txt
296 git add .
297 git commit -m "F2"
298 git checkout -b newfeature
299 git checkout master
300 echo "M4" >> file.txt
301 git add .
302 git commit -m "M4"
303 git checkout feature
304 git rebase master
305 git log --oneline
306 git checkout master
307 git merge newfeature
308 git log --oneline
309 git checkout feature
```

Summary of Key Git Branching and Merging Commands

1. **Create a new branch:**
2. `git branch <branch_name>`
3. **Switch to a branch:**
4. `git checkout <branch_name>`

or

`git switch <branch_name>`

5. **Create and switch to a new branch:**
6. `git checkout -b <branch_name>`

or

`git switch -c <branch_name>`

7. **List all branches:**
8. `git branch`
9. **Merge a branch into the current branch:**
10. `git merge <branch_name>`
11. **Delete a branch:**
12. `git branch -d <branch_name>`
13. **Delete a remote branch:**
14. `git push origin --delete <branch_name>`
15. **Rebase a branch:**
16. `git rebase <branch_name>`

These are the core Git commands you'll need for **branching** and **merging**!

Benefits of Software Configuration Management (SCM):

1. **Version Control:** Tracks changes and allows easy rollback to previous versions.
2. **Collaboration:** Multiple developers can work together without overwriting each other's work.
3. **Code Quality:** Ensures code is in a stable state, preventing broken code from being pushed.
4. **Audit Trail:** Keeps a log of changes, who made them, and why.
5. **Branching and Merging:** Allows parallel development and safe integration of changes.
6. **Backups:** Prevents data loss by storing versions in remote repositories.

7. **Simplified Releases:** Facilitates stable, controlled deployments.
8. **Team Coordination:** Improves workflow through task tracking and integration with project tools.
9. **Conflict Resolution:** Helps resolve code conflicts between team members.
10. **Automation:** Integrates with CI tools for automated testing and deployment.

In short, SCM improves collaboration, maintains code integrity, and enhances project management.

- ☐ **Commit:** Records changes and snapshots the project at a point in time.
- ☐ **Tag:** Marks a specific commit, typically used for releases.
- ☐ **Hooks:** Automates tasks or actions triggered by Git events (e.g., before a commit or push).
- ☐ **Objects:** Git stores data (commits, files, directories) as objects (commits, trees, blobs, tags).

git commit –amend

This command helps to edit the commit message of the last message. The below screenshot displays the commit message modified

RESET

In Git, the git reset command is used to **undo changes** in your repository by moving the current branch pointer to a previous commit. It can also affect the staging area and the working directory. The three main types of resets are: **soft**, **mixed**, and **hard**.

1. Soft Reset (git reset --soft)

- **Effect:** Moves the branch pointer to the specified commit but **keeps changes in the staging area** (index).
- **Use Case:** You want to undo a commit but keep the changes staged for the next commit.
- **Example:**
- `git reset --soft HEAD~1`

This command will undo the most recent commit (HEAD~1 refers to the commit just before the current one) but **leave your changes in the staging area** so you can commit them again.

2. Mixed Reset (git reset --mixed)

- **Effect:** Moves the branch pointer to the specified commit and **unstages changes** (removes them from the staging area) but **keeps the changes in your working directory**.
- **Use Case:** You want to undo a commit and unstage the changes, but keep the modifications in your working directory for further editing.
- **Example:**
- `git reset --mixed HEAD~1`

This command undoes the most recent commit and removes the changes from the staging area, but the changes will still be in your working directory.

- **Note:** `--mixed` is the default behavior for `git reset`, so if you use `git reset` without specifying an option, it will behave like `--mixed`.

3. Hard Reset (`git reset --hard`)

- **Effect:** Moves the branch pointer to the specified commit and **completely removes all changes** in both the staging area and the working directory. It **discards all changes** made since the commit you're resetting to.
- **Use Case:** You want to completely discard changes and reset the repository to a clean state at a specific commit.
- **Example:**
- `git reset --hard HEAD~1`

This command will **remove all uncommitted changes**, reset the branch to the previous commit, and make the working directory match that commit.

Summary:

- **`git reset --soft <commit>`:** Undo the commit but keep changes staged for the next commit.
- **`git reset --mixed <commit>`:** Undo the commit and unstage changes, but keep them in the working directory.
- **`git reset --hard <commit>`:** Undo the commit and **discard all changes** in the working directory and staging area.
- Default mode. You will still keep the changes in your working tree but not on index

Each type of reset serves a different purpose depending on how much of your changes you want to keep or discard.

GIT REVERT

The `git revert` command is used to **undo a commit** by creating a new commit that **reverses** the changes made by a previous commit. Unlike `git reset`, which changes the history of your repository, `git revert` adds a new commit that **preserves history** while effectively undoing the changes from a specific commit.

`git revert <commit_id>`

You can also revert a range of commits. For example:

```
git revert <old_commit_id>..<new_commit_id>
```

When to Use git revert:

- **Undo a commit on a public branch:** You can undo changes from a commit in a shared branch without affecting other contributors.
- **Avoid rewriting history:** git revert is useful when you want to undo changes but **don't want to modify the commit history**, which is particularly important in collaborative environments.

STASHING

Git Stashing is a way to temporarily save changes in your working directory that you are not ready to commit, allowing you to switch branches or perform other tasks without losing those changes. You can later apply or pop these stashed changes back into your working directory when you are ready.

```
git stash
```

```
git stash save "Work in progress on feature X"
```

```
git stash list
```

```
git stash apply
```

```
git stash apply stash@{0}
```

```
git stash pop
```

COMMAADS OF STASHING

mkdir Stash

```
319 cd Sta
```

```
320 cd Stash/
```

```
321 git init
```

```
322 echo "hi" >> file.txt
```

```
323 git add .
```

```
324 git commit -m "hi added"
```

```
325 git remote add origin git@github.com:Akash-Nadigepu/Stash.git
```

```
326 git push -u origin master
327 echo "welcome" > file.txt
328 git pull
329 git stash save "first stash"
330 git status
331 cat file.txt
332 git pull
333 cat file.txt
334 git stash list
335 git stash apply
336 cat file.txt
337 git add .
338 git commit -m "v1"
339 git status
340 git push
341 git stash list
342 git stash pop
343 git stash drop stash@{0}
344 git stash list
345 cat file.txt
346 git stash list
347 git stash save "2nd change"
348 git add .
349 git stash save "2nd change"
350 git stash list
```

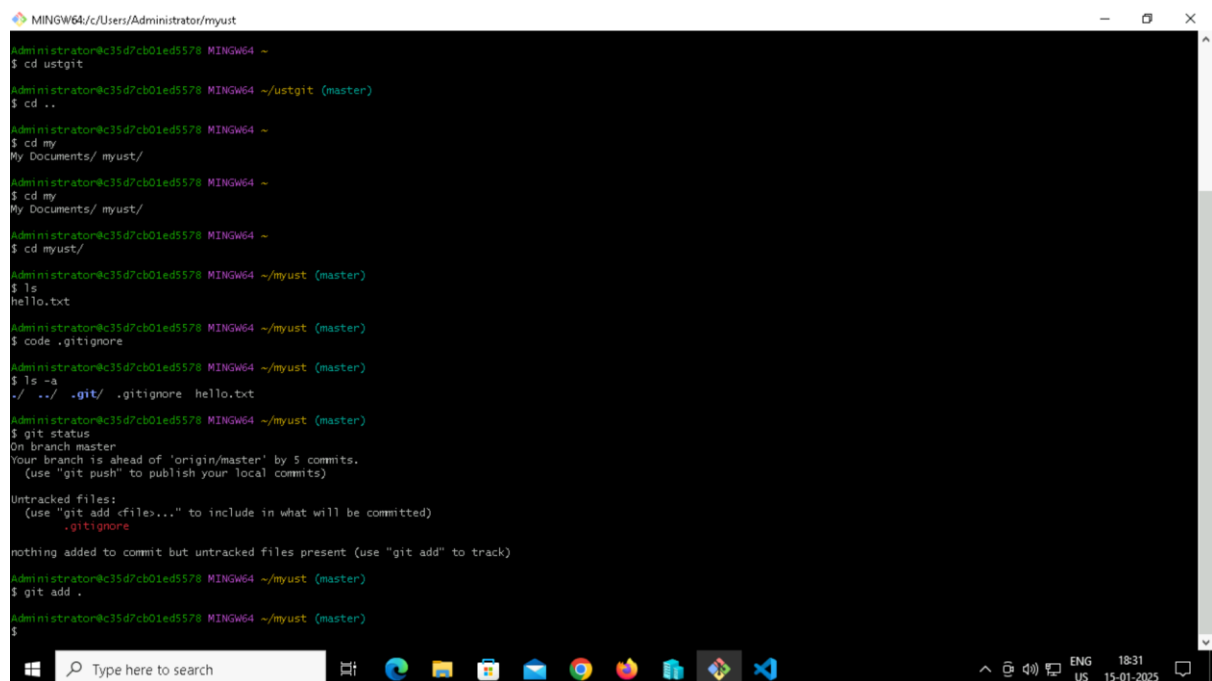


```
351 git stash show stash@{0}
```

```
352 git stash show -p stash@{0}
```

.GITIGNORE

The **.gitignore** file is used to specify which files and directories Git should **ignore** in a repository. It's commonly used to prevent certain files (such as build artifacts, temporary files, or sensitive data) from being tracked by Git.



```
MINGW64/c/Users/Administrator/myust
Administrator@35d7cb01ed5578 MINGW64 ~
$ cd ustgit
Administrator@35d7cb01ed5578 MINGW64 ~/ustgit (master)
$ cd ..
Administrator@35d7cb01ed5578 MINGW64 ~
$ cd my
My Documents/ myust/
Administrator@35d7cb01ed5578 MINGW64 ~
$ cd my
My Documents/ myust/
Administrator@35d7cb01ed5578 MINGW64 ~
$ cd myust/
Administrator@35d7cb01ed5578 MINGW64 ~/myust (master)
$ ls
hello.txt
Administrator@35d7cb01ed5578 MINGW64 ~/myust (master)
$ code .gitignore
Administrator@35d7cb01ed5578 MINGW64 ~/myust (master)
$ ls -la
./  ../ .git/ .gitignore hello.txt
Administrator@35d7cb01ed5578 MINGW64 ~/myust (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
(use "git push" to publish your local commits)

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

nothing added to commit but untracked files present (use "git add" to track)
Administrator@35d7cb01ed5578 MINGW64 ~/myust (master)
$ git add .
Administrator@35d7cb01ed5578 MINGW64 ~/myust (master)
$
```

GIT PULL

The **git pull** command is used to **fetch** and **merge** changes from a remote repository into your current local branch. It's a combination of two commands: **git fetch** (which fetches updates from the remote) and **git merge** (which integrates those updates into your local branch).

PULL REQUEST

A **Pull Request (PR)** is a way of proposing changes to a project in Git-based version control systems like GitHub, GitLab, or Bitbucket. It is used when you want to contribute changes to a project, typically in a collaborative or open-source setting. A PR

allows you to submit your changes, discuss them with others, and get feedback before merging the changes into the main codebase.

GIT TAG

```
MINGW64/c/Users/Administrator/devOps/git/learnGitForUst
Administrator@C35d7cb01ed5578 MINGW64 ~/devOps/git/learnGitForUst (main)
$ git tag -a "learning-tagging" b8ff56c -m abhiram

Administrator@C35d7cb01ed5578 MINGW64 ~/devOps/git/learnGitForUst (main)
$ git tag
@author
learning-tagging

Administrator@C35d7cb01ed5578 MINGW64 ~/devOps/git/learnGitForUst (main)
$ git show @author
tag @author
Tagger: Abhiramkannan cabhiabhiram242@gmail.com
Date: Thu Jan 16 10:24:53 2025 +0530

abhiram

commit b8ff56c1071d7d7f6740bfc843392a2c42a345dd (tag: learning-tagging, tag: @author)
Author: Abhiramkannan cabhiabhiram242@gmail.com
Date: Tue Jan 14 16:06:31 2025 +0530

f4

diff --git a/Desktop - Shortcut.lnk b/Desktop - Shortcut.lnk
new file mode 100644
index 0000000..b8ff56c
Binary files /dev/null and b/Desktop - Shortcut.lnk differ
diff --git a/file1.txt b/file1.txt
index 2f2b3c5..f0938f0 100644
--- a/file1.txt
+++ b/file1.txt
@@ -0,0 +0,0 @@ this is m1 -----
this is m2-----
this m3
gsjjs
-this is f1--this is the change done in main branch
+this is f1--changes done in feature branch
this is f2
\ No newline at end of file

Administrator@C35d7cb01ed5578 MINGW64 ~/devOps/git/learnGitForUst (main)
$ git branch
feature1
* main

Administrator@C35d7cb01ed5578 MINGW64 ~/devOps/git/learnGitForUst (main)
$ |
```