Tutorial 6: Git

CS 108

Spring, 2023-24

Please mark your SAFE Attendance.

TA: Kavya Gupta

Topics to do

- Introduction to Git
- Installing Git
- Concept of Repository
- Working Directory, Local and Origin Repo
- Getting Started with Git
- “git add” and the Staging area
- git commit
- git show
- git diff
- git checkout
- Branching in Git
- git merge
- Resolving Merge Conflicts
- “Undoing” a commit
- Using GitHub with Git

Introduction to Git

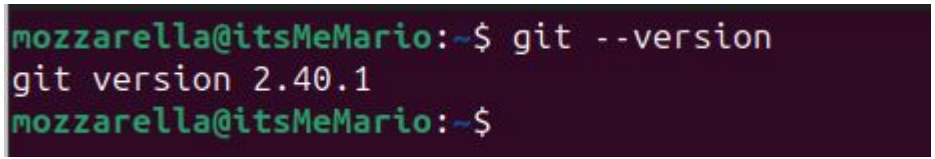
- **What Is Git?**
  - **Git** is a **version control system** (VCS). Imagine it as a **time-travel machine** for your code.
  - It keeps track of every change you make to your files – like a trusty notebook recording your coding journey.
- **Why Use Git?**
  - **Collaboration**: Git lets teams work together seamlessly. Multiple developers can edit the same project without chaos.
  - **Safety Net**: Ever accidentally delete a crucial file? Git's got your back! It stores snapshots of your code, so you can rewind if needed.
  - **Branching Magic**: Create alternate paths (branches) for your code. Experiment without affecting the main project.
  - **Distributed Awesomeness**: Each developer has their own local copy. No more waiting in line to borrow the codebook!

Installation of Git on different OS

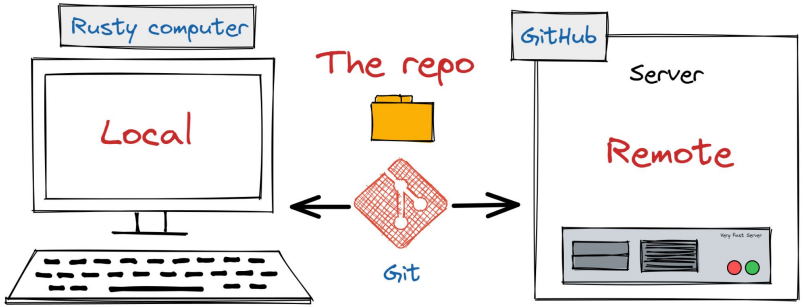
Git should have already installed in your machines as a part of cLab installation. Still these are steps to download it if not already.

- MacOS → Open terminal and type: `brew install git`
- Windows → Follow https://git-scm.com/download/win
- Linux → Type in terminal: `sudo apt-get install git`

Type `git --version` to check if git is installed or not.

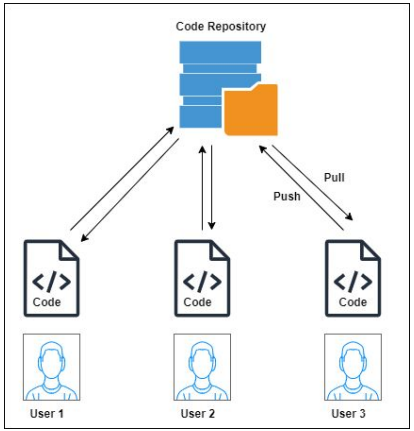


Concept of Repository

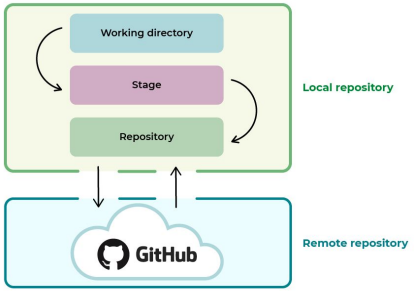


- What is a Git Repository?
  - A **repository** in Git is like a **data structure** that stores metadata for a set of files and directories.
  - It's essentially your project's **home base**, where all the magic happens.
  - Think of it as a **container** that holds not only your code but also the **history of changes** made to those files.
- What Does a Git Repository Contain?
  - **Files**: A repository contains all the files related to your project—source code, documentation, images, and more.
  - **History**: It keeps track of every **commit** you make. Each commit represents a specific state of your project at a given moment.
  - **Branches**: Repositories allow you to create different **branches** to work on separate features or bug fixes.

Working Directory, Local and Origin Repo



- Working Directory:
  - **Definition**: The working directory is your project folder on your computer.
  - **Purpose**: It holds your actual files and directories.
  - **Key Points**:
    - Where you create, edit, and organize files.
    - Changes made here are not automatically part of Git history.
    - Git tracks modifications when you stage and commit them.
- Local Repository:
  - **Definition**: Your local repository resides on your computer. It's where you work with Git.
  - **Purpose**: You make changes, commit them, and manage your project history here.
  - **Key Points**:
    - Contains the entire project history.
    - Includes the working directory (where you edit files) and the .git directory (where Git stores metadata).
    - Acts as your personal sandbox for development.
- Origin Repository:
  - **Definition**: The origin repository typically resides on a remote server (like GitHub).
  - **Purpose**: It serves as the central hub for collaboration and sharing.
  - **Key Points**:
    - Where you push your local changes to share with others.
    - Acts as a reference point for your local repository.
    - Allows collaboration among team members.



Getting Started with Git

```
mozzarella@itsMeMario:~$ git config --global user.name "Kavya Gupta"
mozzarella@itsMeMario:~$ git config --global user.email "22b1053@iitb.ac.in"
mozzarella@itsMeMario:~$
```

1) Configure Your Identity:
   - Set your **global** commit name and email address using the following commands:

```
git config -global user.name “Your name”
git config -global user.email “Your email”
```

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git init
Initialized empty Git repository in /home/mozzarella/Desktop/CS108/Git_Tut/.git/
```

   - Replace `"Your Name"` with your actual name and `"Your email"` with your email address.
   - These settings are important because they are associated with every Git commit you make.

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 5917f5bdfe9f34bf925c1020d15e8741ae414a27 (HEAD -> main)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Thu Feb 8 21:58:54 2024 +0530

    First commit
```

2) Initialize a Git Repository:
   - To create a new Git repository, navigate to your project directory in the terminal and run:
     `git init`
   - This initializes an empty Git repository in the current folder.

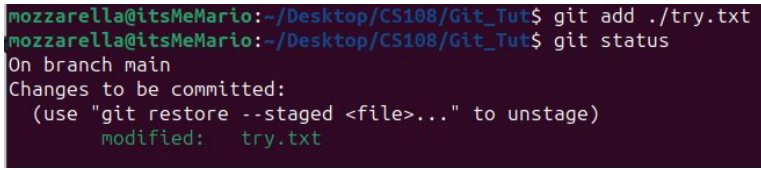
“git add” and the Staging area

- What is `git add`?
  - The git add command is used to **stage** changes in your working directory for the next commit.
  - It marks specific files or directories to be included in the upcoming revision.
  - Think of it as preparing your changes to be committed.
- Staging Area: What is it?
  - The **staging area** (also known as the **index**) is an intermediate step between your working directory and the Git repository.
  - When you run git add, it copies the changes from your work-tree into this staging area.
  - Changes in the staging area are ready to be wrapped up in a new commit using git commit.
  - `git status` can be used to check the status of the staging area
- How to Use git add:
  - To stage a specific file or directory:

    `git add <path>`
  - To stage all files and directories (except those specified in .gitignore):

    `git add .`
  - To interactively choose which parts of changes (hunks) to stage:

    `git add -p`



.gitignore

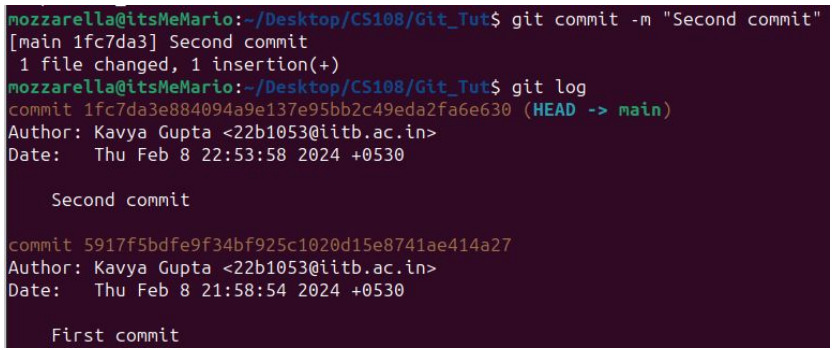
```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ nano .gitignore
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ cat .gitignore
*.log
my_directory/
*.txt
!important_file.txt
```

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git add file.txt
The following paths are ignored by one of your .gitignore files:
file.txt
hint: Use -f if you really want to add them.
hint: Turn this message off by running
hint: "git config advice.addIgnoredFile false"
```

- What is `.gitignore`?
  - A `.gitignore` file is a **plain text file** that tells Git which files or directories to **ignore** when tracking changes.
  - It helps prevent certain files from being accidentally committed to the repository.
- How Does It Work?
  - Each line in a `.gitignore` file specifies a **pattern**.
  - Git checks these patterns to decide whether to ignore a path.
  - Patterns can match specific files, directories, or file types.
- Examples:
  - Ignore all `.log` files:
    `*.log`
  - Exclude a specific directory:
    `my_directory/`
  - Include a previously ignored file:
    `!important_file.txt`

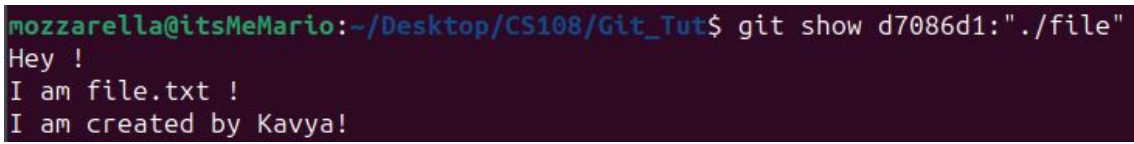
git commit

1) Creating a Commit:
   - Once your changes are staged, create a commit using git commit. The -m flag allows you to add a commit message. For example:
     `git commit -m “Fixed login issue”`
   - Replace "Fixed login issue" with a concise and descriptive message that explains what this commit accomplishes.
2) Skipping the Staging Area:
   - Sometimes, for small changes, using the staging area seems unnecessary. You can directly commit changes without staging by using the -a option:
     `git commit -a -m “Updated index.html with a new line”`
   - Be cautious when skipping the staging area, as it might include unintended changes.
3) Viewing the commit history: Use `git log`



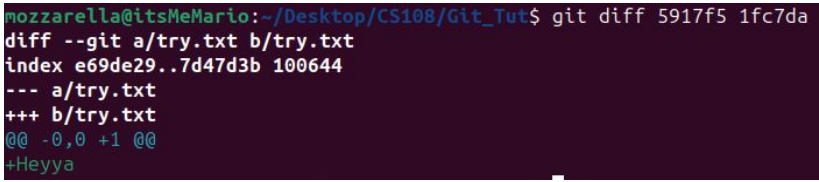
git show

- Viewing Commit Details:
  - To see details of a specific commit, use:
    `git show <commit-hash>`
    - This displays the commit message, changes introduced, and mo
- Listing Files in a Commit:
  - To get a plain list of files modified in a commit:
    `git diff-tree --no-commit-id --name-only -r <commit-hash>`
  - For a user-friendly version (includes commit message), use:
    `git show --pretty="" --name-only <commit-hash>`
- Seeing a file content of a specific commit:
  - `git show <commit-hash>:<path_to_file>`
    - If we use “HEAD” for commit-hash then we’ll access the file from the current working directory.



git diff

- Comparing Working Tree with Index (Staging Area):
  - Use git diff without any options to view the differences between your working tree (current changes) and the index (staging area). This shows what you've modified but haven't yet staged for the next commit:
    `git diff`



- Comparing Two Commits:
  - To compare changes between two specific commits, use:
    `git diff <commit-hash-1> <commit-hash-2>`
    Replace <commit-hash-1> and <commit-hash-2> with the actual commit hashes you want to compare.
- Viewing Staged Changes (Changes Ready for Commit):
  - To see the differences between your staged changes (those in the index) and a specific commit, use:
    `git diff --cached [<commit>]`
    If <commit> is not provided, it defaults to the latest commit (usually HEAD).
- Comparing Working Tree with a Specific Commit:
  - To compare your working tree with a specific commit (e.g., HEAD or another branch), use:
    `git diff <commit> [--] [<path>...]`
    Replace <commit> with the desired commit reference.

git checkout

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git checkout 5917f5
Note: switching to '5917f5'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 5917f5b First commit
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ cat try.txt
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$
```

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 5917f5bdfe9f34bf925c1020d15e8741ae414a27 (HEAD)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Thu Feb 8 21:58:54 2024 +0530

    First commit
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git status
HEAD detached at 5917f5b
```

- Checking Out a Specific Commit:
  - To check out a particular commit, use the following syntax:
    `git checkout <commit-hash>`
    - Replace `<commit-hash>` with the actual hash (SHA-1) of the commit you want to explore.
    - This command places your working tree in the state of that specific commit. You'll be in a state called **"Detached HEAD"**, meaning you're not on any branch but directly at the commit.
- Detached HEAD State:
  - When you check out a commit directly, you're in a **detached HEAD** state. In this mode, any new commits won't belong to any branch.
  - It's useful for inspecting historical states, comparing code, or creating temporary branches.
  - To exit the detached HEAD state and switch back to a branch, use:
    `git checkout <branch-name>`
    - Replace `<branch-name>` with the name of the branch you want to return to.

Branching in Git

- Branching Basics:
  - **Branching** in Git means creating a separate line of development. It allows you to work on features, bug fixes, or experiments without affecting the main codebase.
  - Each branch represents a different version of your project. You can switch between branches to isolate changes and work on them independently
- Creating a New Branch:
  - To create a new branch and switch to it at the same time, use:
    `git checkout -b my-branch-name`
    - Replace my-branch-name with your desired branch name.
- Switching Between Branches:
  - For local branches:
    `git checkout my-branch-name`
  - For remote branches:
    `git checkout –track origin/my-branch-name`
  - Listing Branches:
    `git branch`
- Deleting a Branch:
  `git branch -d my-branch-name`
- HEAD shows the branch we are currently on. It can be seen under `git log`

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git checkout -b "Branch"
Switched to a new branch 'Branch'
```

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git branch
* Branch
  main
```

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 854fe4f9ee0c347a2090d4be632dfdeb8093c4f4 (HEAD -> Branch)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Fri Feb 9 11:42:46 2024 +0530
```

- We go back to a previous commit and create new commits in Detached HEAD State. Git will treat them as a new branch, away from “main” branch.

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git checkout 1fc7da
Note: switching to '1fc7da'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 1fc7da3 Second commit
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 1fc7da3e884094a9e137e95bb2c49eda2fa6e630 (HEAD)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Thu Feb 8 22:53:58 2024 +0530

    Second commit

commit 5917f5bdfe9f34bf925c1020d15e8741ae414a27
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Thu Feb 8 21:58:54 2024 +0530
```

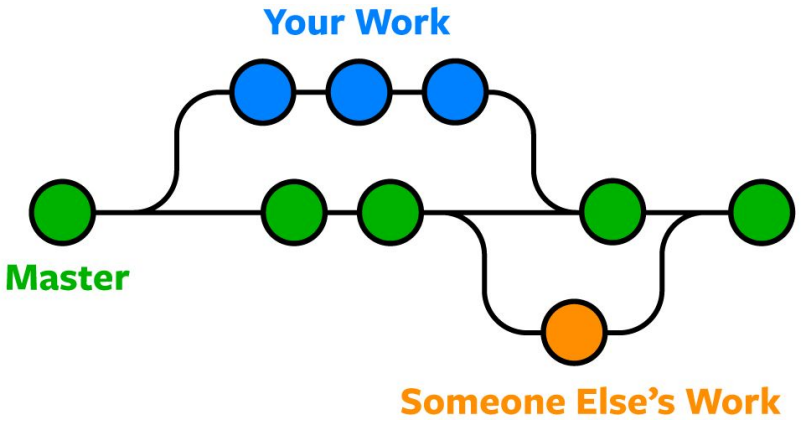
- Before naming/creating a branch, this is what “git branch” show. Detached HEAD commits don’t belong to any branch.

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git branch
* (HEAD detached from 1fc7da3)
  main
```

- After creating the branch, HEAD is no longer in Detached State and the new commits came under the new branch’s log.

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ nano file
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git add .
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git commit -m "Branched commit 2"
[detached HEAD 854fe4f] Branched commit 2
 1 file changed, 1 insertion(+)
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git branch
* (HEAD detached from 1fc7da3)
  main
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git checkout -b "Branch"
Switched to a new branch 'Branch'
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 854fe4f9ee0c347a2090d4be632dfdeb8093c4f4 (HEAD -> Branch)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Fri Feb 9 11:42:46 2024 +0530

    Branched commit 2

commit b27161a6de38bea317ae4b3f245ae4aeb62d89a4
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Fri Feb 9 11:42:04 2024 +0530

    Branched commit 1

commit 1fc7da3e884094a9e137e95bb2c49eda2fa6e630
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Thu Feb 8 22:53:58 2024 +0530

    Second commit

commit 5917f5bdfe9f34bf925c1020d15e8741ae414a27
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Thu Feb 8 21:58:54 2024 +0530

    First commit
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git branch
* Branch
  main
```

git merge

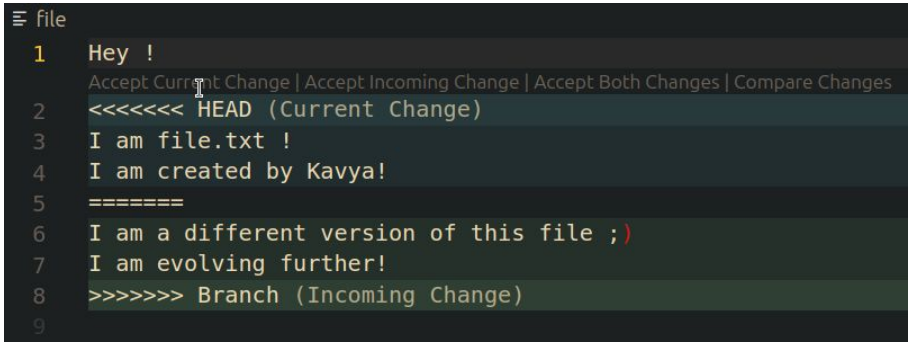



```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git checkout main
Switched to branch 'main'
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git merge Branch
Auto-merging file
CONFLICT (add/add): Merge conflict in file
Automatic merge failed; fix conflicts and then commit the result.
```

- git merge:
  - Merging combines changes from different branches into a single branch.
  - To merge a development branch into the **current branch**, use:
    `git merge dev-branch-name`
    - If there are **no conflicts**, Git automatically creates a new commit with the merged changes.
    - If there are **conflicts**, manual intervention is needed.

Resolving Merge Conflicts

- Merge conflicts occur when competing changes clash in the same file.
- Here's how to handle them:
  - **Competing Line Change Conflicts**:
    - Open the conflicted file in a text editor.
    - Look for markers like <<<<<<< HEAD, =======, and >>>>>>> BRANCH-NAME.
    - Choose which changes to keep or create a new version. (**Use VSCode for this!**)
    - Add and commit the resolved file:
      git add .
      git commit -m "Resolved Merge Conflict"
  - **Removed File Conflicts**:
    - If one branch deletes a file and another modifies it, resolve by keeping the modified version.
    - Add and commit the changes.




```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git add .
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git commit -m "Resolved Merge Conflict"
[main 457b5ce] Resolved Merge Conflict
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 457b5ce35d05ddfa46e02d2608921c4626d24f67 (HEAD -> main)
Merge: d7086d1 854fe4f
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Fri Feb 9 12:24:53 2024 +0530

    Resolved Merge Conflict

commit 854fe4f9ee0c347a2090d4be632dfdeb8093c4f4 (Branch)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Fri Feb 9 11:42:46 2024 +0530

    Branched commit 2
```

“Undoing” a commit

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ echo "Hello again !" >> file
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git add .
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git commit -m "Fifth commit"
[main 4399dc0] Fifth commit
 1 file changed, 1 insertion(+)
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit 4399dc0d2042c03ffaf096e87db66f8294cda288 (HEAD -> main)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Mon Feb 12 12:18:19 2024 +0530

    Fifth commit
```

```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git revert 4399dc0
[main e551534] Revert "Fifth commit"
 1 file changed, 1 deletion(-)
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ git log
commit e551534e525559fe6ec454ce1b2160faf41e251e (HEAD -> main)
Author: Kavya Gupta <22b1053@iitb.ac.in>
Date:   Mon Feb 12 12:18:43 2024 +0530

    Revert "Fifth commit"

    This reverts commit 4399dc0d2042c03ffaf096e87db66f8294cda288.
```

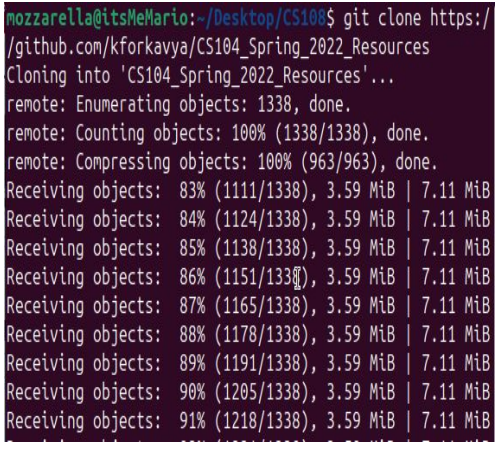
```
mozzarella@itsMeMario:~/Desktop/CS108/Git_Tut$ cat file
Hey !
I am a different version of this file ;)
I am evolving further!
```

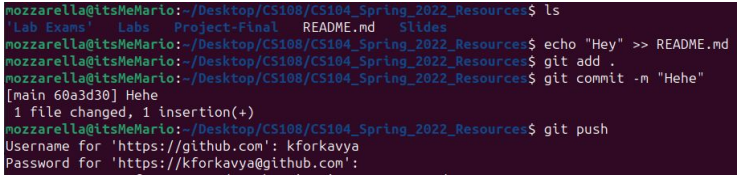
- “Hello again !” line was reverted from “file” !

- Git Revert:
  - Creates a **new commit** that undoes the changes introduced by a previous commit.
  - Adds this new commit to the project history without modifying existing history.
  - Useful when you want to keep a record of the “undo” action.
  - Suppose you have a commit with the hash abc123 that introduced a bug. To revert it, use:
    `git revert abc123`
    - This will create a new commit that undoes the changes from abc123.
- Git Reset:
  - A more complex command with different behaviors based on how it’s invoked.
  - Modifies the index (staging area) or changes which commit a branch head points to.
  - Can alter existing history by changing the commit a branch references.
  - To unstage changes (move them back to working directory) from the last commit:
    `git reset HEAD`
  - To discard the last commit (use with caution, as it rewrites history):
    `git reset --hard HEAD~1`
  - To move the branch pointer to a specific commit (e.g., ghi789):
    `git reset --hard ghi789`

Using GitHub with git

- Clone from GitHub:
  - **Command**: git clone <repository_url>
  - If you're working with a repo that is cloned from GitHub, no need for git init
  - **Explanation**: This command fetches the entire repository from GitHub and creates a local copy on your machine.
  - **Example**: git clone https://github.com/kforkavya/CS104_Spring_2022_Resources
- After adding and committing your changes, you can push them:
  - **Command**: git push
  - **Explanation**: Pushing sends your local commits to the remote repository (GitHub). It's like saying, "Here are my changes; update the online version!"
  - **Example**: git push origin main
- Pull Latest Changes:
  - **Command**: git pull
  - **Explanation**: Pulling fetches any new changes from the remote repository and updates your local copy. It's like saying, "Give me the latest stuff!"
  - **Example**: git pull origin main






Thank You!