Git Distributed Version Control System

1. Introduction to Git

- What is Version Control?
- Git vs. Centralized VCS (CVCS, SVN)
- Key Terminology (Repository, Commit, Branch, Clone)

2. Setting Up Git

- Installation & Configuration (`git config`)
- User Identity Setup (Name, Email, Editor)
- Line Ending Handling (`core.autocrlf`)

3. Basic Git Workflow

- Initializing a Repository (`git init`)
- File States (Untracked, Staged, Committed)
- Staging Changes (`git add`)
- Committing (`git commit`)
- Skipping the Staging Area (`git commit -a`)

4. Viewing and Managing History

- Commit Logs (`git log`, `git log --oneline`)
- Inspecting Changes (`git show`, `git diff`)
- Checking Repository Status (`git status`)

5. Branching

- Creating & Switching Branches (`git branch`, `git switch`)
- Renaming & Deleting Branches (`git branch -m`, `git branch -d`)
- Comparing Branches (`git diff branch1..branch2`)

6. Merging and Conflict Resolution

- Fast-Forward Merge
- 3-Way Merge
- Merge Conflicts (Identification & Resolution)
- Aborting a Merge (`git merge --abort`)

7. Remote Repositories (GitHub)

- Cloning a Repository (`git clone`)
- Adding Remotes (`git remote add`)
- Pushing & Pulling Changes (`git push`, `git pull`)
- Fetching Updates (`git fetch`)

8. Collaboration Workflows

- Forking Repositories
- Pull Requests (PRs)
- Issues & Milestones

09. GitHub Integration with VS Code

- Cloning Repositories in VS Code
- Committing & Pushing from VS Code
- Managing Pull Requests
- Conflict Resolution in VS Code

10. Best Practices & Troubleshooting

- Meaningful Commit Messages
- Branch Naming Conventions
- Handling Large Repositories
- Recovering Lost Commits (`git reflog`)

Chapter 1: Introduction to Git

What is Version Control?

A version control system tracks changes to files over time, allowing developers to:

- See who made changes and when

- Revert to previous versions if something breaks

- Collaborate without overwriting each other's work

Evolution of Version Control Systems

Centralized Version Control (CVCS)

Before Git became popular, most teams used centralized systems like SVN or CVS. These systems worked by having a single central server that stored all versions of every file. Developers would "check out" files from this server to work on them locally, then "check in" their changes back to the server.

The centralized approach had several limitations:

1. Single Point of Failure

Since all files and history were stored on one server, if that server went down, developers couldn't save their changes or access project history until it came back up.

2. Network Dependency

Every operation—viewing history, comparing versions, or saving changes—required connecting to the central server. This made work slow and impossible without internet access.

3. Collaboration Challenges

These systems often used file locking to prevent conflicts, meaning only one person could edit a file at a time. When multiple people did modify the same files, resolving conflicts was difficult and error-prone.

4. Performance Issues

As projects grew larger and accumulated more history, common operations like checking out code or viewing changes became increasingly slow.

5. Limited Branching

Creating branches was so cumbersome that many teams avoided branching altogether, making it hard to work on multiple features simultaneously.

The Git Revolution

In 2005, Linus Torvalds created Git to address these limitations. Git introduced a distributed model that fundamentally changed version control:

1. Full Local Copies

Instead of relying on a central server, every developer gets a complete copy of the repository, including all files, branches, and history. This means you can work entirely offline if needed.

2. Faster Operations

Because nearly all operations happen locally, Git is extremely fast. Committing changes, switching branches, or viewing history takes seconds regardless of project size.

3. Better Collaboration

Git makes branching and merging so easy that developers frequently create branches for new features or bug fixes. The system handles parallel work gracefully, with clear tools to resolve any conflicts.

4. Data Safety

With every clone being a full backup, there's no single point of failure. Even if the main server is destroyed, the project can be restored from any developer's machine.

5. Flexible Workflows

It supports various collaboration models, from small team projects to massive open-source efforts with thousands of contributors. The staging area lets developers precisely control what changes get committed.

Key Git Concepts

Repository: The database storing your project's files and their complete history

Commit: A snapshot of your changes at a specific point in time

Branch: An independent line of development (like a parallel universe for your code)

Clone: Copying a remote repository to your local machine

Why Git Dominates Today

Git's advantages explain why it became the industry standard:

- Free and open-source with an active community

- Blazing fast even for huge projects

- Reliable with built-in data integrity checks

- Flexible enough for any team size or workflow

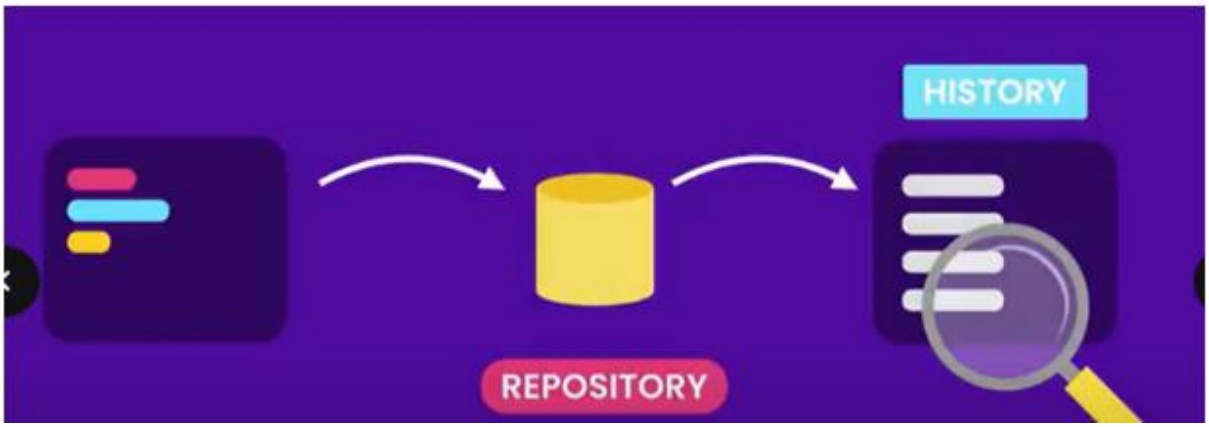
- Powerful tools for tracking and managing changes

The distributed model fundamentally changed how developers work, enabling more experimentation (through branching) and smoother collaboration than was possible with older systems. While there's a learning curve, Git's benefits make it essential for modern software development.

Overall, Git's distributed nature, efficient branching and merging, fast performance, and support for non-linear development workflows have made it the de facto standard for version control in modern software development. Its flexibility and robustness make it ideal for collaborative projects of any size, from small personal projects to large enterprise applications.

A version control system records the changes made to our code over time—using a special database called a repository. We can see the project history: who made what changes, when, and why. If we make a mistake, we can revert our project back to an earlier state.

9. Improved flow and readability



Simply we can say that Track History and Work Together

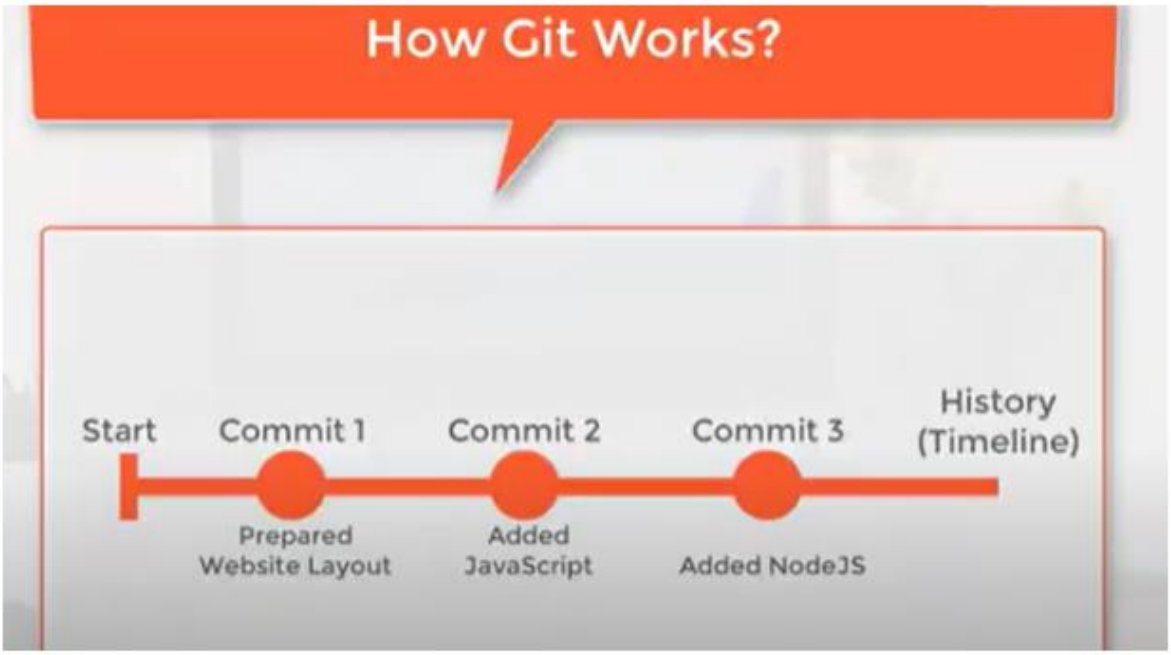
Centralised version system

Distributed version system

Git is the most popular version control system

because

- Free
- Open Source
- Superfast
- Scalable



Note Branching and Merging are slow in other version control system like subversion But Gi is very Fast

5. Using Git

- The Command Line
- Common editors like VS Code(extension git lens)
- Graphical Interfaces
- Gitkraken Git Gui and sourcetree gui (available for windows and mac)

Chapter 2. Setting Up Git

Installation Essentials

1.Download and Install

- All Platforms: Get the latest version from [git-scm.com/downloads](https://gitscm.com/downloads)

- Linux Users: Install via package manager (`apt`, `yum`, or `dnf`)

2.Verify Installation

```bash
git --version
```

Expected output shows your installed version (e.g., `git version 2.40.1`)

Core Configuration (Do This First)

Every Git user must configure these foundational settings:

```bash
git config --global user.name "Your Professional Name"
git config --global user.email your.company@email.com
```

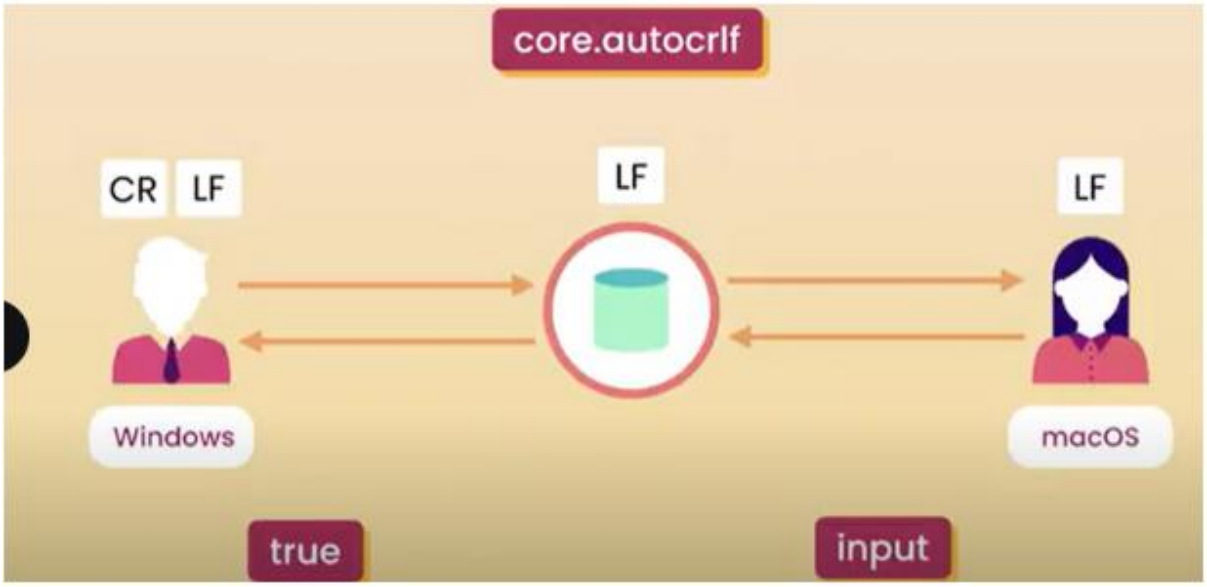
3.git config --global core.editor “code --wait”(wait until we close the window)

4.git config --global -e

5.How git manage end of line

- end of line .....abc\r\n(carriage return and Line feed for windows)

Configure property core.autocrlf(Git Only modify end of line)
``



Why This Matters

These details appear in all your commits and are critical for team accountability and traceability in corporate environments.

Configuration Hierarchy Explained

Git settings cascade through three levels of precedence:

1.System-Level

- Applies to all users on the computer

- Stored in system directories (e.g., `/etc/gitconfig` on Linux)

- Rarely used in practice

```bash

git config --system core.editor "vim"

```

2. Global-Level (Recommended for Most Users)

- Applies to your user account across all projects

- Stored in your home directory (`~/.gitconfig`)

- Used for personal preferences

```bash
```

git config --global core.autocrlf true

```

## 3.Local-Level

-Applies only to a specific repository

- Stored in `.git/config` within the repo

- Overrides global settings when needed

```bash

git config --local user.email "project-specific@email.com"

```

Professional Development Setup

1. IDE Integration

Configure your preferred editor for commit messages:

```bash

VS Code (Cross-Platform)

git config --global core.editor "code --wait"

Alternative Editors

git config --global core.editor "nano"

Beginner-friendly

git config --global core.editor "vim"

Advanced users

```

2. Cross-Platform Line Ending Handling

Prevent invisible whitespace issues:

```bash

Windows Developers (Converts CRLF → LF on commit)

git config --global core.autocrlf true

Mac/Linux Developers (Keeps LF as-is)

git config --global core.autocrlf input

```

3. Security Best Practices

```bash
Cache credentials for 8 working hours (More secure than storing)
git config --global credential.helper "cache --timeout=28800"

Enable GPG commit signing (Corporate compliance)
git config --global commit.gpgsign true
```

Verification and Troubleshooting

Check your effective configuration:

```bash

View all active settings
git config --list --show-origin
Test editor configuration
git config --global -e
```

3. Basic Git Workflow

Staging area or index (staging area allow to review our work before recording a snapshot)

If some of the changes shouldnot be recorded as part of the next snapshot we can unstage them and commit them as part of another snapshots Thats the Basic workflow
```

EXAMPLE:

Hands On

mkdir Moon001

cd Moon001

First Initialized the new empty repository

git init

inside it we have subdirectory .git/ by default it is hidden

- ls
- ls -a (.git)

git add file1 file2 ..files are in the staging area

we review it if everything is ok we commit it or taking a snapshot.

git commit -m “initial commit”

we supply meaningful message to indicate what this snapshot represents.It is necessary for having a essential history .So we fix bugs, implement new features and refactor our code we make commit and each commit clearly explains the state of the project . At That point we have one commit in our repo.

Question

Once we commit the changes the staging area becomes empty???

Answer= Its not correct .we are in a staging area the same snapshot that we stored in the repository.Staging area is very similar to a staging environment .we use when releasing a software to production. Its either a

reflection of what we are currently have in production or the next version that are going to go in production.

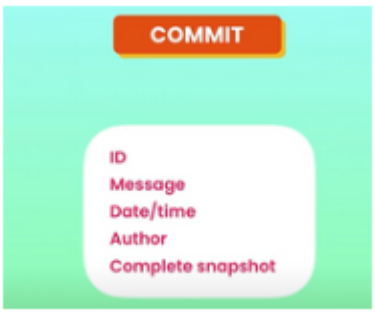
Example:

Lets say as part of fixing the bug we make some changes to file1.note that what we have in staging area is the old version of file1 because we have not staged the changes yet .so once again

- git add file1(so now we have same in staging area thats on working directory) + git commit -m “Fixed the bug that”(now we have 2 commit)

1. Now we no longer need file2 that contain unused code.
2. so we deleted in working directory but it still in Staging area
3. so in this case again use git add file2(in this case deletion)
4. git commit -m “Removed unused code”

- Now we have three commit
- Commits Contains a

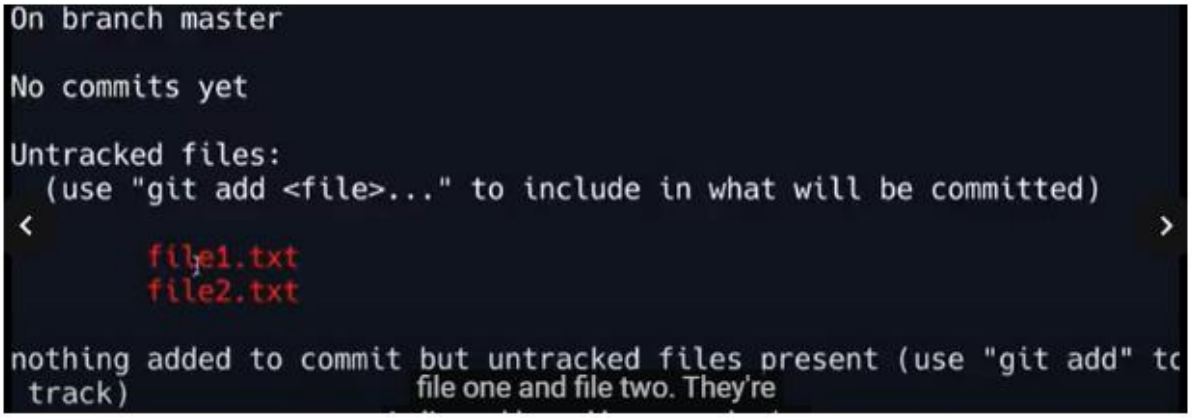


It stores full content of the Project .git is efficient in data storage

- It compresses the content and does not store duplicate Content

Staging File

- echo hello >file1.txt
- echo hello> file2.txt
- git status



- untracked file
- git add file1.txt file2.txt or git add . or git add *.txt
- git status

- Now It is in staging Area
- Let Me Show you one interesting thing if we modify the file1 what happen
- echo world >>file1.txt
- git status

- we run git add file1.txt
- git status

```
› ✔  git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   file1.txt
        new file:   file2.txt
```

- Now we have files in staging area ok alright now commit
- git commit -m “Initial Commit”

```
› ✔  git commit
[master (root-commit) 3b0003b] Initial commit.
 2 files changed, 3 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

Committing is just like to record checkpoints as we go so if you screw up ,we can always go back and recover our code. so try to commit often


```
› ✔  git commit
[master (root-commit) 3b0003b] Initial commit.
 2 files changed, 3 insertions(+)
 create mode 100644 file1.txt
 create mode 100644 file2.txt
```

Committing is just like to record checkpoints as we go so if you screw up ,we can always go back and recover our code. so try to commit often

Question whether we can skip the staging area

- answer is yes but in practice 99% of the case required staging area to review the changes than commit
- echo test >> file1.txt(>> append)
- git commit -am “ Fix the bug that prevented the users from signing up.”(a=all files and m=message)

```
git commit -am "Fix the bug that prevented the users from sig
ng up."
[master 8f092f7] Fix the bug that prevented the users from signing

 1 file changed, 1 insertion(+)
```

REMOVING FILES

- rm file2.txt(its a standard unix command)
- git status

```
git status
On branch master
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  use "git restore <file>..." to discard changes in working direc
        deleted:    file2.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

git ls -files

```
git ls-files
file1.txt
file2.txt
```

- So git add file2.txt
- git ls -files(now its a staging area)
- file1.txt

```
git status
branch master
hanges to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    file2.txt
```

git commit -m “Remove unused code”

NOW IF WE REMOVE IN ONE STEP WE USED THE COMMAND

git rm file2.txt (it removes from both staging as well as working area)

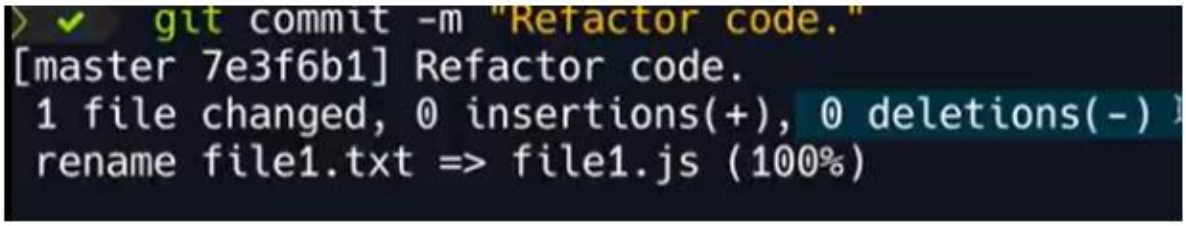
RENAMING AND REMOVING FILES

- ls
- file1.txt
- mv file1.txt main.js
- git status

1. git add file1.txt 2.git add main.js

git mv main.js file1.js

git commit -m “Refactor code”



4. Viewing and Managing History

IGNORING FILES

mkdir logs

echo hello >logs/dev.log

```
git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        logs/
```

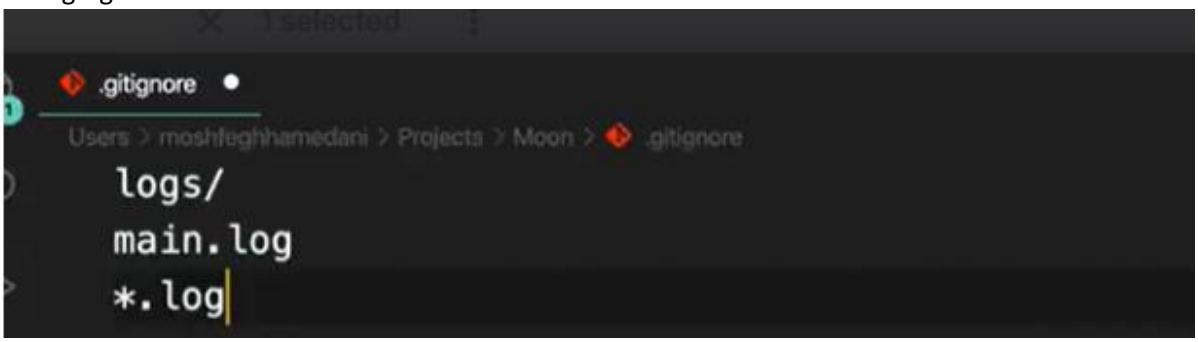
we dont want to add this in staging area because we dont want to track this . To prevent this we used a special file

.gitignore

echo logs/ > .gitignore

Now open this file using vscode

code .gitignore



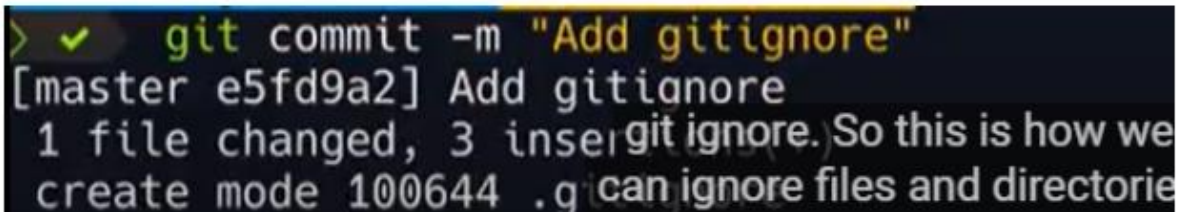
git status

```
git status
On branch master
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)
```

git add .gitignore

git commit -m " Add git ignore"

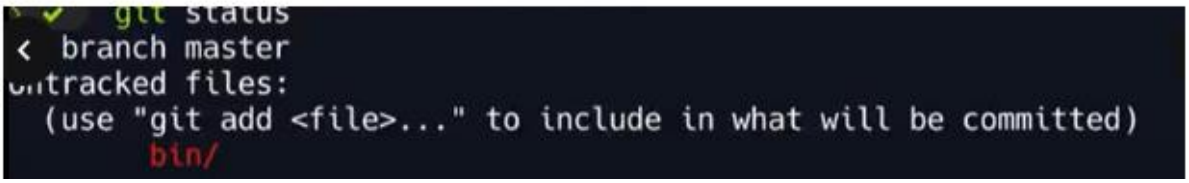


Note:: its only work if you included files in the repository .if u accidentally include the files in the repo and then later added to git ignore ...git not ignore that

mkdir bin

echo hello >bin/app.bin

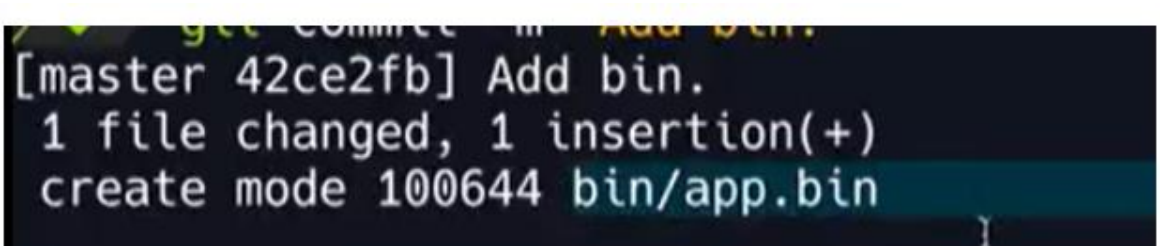
git status



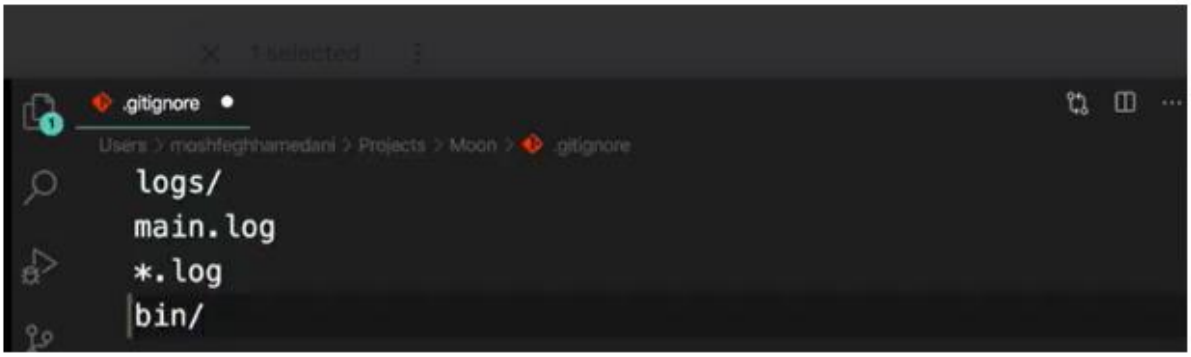
git add .

git commit -m “add bin”

``````````````````




so back to code.gitignore



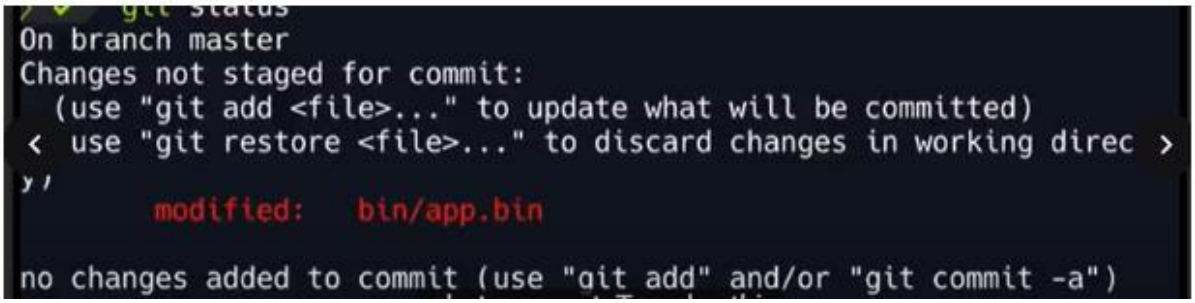

- git status


git add .

git commit -m “Include bin / in gitignore.”

echo helloworld > bin/app.bin
``````````````````

git status



file is modified this is what we dont want to solve this problem so delete or remove this file from staging area

git ls -files

we should remove it here

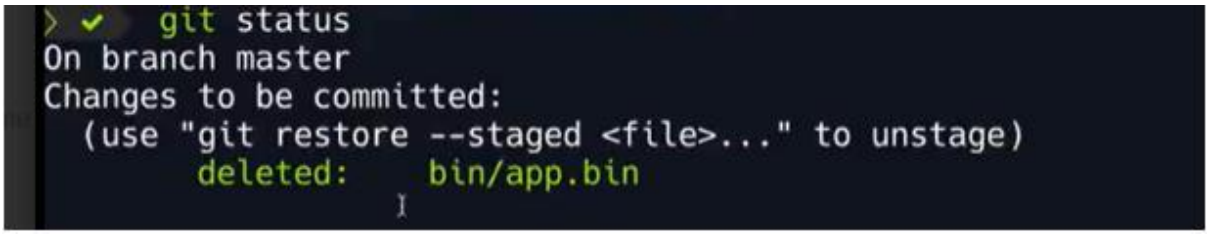
git rm -h (we want to remove it on staging area)

git rm --cached bin/

so git rm --cached -r bin/

git ls-files

git status



git commit -m “Remove the bin directory that was accidentally committed”

```
git commit -m "Remove the bin directory that was acci
ommitted."
[master 921a2ff] Remove the bin directory that was accident
tted.
 1 file changed, 1 dele
```
Okay, from this point forward,

echo test > bin/app.bin

git status

```
git status
On branch master
nothing to commit, working tree clean
```

Note:--->github.com/github/gitignore

example you can ignore any files u want

Status

git status -s

echo sky >>file1.js

echo sky > file2.js

git status

```
git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directo
)
        modified:   file1.js
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        file2.js
```

git status -s

```
git status -s
 M file1.js
?? file2.js
```

left coloumn represent the staging area and right coloumn represent the Staging area

M....modified in coloum 1 we have in working directory not in staging area thats why nothing in left coloumn

?? because file2 is a new file

git add file1.js

git status -s

```
git status -s
M  file1.js
?? file2.js
```

git add file1.js

git status -s

```
git status -s
M  file1.js
A  file2.js
```

A-> represent added

Viewing The Unstaged Changes

Always reviews in a Staging area before commit (best practise)

git diff --staged

```
diff --git a/file1.js b/file1.js
index badfb70..47c3216 100644
--- a/file1.js
+++ b/file1.js
@@ -1,3 +1,5 @@
 hello
 world
 test
+sky
 ean
diff --git a/file2.js b/file2.js
new file mode 100644
index 0000000..f5e95e7
--- /dev/null
+++ b/file2.js
@@ -0,0 +1 @@
```

git diff

git status -s

```
> ✔  git status -s
M  file1.js
A  file2.js
```

code file1.js or echo hello world >file1.txt

git status -s

```
> ✔  git status -s
MM file1.js
A  file2.js
```

git diff

```
> ✔  git diff
diff --git a/file1.js b/file1.js
index 47c3216..8636dbe 100644
<-- a/file1.js
+++ b/file1.js
@@ -1,4 +1,4 @@
-hello
+hello world
 world
 test
 sky
```

Viewing History

git log

git log –oneline

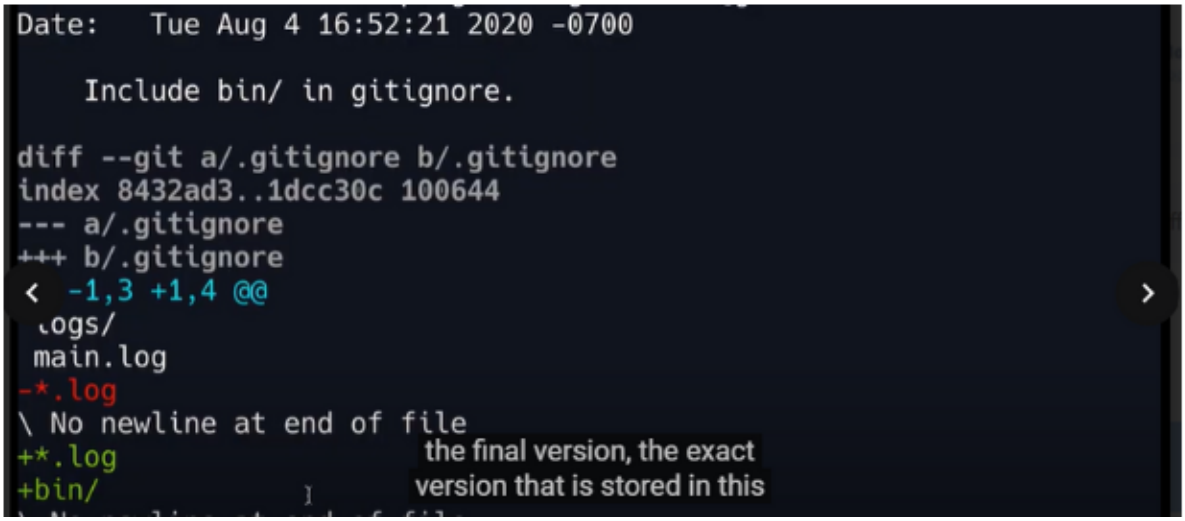
```
> ✔  git log --oneline
921a2ff (HEAD -> master) Remove the bin directory that was accidental
y committed.
d601b90 Include bin/ in gitignore.
42ce2fb Add bin.
e5fd9a2 Add gitignore
7e3f6b1 Refactor code.
< 'c8ef Remove unused code.                                          >
0,092f7 Fix the bug that prevented the users from signing up.
3b0003b Initial commit.
```

git log --oneline --reverse

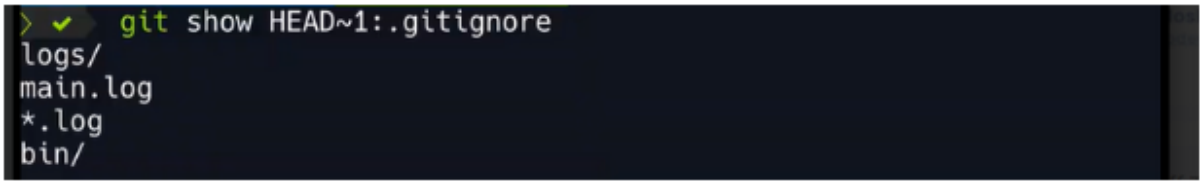
Viewing A Commit

git show d601b

- git show HEAD~1



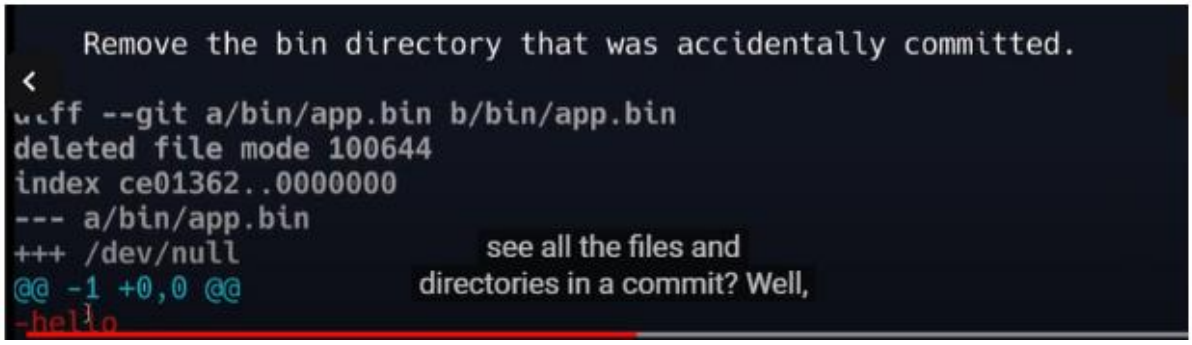
gitshow HEAD~1: .gitignore or git show HEAD~1:bin/app.bin



Each Commit Contains a Complete snapshot of our working directory

we run show command we only see the changes

git show HEAD



git ls-tree (list all files in a tree)

git ls-tree HEAD~1

blob = file

directory= Tree

git show 1dcc30

git show 64629

So using show command we can view an object in git database

Commits

Blobs(Files)

Trees (Directories)

Tags

UNSTAGING FILES

Always Review the stuff that you have in the staging area before making a commit. we realize that the change in file1 shouldnot go in the next commit , perhaps these changes are logically part of a

different task.

```
> ✔  git status -s
MM file1.js
A  file2.js
```

In this case we want to undo the Add operation because earlier we used add command to the staging area we are going to undo this operation.

- git restore --staged file1.js 0r git restore --staged or git restore --staged file1.js file2.js
- git restore --staged file1.js
- git status -

```
> ✔  git status -s
 M file1.js
A  file2.js
```

How Restore Command Works??

Restore command takes the copy from the next environment. so in this case of staging environment what is the next environment ,the last commit? what do we have in the repo so when a restored file1 in the staging area, git took the last copy of this file from the last snapshot and put in the staging area.

```
> ✔  git status -s
 M file1.js
A  file2.js
```

Discarding Local Changes

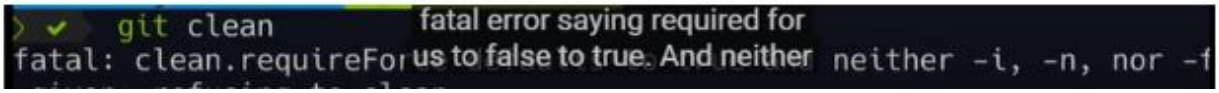
Undo this changes

- git restore file1.js
- git restore .
- git status -s

```
✔  git status -s
?? file2.js
```

file2 is still here why? this is a new untracked file. So Git Hasnot been tracking this. So when we tell Git To reassert this file Git Doesnot know where to get a previous version of his file. It doesnot exist in our Staging environmentt or in our repo so to remove all these new untracked files .

git clean



Git clean -fd

git status -s
Restoring a file to an earlier version

Git tracks a file it stores every version of that file in its database.That means screw things we can always restore a file or a directory to a previous version .

- Task ....Delete a file and how to restore it(18)
- rm file1.js(delete only from working directory)
- used git rm file1.js(deleted both from working directory as well as staging area)
- git status -s

- D.....deleted file in staging area
- git commit -m “Delete file1.js”

now restore this changes

git log –oneline

git restore --source=HEAD~1 file1.js

git status -s

Q=1: What is the difference between centralized and distributed version control systems?

A: Centralized VCS uses a single server for all operations while distributed VCS gives each developer a full repository copy, enabling offline work and faster operations.

Q=2: How do you configure Git to use VS Code as the default editor?

A: git config --global core.editor "code --wait"

Q=3: What command shows the current status of your working directory and staging area?

A: git status

Q=4: Explain the three main file states in Git.

A: Untracked (new files), Staged (changes ready to commit), and Committed (changes safely stored).

Q=5: How do you create and switch to a new branch in one command?

A: git switch -c <branch-name>

Q=6: What is the purpose of the .gitignore file?

A: It specifies files and directories that Git should ignore and not track.

Q: How do you view a condensed version of the commit history?

A: git log --oneline

Q=7: What command would you use to discard local changes in a file?

A: git restore <file>

Q=8: How do you remove a file from Git without deleting it from your working directory?

A: git rm --cached <file>

Q=9: What is the difference between git pull and git fetch?

A: git fetch downloads changes without merging, while git pull downloads and merges changes.

Q=10: How would you restore a deleted file from the previous commit?

A: git restore --source=HEAD^1 <file>

Q=11: What command shows the differences between your working directory and the staging area?

A: git diff

Q=12: How do you amend the most recent commit message?

A: git commit --amend -m "new message"

Q=13: What is the purpose of git stash?

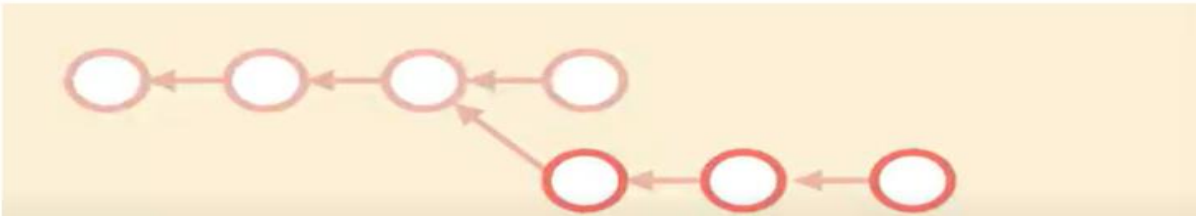
A: It temporarily shelves changes so you can work on something else, then reapply them later.

Q=14: How do you resolve a merge conflict after it occurs?

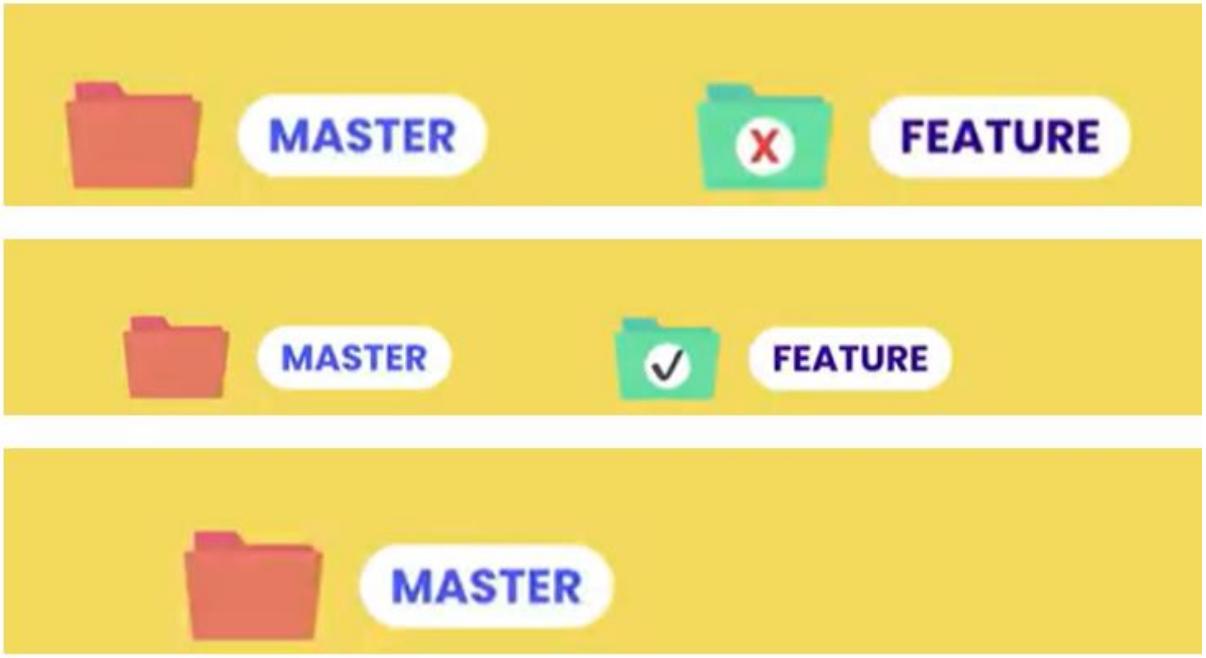
A: Edit the conflicted files, mark them as resolved with git add, then complete the merge with git commit.

Chapter 5. Branching

Branching in **version control systems (like Git)** allows you to **create a separate line of development** that diverges from the main line (often called main or master). This enables you to work on changes **in isolation** without affecting the main codebase.



So we have main workspace



Branching is a **version control mechanism** that allows developers to **work on different tasks in isolation** without disrupting the main (stable) codebase.

Key Concepts:

1. **Master/Main Branch (Stable Workspace)**

Represents the **production-ready** code.

Always kept in a **stable, releasable** state. New team members should start here to ensure they work on a clean, functional codebase.

2. Feature Branch (Isolated Workspace)

   Created to **develop a new feature, fix, or experiment** without affecting main.

   Code here may be **unstable** during development.

   Only merged back into main after **testing and bug fixes**.

3. **Merging (Bringing Changes Back to Main)**

   Once the feature is **complete, tested, and stable**, it is **merged** into main.

   Ensures main remains **clean and deployable** at all times.

Why Use Branching?

Isolation – Work on features/fixes without breaking main.
Stability – Keep main always ready for release.
Parallel Development – Multiple teams can work on different features simultaneously.
Safety – If a feature fails, it doesn't affect the main code.

Git Branches Are Different

The way Git manages branches is **very different** from other version control systems like **Subversion (SVN)**.

In **Subversion**, when you create a new branch, it **makes a full copy** of your entire working directory and stores it separately.

This process is **slow** and consumes extra **disk space**, wasting time.

In **Git**, branching is **lightweight and fast** because:

Git **does not copy all files**—instead, it creates a **pointer** to the current commit.

Branches are **just references** (like bookmarks) that track changes efficiently.

Switching between branches is **almost instant**, even in large projects.

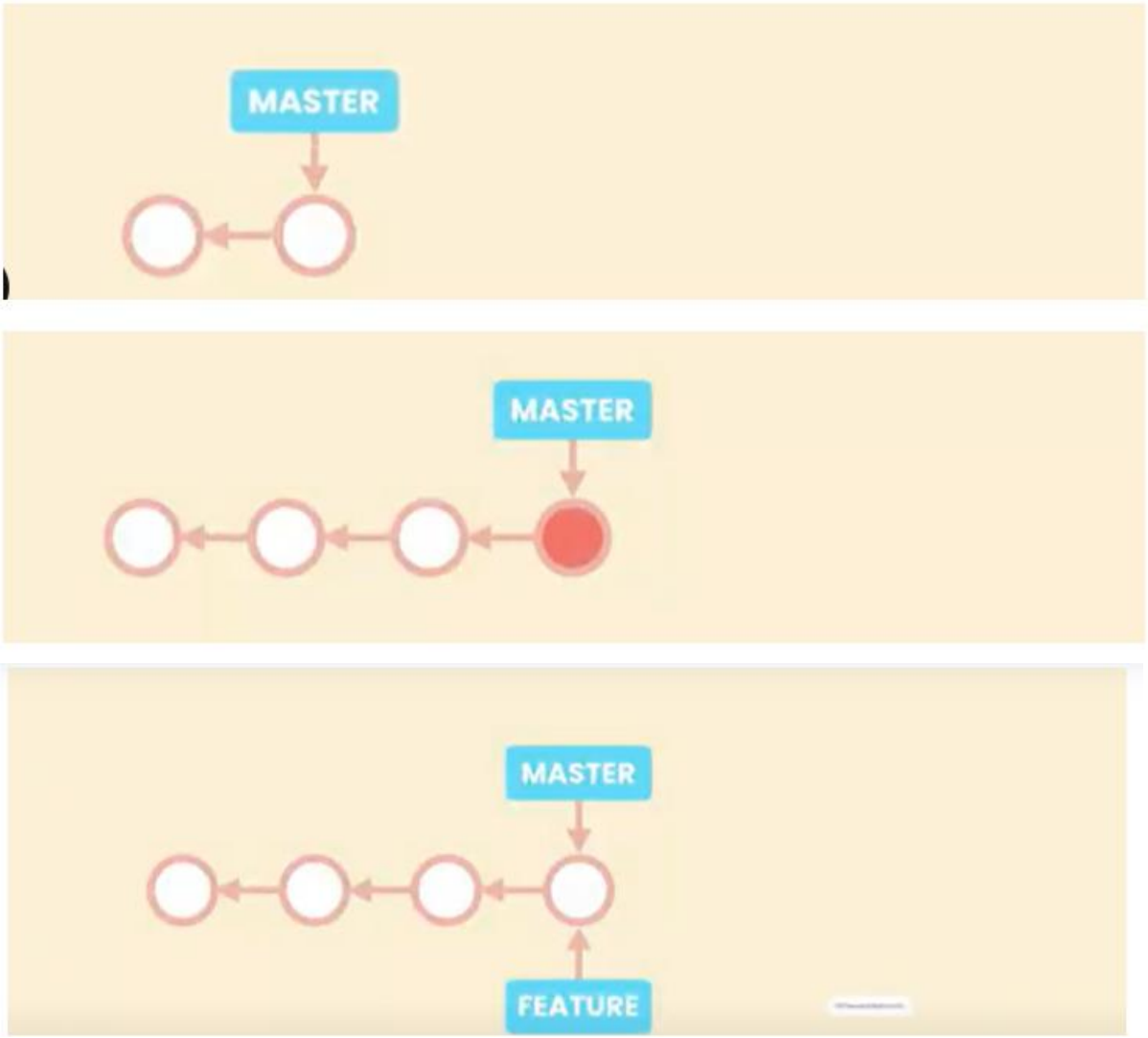
Git Branches Are Super Fast and Cheap

In Git, a **branch is just a pointer to a commit**.

The master **(or** main**) branch** is simply a pointer to the **latest commit** in the main line of work.

Creating a new branch **does not copy files**—it just adds a new pointer.

This makes branching **lightning-fast** and **resource-efficient** compared to older systems like SVN.



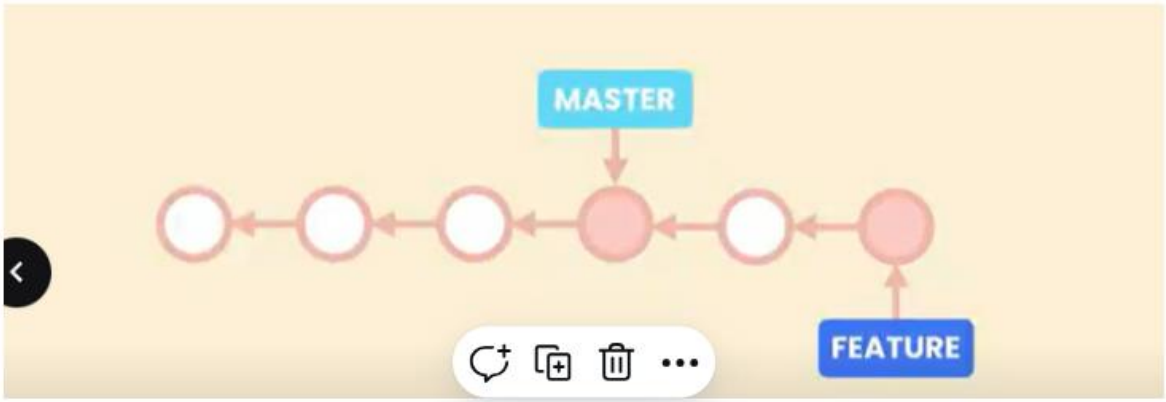
How Git Branches Work

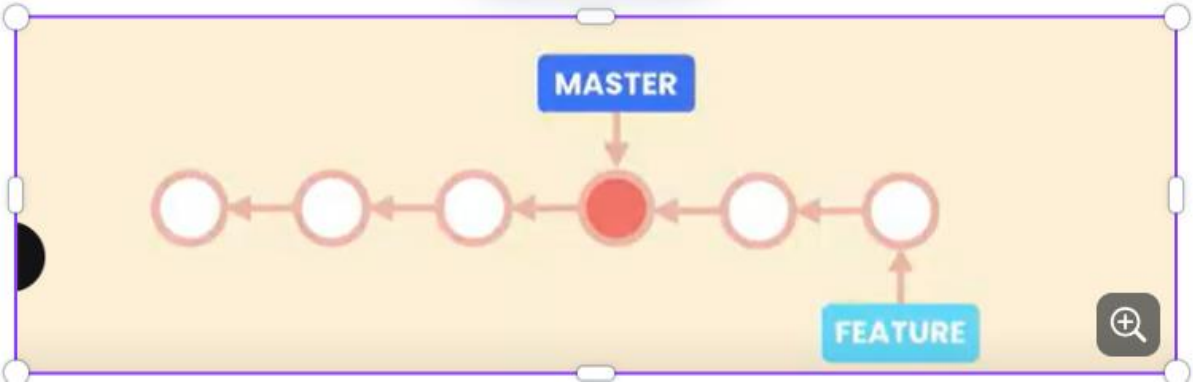
When we make new commits, **this pointer moves forward**, tracking the latest code in the main line of work. Each commit stores a **snapshot** of the project at that point.

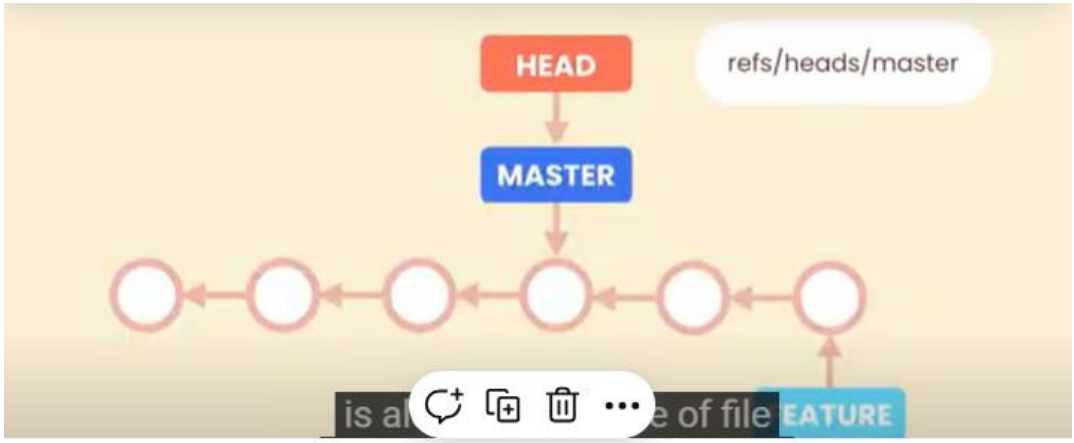
When you create a **new branch**, Git simply creates a **new movable pointer** (like a bookmark).

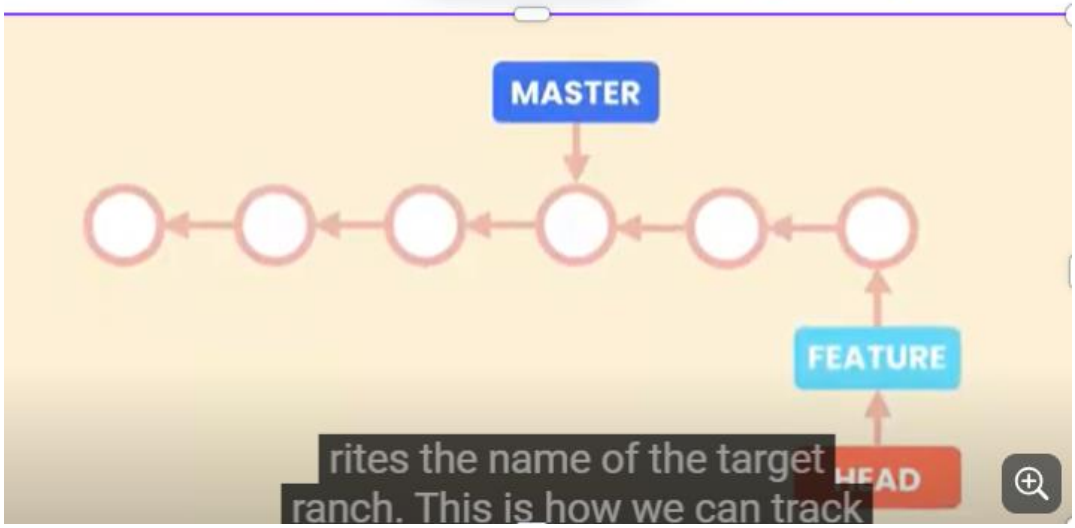
This pointer is just a **tiny file (40-byte SHA-1 hash)** pointing to a commit.

That's why **branching in Git is blazingly fast**—it's just creating a lightweight reference!









How Git Tracks Branches and Commits

When we **switch to a branch** and make new commits, **that branch's pointer moves forward** to track the latest changes, while the master pointer stays in place. This way, Git always knows the latest commit in each branch.

Switching back to master:
Git **restores the working directory** to match the snapshot of the commit master points to.

We always have **a single working directory**—it just updates based on the active branch.

How Git Knows the Current Branch:
Git uses a special pointer called HEAD, which is a small file storing the **name of the current branch** (e.g., ref: refs/heads/master).

Switching Branches:
When you run git checkout <branch> or git switch <branch>, Git:

Moves the HEAD **pointer** to the new branch.

Updates the working directory to reflect that branch's latest commit.

Modifies the tiny HEAD **file** to point to the new branch name.

This is how Git **tracks which branch you're working on** at any time.

Some Hands-On

We just got a bug report . Now to fix this bug

first make a new branch called bug fix

git branch bugfix

git branch

```
git branch
  bugfix
* master
```

1. git status

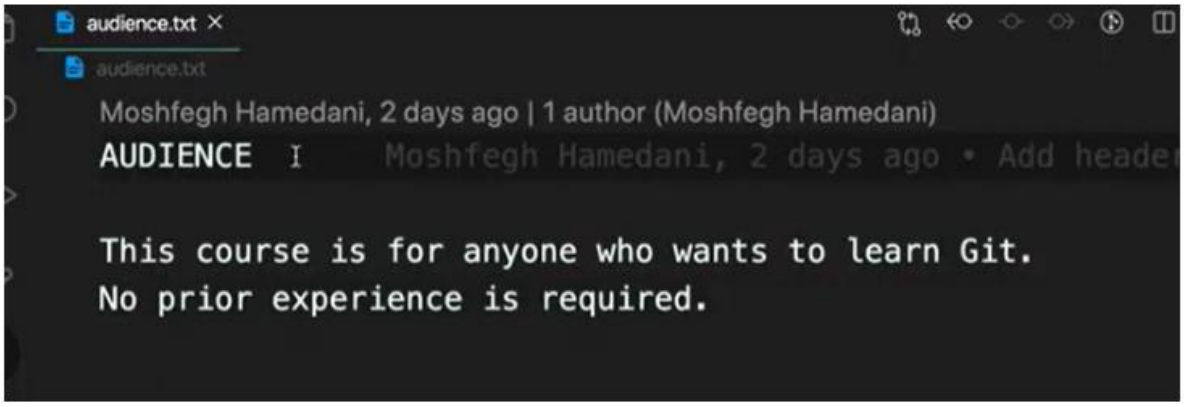
```
git status
On branch master
nothing to commit, working tree clean
```

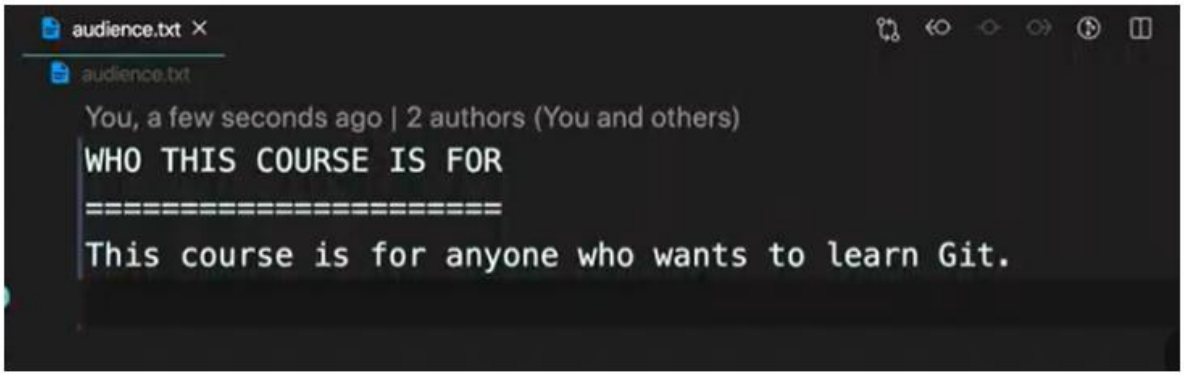
git switch bugfix

```
git switch bugfix
Switched to branch 'bugfix'
```

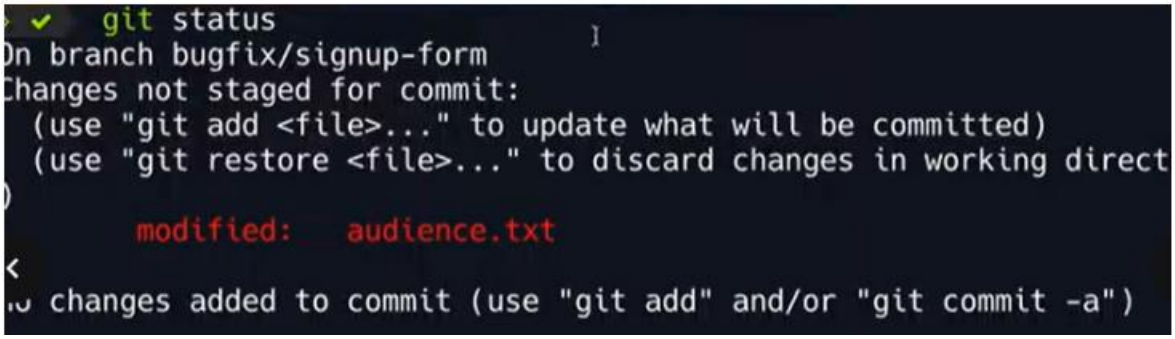
rename the branch
git branch -m bugfix/signup-form

1. code audience.txt





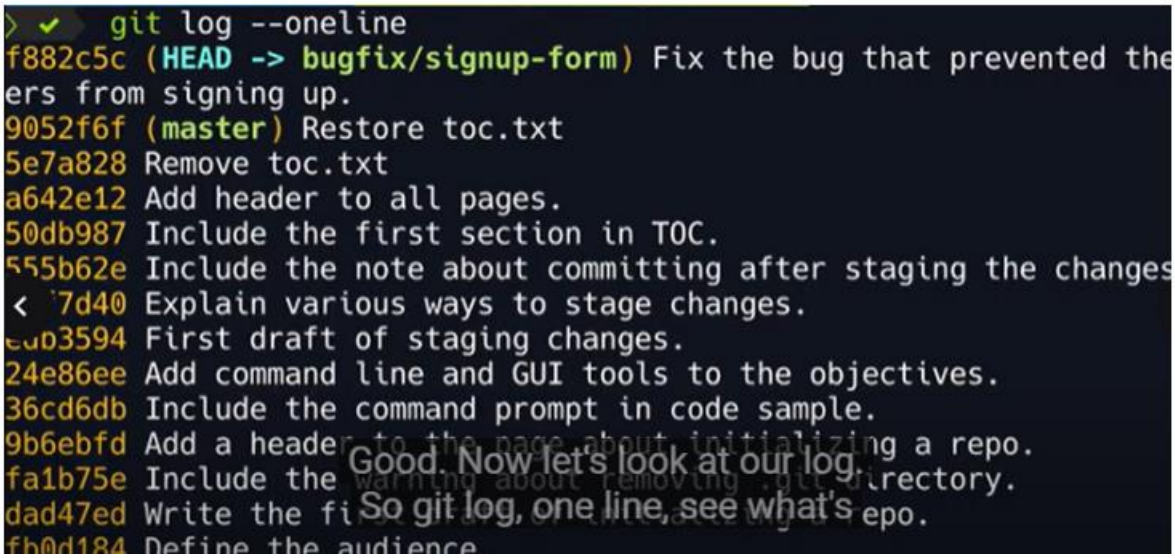
git status



git add .

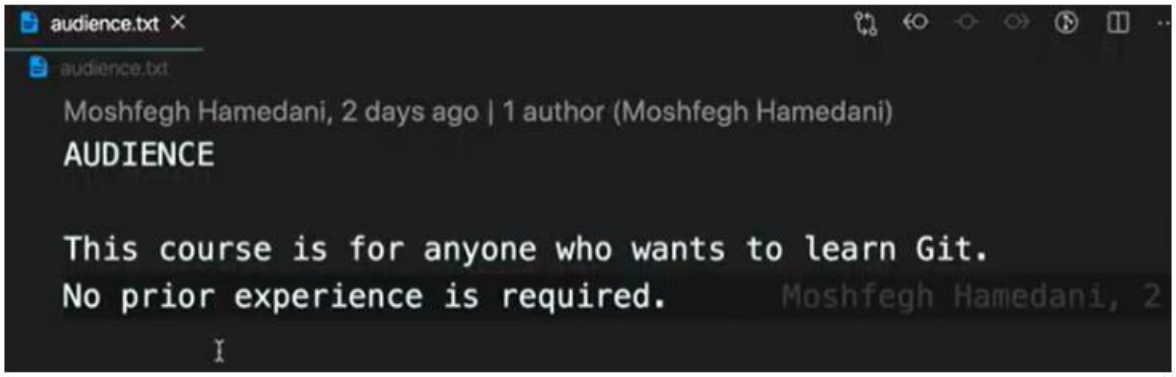
git commit -m “Fix The bug prevented the users from sign-up.”

- git log –oneline

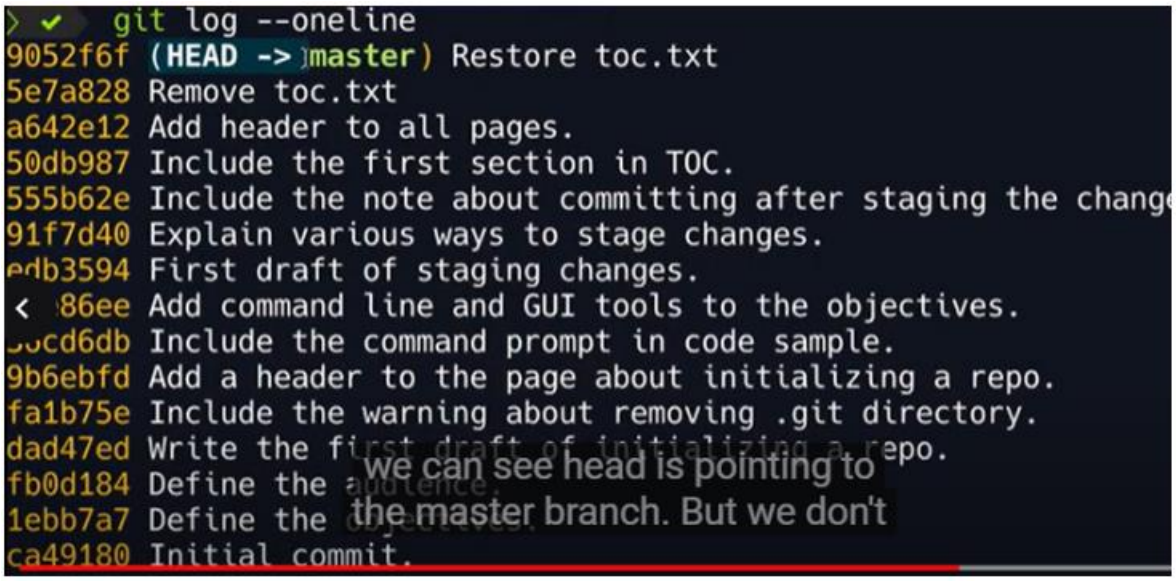


git switch master

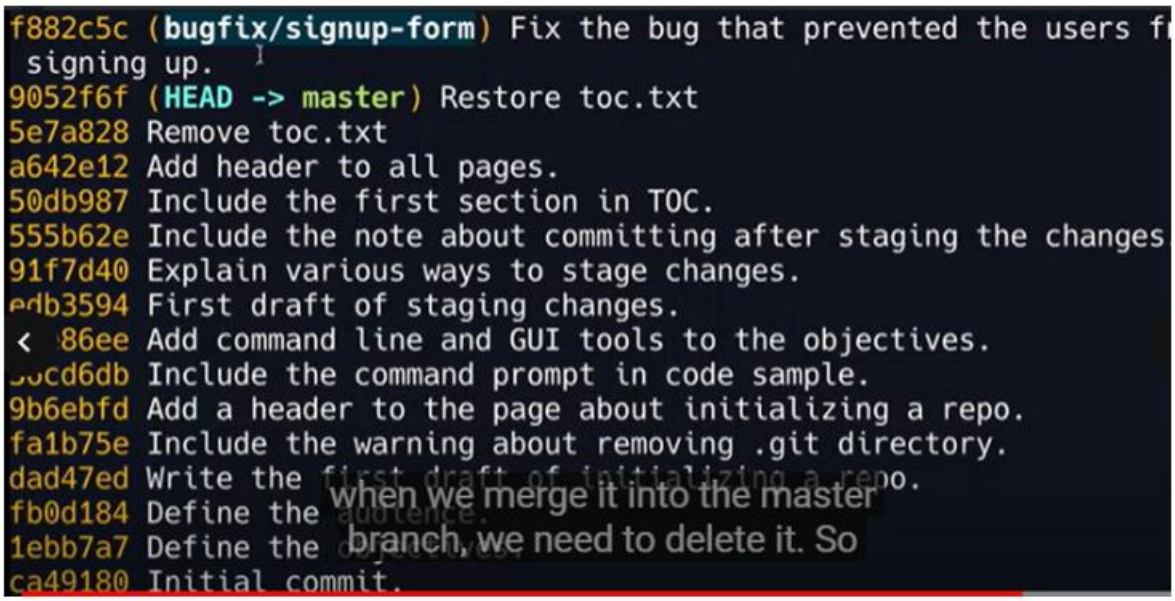
code audience.txt



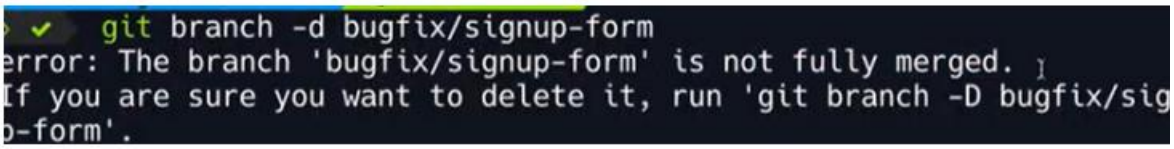
git log –oneline



git log –oneline –all



When we're done with the bugfix/signup-form branch and have merged it into master, we should delete it to keep the repository clean



Error: Branch Not Fully Merged

You're seeing this error because the bugfix/signup-form branch contains changes that haven't been merged into master. By default, Git prevents accidental deletion of unmerged branches to protect your work.

Safe Deletion

First merge your changes:
git checkout master

git merge bugfix/signup-form

git branch -d bugfix/signup-form # Now this will work
If you're absolutely sure you want to discard the unmerged changes:

git branch -D bugfix/signup-form # Note: Capital 'D' forces deletion

COMPARING BRANCHES

- git log master ..bugfix/signup-form

```
> ✔  git log master..bugfix/signup-form
commit f882c5c337fd2a80e8cac5f41a125c68f25e591f (bugfix/signup-form)
Author: Mosh Hamedani <programmingwithmosh@gmail.com>
Date:   Thu Aug 20 14:20:25 2020 -0700

    Fix the bug that prevented the users from signing up.
```

git diff master ..bugfix/signup-form

```
> ✔  git diff master..bugfix/signup-form
diff --git a/audience.txt b/audience.txt
index 4cfef55..709705e 100644
--- a/audience.txt
+++ b/audience.txt
@@ -1,4 +1,3 @@
-AUDIENCE
<
-WHO THIS COURSE IS FOR
+=======================
 This course is for anyone who wants to learn Git.
-No prior experience is required
\ No newline at end of file
```

colon colon bug fix slash signup form. So now we know that once

- git diff bugfix/signup-form

```
> ✔  git diff bugfix/signup-form
diff --git a/audience.txt b/audience.txt
index 709705e..4cfef55 100644
--- a/audience.txt
+++ b/audience.txt
@@ -1,3 +1,4 @@
-WHO THIS COURSE IS FOR
< ==========]==========
+AUDIENCE
+
 This course is for anyone who wants to learn Git.
+No prior experience is required
\ No newline at end of file
```

he changes in terms of the

git diff --name-status bugfix/signup-form

Stashing in Git

When you switch branches, Git **resets your working directory** to match the snapshot from the target branch's latest commit.

The Problem:

If you have **uncommitted local changes** (modified/untracked files), Git:

Prevents branch switching (to avoid losing work).

Shows an error:

error: Your local changes would be overwritten. Commit or stash them.

The Solution: git stash

Stashing **temporarily shelves** your changes so you can switch branches safely:

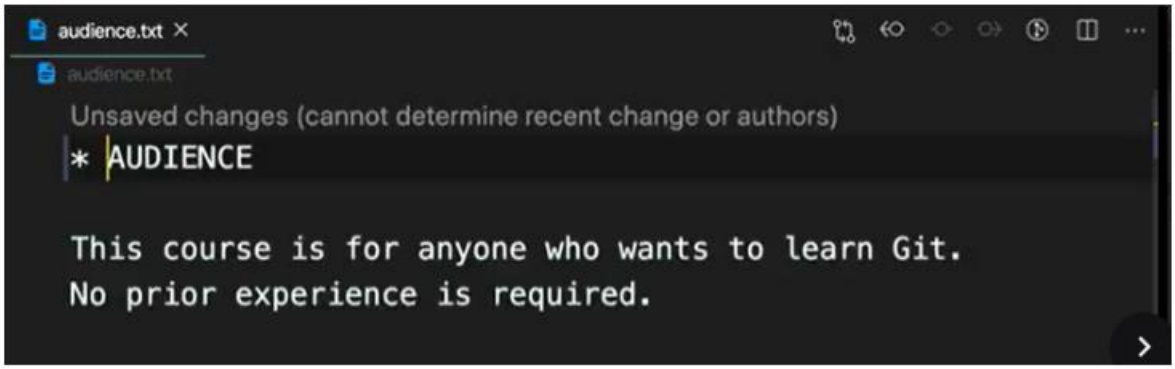
bash

Copy

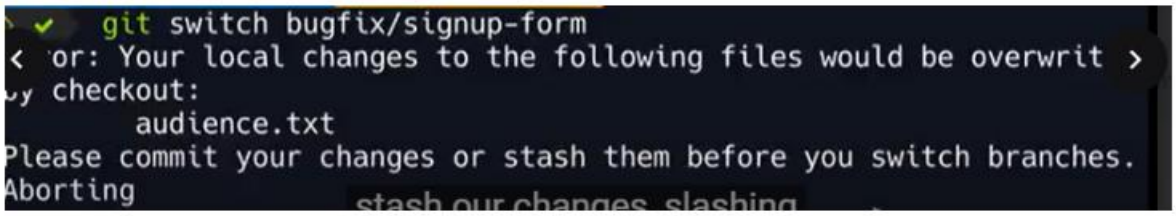
Download

```
git stash        # Saves uncommitted changes
git switch main  # Now you can switch branches
git stash pop    # Restores changes later
```

codeaudience.txt



git switch bugfix/signup-form

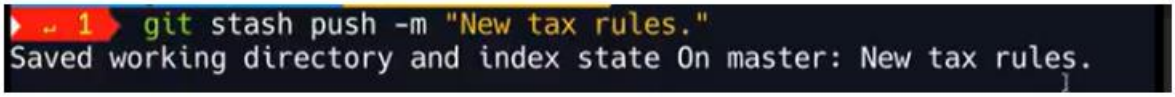


Stashing Changes

Stashing means temporarily storing your uncommitted changes in a safe place within your Git repository. These stashed changes are **not part of your commit history** - they're kept separate until you're ready to use them.

To stash your current changes with a descriptive message:

git stash push -m "New tax rules"



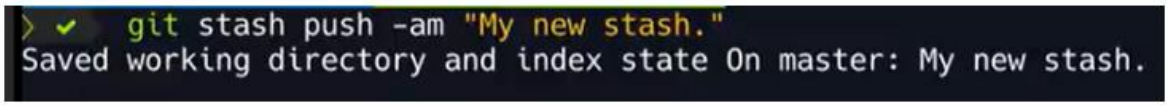
Remember by default new untracked file are not included in your stash

echo hello > newfile.txt

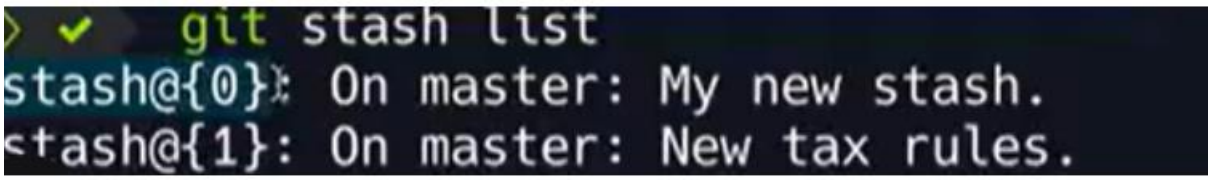
git status -s

?? newfile.txt

git stash push -am “My new stash.”



git stash list



Each stash has a **unique identifier** in the format stash@{n}, where n is the index number in curly braces.

For example:

stash@{0}: On master: New tax rules

stash@{1}: On dev: UI fixes

Switching Branches After Stashing

- Once your working directory is clean (all changes stashed), you can safely switch branches:
  git switch bugfix/signup-form

```
git switch bugfix/signup-form
Switched to branch 'bugfix/signup-form'
```

  git switch master

- git stash show stash@{1} or git stash show 1

```
git stash show 1
audience.txt | 2 +-
1 file changed, 1 insertion(+), 1 deletion(-)
```

  git stash apply 1

```
git stash apply 1
On branch master
nges not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working director
)
        modified:
```

beautiful. Now when we're done

  git stash drop 1(modified audience.txt)

```
git stash drop 1
Dropped refs/stash@{1} (90ad9efd7e0850126a565315e18e271521788dec)
```

  git stash list

```
git stash list
stash@{0}: On master: My new stash.
```

git stash clear

Chapter 6. Merging and Conflict Resolution

Merging is the process of bringing changes from one branch into another. Git supports two primary merge strategies:

Fast-Forward Merge

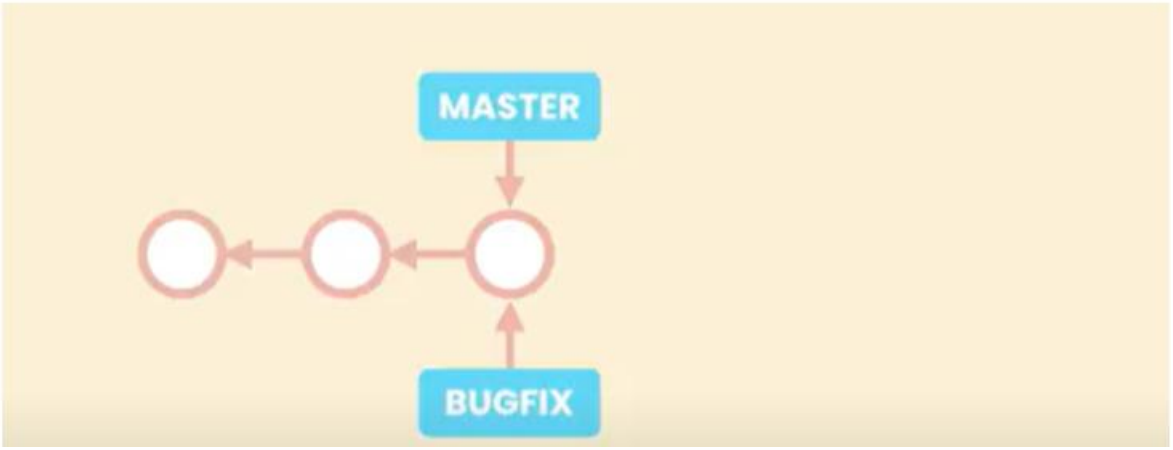
Occurs when the target branch has no new commits since the source branch was created

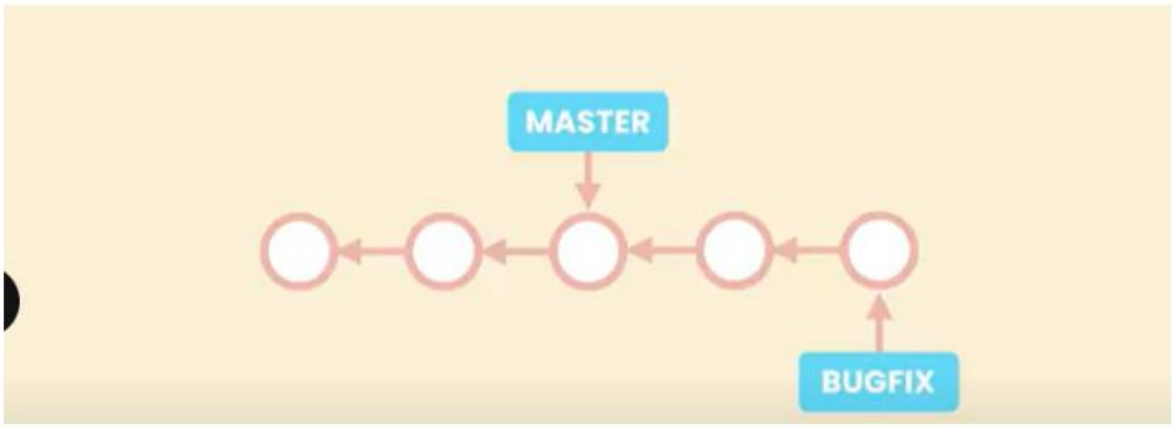
Git simply moves the branch pointer forward

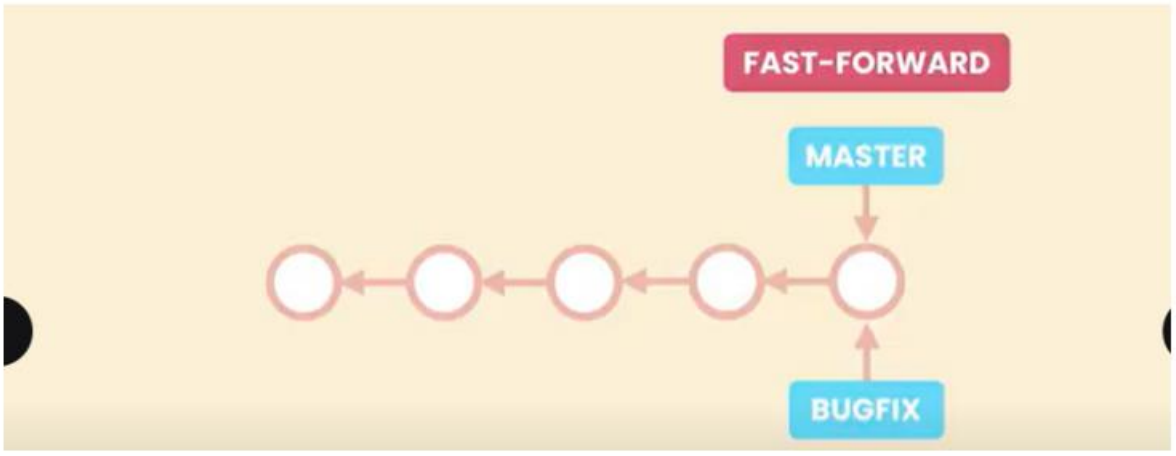
3-Way Merge

Used when both branches have new commits

Git creates a new merge commit that combines both histories







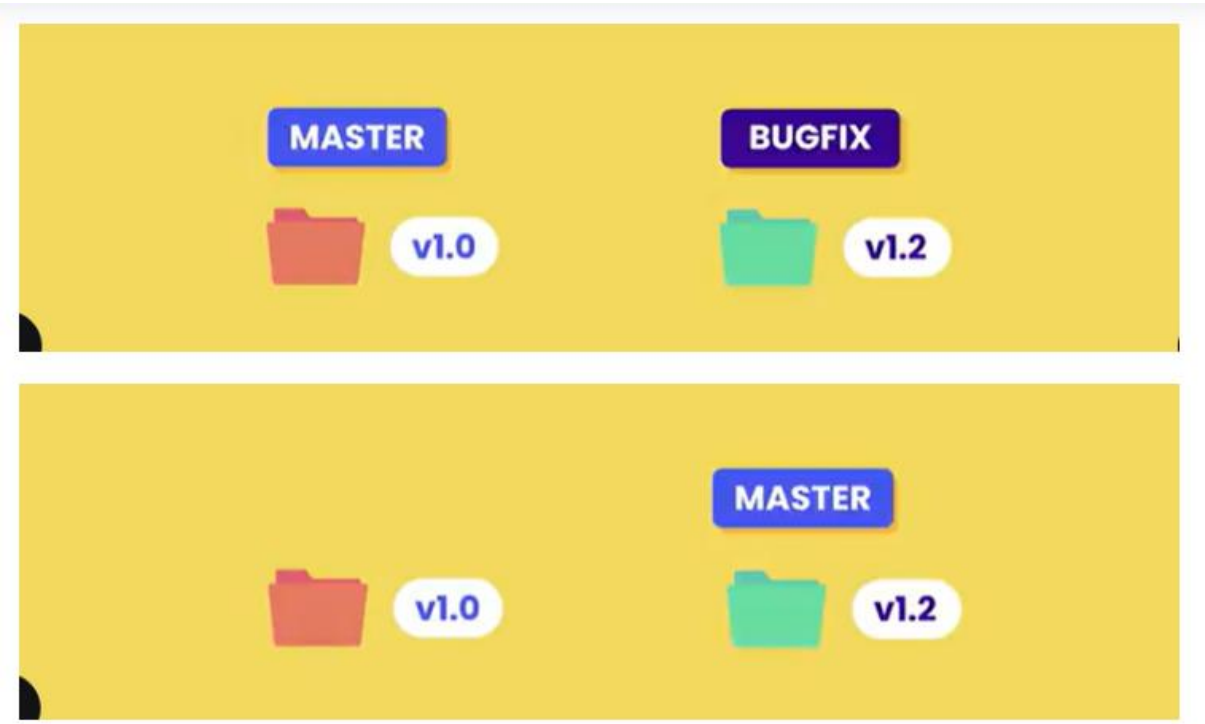
Here is the **Master branch** with three commits. We create a new branch called **bugfix**. Remember, a branch in Git is just a pointer to a commit. At this point, both the **Master** and **bugfix** pointers are pointing to the same commit.

Now we switch to the **bugfix** branch and make a couple of commits. When we need to bring these changes back to **Master**, we notice that:

The branches **have not diverged**

There is a **direct linear path** from bugfix to Master

To merge these changes, Git simply moves the **Master pointer forward** to match the bugfix pointer. This is called a **fast-forward merge**.



Fast-Forward Merge Scenario

Initial State

The code in our Master branch is at **Version 1**

We create a copy (branch) called bugfix

Both Master and bugfix point to the same commit (Version 1)

Development Phase

We make changes in the bugfix branch, advancing to **Version 1.2**

Master remains at Version 1

The commit history forms a straight line: Version 1 ← Version 1.2

Merging Process

Since the branches haven't diverged (no parallel development occurred):

Git performs a **fast-forward merge**

The Master pointer simply moves forward to match bugfix

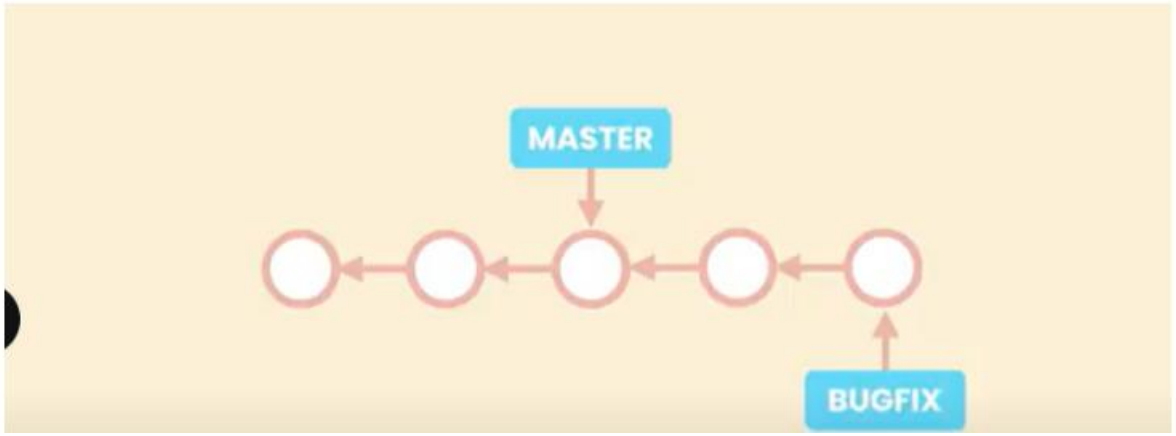
This is equivalent to "renaming" the latest version (1.2) as our new Master

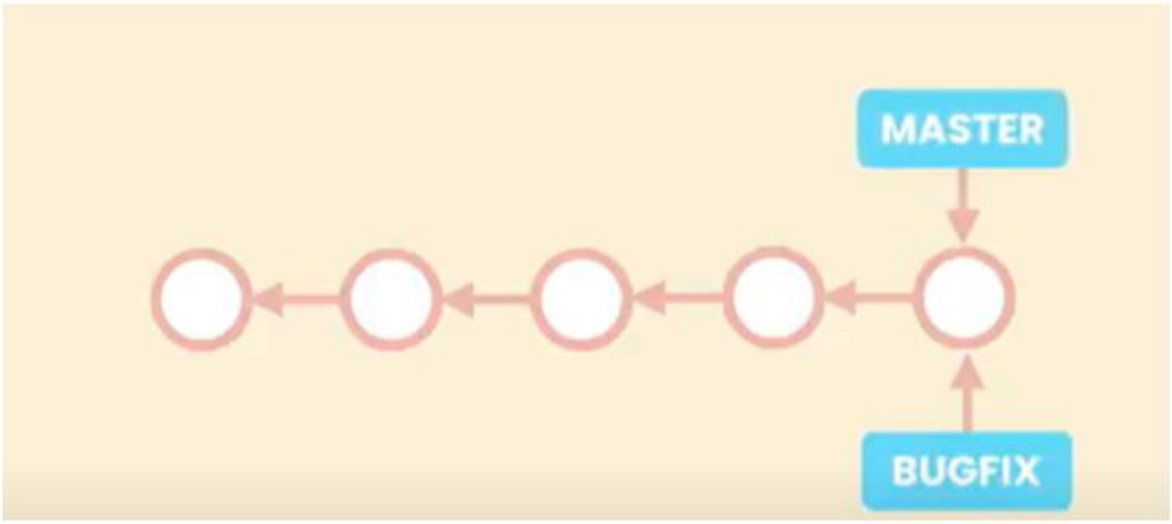
The bugfix pointer can then be safely removed
No merge commit is created in fast-forward merges
Branch pointer movement is atomic and instantaneous

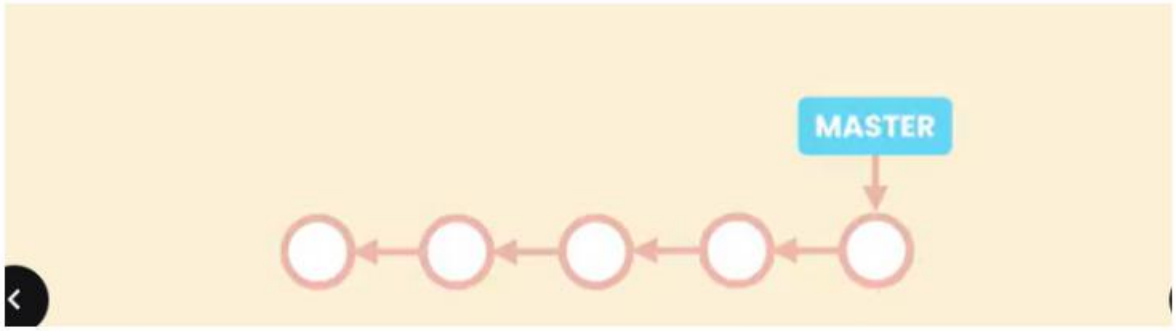
Linear history is preserved (no branching in the commit graph)
git checkout master # Switch to master

git merge bugfix # Fast-forward merge

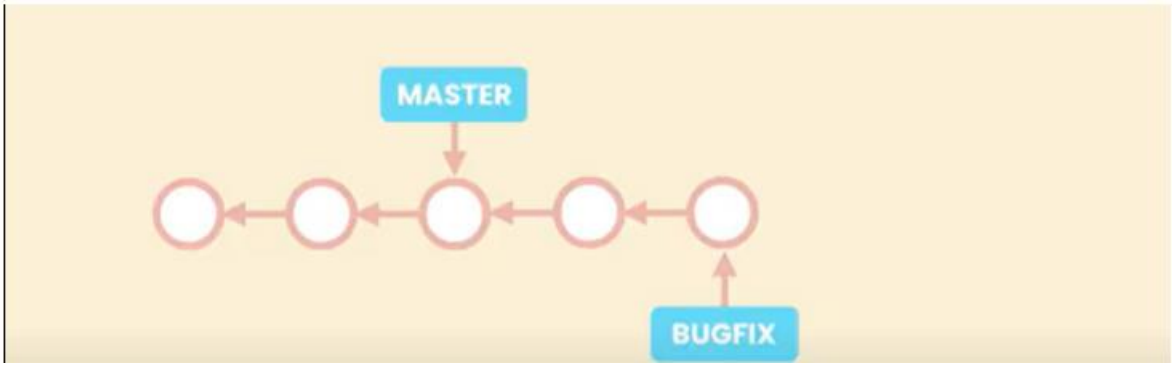
- git branch -d bugfix # Delete the bugfix pointer







3 way Merge



Diverged Branches Scenario

Current State

The bugfix branch is **2 commits ahead** of master

Before merging, we **switch back to** master and add a new commit

Now the branches have **diverged**:

master has new changes not in bugfix

bugfix has changes not in master

* (master) New master commit

|

| * (bugfix) Bugfix commit 2

| * Bugfix commit 1

|/

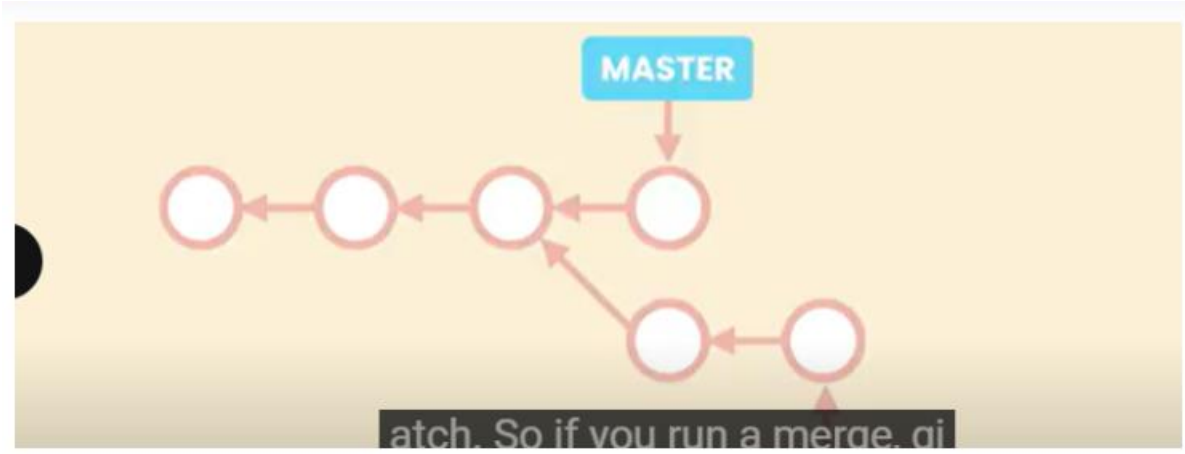
* Original common commit

Merge Implications

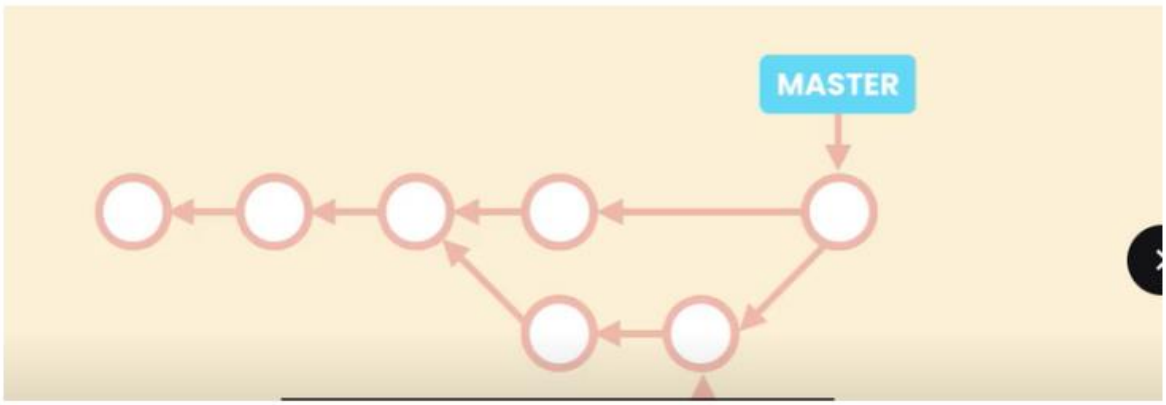
Git can no longer do a **fast-forward merge**

A **3-way merge** will be required

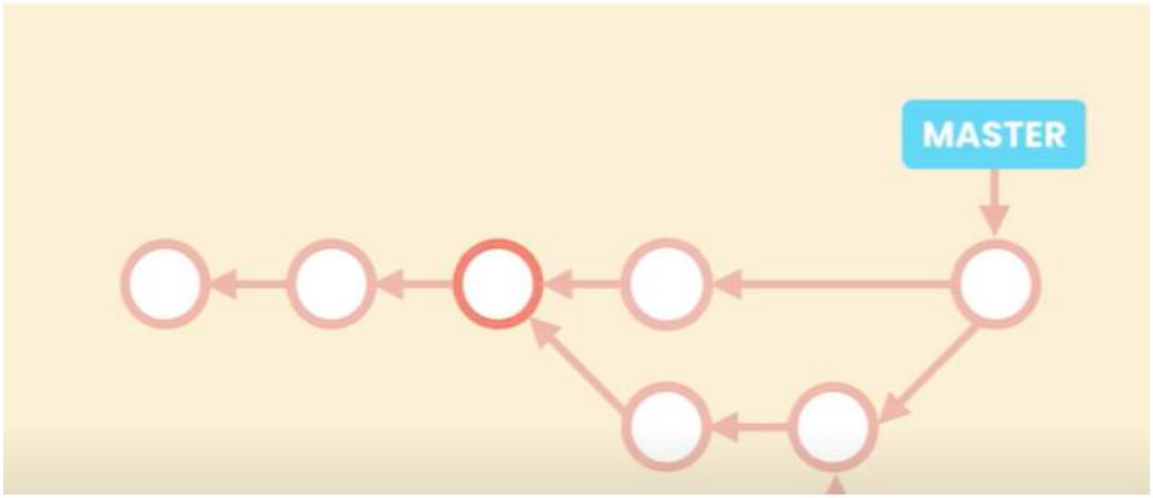
This creates a **new merge commit** combining both histories



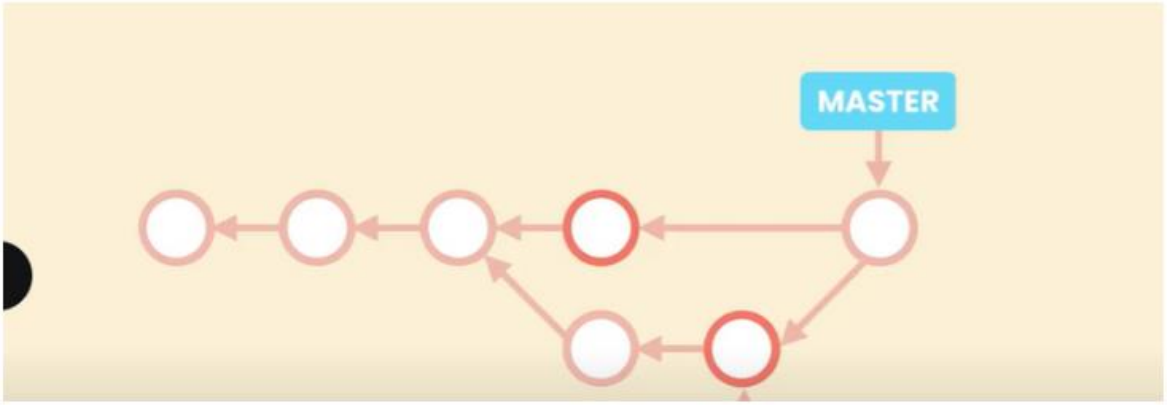
BUGFIX



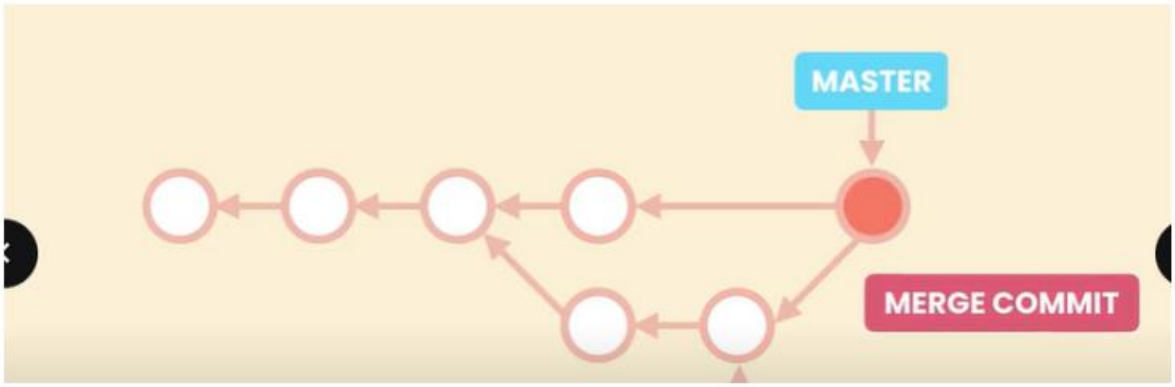
BUGFIX



BUGFIX



BUGFIX



BUGFIX

Merge Behavior in Git

When branches have diverged:

Git **cannot** simply move the master pointer forward to match bugfix because:

This would lose the latest commit in master

The branches contain independent changes that must be preserved

Instead, Git creates a **new merge commit** that:

Combines changes from both branches

Preserves all history

Why "3-Way Merge"?

This merge type uses three commits:

The **common ancestor** (before changes)

The **tip of** master (after changes in main branch)

The **tip of** bugfix (after changes in feature branch)

Git:

Compares these three snapshots

Intelligently combines the changes

Creates a new **merge commit** that ties both branches together

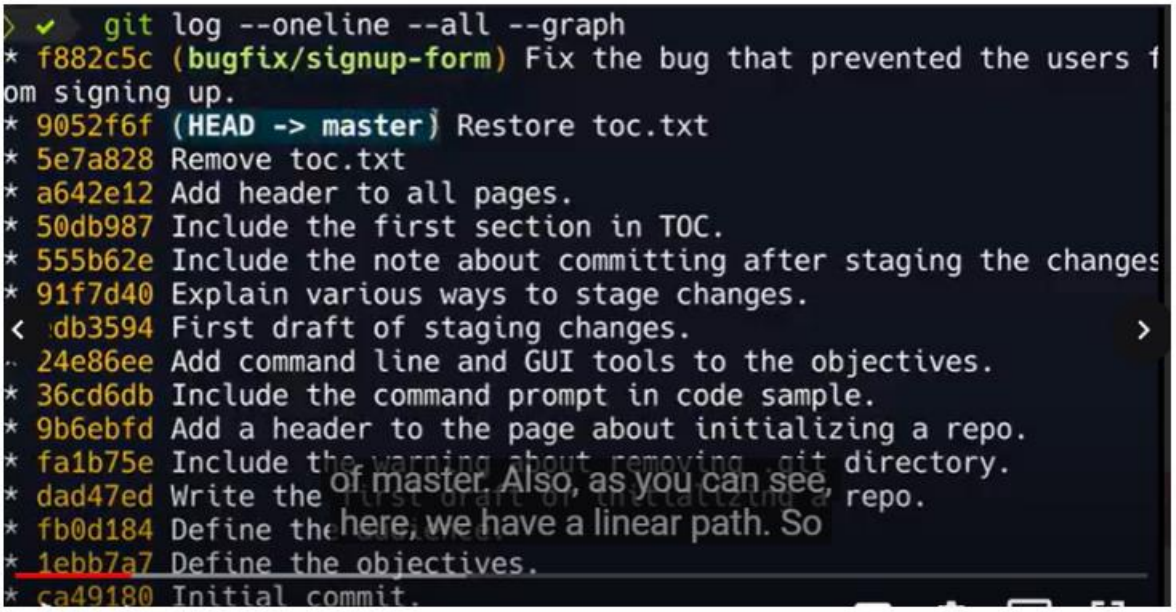
Merge Types Summary

Fast-forward merges: When branches have not diverged (linear history)

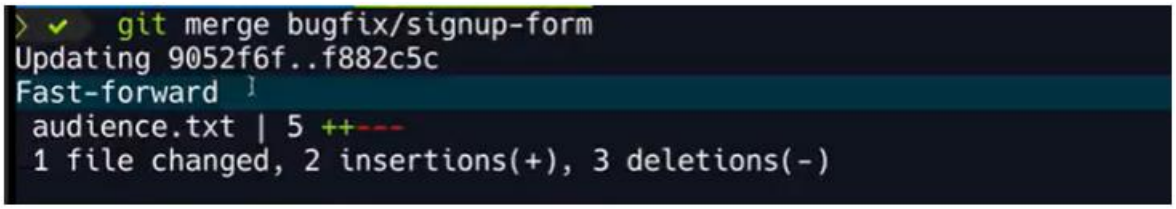
3-way merges: When branches have diverged (parallel development)

Example of fast forward merges

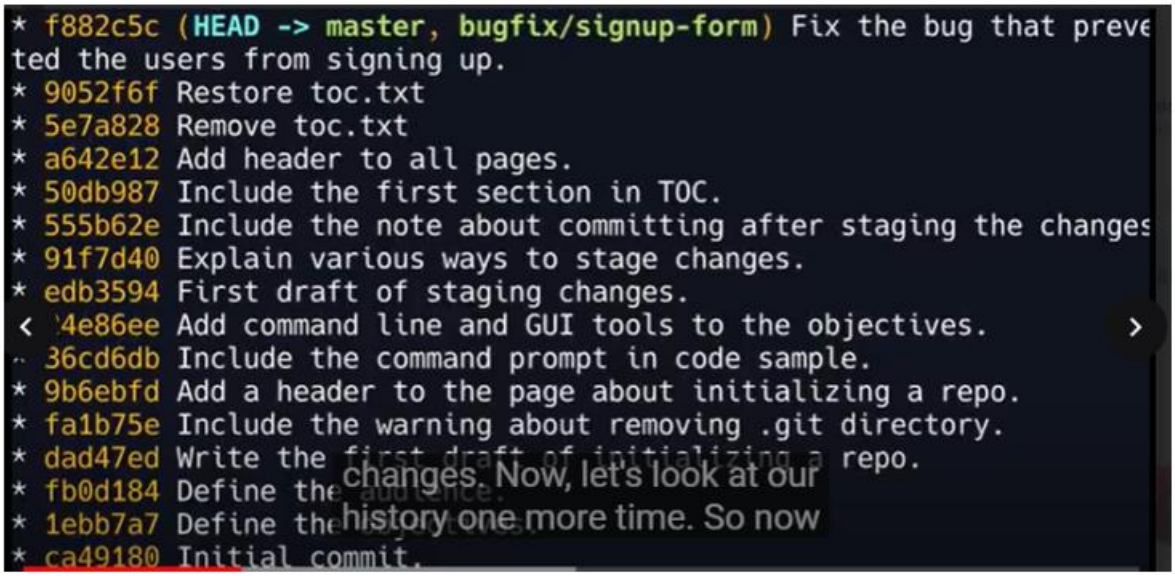
git log --oneline --all --graph



git merge bugfix/signup-form



git log –oneline --all --graph

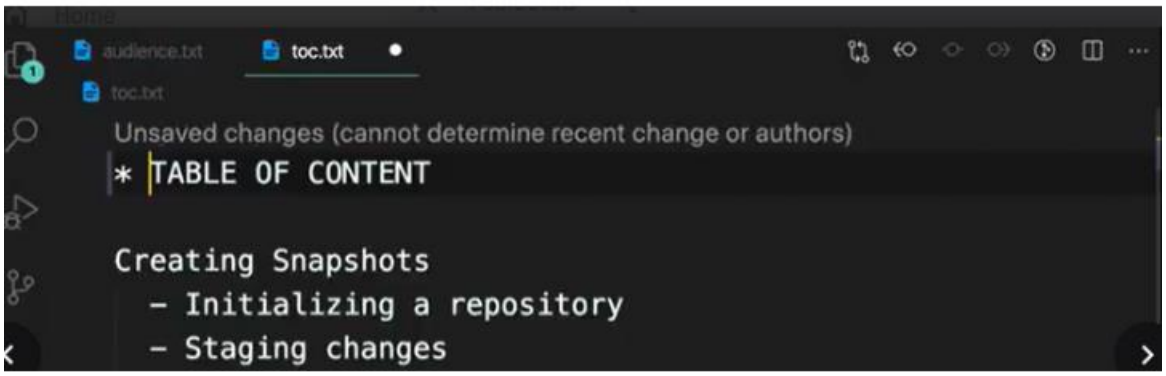


git branch bugfix(but after that we have to switch that branch so we use another way)

git switch -C bugfix/login-form

- 

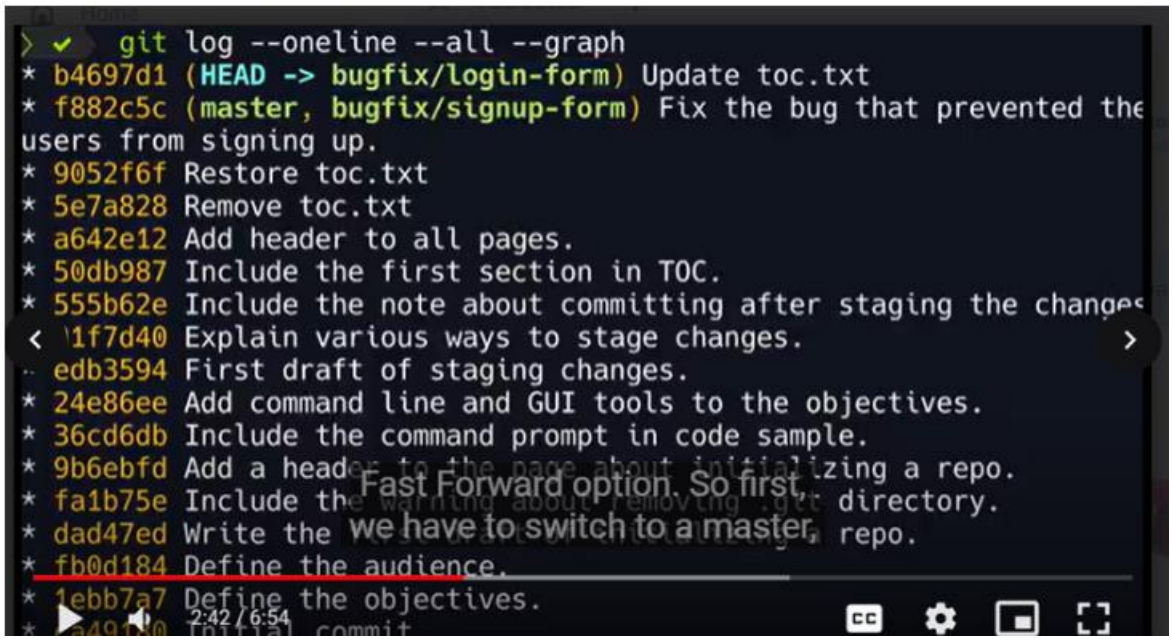
code toc.txt



git add .

git commit -m “update toc.txt”

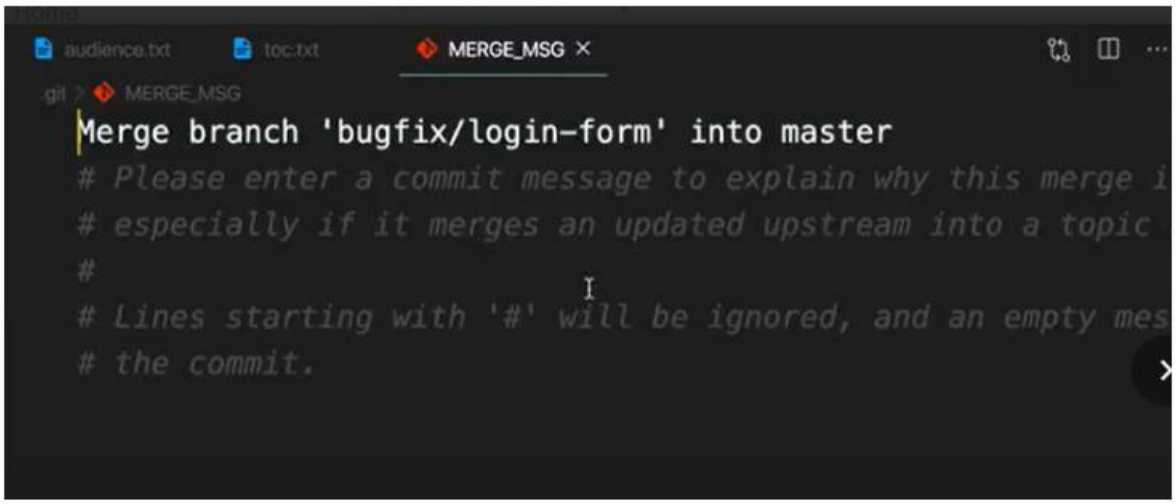
git log --oneline --all –graph



git switch master

```
git switch master
Switched to branch 'master'
```

git merge --no-ff bugfix/login-form



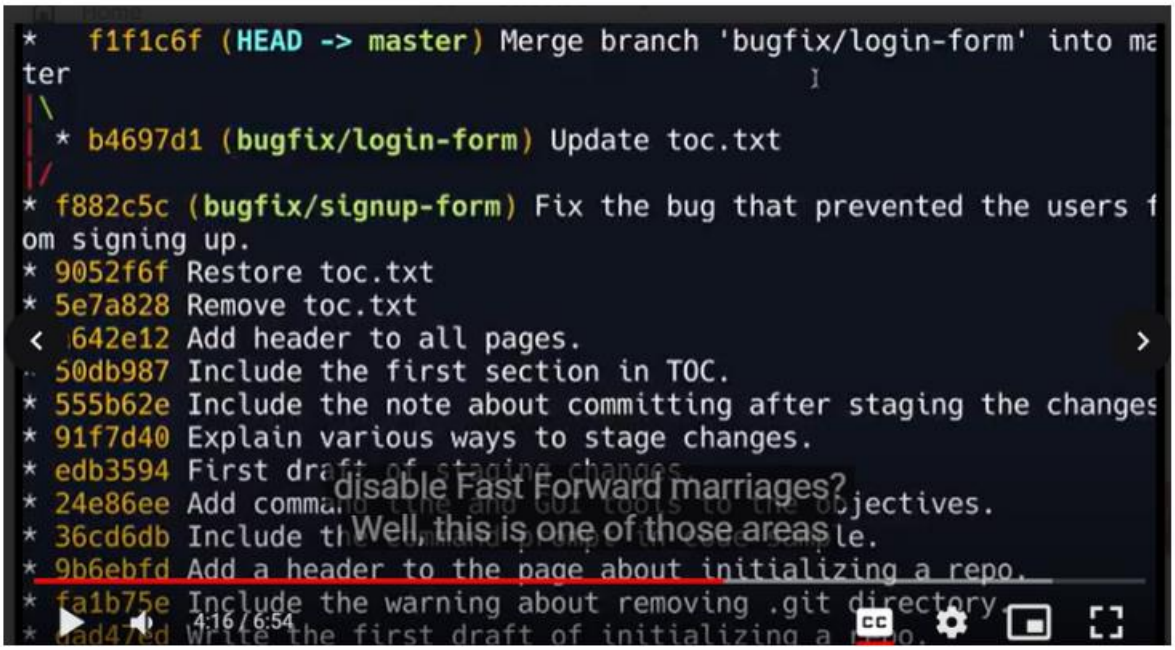
To **force Git to create a merge commit** even when a fast-forward is possible:
git merge --no-ff bugfix
This will:

Combine all changes from the target branch (bugfix)

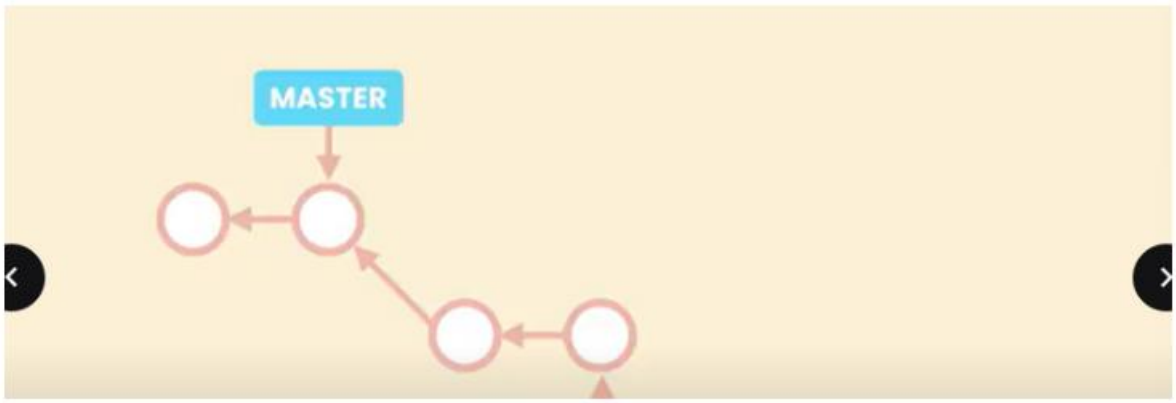
Create a dedicated merge commit in master

Preserve branch topology in the history

To visualize the merge structure
git log --oneline --all –graph



example....Two branches Master and Feature and our branches are not diverge so there is a direct linear path from feature to Master.



Feature

Fast-Forward vs. Explicit Merge Commit

When NOT Using Fast-Forward Merge:

We get a **merge commit** that combines all changes from the feature branch (F1 and F2)

If we later need to remove this feature:

We can simply revert the single merge commit

This creates a new commit that is the **exact opposite** of the original merge

Cleanly undoes all changes from that feature in one operation

When Using Fast-Forward Merge:

The master pointer simply moves to the feature branch's HEAD

To remove the feature later:

We must revert **multiple individual commits** (F1, F2, etc.)

More complex as we need to undo each change separately

Higher chance of missing something or creating conflicts

Key Points:

Both approaches are valid with different trade offs

Merge commits provide cleaner undo capability

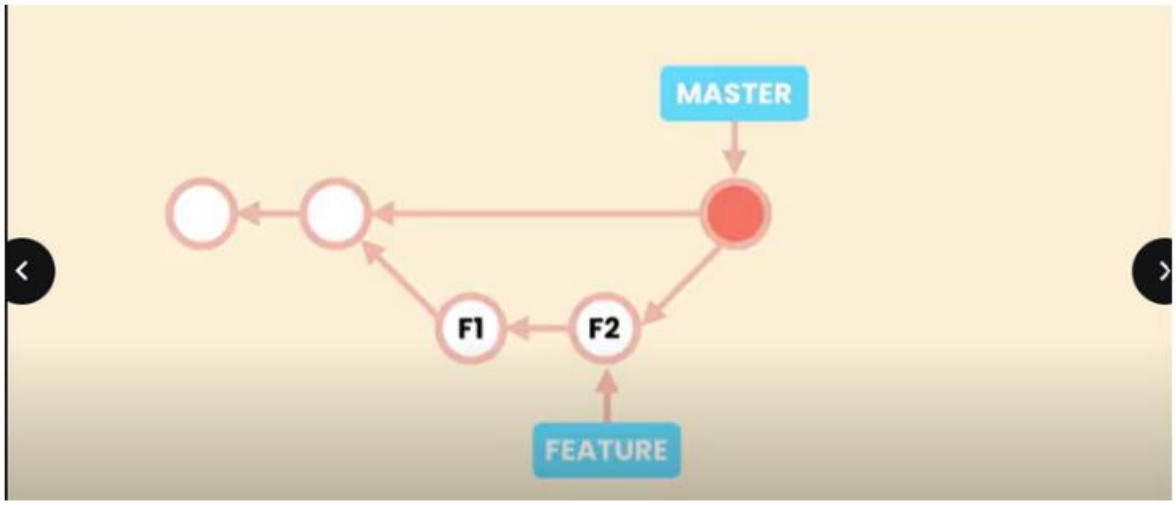
Fast-forward keeps history linear but makes reversions harder

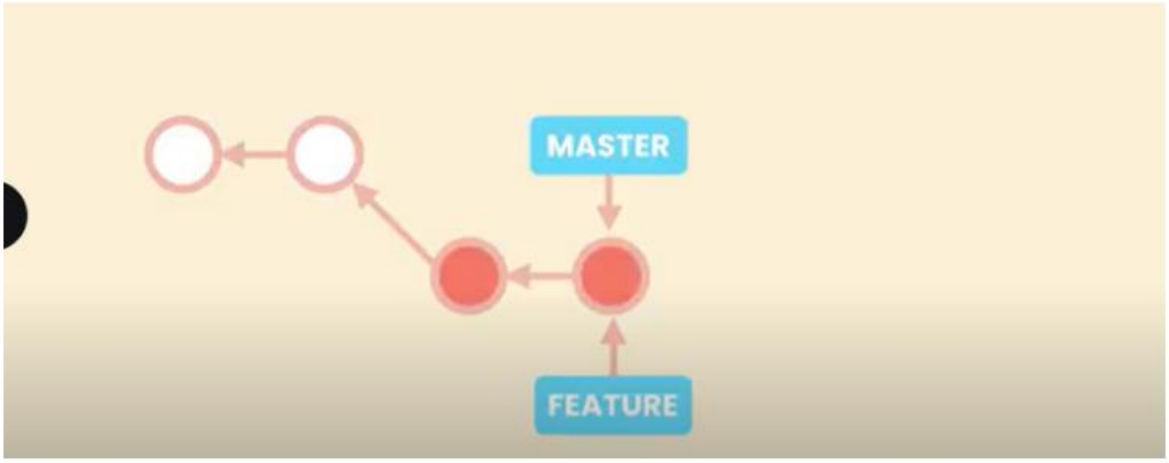
Reverting creates a new commit that counteracts the original changes

Example Commands:

git merge --no-ff feature

git revert -m 1 <merge-commit-hash>



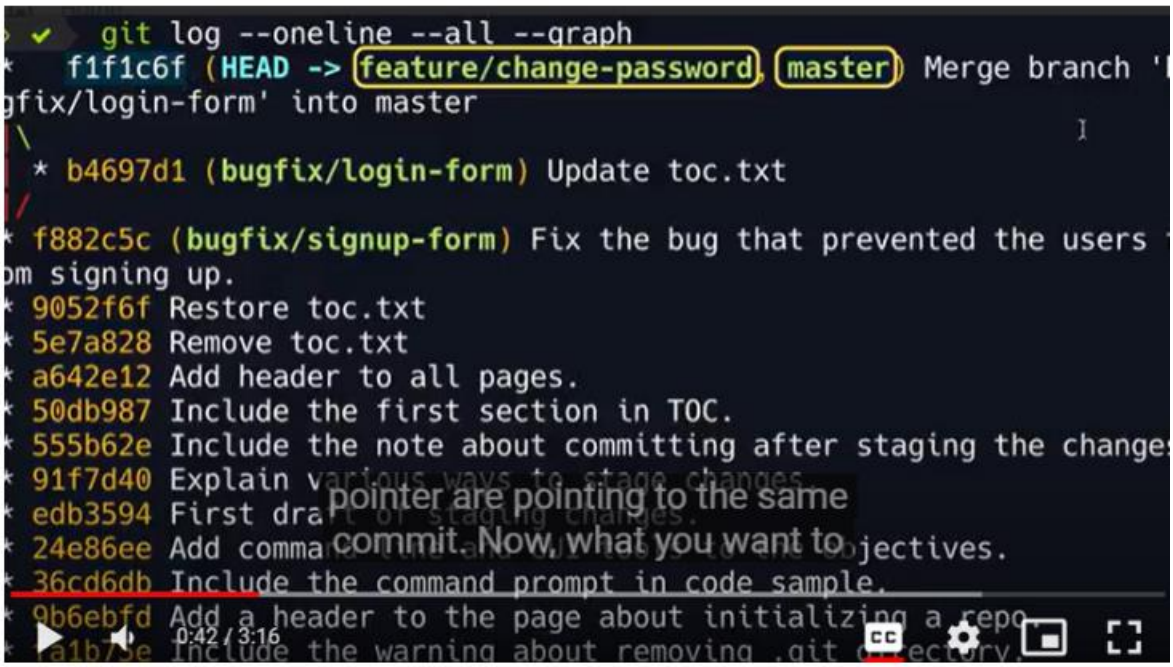


3 way -Merge

git switch -C feature/change-password

```
git switch -C feature/change-password
Switched to a new branch 'feature/change-password'
```

git log --oneline --all --graph



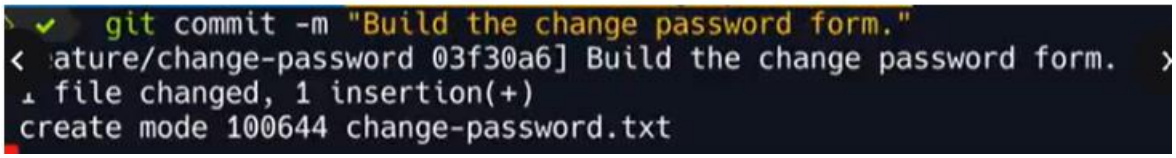
change password pointer and master merge pointer on the same point(commit)

HOW TO BRANCHES CAN DIVERGE

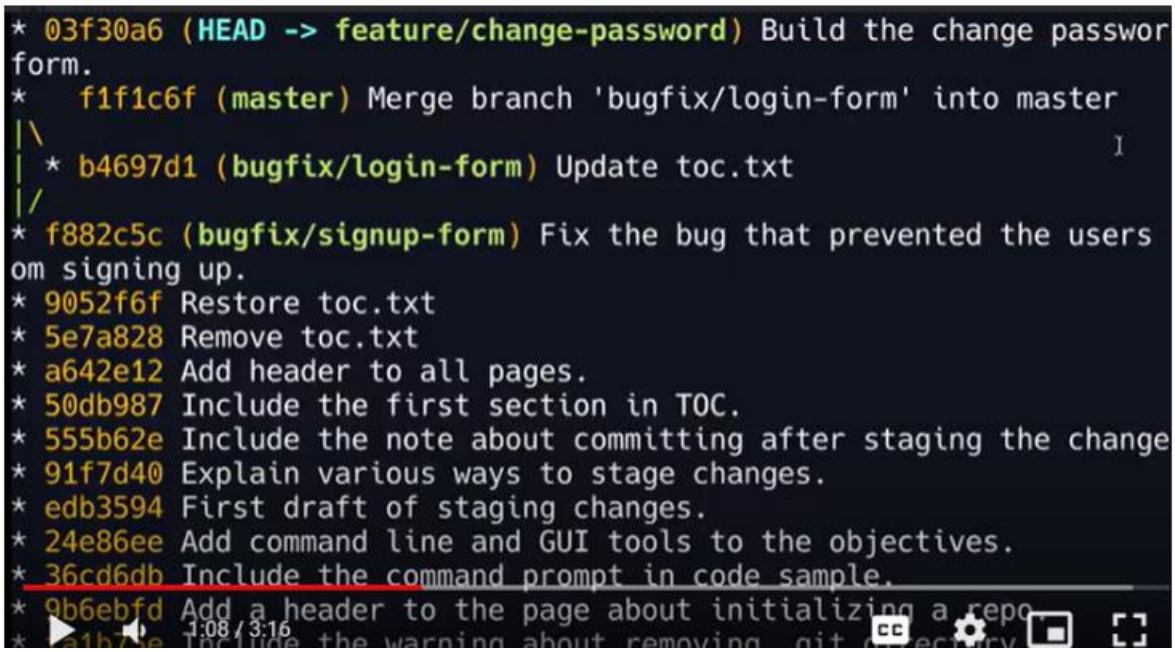
echo hello > change-password.txt

git add .

- git commit -m “Build the change password form.”



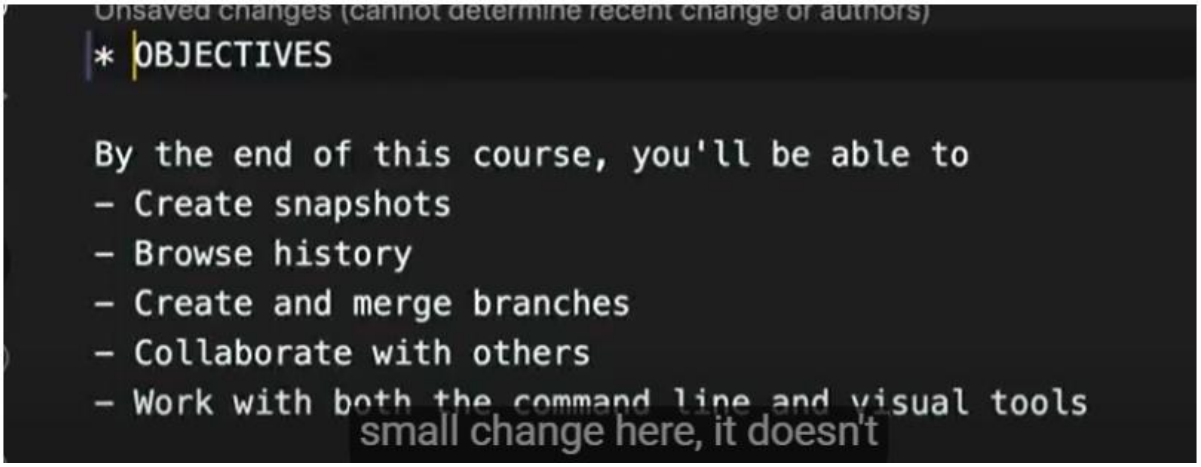
git log --oneline --all --graph

- 

  for diverge branch go back to master branch

  git switch master

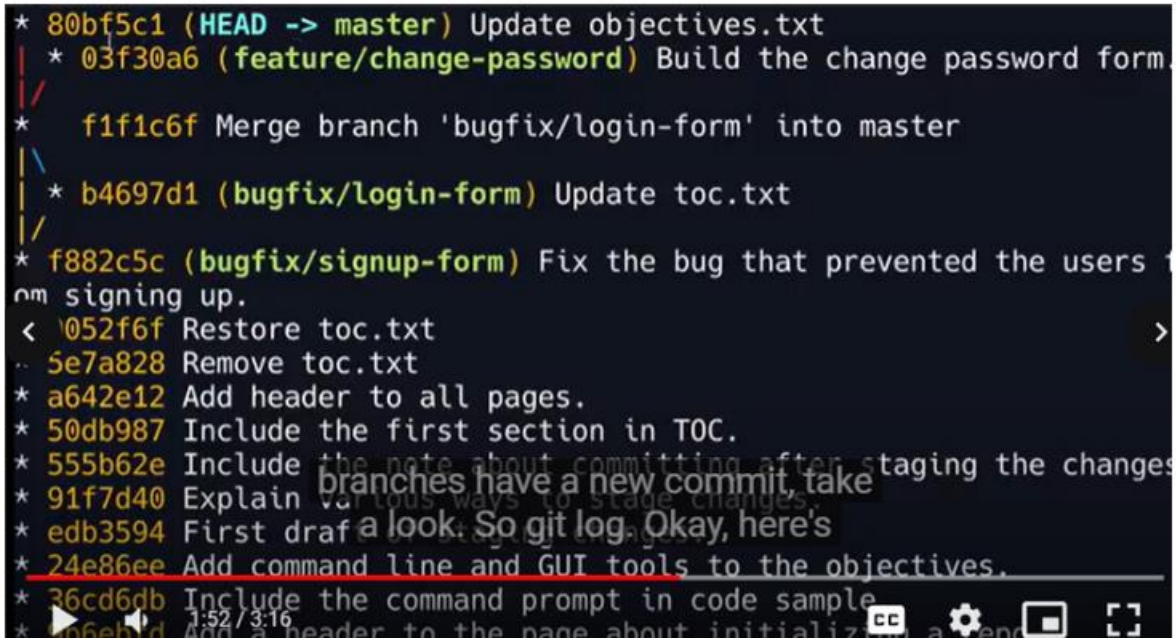
- code objectives.txt



  git add .

  git commit -m “update objectives.txt”

  git log --oneline --all –graph



Diverged Branches Scenario

You can see the branches have diverged:

This commit (f1f1c6f) was added to feature/change-password branch

Meanwhile, we added a new commit to master

Now there's no direct linear path from feature/change-password to master - they've developed in parallel.

Key Observations:

If you're on feature/change-password, you can't reach master's commits directly

The branches form a fork in the commit history:

* (master) New master commit

|

| * (feature/change-password) f1f1c6f - Change password feature

|/

* Common ancestor commit

Merging the Branches:

git checkout master

git merge feature/change-password

This will:

Perform a **3-way merge** (not fast-forward)

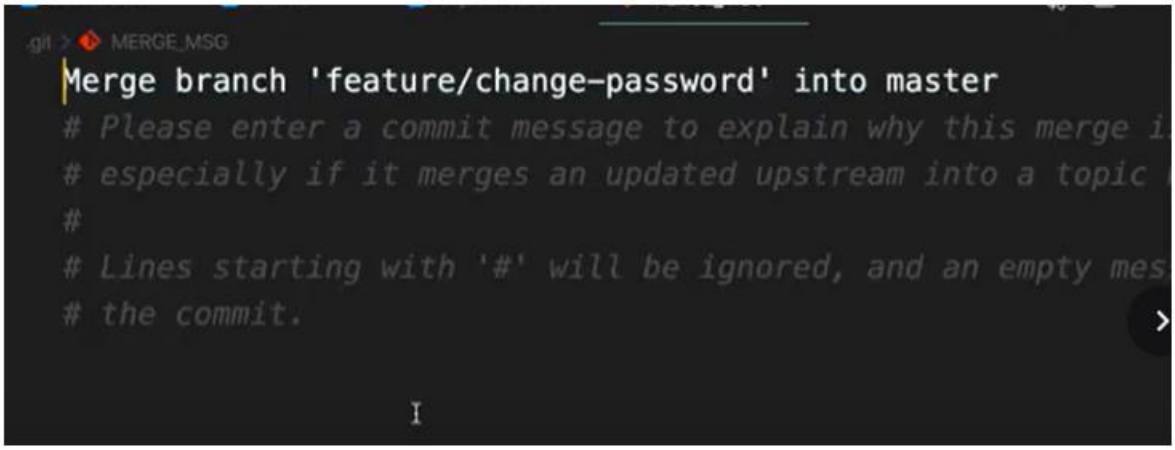
Create a new **merge commit** that ties both histories together

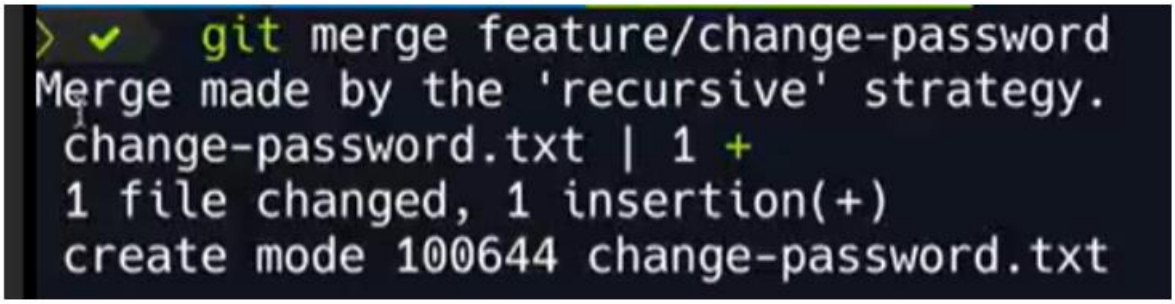
Preserve all work from both branches

Why This Matters:

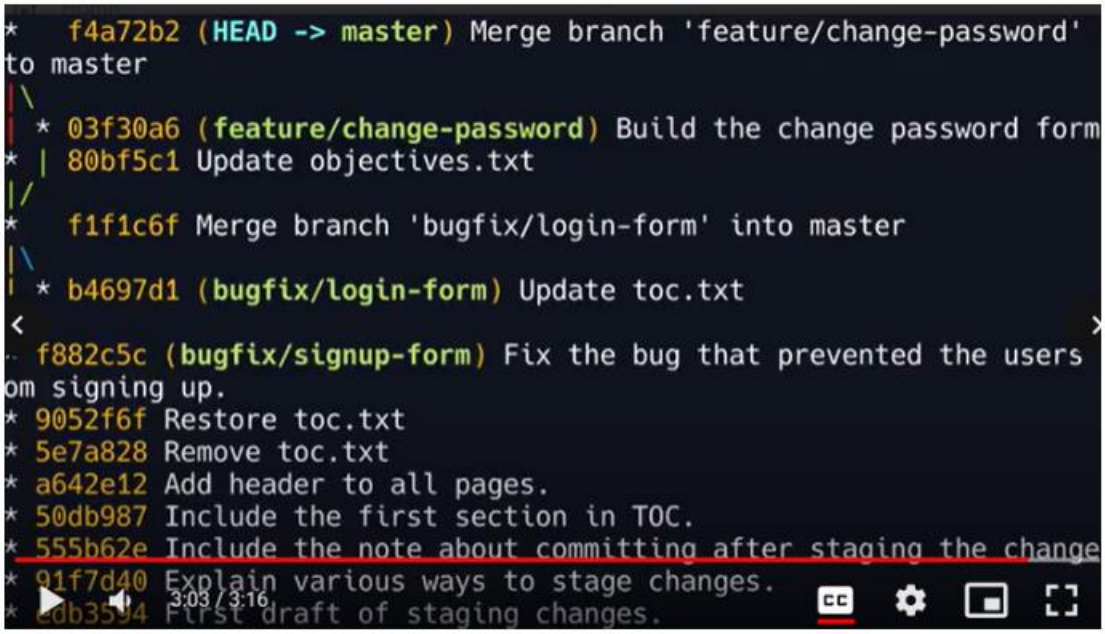
Maintains complete project history
Preserves context of parallel development
Requires conflict resolution if changes overlap

The merge will succeed if changes don't conflict, otherwise you'll need to resolve the differences manually.

- 



git log --oneline --all --graph



VIEWING THE MERGED BRANCHES

git branch –merged

git branch -d bugfix/login-form

git branch --no-merged

MERGE CONFLICTS

SCENARIO
Merge Conflict Scenarios

Git encounters conflicts in these cases:

1. **Line-Level Conflicts**

   Same line changed differently in both branches
   Example:
   Branch A: change1, change2
   Branch B: change, Delete

2. **File Deletion vs Modification**

   File changed in one branch but deleted in the other

3. **Duplicate File Addition**

   Same filename added in both branches with different content
   Example: Branch A: Add1, Add

Branch B: Different content for same file
How Git Handles Conflicts

Stops the merge process automatically

Marks conflicted files in your working directory

Requires manual resolution - you must:

git add <resolved-file>

git commit

Conflict Markers in Files

Git inserts conflict markers showing both versions:
<<<<<<< HEAD

Version from current branch

=======

Version from merging branch

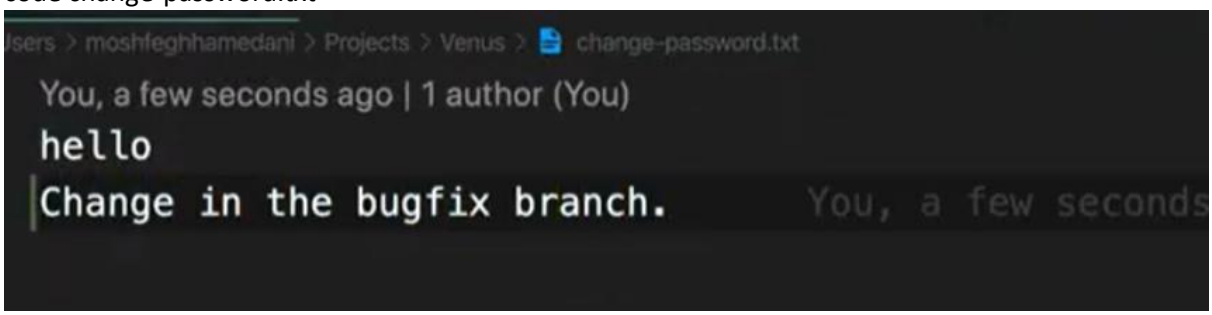
>>>>>>> branch-name

Examples

On the master

git switch -C bugfix/change-password

- code change-password.txt



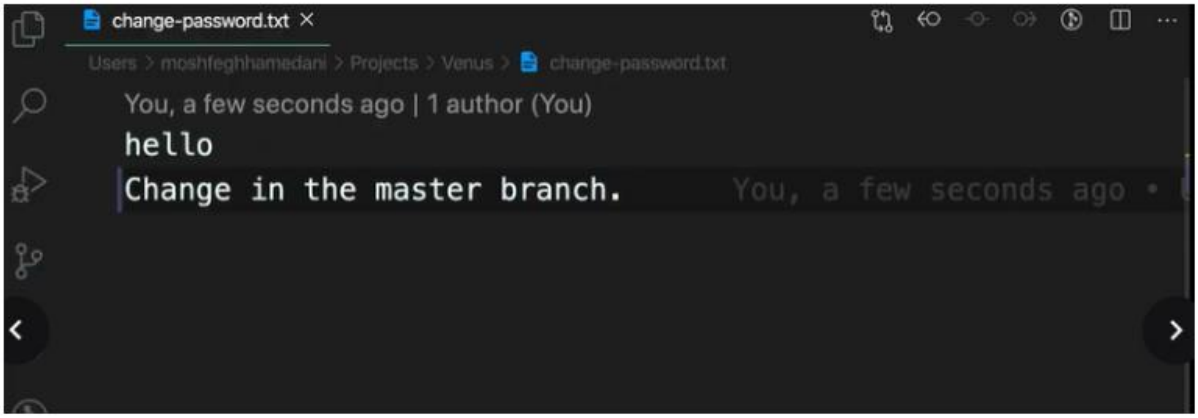
git add .

git commit -m “update change-password.txt”

Now switch to master branch

git switch master

code change-password.txt

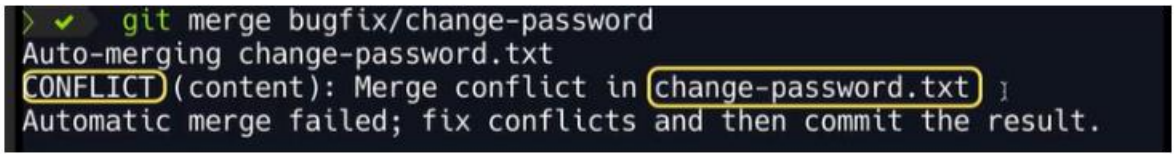


git add .

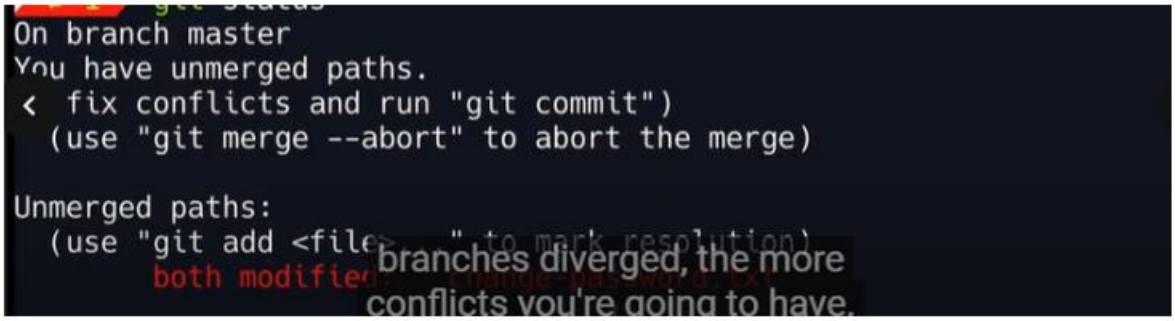
git commit -m “Update change-password.txt”

Both changes merge in diffrent ways

git merge bugfix/change-password

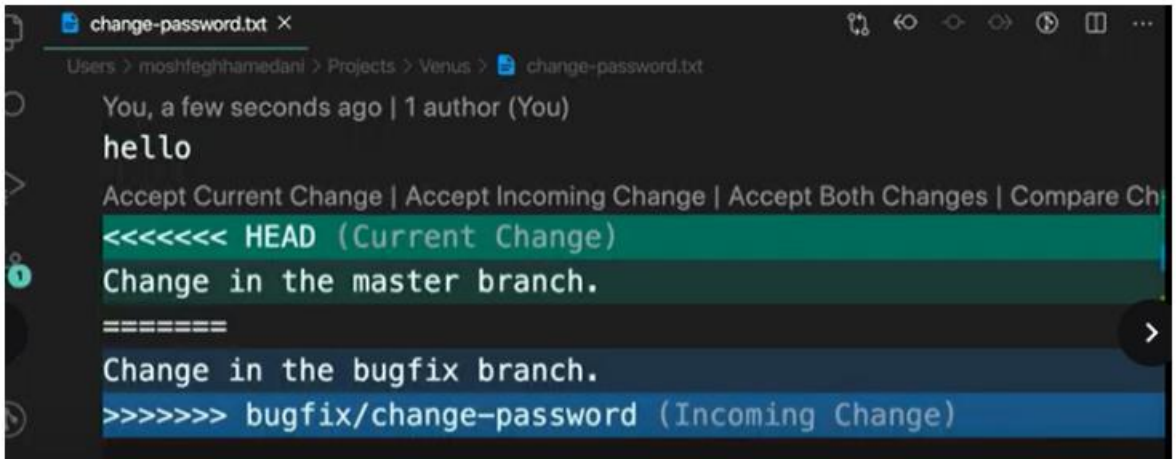


Now we have a conflict.
I know we have to jump in and manually combine the changes. We're in the middle of a merge process, so let's run:
git status



this is very simple scenario in real world we have 10 files listed here basically the more branches diverged the more conflict you are going to have .

code change-password.txt



Merge Conflict Markers Explained

There are markers showing changes from both branches:

HEAD (current branch) changes appear between:

<<<<<<< HEAD

[Our brach's code]

The other branch's changes appear after

[Their branch's code]

>>>>>>> branch-name
In real situations:

You'll see **multiple conflict sections** in a file

Each conflicting "chunk" of code will have these markers

Resolving in VSCode:

Open the conflicted file

You can:

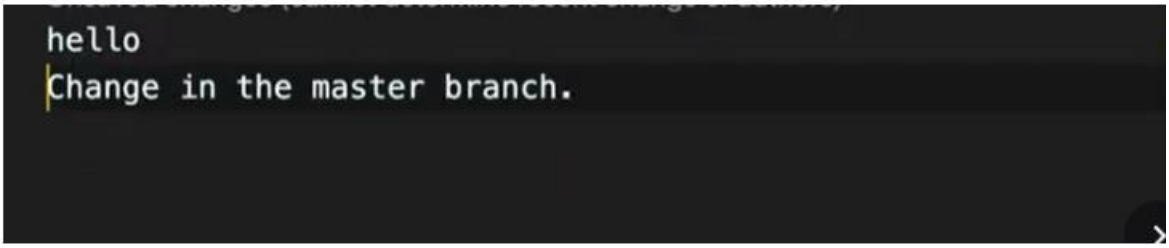
Accept current change (keep HEAD version)

Accept incoming change (use their version)

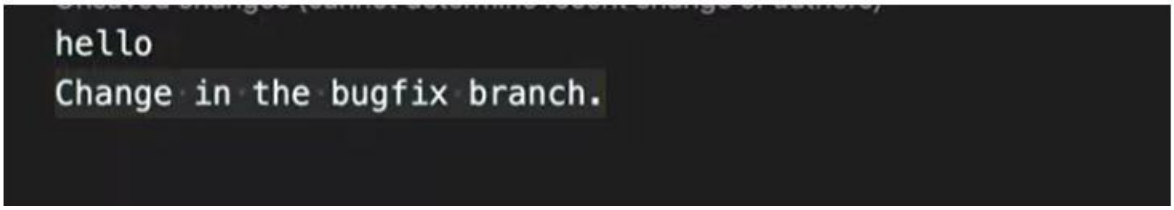
Combine changes manually (edit the code directly)

Save the file after resolving
git add <file>

git commit
Your original conflict marker explanation
VSCode resolution workflow
All technical accuracy about Git's behavior

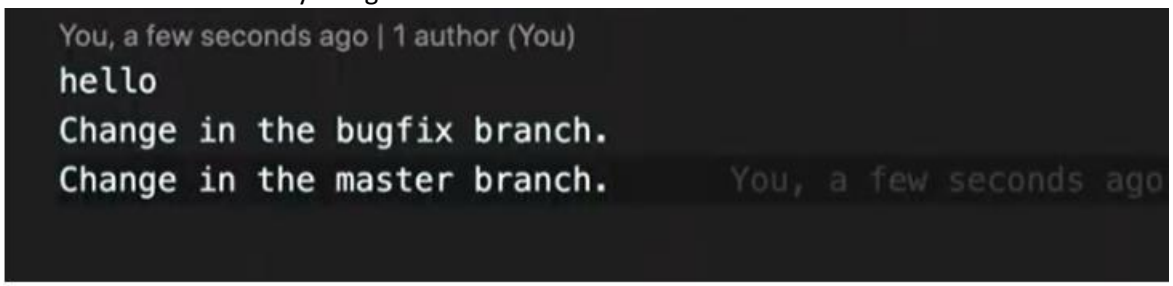


2nd way accept incoming changes



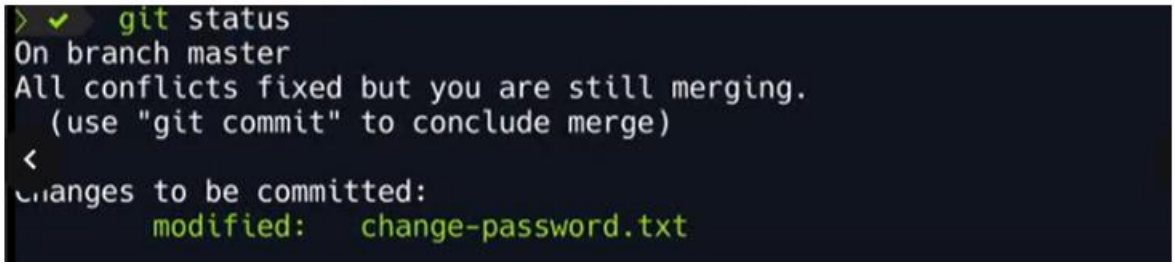
3rd option accept both changes

we can also edit manually merge

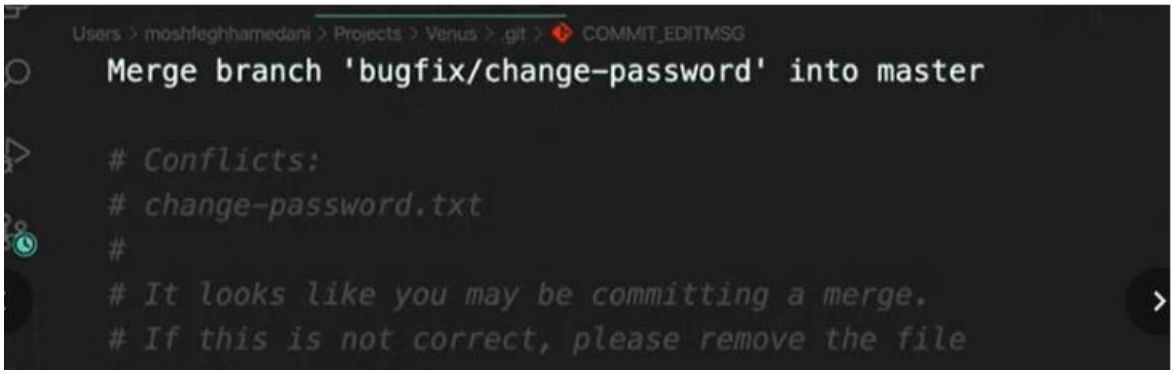


if you add another code that changes is evil commit because it introducing changes that didnt exist in any branches.

git add change-password.txt



git commit



Aborting A Merge

we have a merge conflict to abort the merge we simply type

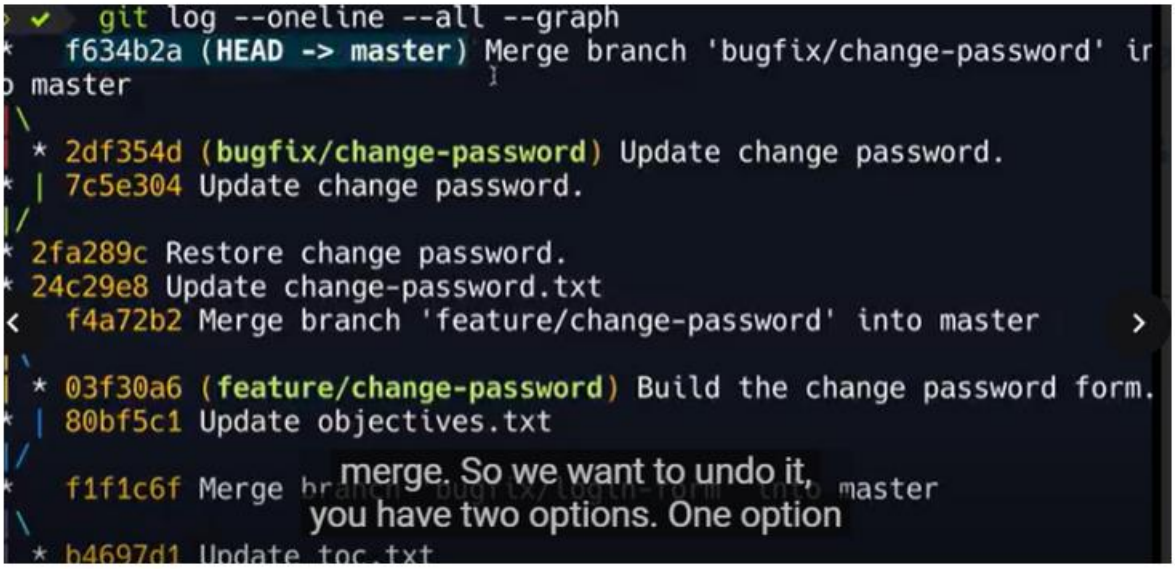
git merge --abort

Now we back to the stage where we start the merge

Undoing A faulty Merge

Sometime we do a merge and then find out that our code doesn't compiled or our application doesnt work. What happen if u screw the merge if we dont combine the changes properly ?? situation like this we need to undo the merge and then remerge. so lets have a quick look at our history.

git log --oneline --all --graph



Undoing a Faulty Merge Commit

At the top of our history, we have a merge commit. Let's assume this merge introduced problems and we need to undo it. We have two options:

Remove the commit (Rewriting History)

Dangerous for shared repositories

Acceptable for local work

Eliminates the commit completely
git reset --hard HEAD~1

Revert the commit (Safe Approach)

Creates a new commit that undoes all changes

Preserves history (crucial for shared/remote repos)
git revert -m 1 <merge-commit-hash>
(HEAD -> master)

* [Faulty merge commit] ← Both pointers here

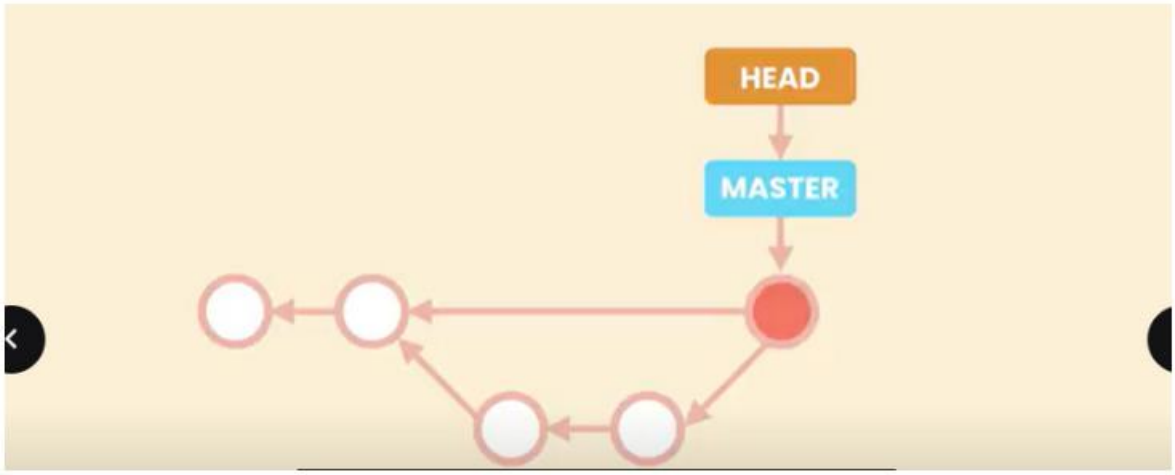
|

* [Previous commit]
Key Considerations

Rewriting history (git reset) is problematic if commits were pushed

Reverting is always safe but leaves the original mistake in history

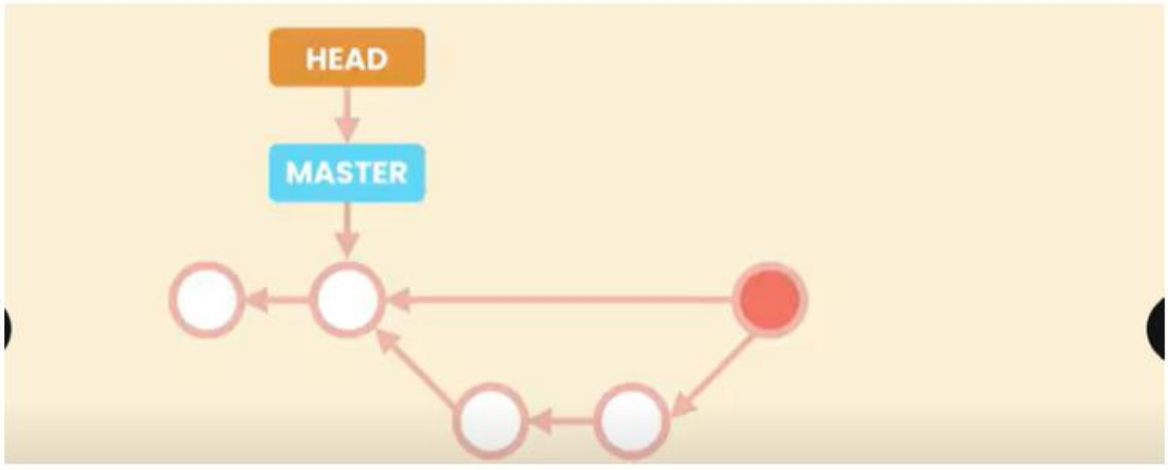
- For merge commits, use -m 1 to specify the parent branch



Feature

Now we are going to use reset command to move both these pointers and have them point the last commit on the Master Branch before we started the merge.

Now look at our merge commit .we dont have any other comments or any other comments or any pointers pointing to this commit.So from this point of view this commit is garbage in a while get looks for this commit and automatically removes from the repo



FEATURES

git reset --hard HEAD ~1

RESETTING

Soft
Mixed

- Hard

--soft Reset
git reset --soft HEAD~1
What happens:

The HEAD pointer moves to the previous snapshot

Staging area and working directory remain unchanged

Effect:

Changes from undone commit appear staged (ready to recommit)
--mixed Reset (Default)
git reset HEAD~1 # '--mixed' is implied
What happens:

HEAD moves to previous snapshot

Staging area is updated to match the new snapshot

Working directory remains unchanged

Effect:

- Changes from undone commit appear as unstaged modifications

git reset --mixed HEAD~1

So if u have any local changes in our working directory they are not affected

- If used git reset --hard HEAD~1

git reset --hard HEAD~1

-
```
> ✔  git reset --hard HEAD~1
HEAD is now at 7c5e304 Update change password.
```

git log --oneline --all –graph

```
> ✔  git log --oneline --all --graph
* 7c5e304 (HEAD -> master) Update change password.
| * 2df354d (bugfix/change-password) Update change password.
|/
* 2fa289c Restore change password.
* 24c29e8 Update change-password.txt
*   f4a72b2 Merge branch 'feature/change-password' into master
|\
| * 03f30a6 (feature/change-password) Build the change password form
<   80bf5c1 Update objectives.txt                                    >
|/
*   f1f1c6f Merge branch 'bugfix/login-form' into master
|\
| * b4697d1 Update toc.txt
|/
* f882c5c (bugfix/s                                    evented the users f
rom signing up.
```

we can recover it, I'm gonna show you. So we type Git reset

```
> ✔  git reset --hard f634b2a
HEAD is now at f634b2a Merge branch 'bugfix/change-password' into mas
ter
```

```
git log --oneline --all --graph
*   f634b2a (HEAD -> master) Merge branch 'bugfix/change-password' ir
to master
|\
| * 2df354d (bugfix/change-password) Update change password.
* | 7c5e304 Update change password.
|/
* 2fa289c Restore change password.
* 4c29e8 Update change-password.txt
*   f4a72b2 Merge branch 'feature/change-password' into master
|\
| * 03f30a6 (feature/change-password) Build the change password form.
* | 80bf5c1 Update objectives.txt
|/
*   f1f1c6f Merge branch 'bugfix/login-form' into master
|\
| * b4697d1 Update toc.txt
```

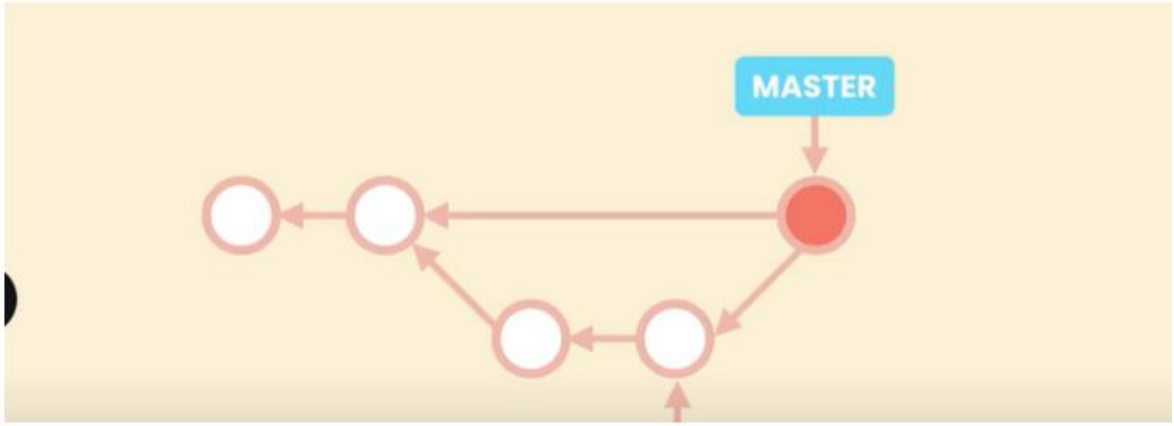
Second option Revert the commit

git revert HEAD

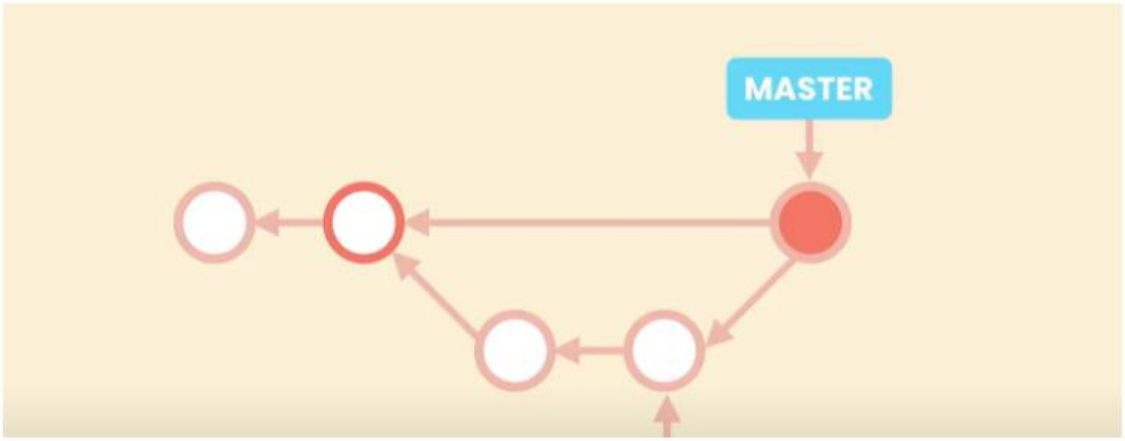
```
git revert HEAD
error: commit f634b2afbc815f489641ae7ade2ea9061ee383c0 is a merge but
 no -m option was given.
fatal: revert failed
```

Here is our Merge Commit below.

A merge commit has two parents one on Master Branch , the other on the feature branch .So to revert this merge commit we have to tell git how we want to revert the changes . In this case we want to revert to this commit over here. On the last commit on the Master Branch before we started the merge .this commit over here is the first parent of our Merge Commit because our merge commit



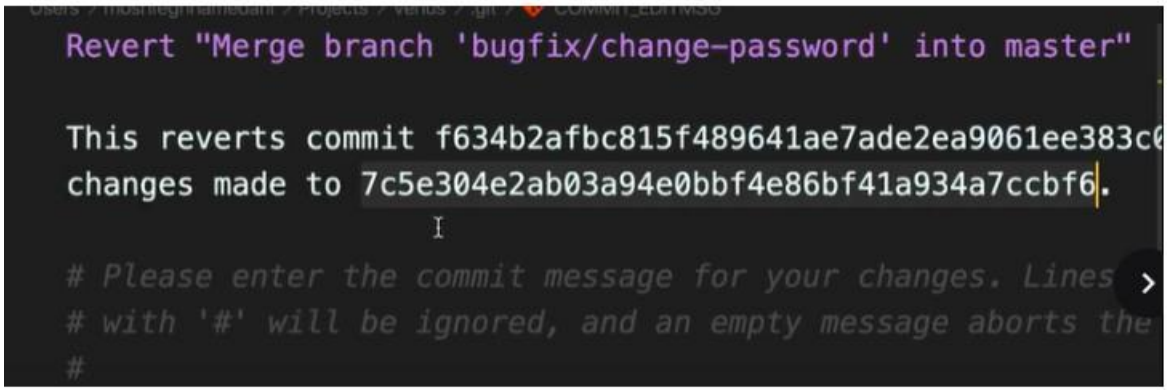
- FEATURE



- FEATURE

is on the master branch .so the first parent should also be on the Master branch ,

git revert -m 1 HEAD(target the last commit)

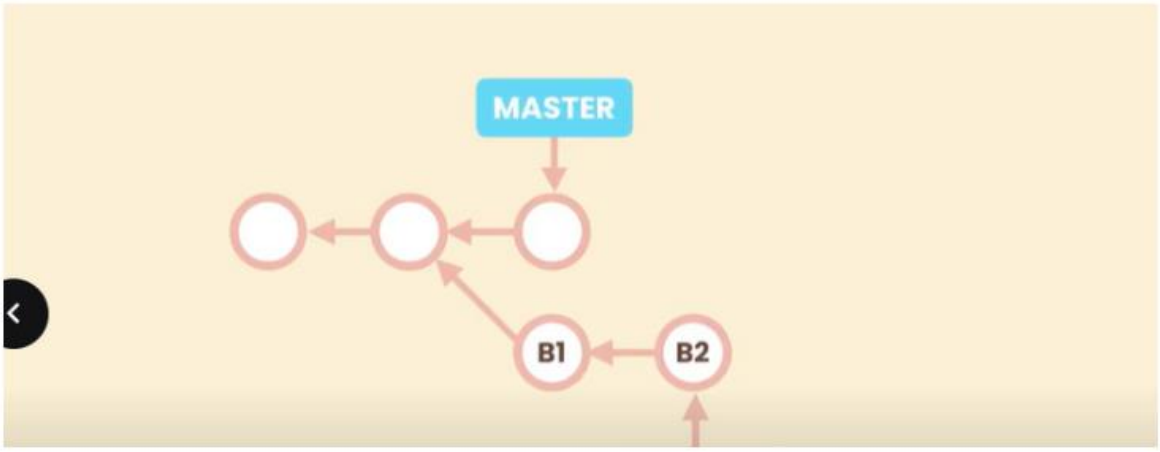


git log --oneline --all –graph

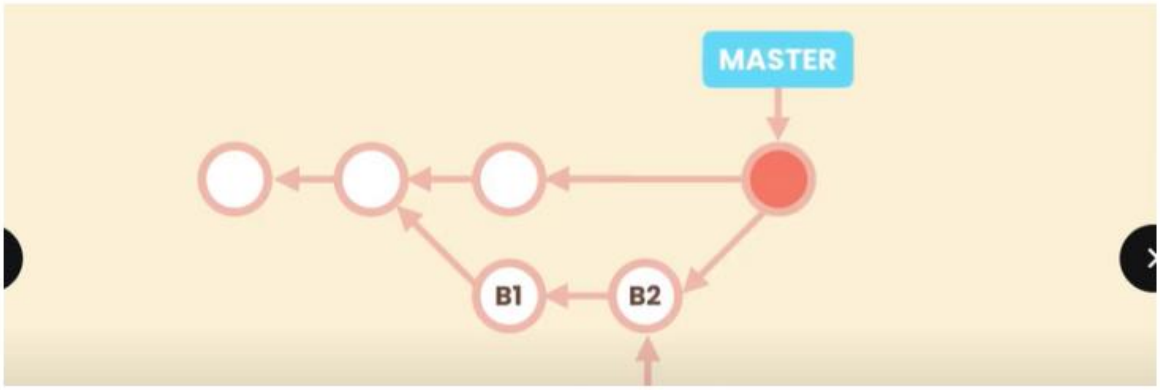
```
* 3a44949 (HEAD -> master) Revert "Merge branch 'bugfix/change-passwo
rd' into master"
*   f634b2a Merge branch 'bugfix/change-password' into master
|\
| * 2df354d (bugfix/change-password) Update change password.
* | 7c5e304 Update change password.
|/
* 2fa289c Restore change password.
* 24c29e8 Update change-password.txt
*   f4a72b2 Merge branch 'feature/change-password' into master
|\
| * 03f30a6 (feature/change-password) Build the change password form.
* | 80bf5c1 Update objectives.txt
|/
*   f1f1c6f Merge branch 'bugfix/login-form' into master
|\
| * b4697d1 Update toc.txt
```

SQUASH MERGING

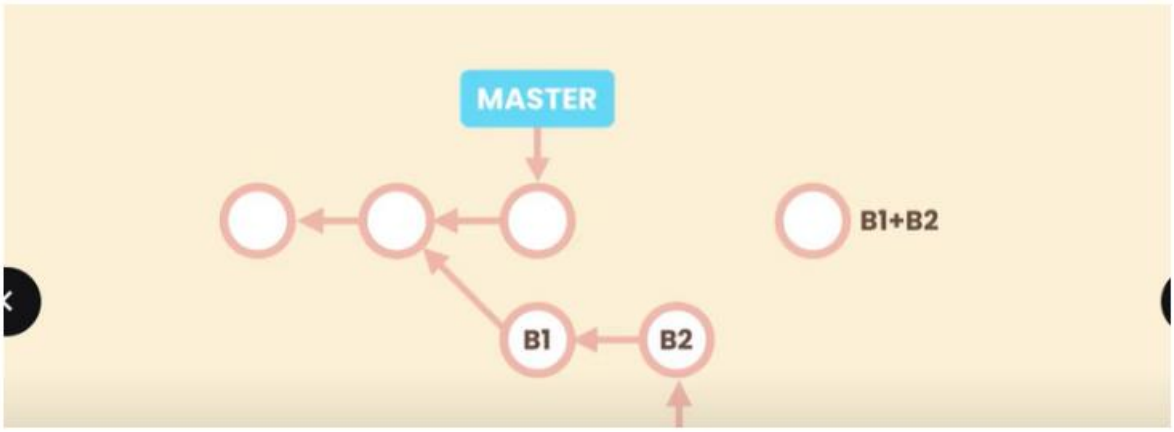
Squash Merging combines all commits from a feature branch into a single commit when merging to the main branch. It keeps history cleaner by reducing clutter while preserving changes.



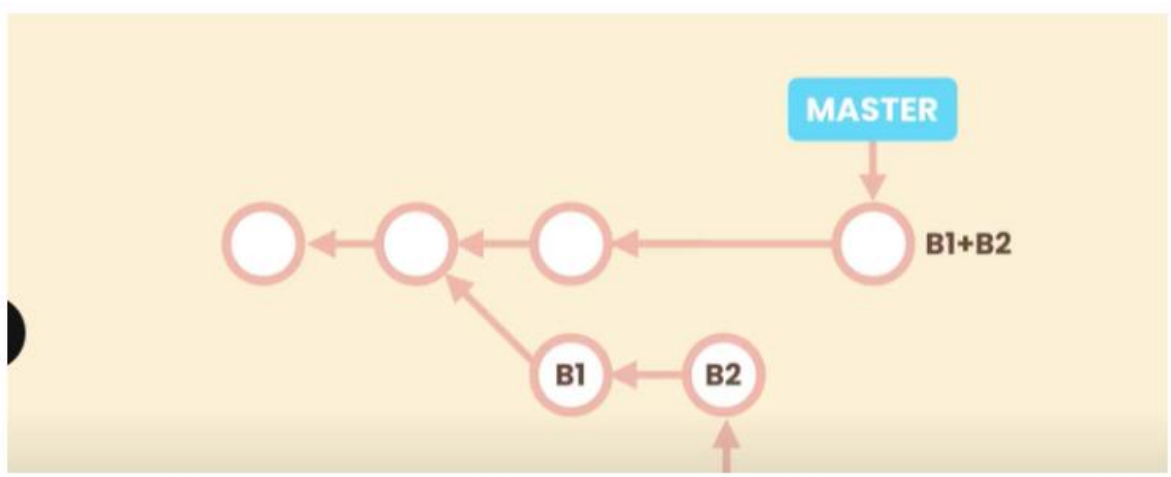
BUGFIX



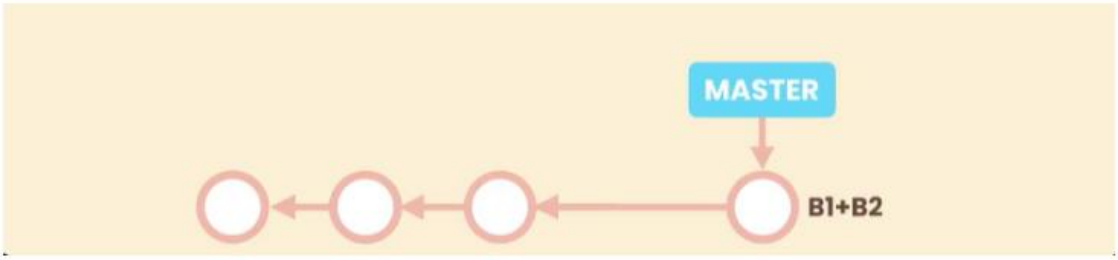
Bugfix



BUGFIX



BUGFIX



We have a bugfix branch where commits B1 & B2 are part of Master's history, but these commits might be:Too fine-grained

Not representing logical changes

Containing mixed changes

When we don't care about preserving the bugfix branch's messy history, we can use **squash merging**:

Undo the original merge (if already done)

Create a new single commit combining all bugfix changes

Apply this clean commit on top of master

git switch -C bugfix/photo-upload

```
› ✔  git switch -C bugfix/photo-upload
Switched to a new branch 'bugfix/photo-upload'
```

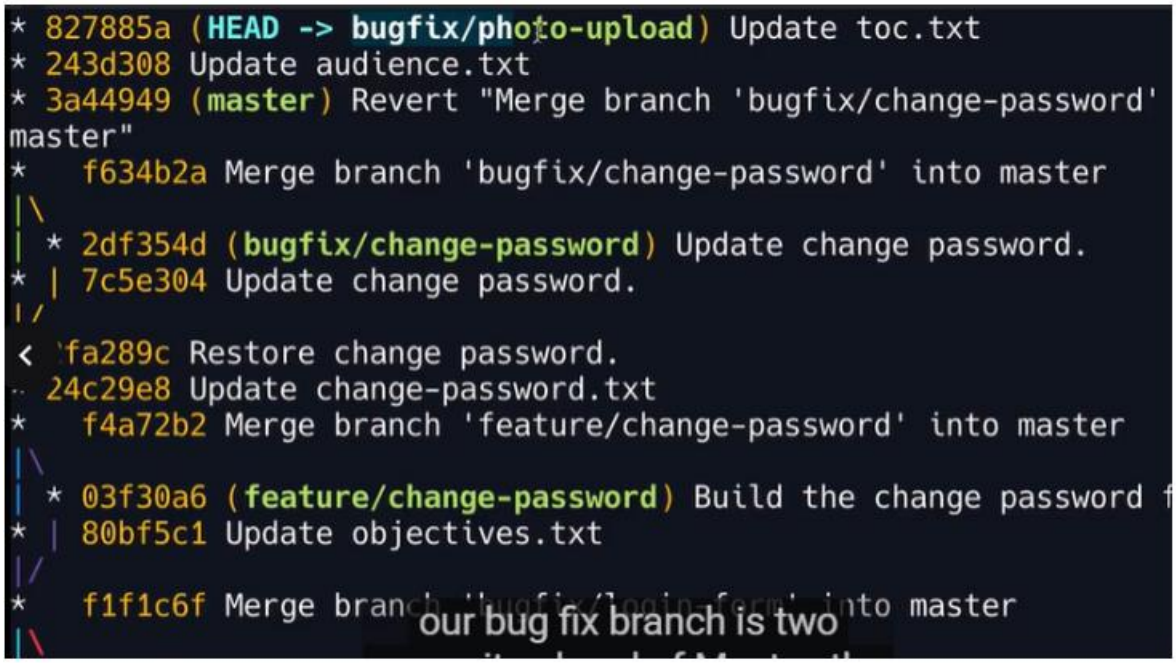
echo bugfix>> audience.txt

git commit -am “update audience txt”

echo bugfix >>toc.txt

git commit -am “update toc.txt”

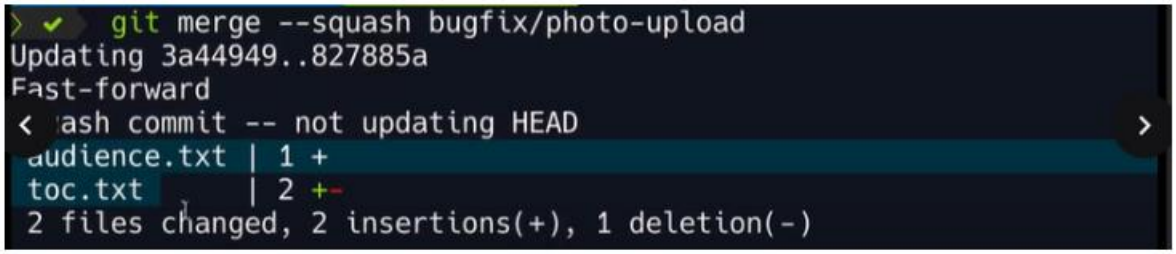
- git log --oneline --all –graph



git switch master

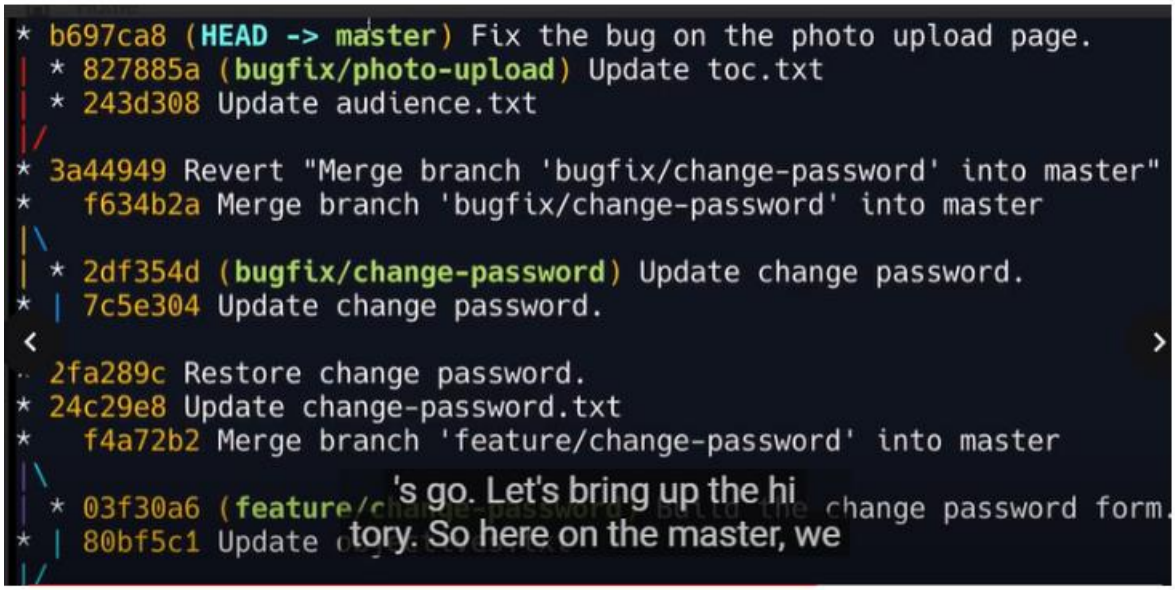
git merge --squash bugfix/photo-upload

- git status -s



git commit -m “fix the bug on photo upload page”

git log --oneline --all --graph

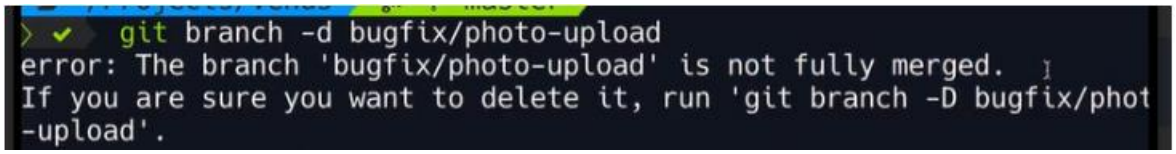


we have a new commit that combines all the changes for fix the bug so once we remove the bugfix branch we have a linear history .But before doing that let me show u different thing

look bugfix/photo-upload branch is not in this list because technically we dont have a merge commit that connects these branches.So when doing a squash merge its super important to remove your target branch otherwise that branch will be sitting there and may create the confusion in the future .

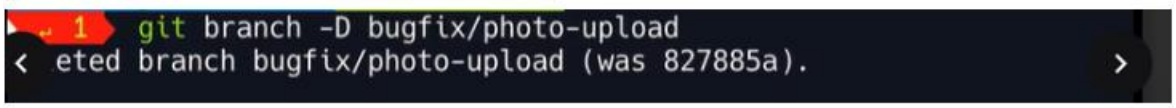
example you might run git branch --no-merged and think u havent merged this branch into master.This is confusing so we type

git branch -d bugfix/photo-upload (D for permanent delete)(error)

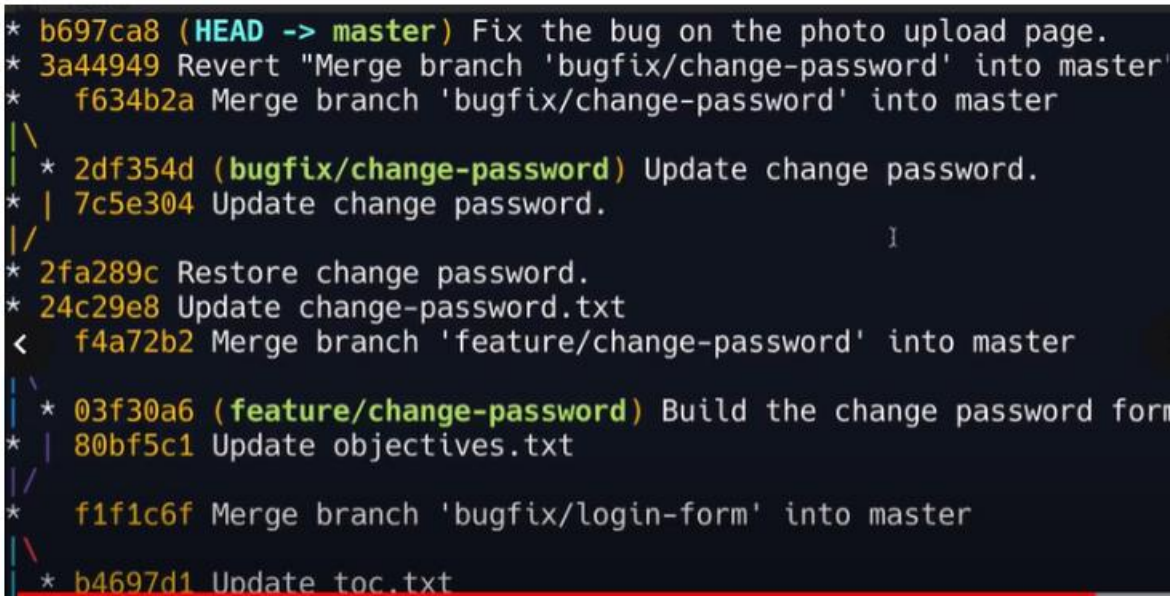


because git doesnot say this as a real merge

so we enforced a deletion



git log --oneline --all --graph



Squash Merge with Conflict Resolution

We now have **a simple linear history** with a single commit containing all bugfix changes for the photo upload page.

When performing a squash merge onto master:

Conflicts may occur (just like regular merges)

Resolution process is identical:

Git pauses the merge

You manually resolve conflicts in files

Mark as resolved with git add

Complete with git commit

git merge --squash bugfix/photo-upload

REBASING:

Rebasing: An Alternative to Merging

Rebasing is another technique for integrating changes from one branch into another. Here's how it works:

Current Situation

We have a feature branch with two commits

If we merge normally, we get non-linear history

The Challenge

While the history isn't overly complicated

With multiple branches and complex scenarios

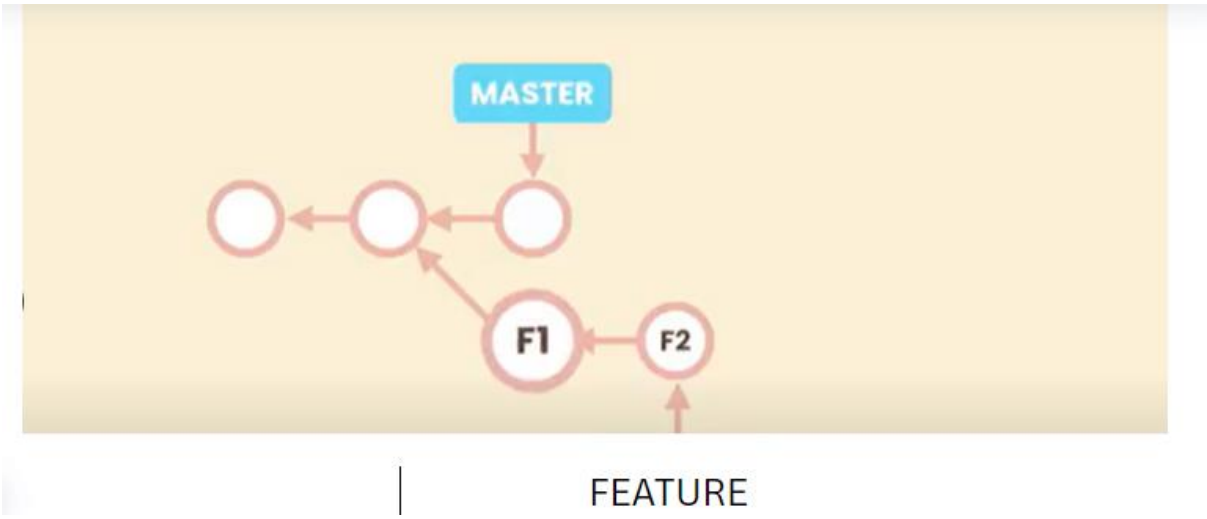
History can become hard to read and follow

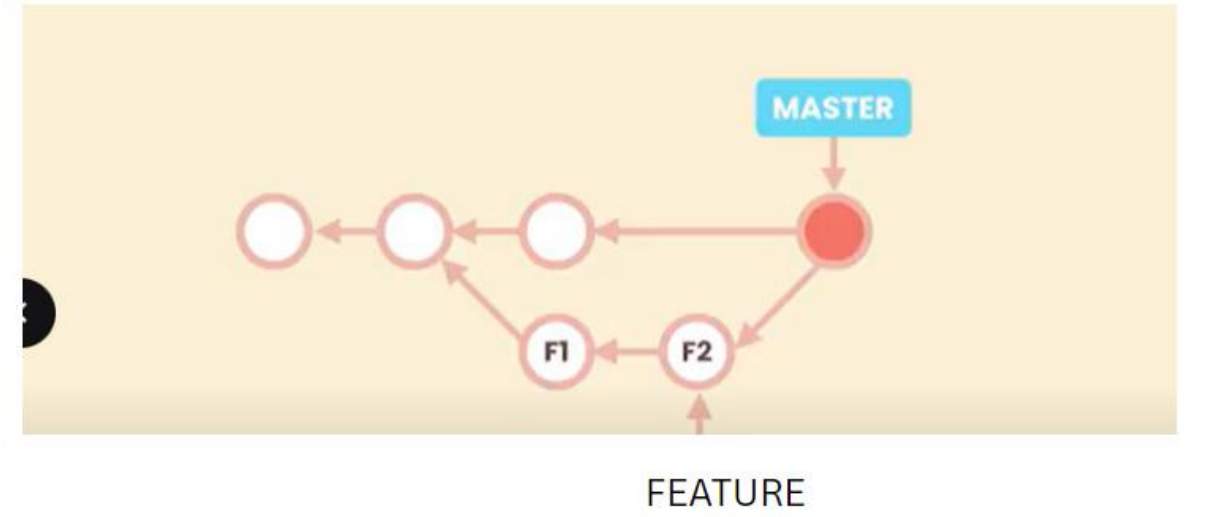
Rebasing Solution

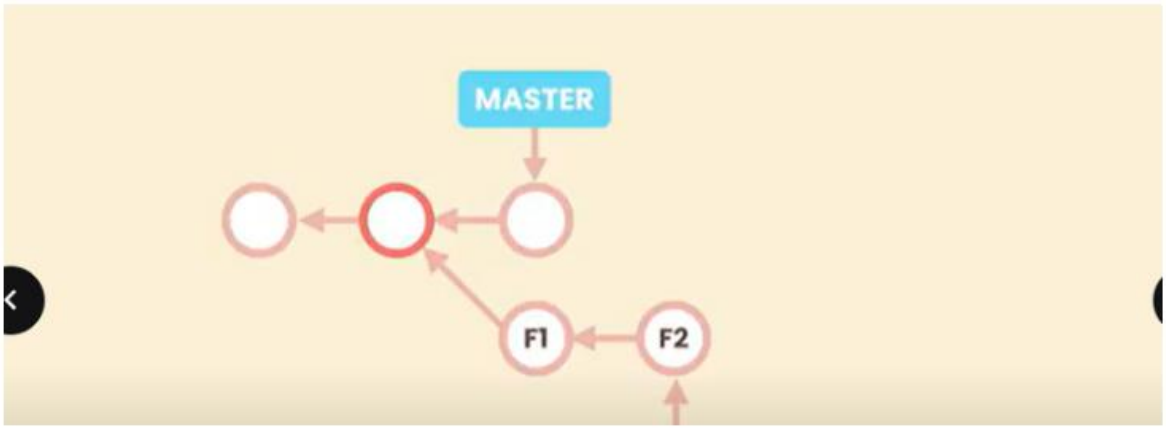
Results in clean, linear history

Replays feature commits on top of target branch
git checkout feature

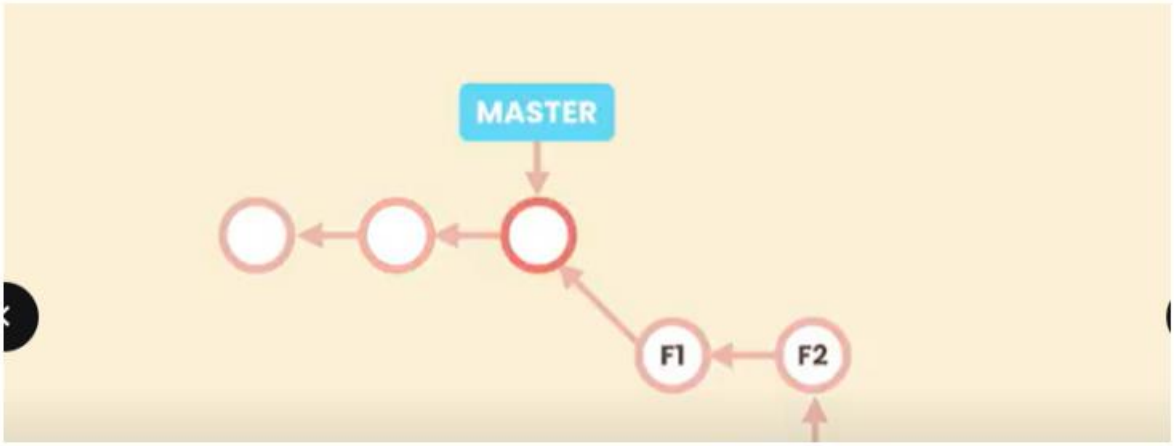
git rebase master







FEATURE



FEATURE

Rebasing for Linear History

Using the rebase command to change the base of our feature branch:

git checkout feature

git rebase master

Purpose:

1. Bases our feature branch on the latest master commit
2. Creates a direct linear path from feature → master

Result:

Enables **fast-forward merge** into master

Maintains **clean, linear history**

Avoids unnecessary merge commits

Before rebase:

master: A---B

feature: \---C---D

After rebase:

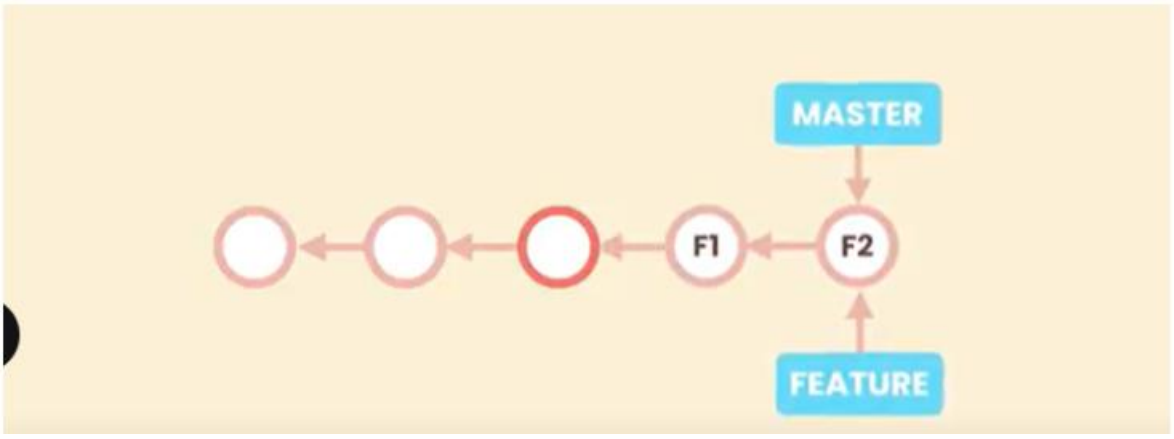
master: A---B

feature: C'---D'

After merge:

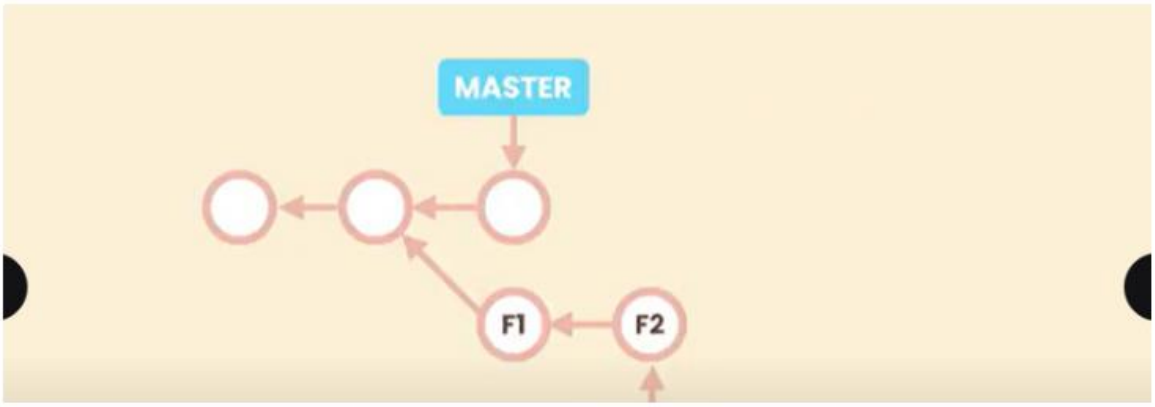
master: A---B---C'---D'

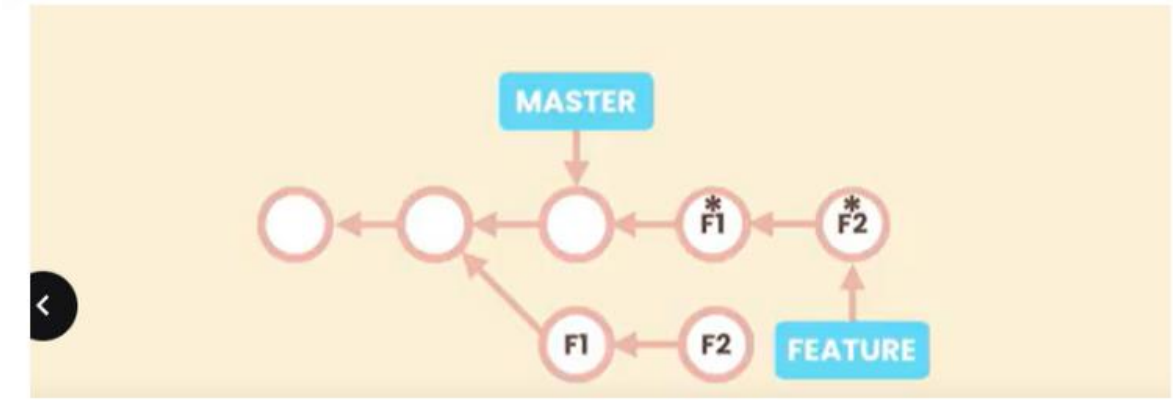
This process is called **rebasing** - rewriting feature commits on top of master's tip.



It looks like a good idea, but be cautious with **rebasing** because **rebasing rewrites history**.
That means you should use rebasing **only for branches or commits that are local** in your repository.
If you have **shared this commit with other people** on your team — if you have **pushed your changes** — you **shouldn't use rebasing**.
Otherwise, you're going to make a big mess.

Let me explain why **rebasing rewrites history**. This is the state **before** rebasing





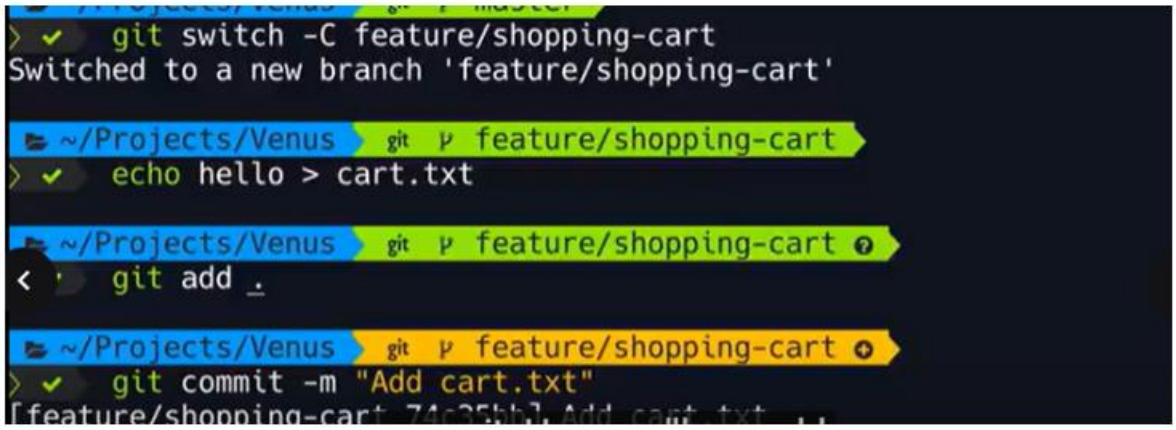
When you're rebasing the **feature** branch, Git is **not** going to change the base or parent commit F1, because **commits in Git are immutable** — they cannot be changed.
So, Git is going to look at this commit and create **new commits** that look like F1 and F2.

We don't have any branches or other commits pointed to the **original** F1 and F2, so at some point in the future, Git is going to **remove** those commits.

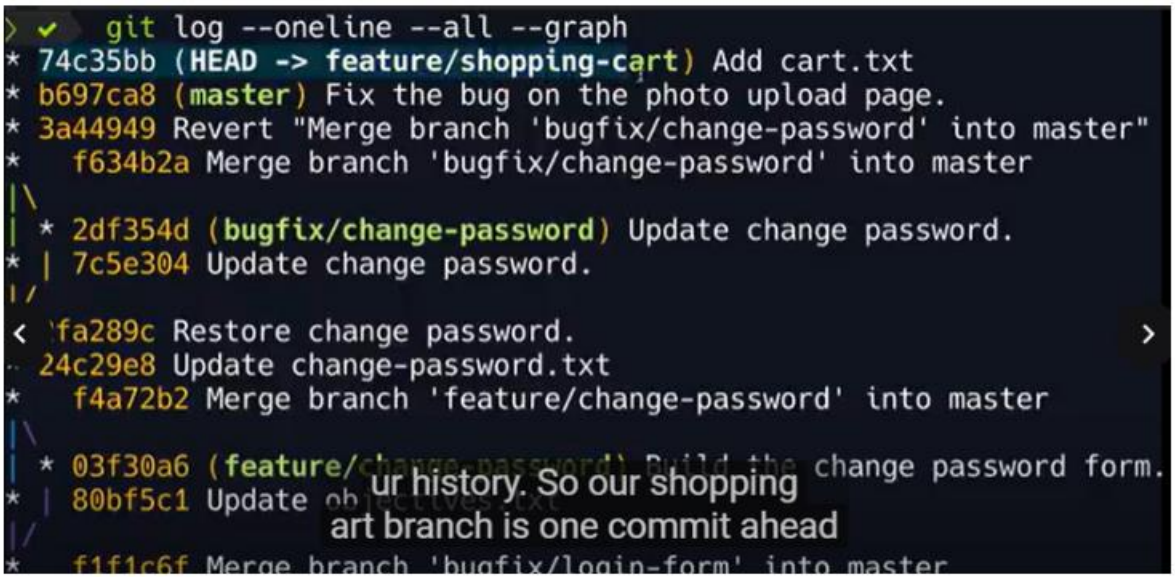
So, we are essentially **rewriting history** — these two commits (F1 and F2) are **not the same** as before.

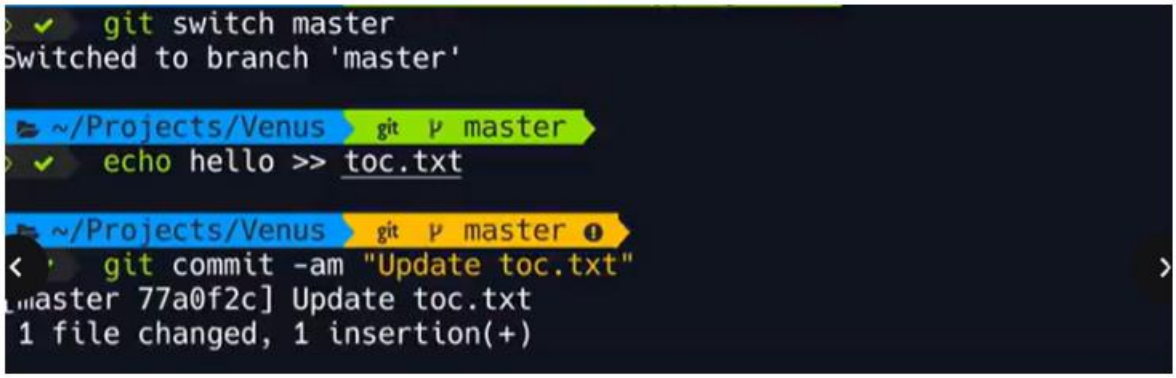
Now, if we had already **shared** F1 and F2 with others, and someone created a new commit on top of F2,
then after rebasing, **their Git history is going to get screwed**.

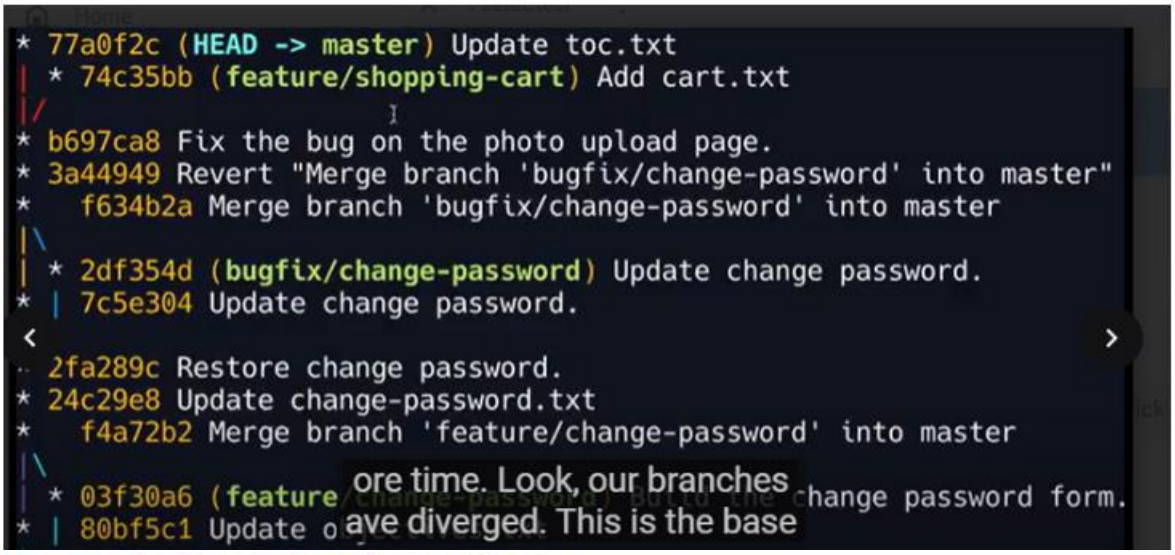
Let's create a new feature branch called **shopping-cart**.

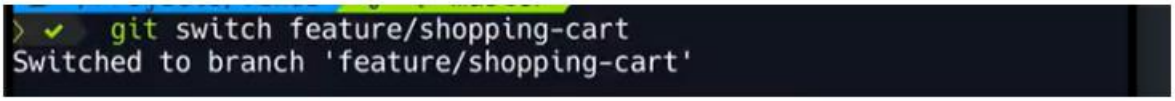


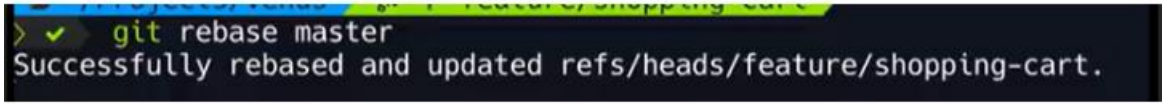
git log --oneline –graph





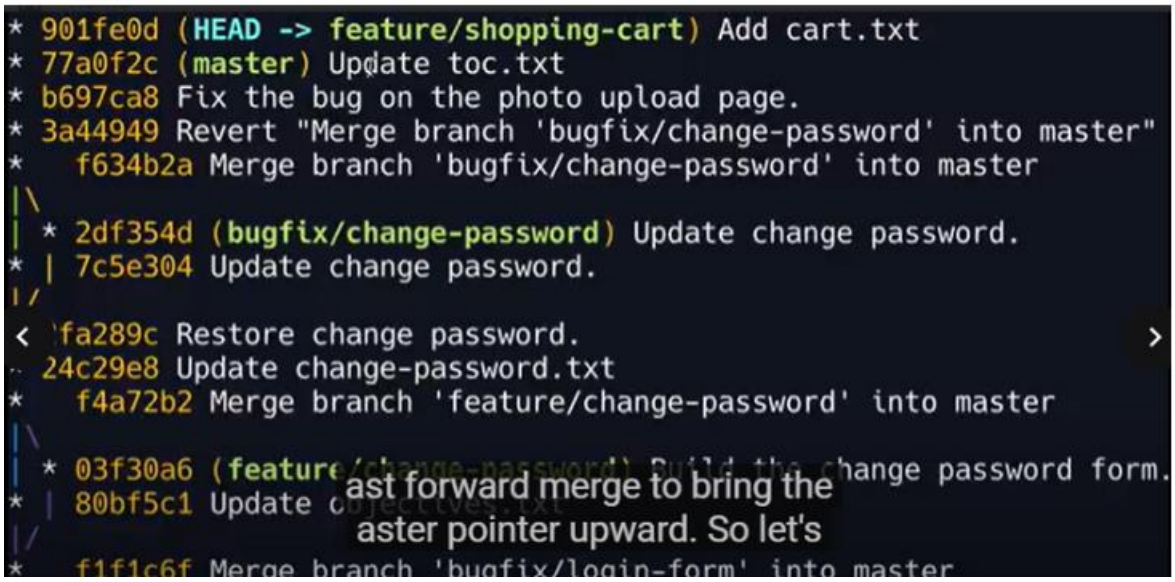




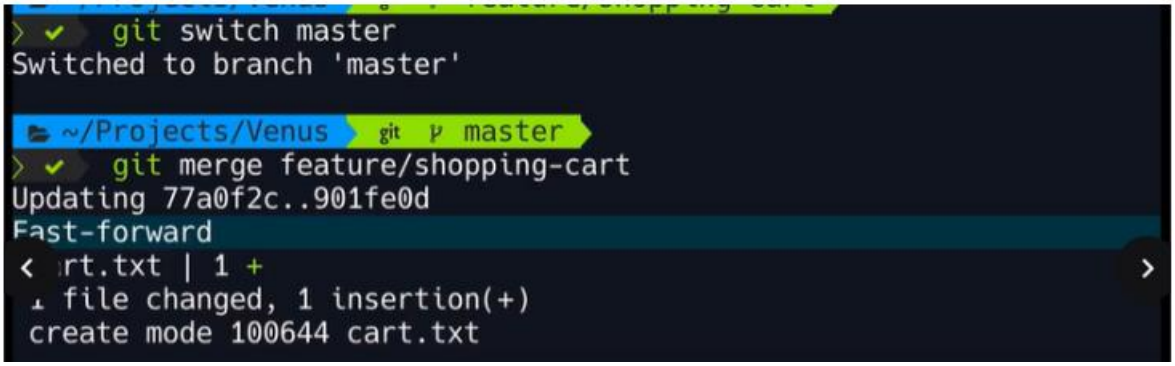


that is not the case in the real world .Most of the time rebase endup with the conflicts .

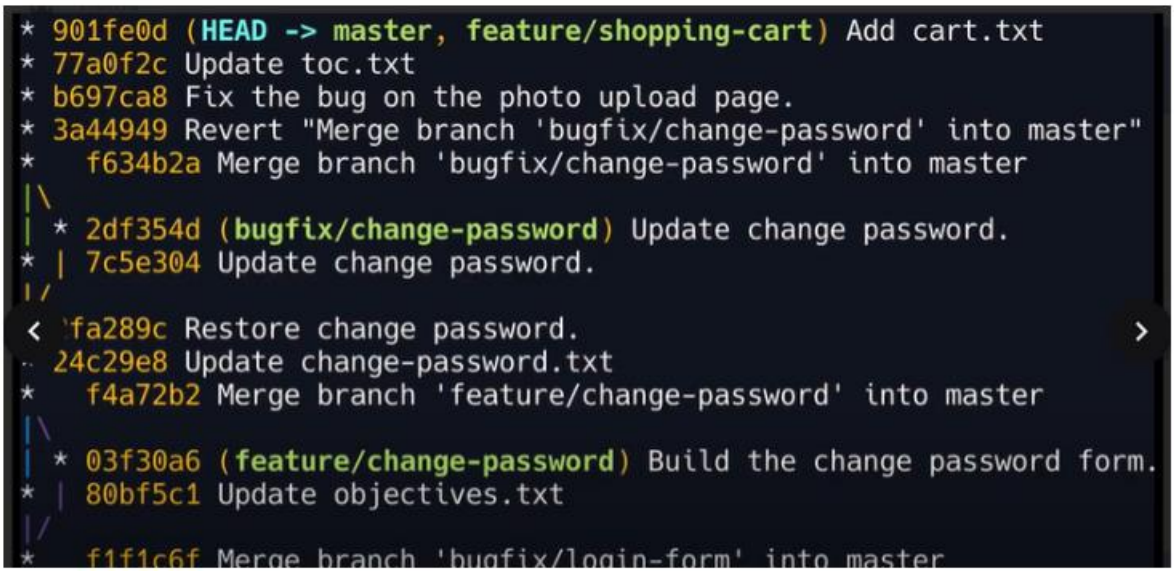
git log --oneline --all --graph



now fast forward merge so

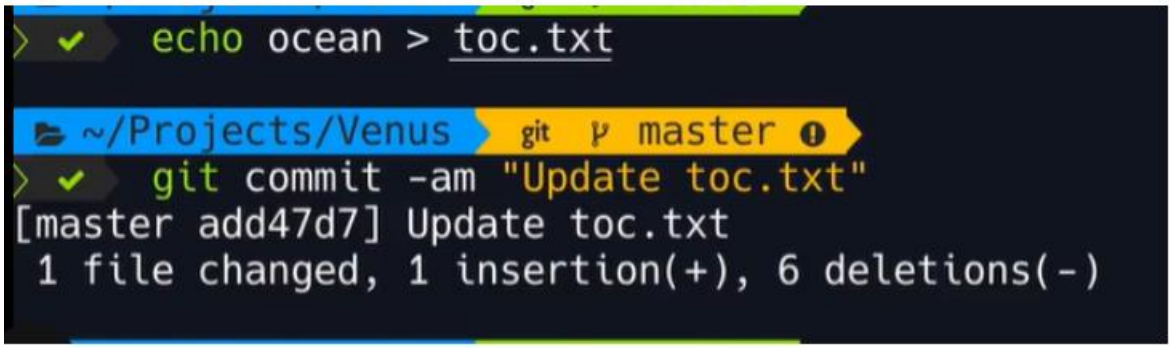


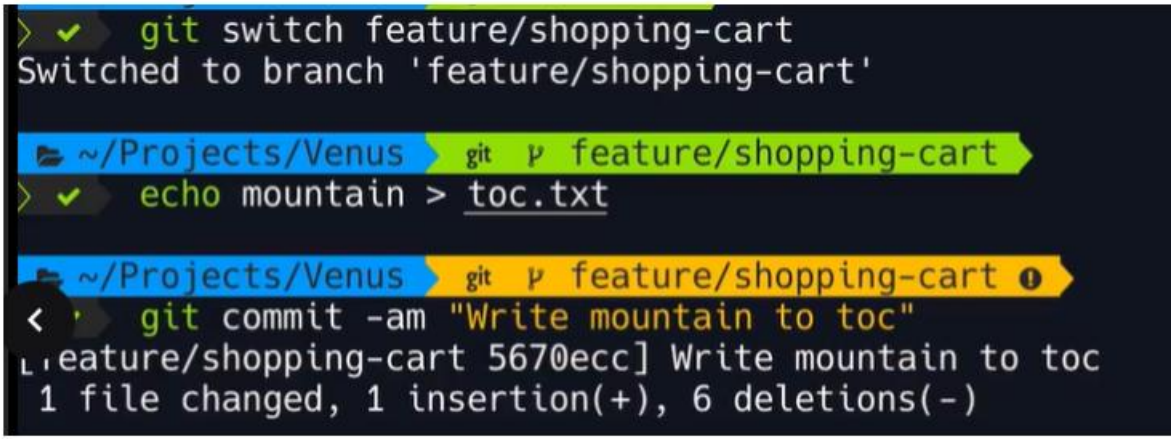
git log --oneline --all --graph



so both the branches are on the same and we have a simple linear history.

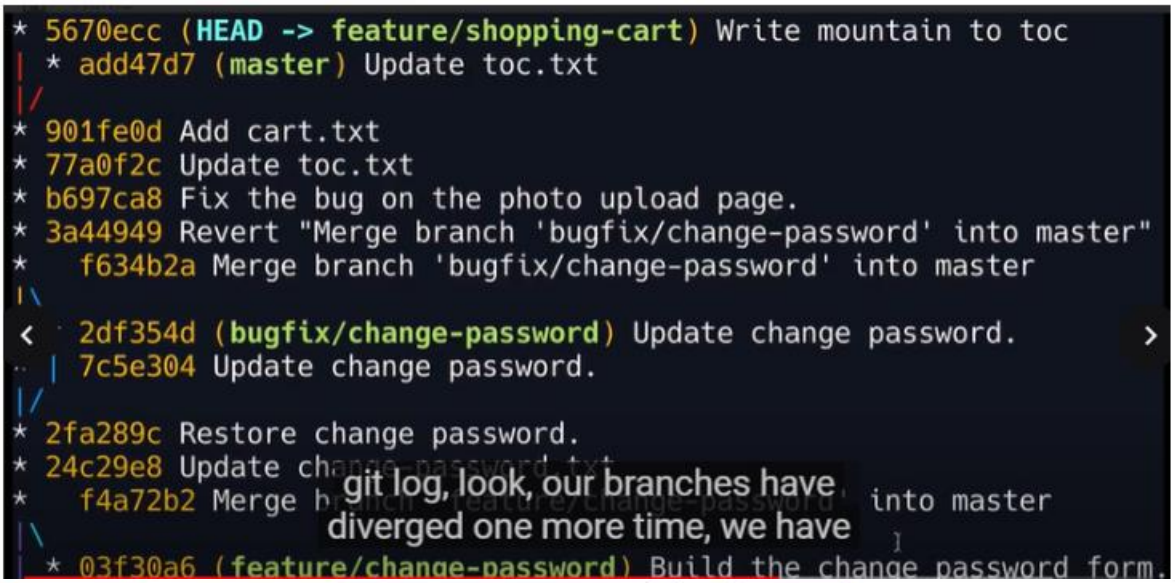
IF CONFLICTS WHAT HAPPENED

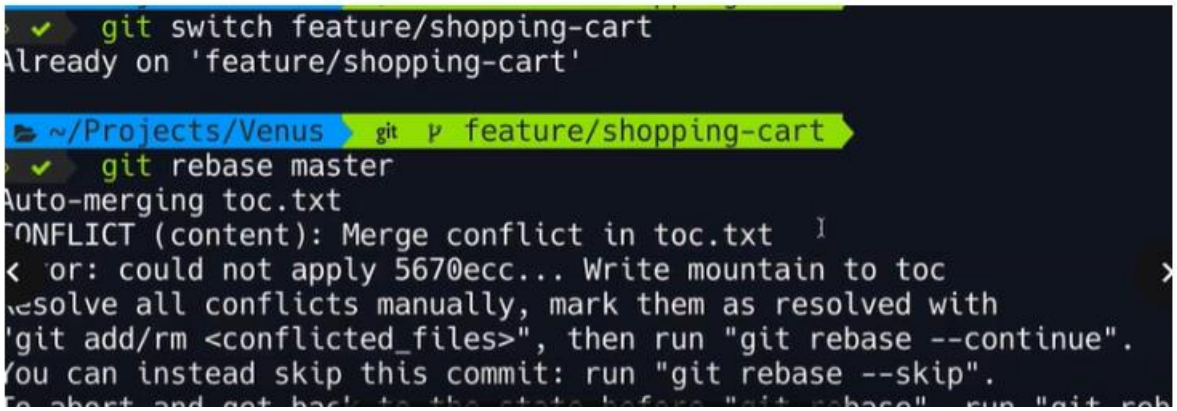




Now branch r diverged and we have conflicting change so verify it

git log --oneline --all --graph





so we should resolve the conflict exactly like before we r going to launch merge tool

git merge tool

git rebase --abort(if we dont have enough time for resolving all these conflicts and going through a complete rebase we can abort the rebase here)

git status

```
git status
On branch feature/shopping-cart
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        toc.txt.orig

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

A mergetool — in this case, pmerge — **creates this file automatically as a backup**.
(So, look at the contents of this file:)
cat toc.txt.orig

```
cat toc.txt.orig
<<<<<<< HEAD
ocean
=======
mountain
>>>>>>> 5670ecc... Write mountain to toc
```

This is our toc file **before** we resolve the conflict.
We **don't want this file in our repo**, so we remove it:
git clean -fd
For the future, to **eliminate these automatic backup files** created by the pmerge tool, we use:
git config --global mergetool.keepBackup false

…….
Here are 20 multiple-choice questions covering the key concepts from the branching and merging discussions:

Multiple Choice Questions: Branching & Merging

Branching (5 Questions)

1. What happens when you create a new branch in Git?

a) Copies all repository files to a new folder

b) Creates a new pointer to the current commit

c) Downloads remote changes automatically

d) Deletes the master branch

Answer: b

2. Which command safely deletes a merged branch?

a) `git branch -D`

b) `git branch -m`

c) `git branch -d`

d) `git switch -d`

Answer: c

3. How does Git represent branches internally?

a) As full copies of the working directory

b) As 40-byte files containing commit hashes

c) As SQL database entries

d) As ZIP archives

Answer: b

4. What does `git switch -C feature/login` do?

a) Creates and switches to a new branch, forcing overwrite

b) Compares two branches

c) Deletes a remote branch

d) Shows commit history

Answer: a

5. Which command shows differences between branches?

a) `git log branch1..branch2`

b) `git diff branch1..branch2`

c) `git merge --diff branch1 branch2`

d) `git status --compare`

Answer: b

Merging (8 Questions)

6. A fast-forward merge is possible when:

a) Branches have diverged

b) Target branch has no new commits

c) Merge conflicts exist

d) Working directory is dirty

Answer: b

7. What does a 3-way merge use? (Choose 3)

a) Two branch tips

b) Common ancestor

c) Staging area content

d) Remote repository state

Answers: a, b

8. Merge conflicts occur when:

a) Same file is modified in both branches

b) Branches have different names

c) Commits are more than 1 week old

d) Git config disallows merging

Answer: a

9. To abort a conflicted merge:

a) `git reset --hard`

b) `git merge --abort`

c) `git revert HEAD`

d) `git commit --skip`

Answer: b

10. What does `git merge --no-ff` do?

a) Forces fast-forward

b) Creates a merge commit always

c) Skips conflict detection

d) Deletes source branch

Answer: b

11. Conflict markers include:

a) `<<<<<<< HEAD`

b) `=======`

c) `>>>>>>> branch`

d) All of above

Answer: d

12. After resolving conflicts, you must:

a) `git add` then `git commit`

b) `git --continue`

c) `git push --force`

d) Delete the branch

Answer: a

13. Squash merging is ideal for:

a) Long-running feature branches

b) Bugfix branches with messy history

c) Release branches

d) All merge scenarios

Answer: b

Rebasing (4 Questions)

14. Rebasing creates:

a) Non-linear history

b) New commit hashes

c) Merge conflicts always

d) Tagged releases

Answer: b

15. The golden rule of rebasing is:

a) Rebase shared branches often

b) Never rebase public history

c) Always squash when rebasing

d) Skip conflict resolution

Answer: b

16. After rebasing, you typically:

a) Create a new repository

b) Force push the branch

c) Delete all tags

d) Revert HEAD

Answer: b

17. Rebasing is preferred over merging when:

a) Preserving exact commit timing is critical

b) You want cleaner project history

c) Working with large binary files

d) Multiple teams collaborate on a branch

Answer: b

Advanced (3 Questions)

18. `git reset --soft HEAD~1` affects:

a) Working directory + staging

b) Only HEAD pointer

c) HEAD + staging

d) Remote repository

Answer: c

19. To undo a public merge commit:

a) `git reset --hard`

b) `git revert -m 1 <hash>`

c) `git branch -D`

d) `git stash pop`

Answer: b

20. Stashing is used to:

a) Share unfinished work

b) Temporarily save uncommitted changes

c) Create backup repositories

d) Tag releases

Answer: b

7. Remote Repositories (GitHub)

1. Introduction to GitHub

GitHub is a powerful platform that serves as a hub for software development and version control. It enables developers to collaborate on projects, track changes, and manage code efficiently. Below are the key elements that define GitHub:

Key Features of GitHub

- **Version Control**: Built on Git, GitHub tracks changes in code over time, allowing developers to revert to previous versions if necessary.
- **Collaboration**: Multiple developers can work on the same project simultaneously, making it easier to manage contributions from different team members.
- **Open Source**: GitHub hosts millions of open-source projects, enabling developers to share their work and contribute to others' projects.
- **Community Engagement**: Users can follow other developers, star repositories they find interesting, and participate in discussions through issues and pull requests.

Basic Terminology

- **Repository (Repo)**: A storage space for your project files and their revision history.
- **Branch**: A parallel version of the repository where you can make changes without affecting the main codebase.
- **Commit**: A snapshot of your repository at a specific point in time, along with a message describing the changes made.
- **Pull Request (PR)**: A request to merge changes from one branch into another, allowing for code review and discussion before integration.

Getting Started

To start using GitHub:

1. Create an account at GitHub.com.
2. Set up Git on your local machine.
3. Create a new repository or clone an existing one to begin collaborating.

Prerequisites

Git installed on your machine (Download Git)

A GitHub account

Basic command-line knowledge

Copy a remote repository to your local machine.
git clone <repository-url>
Example:
git clone https://github.com/octocat/Hello-World.git

Lab:

Go to any public repository on GitHub.

Click on the **"Code"** button and copy the URL.

Open your terminal and run:
git clone <paste-url-here>

cd <repo-name>

ls

Connect your local repository to a remote repository on GitHub.
git remote add <name> <url>
Example:
git remote add origin https://github.com/username/my-repo.git

Lab:

Create a new repository on GitHub (don't initialize with README).

On your local system:

git init my-repo

cd my-repo

git remote add origin https://github.com/<your-username>/my-repo.git

git push: Upload local changes to the remote repo.
git push origin main

git pull: Download and merge remote changes to local.

git pull origin main

Example Workflow:
echo "# My Project" >> README.md

git add README.md

git commit -m "Add README"

git push origin main

Lab:

1. Make a change to any file.
2. Run:
   git add .

   git commit -m "My update"

   git push origin main

   Ask your peer to pull your changes:
   git pull origin main

   Download remote changes **without merging** into your current branch.
   git fetch origin
   Example:
   git fetch origin

   git log origin/main

   Lab:

   Your teammate pushes a new commit.

   Instead of git pull, try:
   git fetch origin

   git diff origin/main

Chapter 8. Collaboration Workflows

1. Introduction to GitHub Collaboration Workflows

What are Collaboration Workflows? GitHub Collaboration Workflows are a set of practices and tools that enable developers to work together efficiently on shared codebases. These workflows enhance productivity, ensure code quality, and foster community engagement.

2. Forking Repositories

Purpose

Forking allows a user to create a personal copy of someone else's repository. It's typically used to propose changes to someone else's project or to use a project as a starting point for your own idea.

Key Concepts

- Forked repositories remain connected to the original (upstream) repository.

Ideal for contributing to open-source projects.

Steps

1. Go to the repository you want to fork.
2. Click the **Fork** button (top-right corner).
3. GitHub creates a copy under your GitHub account.
4. Clone the fork to your local machine:

git clone https://github.com/your-username/forked-repo.git

Lab Activity

Fork the octocat/Spoon-Knife repository.

Clone it locally.

Explore and understand the project structure.

Pull Requests (PRs)

Purpose

A Pull Request (PR) is a way to propose changes to a repository. It allows team members to review, discuss, and merge changes.

Key Concepts

PRs are typically made from feature branches or forked repositories.

Enables code review and feedback before merging.

Steps

Create a new branch:
git checkout -b feature-branch

Make and commit changes:

git commit -am "Added new feature"

Push the branch to your fork:
git push origin feature-branch
Go to GitHub and open a Pull Request from your branch to the original repository's main branch.

Lab Activity

Modify the README file in your forked repo.
Create a new branch and push changes.
Open a Pull Request.
Review and comment on a teammate's PR.

Issues & Milestones

Purpose

Issues are used to track tasks, bugs, or enhancements. Milestones group issues and PRs into a common goal.

Key Concepts

Issues can be assigned, labeled, and referenced in commits/PRs.
Milestones are used for planning and tracking progress.

Steps to Use Issues

1. Navigate to the **Issues** tab of a repository.
2. Click **New Issue**, provide a title and description.
3. Use labels like bug, enhancement, or question.
4. Assign the issue and link to a milestone.

Steps to Use Milestones

1. Go to the **Milestones** section.
2. Create a new milestone and set a due date.
3. Link issues and PRs to the milestone.

Lab Activity

Create a new issue describing a bug or enhancement.

Assign labels and yourself.
Create a milestone named "v1.0 Release" and link the issue.

By mastering these collaboration workflows—Forking, Pull Requests, and Issues & Milestones—you gain essential skills for contributing to modern software development. These workflows not only streamline development but also promote transparency and accountability in teams.

Multiple Choice Questions

1. **What is the purpose of forking a repository?**
    - A. To delete the original repository
    - B. To create a personal copy of a repository
    - C. To reset the main branch
    - D. To merge branches automatically
    - **Answer:** B
2. **What command is used to clone a repository locally?**
    - A. git init
    - B. git commit
    - C. git clone
    - D. git fork
    - **Answer:** C

3. **What does a Pull Request allow developers to do?**
    - A. Create a new repository
    - B. Merge code without approval
    - C. Propose and review code changes
    - D. Delete issues
    - **Answer:** C
4. **Where do you go to create a new issue in GitHub?**
    - A. Commits tab
    - B. Pull Requests tab
    - C. Actions tab
    - D. Issues tab
    - **Answer:** D
5. **Which of the following is used to group related issues and pull requests?**
    - A. Tags
    - B. Milestones
    - C. Branches
    - D. Releases
    - **Answer:** B
6. **What keyword can be used in commits to automatically close an issue?**
    - A. Close now
    - B. Done #1
    - C. Fixes #issue-number
    - D. Shut #1
    - **Answer:** C
7. **What is a key benefit of using branches for each feature or fix?**
    - A. Avoids internet usage
    - B. Prevents data loss
    - C. Isolates changes for easier management
    - D. Removes the need for pull requests
    - **Answer:** C
8. **What happens after pushing your feature branch to GitHub?**

- A. It automatically merges to the main branch
- B. It creates an issue
- C. You can open a pull request
- D. It deletes the upstream repository
- **Answer:** C

9. **How can contributors discuss a proposed code change?**
    - A. Through milestones
    - B. By commenting on pull requests
    - C. By editing directly
    - D. Through commits only
    - **Answer:** B
10. **Which tab in GitHub shows all assigned issues and milestones?**
    - A. Projects
    - B. Wiki
    - C. Issues
    - D. Insights
    - **Answer:** C

Chapter 11. GitHub Integration with VS Code

Module 1: Introduction to GitHub & VS Code IntegrationObjective: Understand how VS Code enhances GitHub workflows for seamless collaboration.

Key Benefits

Real-time collaboration (Live Share, Co-editing)
Built-in Git tools (No CLI dependency)
Streamlined PR management
Visual conflict resolution

Lab Setup:

Install VS Code

Install the GitHub Pull Requests & Issues extension

Sign in to GitHub via VS Code (Click the Accounts icon in the bottom-left)

Module 2: Cloning Repositories in VS Code

Steps to Clone a Repo

1. Open VS Code → Press Ctrl+Shift+P → Type **"Git: Clone"**

2. Paste the GitHub repo URL (e.g., https://github.com/user/repo.git)
3. Select a local folder to save the repo.

Lab Exercise:

Clone this sample repo: https://github.com/octocat/Spoon-Knife

Verify files appear in VS Code's Explorer.

Module 3: Committing & Pushing from VS Code
Workflow

1. Stage Changes:

   Open the Source Control tab (Ctrl+Shift+G).

   Click + next to modified files.

2. Commit:

   Enter a commit message → Click the checkmark (✓).

3. Push to GitHub:

   Click the "..." menu → Push.

Lab Exercise:

1. Create a README.md file in your cloned repo.
2. Stage, commit (message: "Added README"), and push.
3. Verify changes on GitHub.com.

Module 4: Managing Pull Requests in VS Code

Creating a PR

Create a new branch:
git checkout -b new-feature
Make changes → Commit → Push the branch.

Click the **GitHub icon** in the Activity Bar → **"Create Pull Request"**.

Reviewing PRs

View inline comments.

Approve/Reject via the **GitHub Pull Requests** tab.

Lab Exercise:

Create a PR for your README.md changes.

Add a reviewer (use a peer's GitHub username).
Module 5: Conflict Resolution in VS Code

Steps to Resolve Conflicts

Fetch latest changes:

git pull origin main
VS Code highlights conflicts. Click **"Accept Current/Incoming Change"**.

Stage resolved files → Commit → Push.
Lab Exercise (Simulate a Conflict):

1. Edit the same line in README.md on GitHub.com and your local repo.
2. Attempt to push → Resolve conflicts using VS Code's merge editor.

Final Lab: End-to-End GitHub + VS Code Workflow

Scenario: Collaborate on a Python script with a team.

1. Clone a repo.
2. Create a branch (git checkout -b fix-bug).
3. Edit script.py → Commit → Push.
4. Create a PR → Resolve a conflict.

Merge to main.

Chapter10. Best Practices & Troubleshooting

Module 1: Meaningful Commit Messages

Objective: Learn to write clear, actionable commit messages that improve collaboration and code history tracking.

Best Practices

Structure: Use the imperative mood ("Fix bug" not "Fixed bug")
Format: Follow the 50/72 rule (50 char subject, 72 char body)
Content: Explain *why* not just *what*
Reference: Link issues (e.g., "Fixes #123")
Examples:
feat: add user authentication middleware

- Implement JWT token verification

- Add error handling for expired tokens

- Related to #45 (auth system overhaul)
Lab Exercise:

1. Create a new file auth.js with basic login function

2. Make three intentional changes (e.g., add validation, error handling)
3. Commit each with proper messages using the format above

Module 2: Branch Naming Conventions

Objective: Standardize branch names for better organization and CI/CD compatibility.

Recommended Convention

<type>/<short-description>-<issue-id>

Types:

feat/ - New features

fix/ - Bug fixes

chore/ - Maintenance tasks

docs/ - Documentation changes

Examples:
feat/user-profile-avatar-89

fix/login-timeout-error-142

Lab Exercise:

1. Create branches for these scenarios:

   Adding dark mode (issue #201)

   Fixing payment API bug (issue #156)
2. Verify names match convention:
3. `git branch`

Module 3: Handling Large Repositories

Objective: Manage repositories with large files/history without performance degradation.

Solutions

Git LFS (Large File Storage):

git lfs install

git lfs track "*.psd"

git add .gitattributes

Shallow Cloning
git clone --depth 1 https://github.com/large-repo

Module 4: Recovering Lost Commits (git reflog**)**

Objective: Master commit recovery techniques using Git's safety net.

Step-by-Step Recovery

1. View reflog:

   git reflog

   # Shows: HEAD@{0}: reset: moving to HEAD~2

   Identify lost commit hash:
   abc1234 HEAD@{5}: commit: Add critical feature

   Restore:
   git checkout abc1234

   git branch recovery-branch
   Lab Exercise:

1. Make 3 test commits
2. Accidentally run:

   git reset --hard HEAD~2

   Use reflog to recover the "lost" commit

   ……..END>…….