1. **Git Distributed Version Control system**

Fundamental Concepts

creating a Snapshots(committing)

Browsing Project history

Branching and Merging

Collaborating Using Github

Rewriting History

Fundamental Concepts

Before the widespread adoption of Git, developers primarily used centralized version control systems (VCS) such as Subversion (SVN) and Concurrent Versions System (CVS). These systems had a central repository where all code changes were stored, and developers would check out a copy of the codebase, make changes locally, and then commit those changes back to the central repository.

limitations:

1.Centralized Repository: All changes were stored in a central server, which meant that if the central server went down, developers couldn't access the codebase or commit changes.

2. Concurrency Issues: Concurrent edits by multiple developers could lead to conflicts that were difficult to resolve, especially if they occurred frequently.

3. Limited Branching and Merging: Branching and merging in centralized systems were often complex and error-prone, leading to difficulties in managing parallel development efforts.

4. Performance: As the project history grew, operations such as checking out or updating the codebase became slower due to the centralized nature of the system.

Git, introduced by Linus Torvalds in 2005, addressed these limitations by providing a distributed version control system (DVCS) with the following advantages:

1.Distributed Nature: Every developer has a complete copy of the repository, including its full history. This means that even if a central server goes down, developers can continue working with their local copies.

2. Efficient Branching and Merging: Git simplifies branching and merging, making it easier for developers to work on multiple features concurrently and merge changes back into the main codebase.

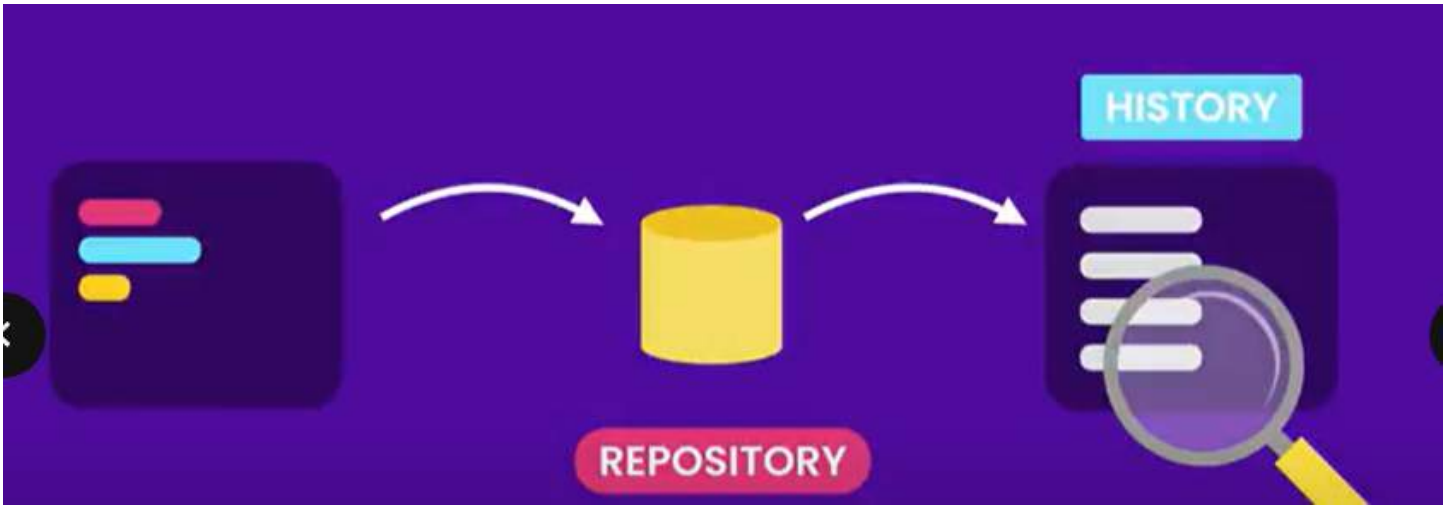
3. Fast Performance: Git is designed to be fast, even with large codebases and extensive histories, due to its distributed architecture and efficient data storage mechanisms.

4. Support for Non-linear Development: Git supports non-linear development workflows, allowing for various branching strategies like feature branches, release branches, and hotfix branches.

5. Built-in Staging Area: Git has a staging area where changes can be reviewed and selectively committed, providing finer control over the commit process.

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.

Version Control System Records The Changes made to our code over time-----special database called repository-----we can see the project history who has made what changes when and why and if screw something up we can revert our project back to earlier state.



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

1. Note Branching and Merging are slow in other version control system like subversion But Git is very Fast
2. 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)

-

Install Git

git --version

I Used Git bash to run git command (windows)

A. Need 04 Setting

- Name
- Email
- Default editor
- Line Ending

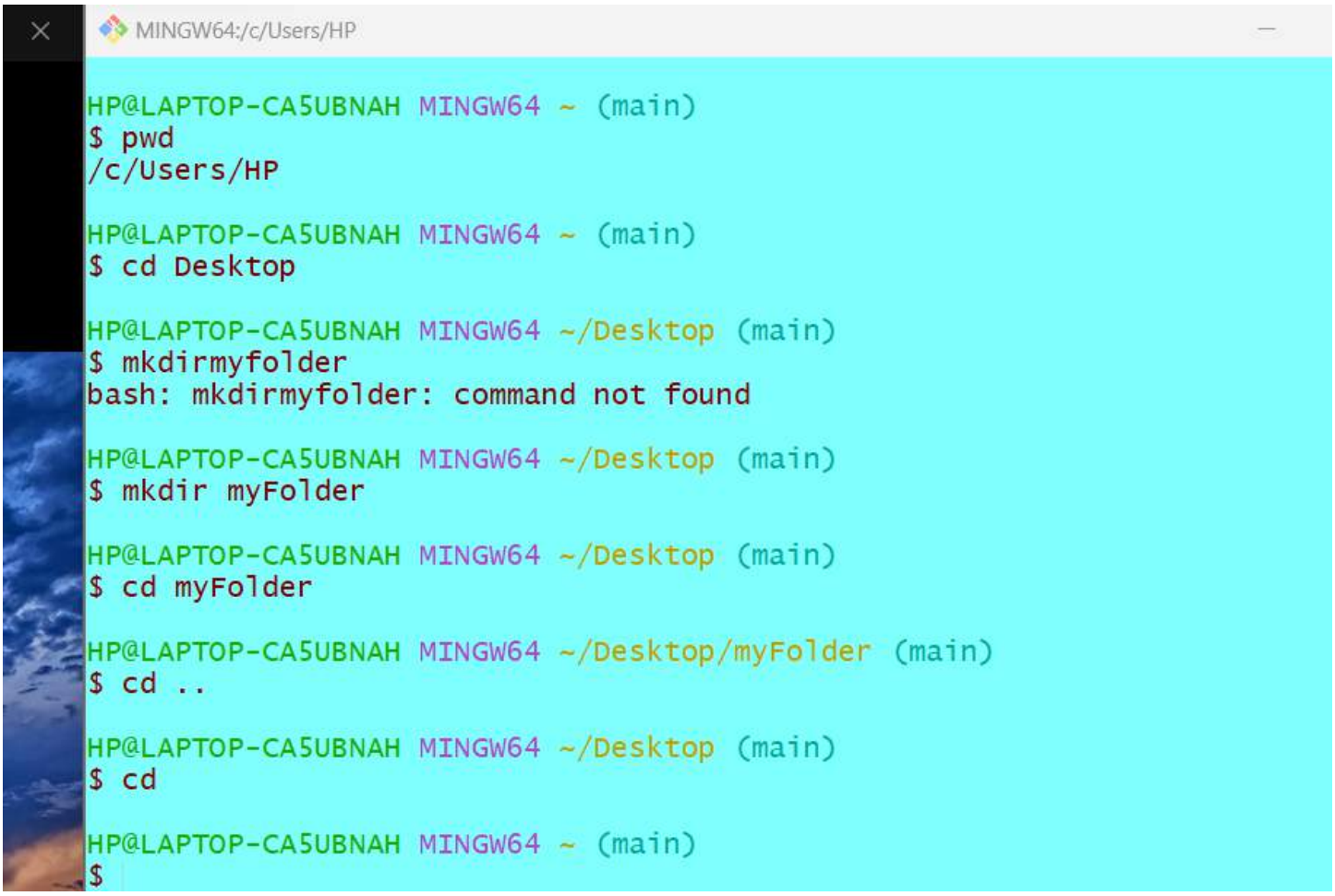
3 Levels

1.System.....All Users

2.Global....All Repositories of the current user

3.Local .....the Current Repository

Basics Of terminals



On Terminal Window

1.git config --global user.name “MazharHuda”

2.git config --global user.email cloudindia001@gmail.com

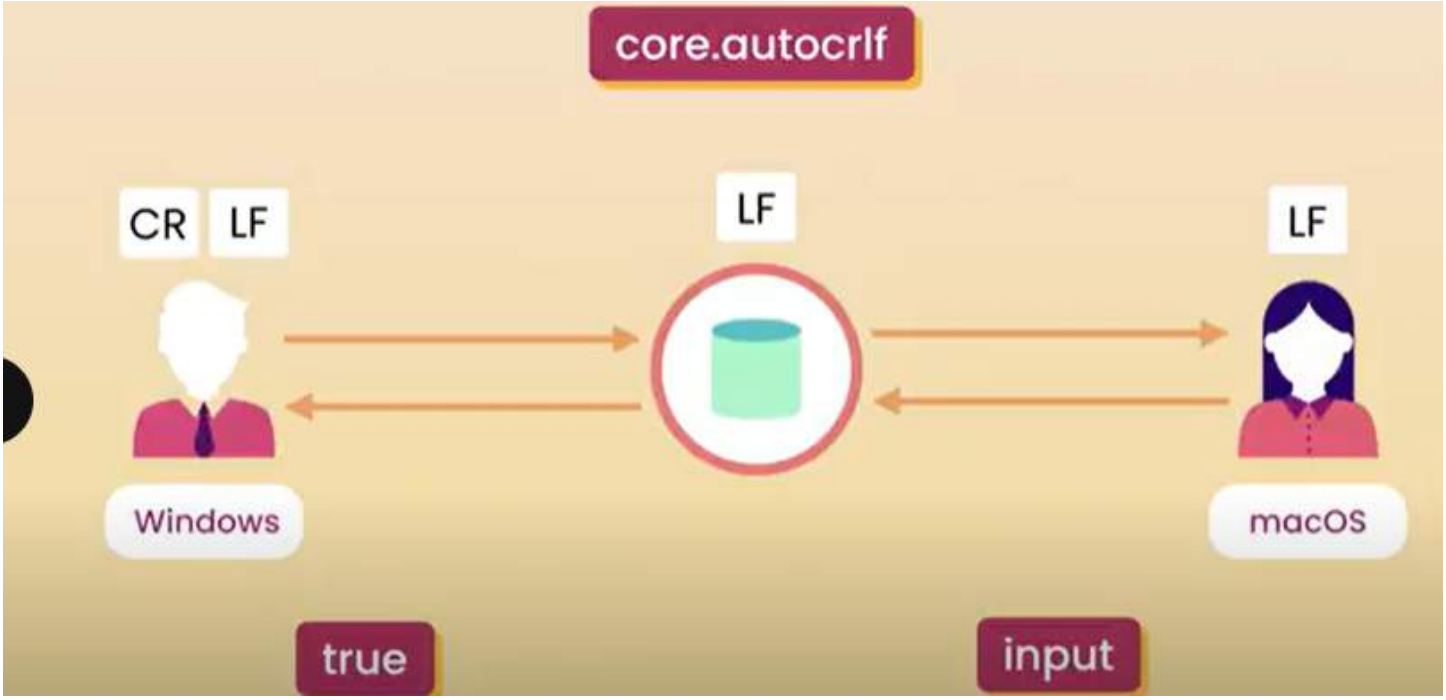
3.git config --global core.editor “code --wait”(wait until we close the window)

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)



git config --global core.autocrlf true(window)

git config --global core.autocrlf input(Mac)

git config --help

or git config -h

B.CREATING A SNAPSHOT(COMMIT)

Hands On

mkdir Moon001

cd Moon001

First Initialized the new empty repository

git init

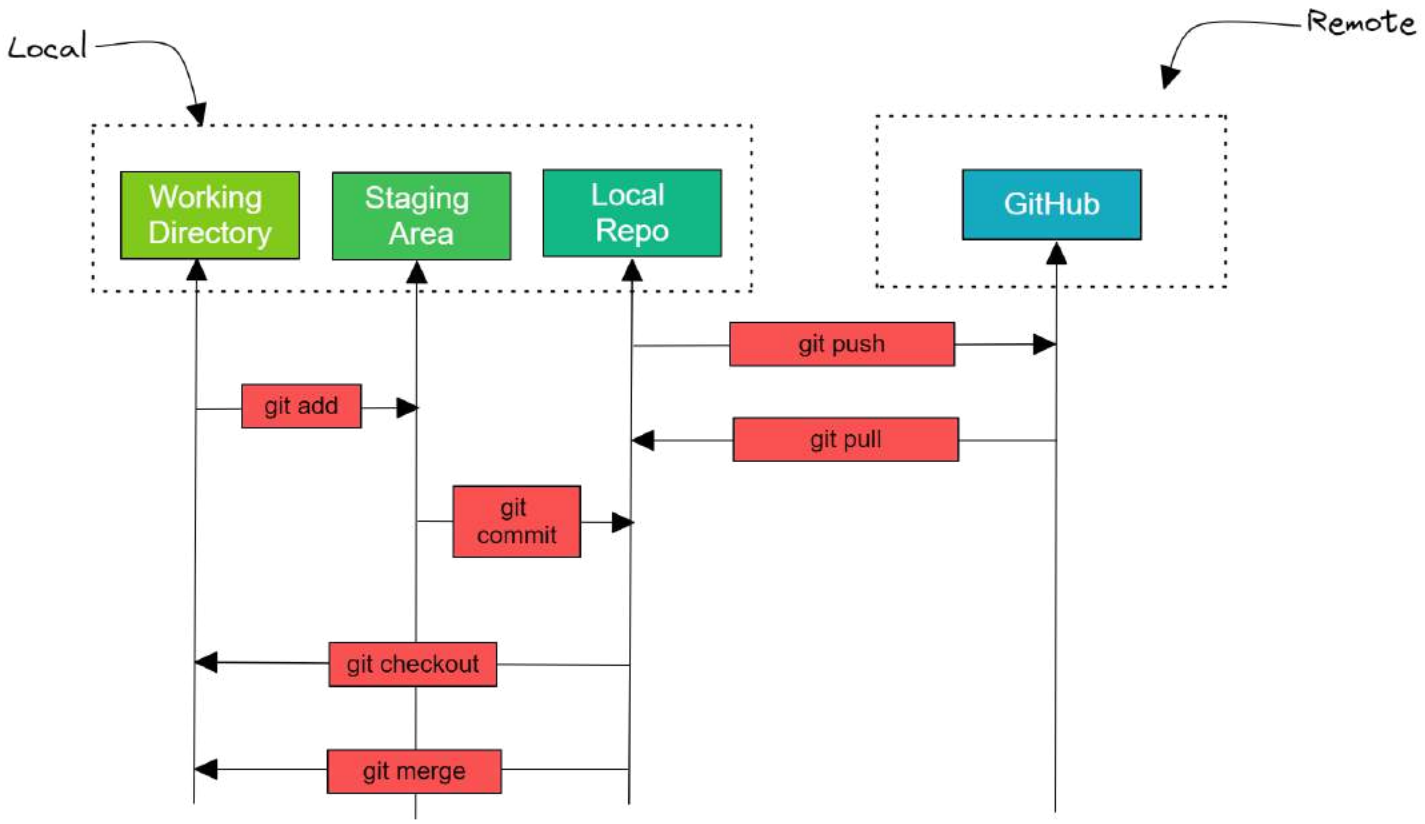
inside it we have subdirectory .git/ by default it is hidden

- ls
- ls -a (.git)

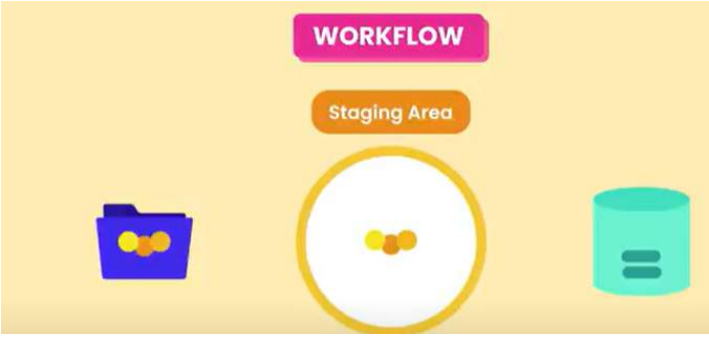
open.git (just see in gui)(dont need to understand this structure)

dont touch hidden directory .git if we delete it rm -rf .git

you have to again initialize it git init



BASIC GIT WORKFLOW



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

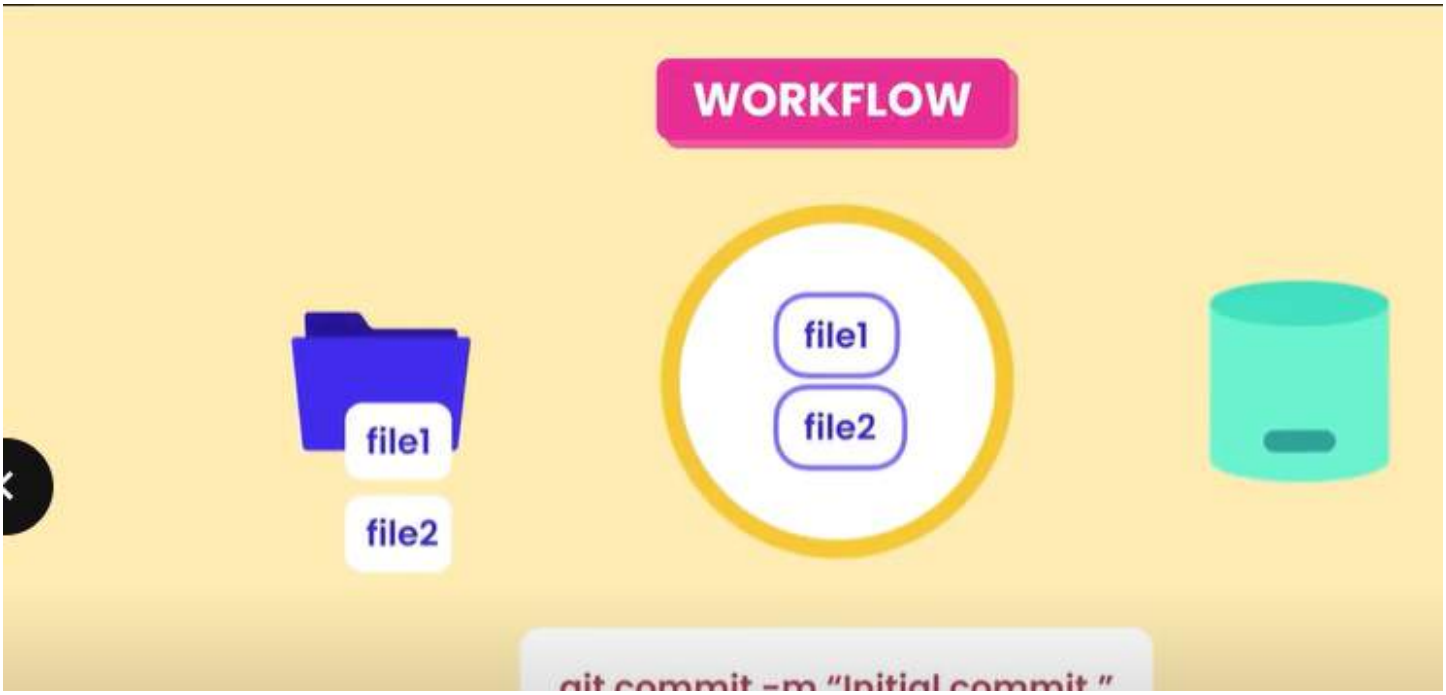
If some of the changes shouldnot be recorded as part of the next snapshot we can unstage them and commit them as part of another snapshots Thats the Basic workflow

EXAMPLE:

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???

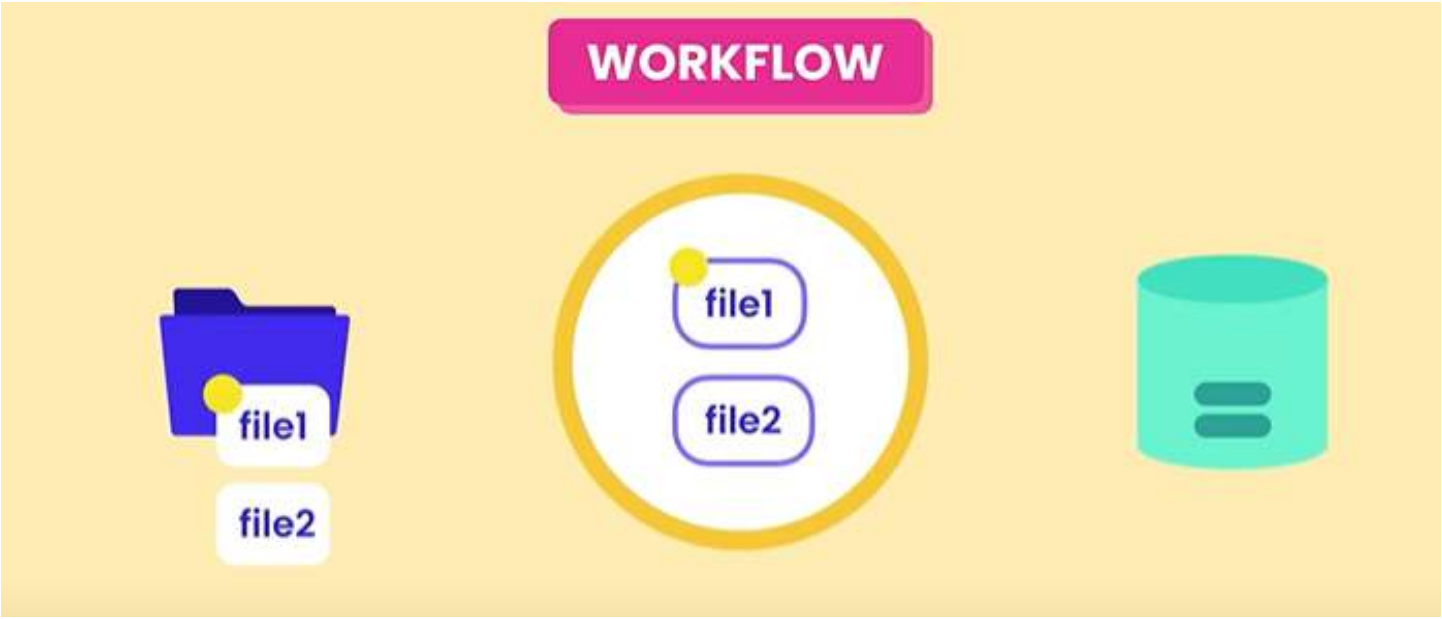
Answer= Its not correct .we are in a staging area the same snapshot that we stored in the repository.Staging area is very similar to a staging environment .we use when releasing a software to production. Its either a

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

Example:

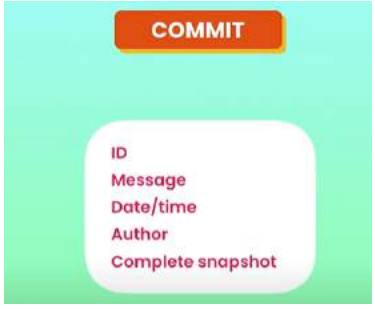
Lets say as part of fixing the bug we make some changes to file1.note that what we have in staging area is the old version of file1 because we have not staged the changes yet .so once again

- 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 doesnot store duplicate Content

Staging File

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

```
On branch master

No commits yet

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

        file1.txt
        file2.txt

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

file one and file two. They're

untracted file

- git add file1.txt file2.txt or git add . or git add *.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
```

be committed. So if we run git

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

```
No commits yet

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

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

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working dire
ctory)
                                in the staging area, because
        modified:   fil         they're indicated by green. But
```

we run git add file1.txt

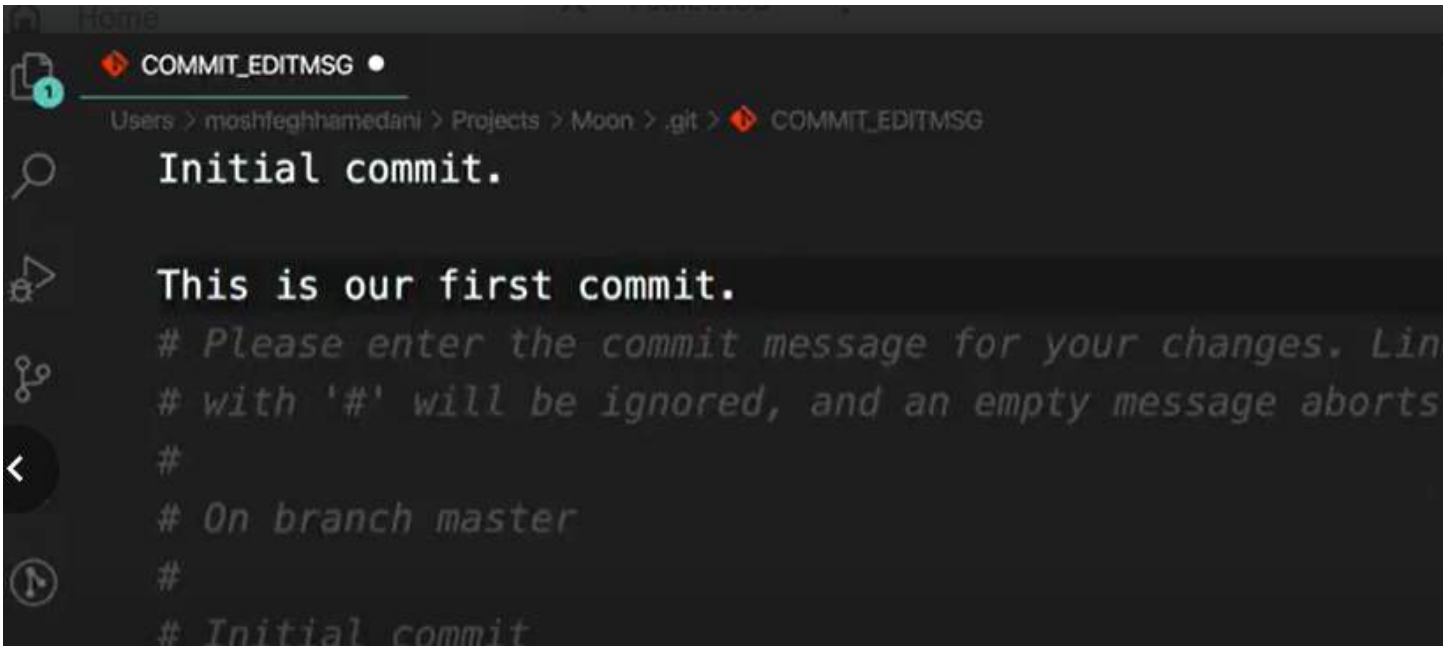
git status

```
> ✔  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”

if only write git commit (it opens the vs code(our default editor)




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

- git commit -m “Remove unused code”

1. 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

```
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 director
y)
        deleted:    file1.txt

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

- git add file1.txt
- git add main.js

```
git status
branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    file1.txt -> main.js
```

- git mv main.js file1.js

```
git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    file1.txt -> file1.js
```

- git commit -m “Refactor code”

```
git commit -m "Refactor code."
[master 7e3f6b1] Refactor code.
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename file1.txt => file1.js (100%)
```

C. IGNORING FILES

- mkdir logs
- echo hello >logs/dev.log

```
git status
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

```
.gitignore
Users > moshfeghhamedani > Projects > Moon > .gitignore
logs/
main.log
*.log
```

- 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”

```
git commit -m "Add gitignore"
[master e5fd9a2] Add gitignore
 1 file changed, 3 inse
 create mode 100644 .g
```

git ignore. So this is how we can ignore files and directorie

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

- git add .
- git commit -m “add bin”

```
[master 42ce2fb] Add bin.
 1 file changed, 1 insertion(+)
 create mode 100644 bin/app.bin
```

- so back to code.gitignore

```
.gitignore
Users > moshfeghhamedani > Projects > Moon > .gitignore
logs/
main.log
*.log
bin/
```

- 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
y)
        modified:   .gitignore
<
no changes added to commit (use "git add" and/or "git commit -a")
```

- git add .
- git commit -m “Include bin / in gitignore.”
- echo helloworld > bin/app.bin
- 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 direc
y)
        modified:   bin/app.bin

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

- 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

```
git ls-files
.gitignore
bin/app.bin
file1.js
```

- we should remove it here
- git rm -h (we want to remove it on staging area)
- git rm --cached bin/

```
SIGHUP(1) git rm --cached bin/
fatal: not removing 'bin/' recursively without -r
```

- so git rm --cached -r bin/
- git ls-files

```
git ls-files
.gitignore
file1.js
```

- git status

```
git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    bin/app.bin
```

- 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

github.com/github/gitignore/blob/master/Java.gitignore

23 lines (18 sloc) 278 Bytes Raw Blame

```
# Compiled class file
*.class

# Log file
*.log

# BlueJ files
*.ctxt
```

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
```

- M.....now in staging area
- echo ocean >> file1.js
- git status -s

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

- git add file1.js
- git status -s

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

- git add file2.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 @@
+sky
```

- 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
- paste ss
- 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.
138c8ef Remove unused code.
8f092f7 Fix the bug that prevented the users from signing up.
3b0003b Initial commit.
```

- git log --oneline --reverse

```
git log --oneline --reverse
3b0003b Initial commit.
8f092f7 Fix the bug that prevented the users from signing up.
138c8ef Remove unused code.
7e3f6b1 Refactor code.
e5fd9a2 Add gitignore
42ce2fb Add bin.
d601b90 Include bin/ in gitignore.
921a2ff (HEAD -> master) Remove the bin directory that was accidenta
y committed.
```

Viewing A Commit

- git show d601b
- git show HEAD~1

```
Date:   Tue Aug 4 16:52:21 2020 -0700

    Include bin/ in gitignore.

diff --git a/.gitignore b/.gitignore
index 8432ad3..1dcc30c 100644
--- a/.gitignore
+++ b/.gitignore
@@ -1,3 +1,4 @@
 logs/
 main.log
-*.log
\ No newline at end of file
+*.log
+bin/
\ No newline at end of file
```

the final version, the exact version that is stored in this

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

```
> ✔  git show HEAD~1:.gitignore
logs/
main.log
*.log
bin/
```

- Each Commit Contains a Complete snapshot of our working directory
- we run show command we only see the changes
- git show HEAD

```
    Remove the bin directory that was accidentally committed.

diff --git a/bin/app.bin b/bin/app.bin
deleted file mode 100644
index ce01362..0000000
--- a/bin/app.bin
+++ /dev/null
@@ -1 +0,0 @@
-hello
```

see all the files and directories in a commit? Well,

- git ls-tree (list all files in a tree)
- git ls-tree HEAD~1

```
git ls-tree HEAD~1
100644 blob 1dcc30c4cd47f8915741af2cfef91e16e0dc7d89    .gitignore
040000 tree 64629cd51ef4a65a9d9cb9e656e1f46e07e1357f    bin
100644 blob badfb70fd8b1725682b26674f7b2882e94078579    file1.js
```

- blob = file
- directory= Tree
- git show 1dcc30

```
git show 1dcc30
< js/
main.log
*.log
bin/
```

- git show 64629

```
git show 64629
tree 64629

app.bin
```

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

1. Commits
2. Blobs(Files)
3. Trees (Directories)
4. 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 Or git restore --staged or git restore --staged file1.js file2.js
- git restore --staged file1.js
- git status -s

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

- How Restore Command Works??

1. 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
```

That is what happened Now look at file2 is a new file .This file2 is new in the staging area .This File Doesnot exist in the last commit.So What do you think will happen when ever you restore this file??? Well because

we dont have a copy of this file in our repository or in a last commit .

Git is going to remove this file from the staging Area and back to its previous which is a new untracked file .Let me show you

- git restore --staged file2.js
- git status -s

```
git status -s
 M file1.js
?? 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
fatal: clean.requireFor neither -i, -n, nor -f
 given: refusing to clean
```

fatal error saying required for us to false to true. And neither

- git clean -fd

```
> ↵ SIGHUP(1)  git clean -fd
Removing file2.js
```

```
> ✔  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

```
> ✔  git status -s
D  file1.js
```

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

```
✔  git commit -m "Delete file1.js"
[master 905cf09] Delete file1.js
 1 file changed, 3 deletions(-)
 delete mode 100644 file1.js
```

- now restore this changes
- git log --oneline

```
git log --oneline
905cf09 (HEAD -> master) Delete file1.js
921a2ff Remove the bin directory that was accidentally committed.
d601b90 Include bin/ in gitignore.
42ce2fb Add bin.
e5fd9a2 Add gitignore
7e3f6b1 Refactor code.
c8ef Remove unused code.
092f7 Fix the bug that prevented the users from signing up.
3b0003b Initial commit.
```

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

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

B.BROWSING HISTORY

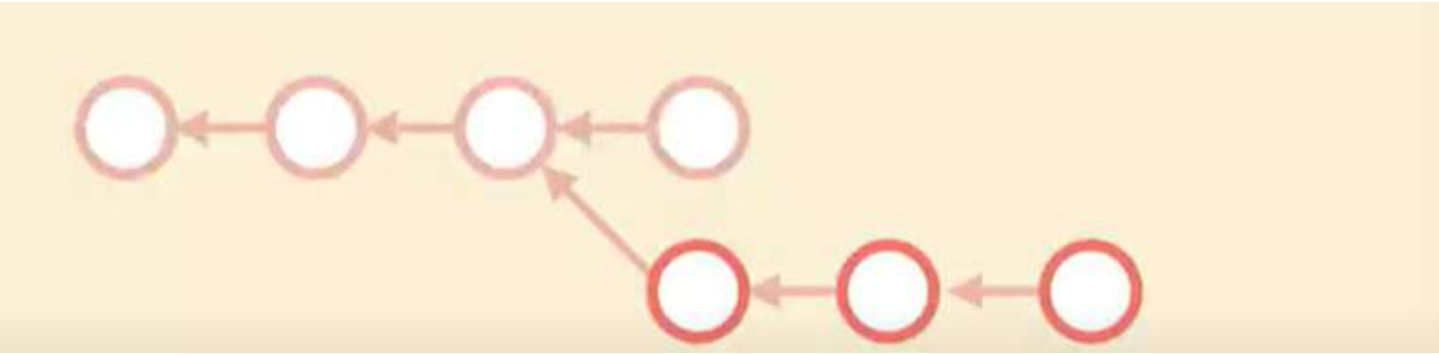
C. BRANCHING AND MERGING

- Use Of Branches
- Compare Branches
- Merge Branches
- Resolve Conflicts
- undo a faulty merge

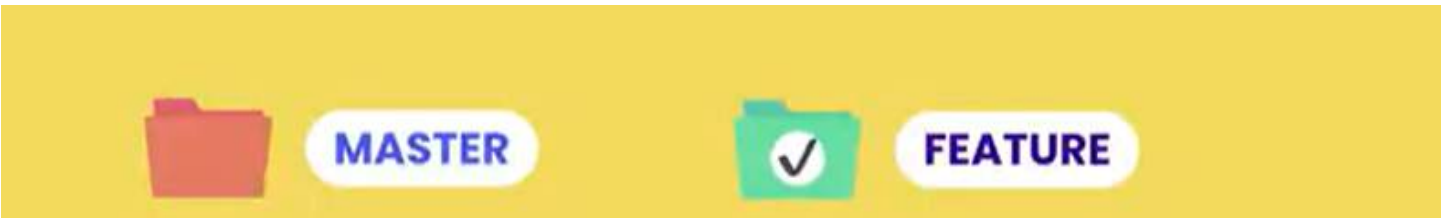
1. Essential Tools (Stashing and Cherry Picking)

What is Branching?

Branching allows diverge from the main line of work and work on something else in isolation.Conceptually you can think of a branch like a separate isolated workspace.



So we have our main workspace



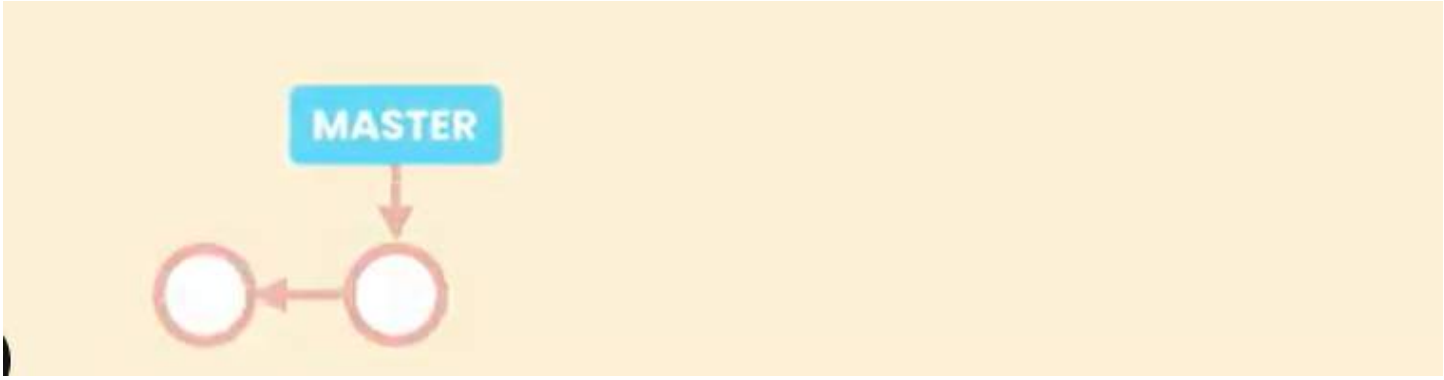
We have main work space which is Master, we can have another workspace We are working on a new FEATURE in isolation.

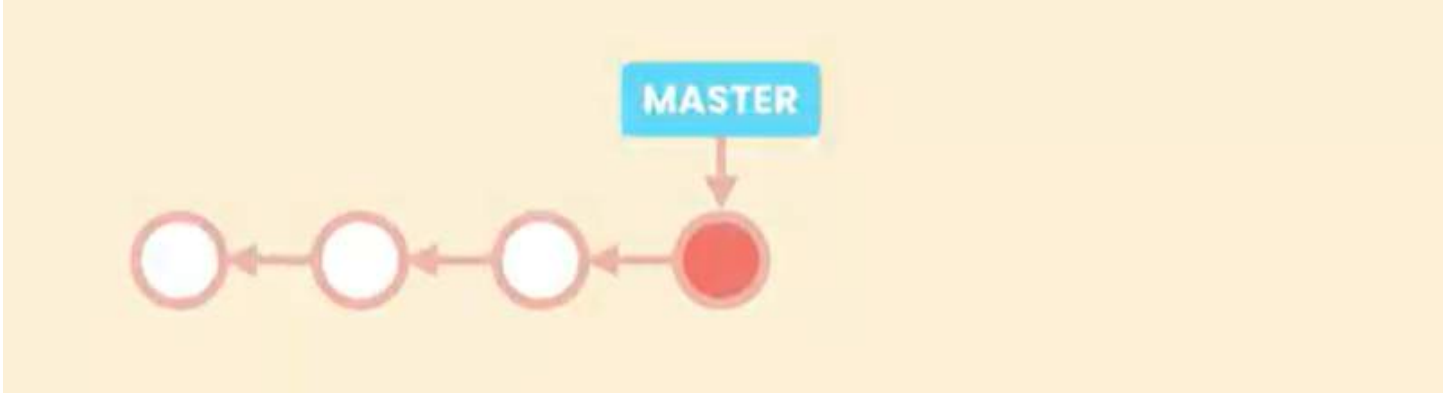
1. While we are developing this new feature our code may get unstable. So we dont want to release the code in this workspace .
2. We continue working here when we are done, we test our code and after we fix all the bugs then we bring the changes in this workspace into the master This is called merging .
3. So branching allows us to work on different work items .Without messing up with the main line of work, we keep the main line as stable as possible, so we can release it any time. Also anyone joining our team can start off on a stable codebase. Thats the idea of branching .

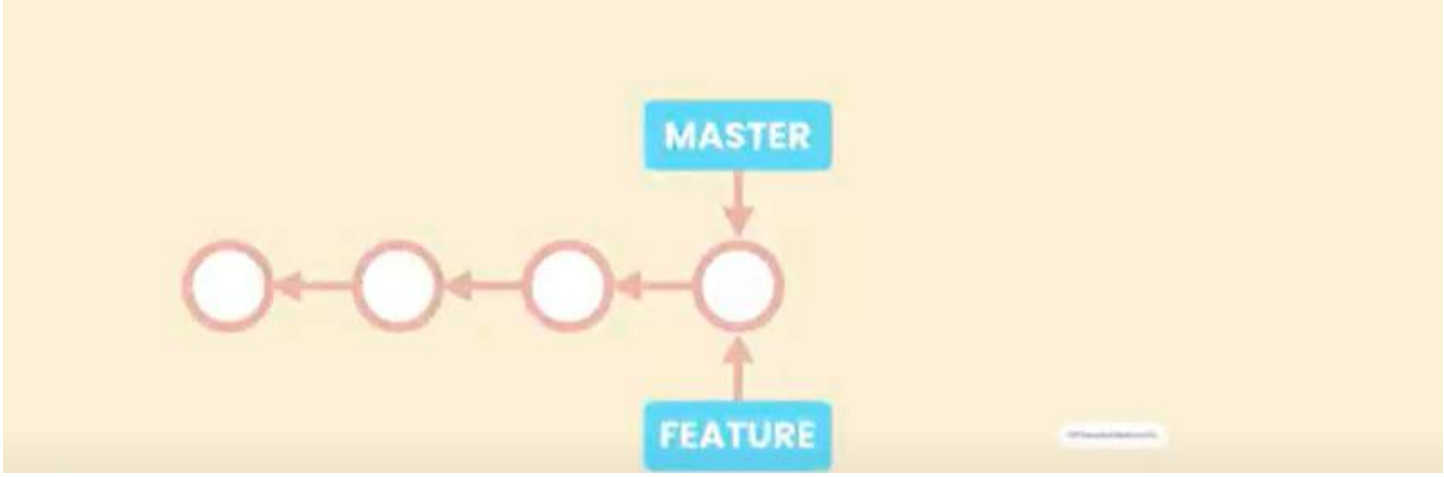
GIT BRANCHES ARE DIFFERENT

The way git manages branches is very different from other version control system like subversion .In subversion when we create a new branch ,subversion takes a copy of our entire working directory and stores it somewhere else. Its slow and waste a lot of time.

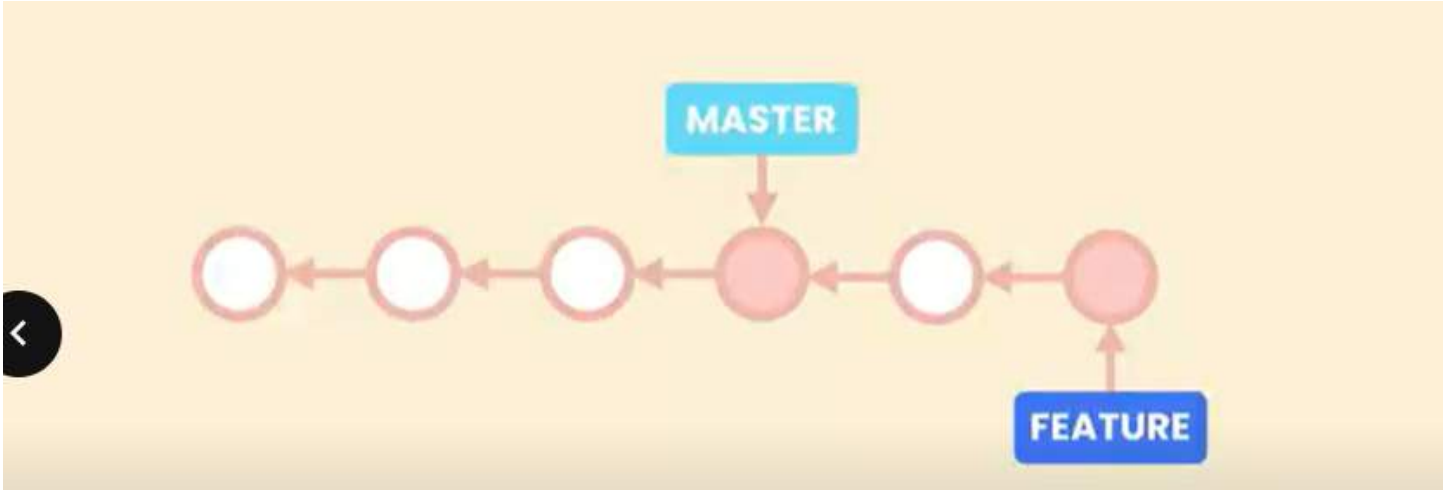
- Git branches are superfast and cheap because a branch and GIT is just a pointer to a Commit. So the master branch is just a pointer to the last commit in the main line of work.

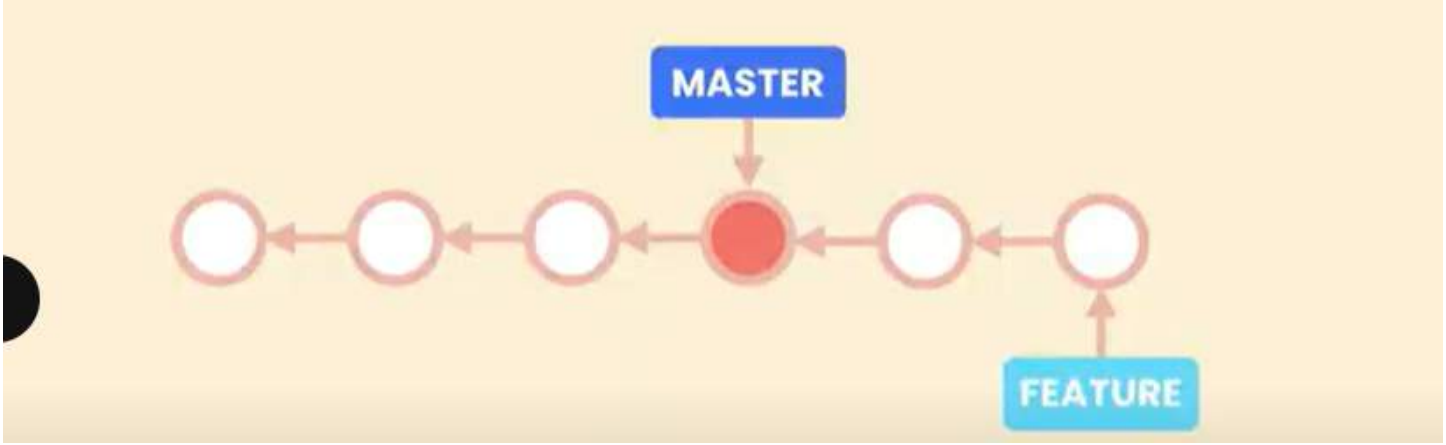


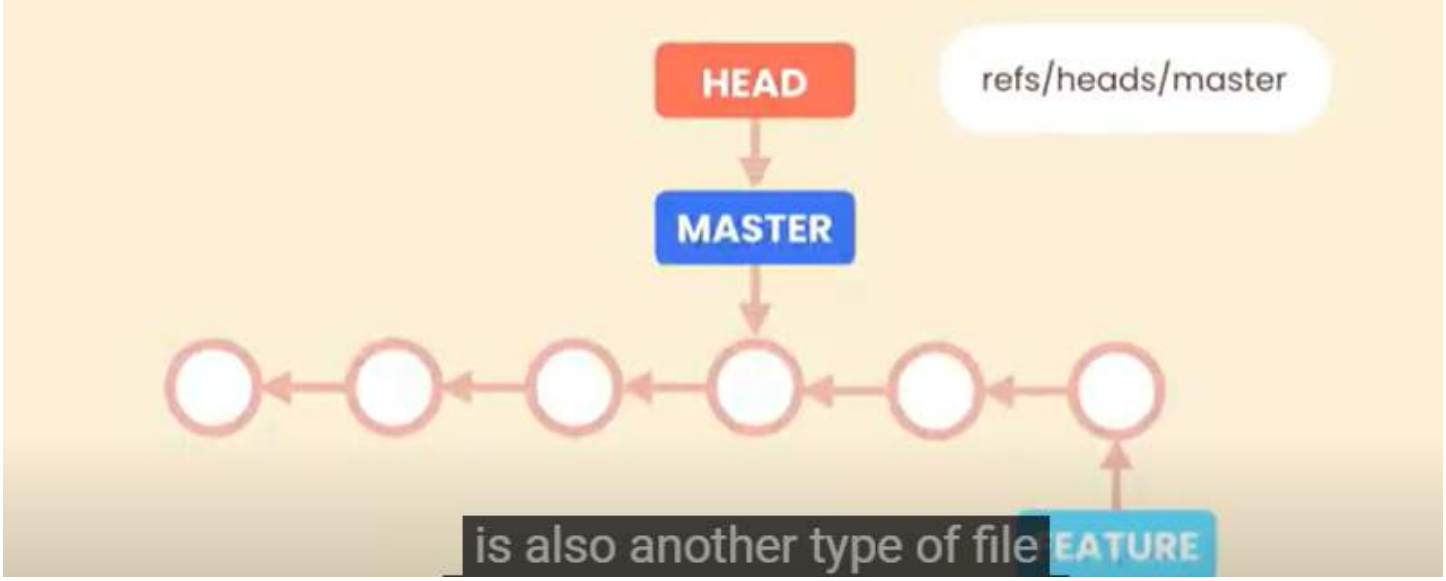


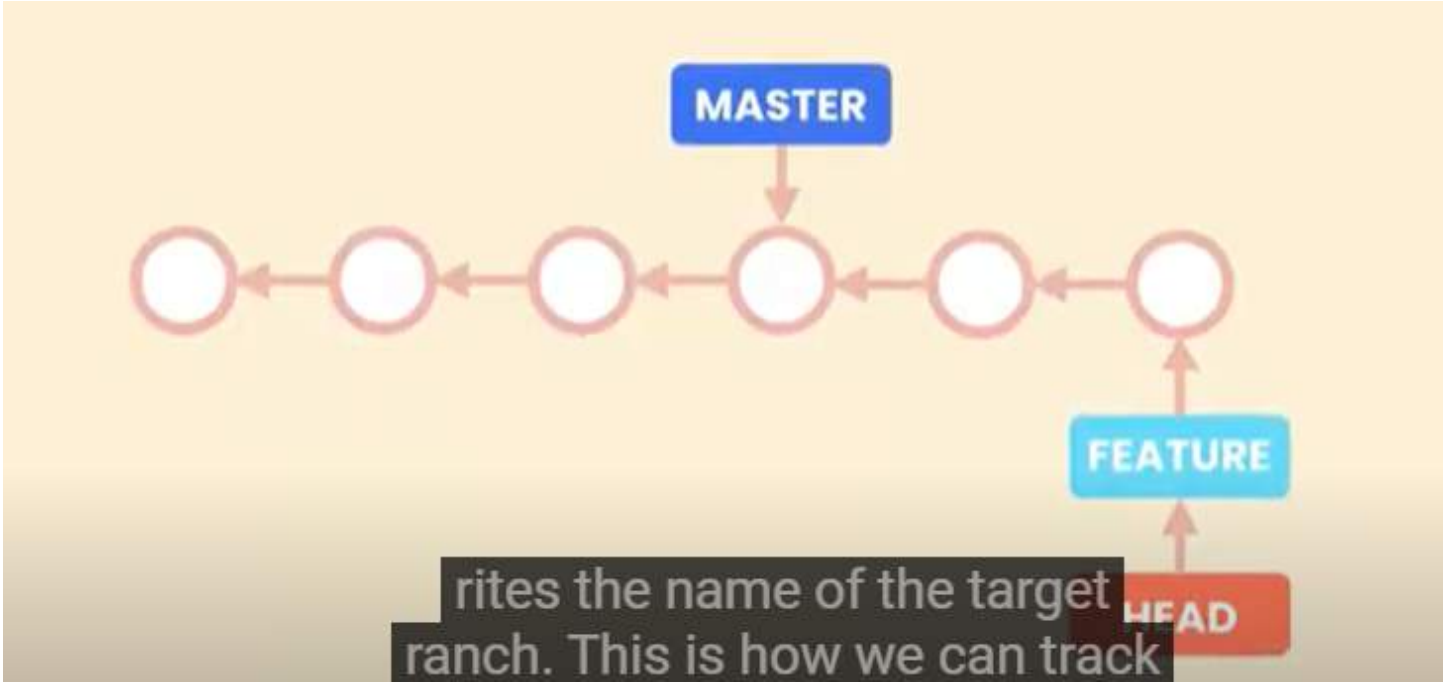


- we make new commits get most this pointer forward.
- So it knows what is the latest code in the main line of work. Snapshots stores in this commit. When we create a new branch git creates a new pointer that can be move around.
- This Pointer is just a tiny file that contains a 40 byte commit id . Thats why creating a branch in Git is blazingly fast.


MASTER
FEATURE


MASTER
FEATURE


HEAD
refs/heads/master
MASTER
is also another type of file
EATURE



- Now when we switch to this branch and make new commits get most of this point forward, the master pointer stays where it is so git knows the latest code in each branch.
- Let me switch back to master .It takes the snapshot from the commit that master point to and reset our working directory to that snapshot.
- So we always have a single working directory.
- Now how does git know which branch we are currently working on using a special pointer called HEAD.
- This pointer is another type of file that contains the name of a branch like Master.
- Let me switch to different branch. Git Moves the HEAD Pointer out. so it updates the tiny files the name of the target branch .This is how we can track that branch we are currently working on .
- Some Hands-On

1. 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
```

1. git switch bugfix

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

1. Rename the branch
2. git branch -m bugfix bugfix/signup-form

```
git branch -m bugfix bugfix/signup-form
```

1. code audience.txt

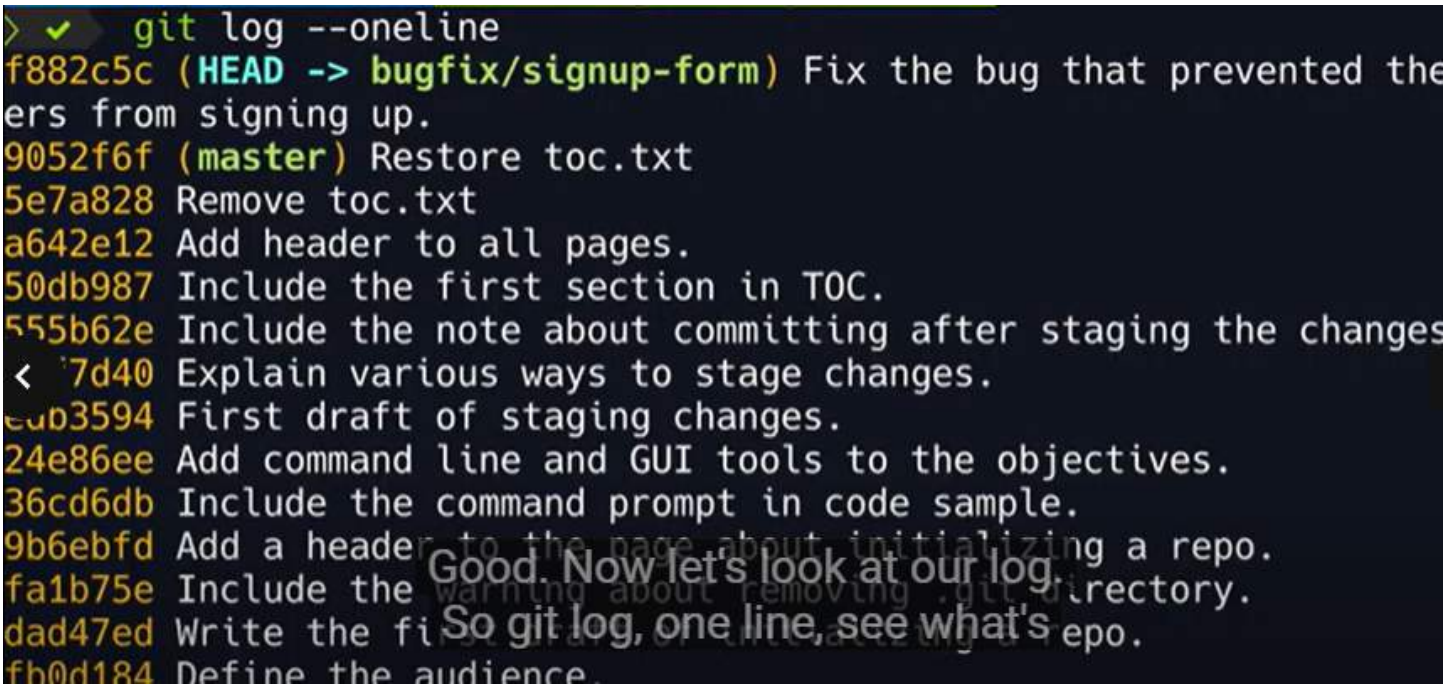
```
audience.txt ×
audience.txt
Moshfegh Hamedani, 2 days ago | 1 author (Moshfegh Hamedani)
AUDIENCE
Moshfegh Hamedani, 2 days ago • Add header

This course is for anyone who wants to learn Git.
No prior experience is required.
```

```
audience.txt ×
audience.txt
You, a few seconds ago | 2 authors (You and others)
WHO THIS COURSE IS FOR
======================
This course is for anyone who wants to learn Git.
```

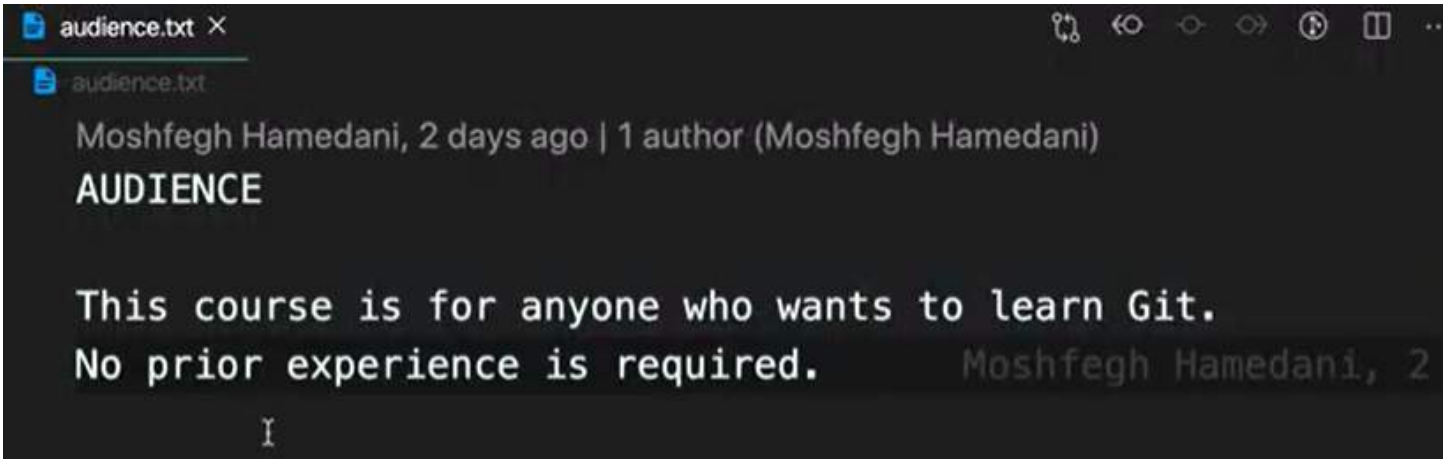
1. git status

```
git status
On branch bugfix/signup-form
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working direct

        modified:   audience.txt

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

1. git add .
2. git commit -m “Fix The bug prevented the users from sign-up.”
3. git log --oneline



- git switch master
- code audience.txt



- git log --oneline

```
git log --oneline
9052f6f (HEAD -> master) Restore toc.txt
5e7a828 Remove toc.txt
a642e12 Add header to all pages.
50db987 Include the first section in TOC.
555b62e Include the note about committing after staging the change
91f7d40 Explain various ways to stage changes.
edb3594 First draft of staging changes.
< 86ee Add command line and GUI tools to the objectives.
cd6db Include the command prompt in code sample.
9b6ebfd Add a header to the page about initializing a repo.
fa1b75e Include the warning about removing .git directory.
dad47ed Write the first draft of initializing a repo.
fb0d184 Define the audience.
1ebb7a7 Define the objectives.
ca49180 Initial commit.
```

we can see head is pointing to the master branch. But we don't

- git log --oneline --all

```
f882c5c (bugfix/signup-form) Fix the bug that prevented the users f
 signing up.
9052f6f (HEAD -> master) Restore toc.txt
5e7a828 Remove toc.txt
a642e12 Add header to all pages.
50db987 Include the first section in TOC.
555b62e Include the note about committing after staging the changes
91f7d40 Explain various ways to stage changes.
edb3594 First draft of staging changes.
< 86ee Add command line and GUI tools to the objectives.
cd6db Include the command prompt in code sample.
9b6ebfd Add a header to the page about initializing a repo.
fa1b75e Include the warning about removing .git directory.
dad47ed Write the first draft of initializing a repo.
fb0d184 Define the audience.
1ebb7a7 Define the objectives.
ca49180 Initial commit.
```

when we merge it into the master branch, we need to delete it. So

- When we are done with the bugfix branch when we merge to the master branch we need to delete it.
- git branch -d bugfix/signup-form

```
git branch -d bugfix/signup-form
error: The branch 'bugfix/signup-form' is not fully merged.
If you are sure you want to delete it, run 'git branch -D bugfix/sig
p-form'.
```

- Error branch bug fix is not fully merged you r seeing the error because we have some changes in the bug fix branch that are not merged with a master branch.

- By default it prevent us from accidentally deleting the branch unless we merge it first. But if you are pretty sure that we dont want to change the branch we can force deletion By D
- git branch -D bugfix/signup-form

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
```

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

```
git diff --name-status bugfix/signup-form
M       audience.txt
```

STASHING

When we switch branches git reset our working directory to the snapshot stored in the last commit of the target branch. If you have local changes in our working directories that we haven't committed yet , these changes could get lost in these situation. It doesnot allow us to switch branches.

- code audience.txt

```
audience.txt ×
audience.txt
Moshfegh Hamedani, 2 days ago | 1 author (Moshfegh Hamedani)
AUDIENCE
Moshfegh Hamedani, 2 days ago • Add header to all page
This course is for anyone who wants to learn Git.
No prior experience is required.
```

```
audience.txt ×
audience.txt
Unsaved changes (cannot determine recent change or authors)
* AUDIENCE

This course is for anyone who wants to learn Git.
No prior experience is required.
```

git switch bugfix/signup-form

```
git switch bugfix/signup-form
or: Your local changes to the following files would be overwrit
y checkout:
        audience.txt
Please commit your changes or stash them before you switch branches.
Aborting
```

Stashing means storing it in a safe place. So we are going to store this somewhere in our git repository. But it is not going to be part of history.

git stash push -m “New tax rules.”

```
1 git stash push -m "New tax rules."
Saved working directory and index state On master: New tax rules.
```

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

- echo hello > newfile.txt
- git status -s

```
git status -s
newfile.txt
```

- ?? newfile.txt
- git stash push -am “My new stash.”

```
git stash push -am "My new stash."
Saved working directory and index state On master: My new stash.
```

- git stash list

```
> ✔  git stash list
stash@{0}: On master: My new stash.
stash@{1}: On master: New tax rules.
```

Two Stashes. Each stash has a unique identifier stash add .In Culy braces

we have an index. our working directory is clean so we can switch to the bug fix branch.

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:   audience.txt
```

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

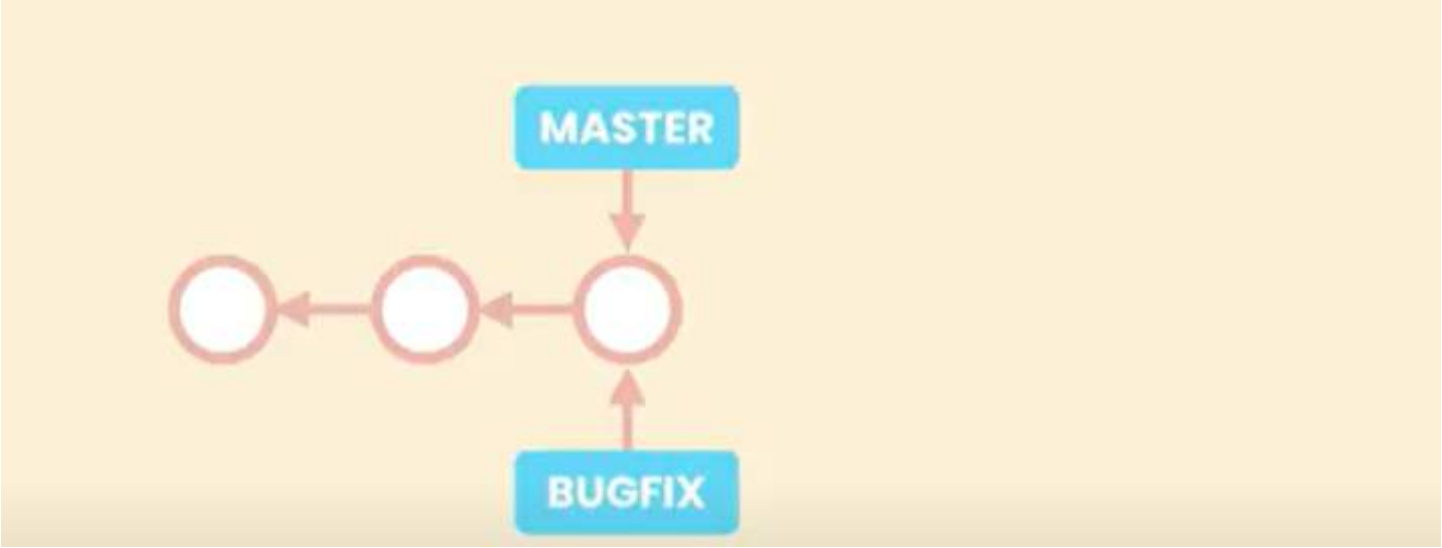
```
> ✔  git stash list
stash@{0}: On master: My new stash.
```

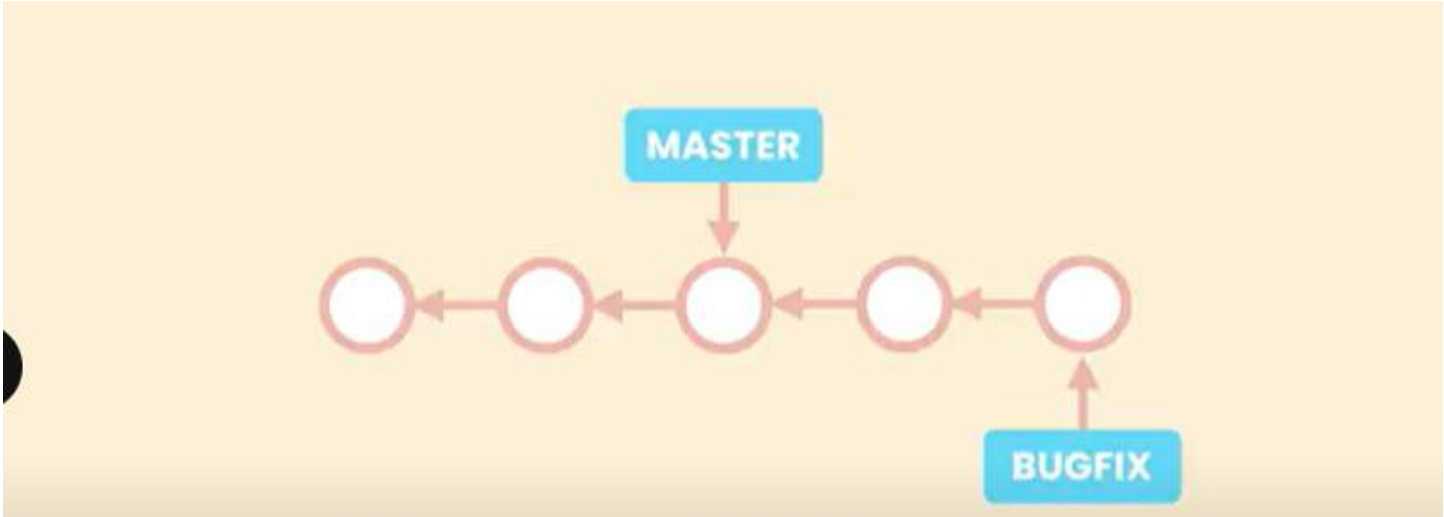
- git stash clear

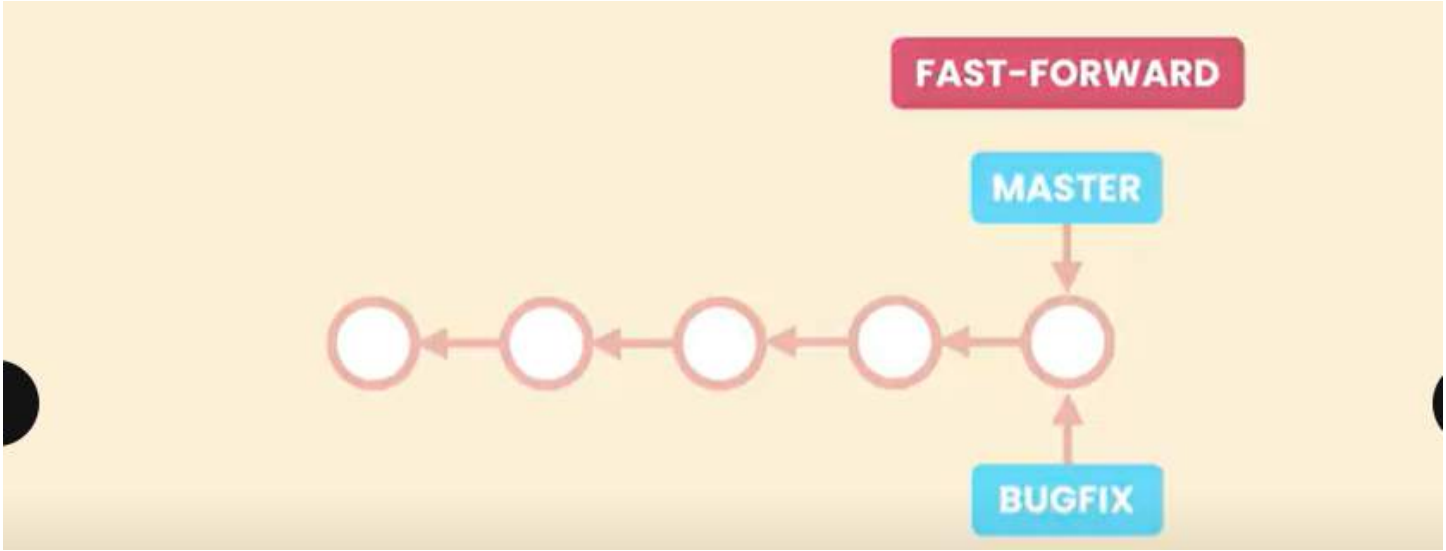
C.MERGING

Merging is all about bringing changes from one branch to another. In git two types of merges.

1. Fast-Forward Merges
2. 3-way Merges

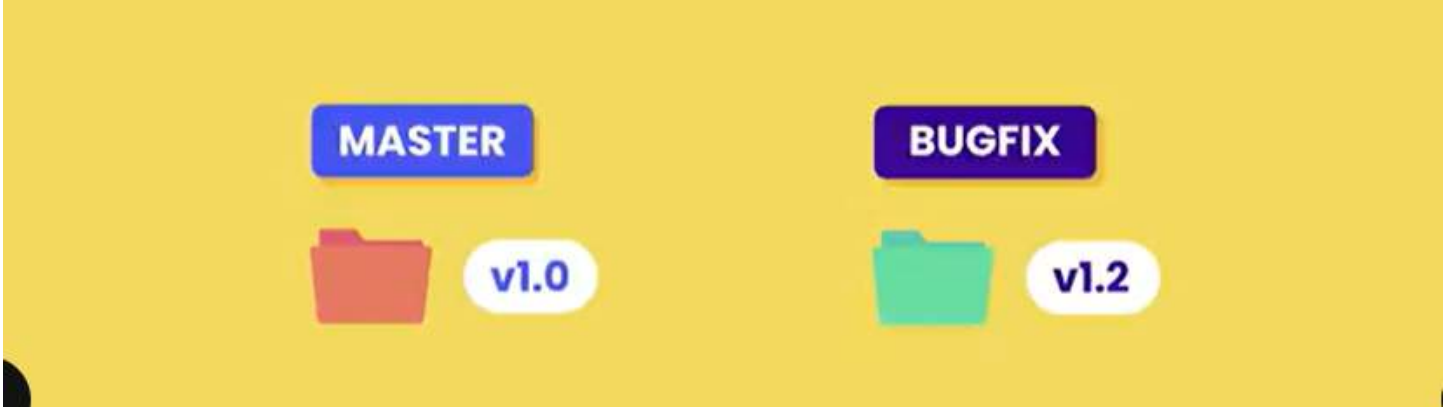


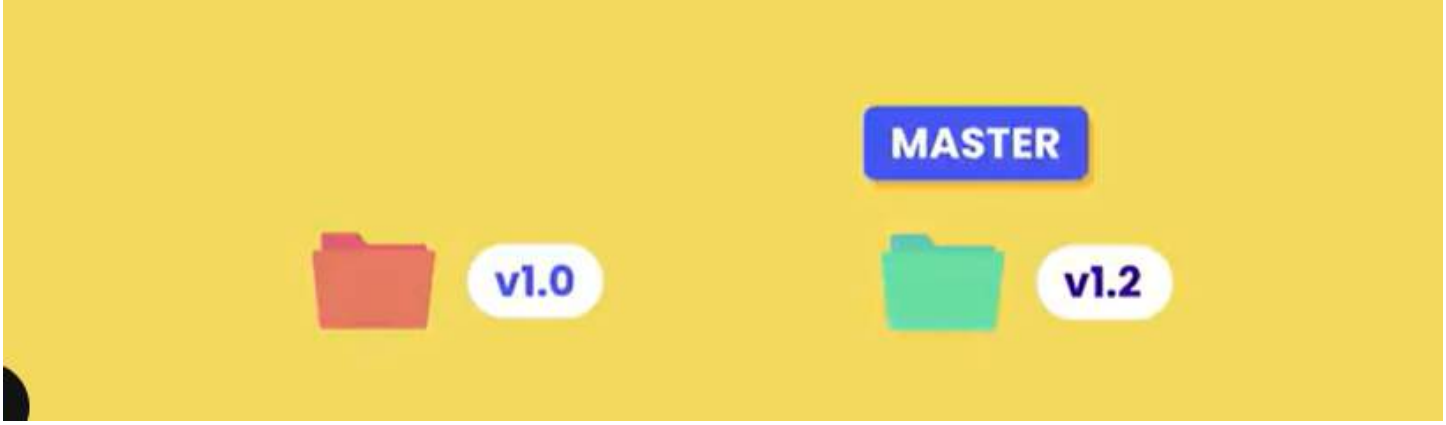




1. Here is the Master Branch with three commits & we create a new branch called bugfix.Branch just a pointer to a commit.Both Master and bug fix pointers are pointing to the same commit.

2. Now We will switch to the bug fix branch and make a couple of commits and now we need to bring the changes back to Master. These Branches have not diverged and there is a direct linear path from bugfix to Master .Merge the changes is to bring the Master pointer forward. This is call a fast forward merge.

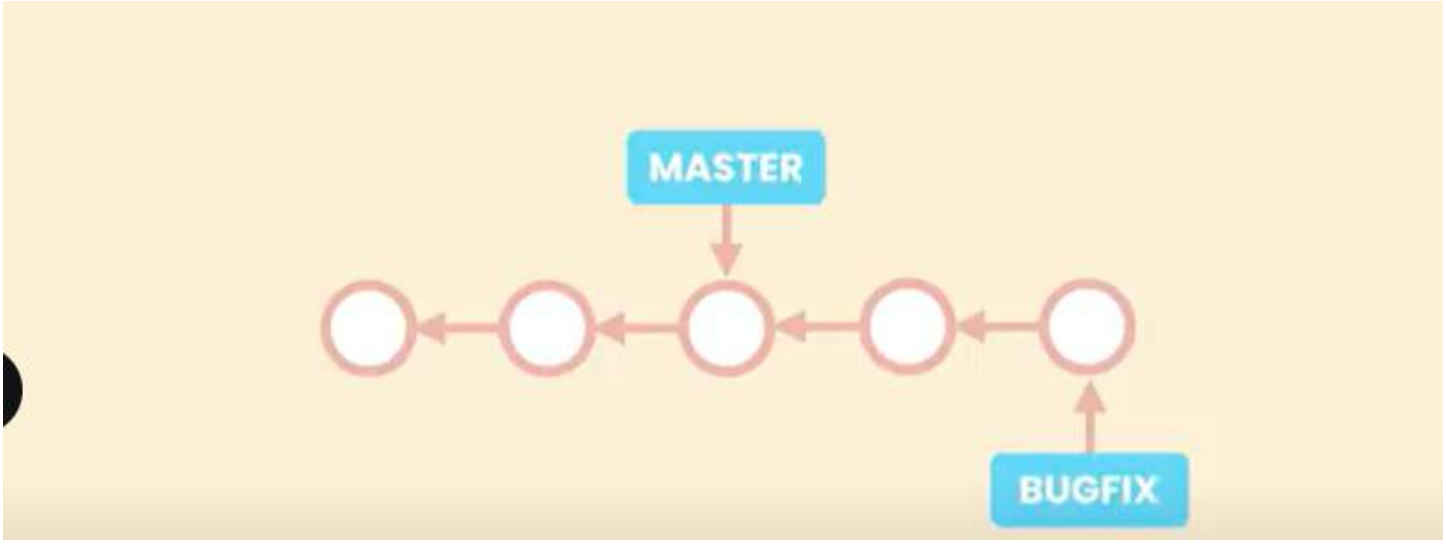


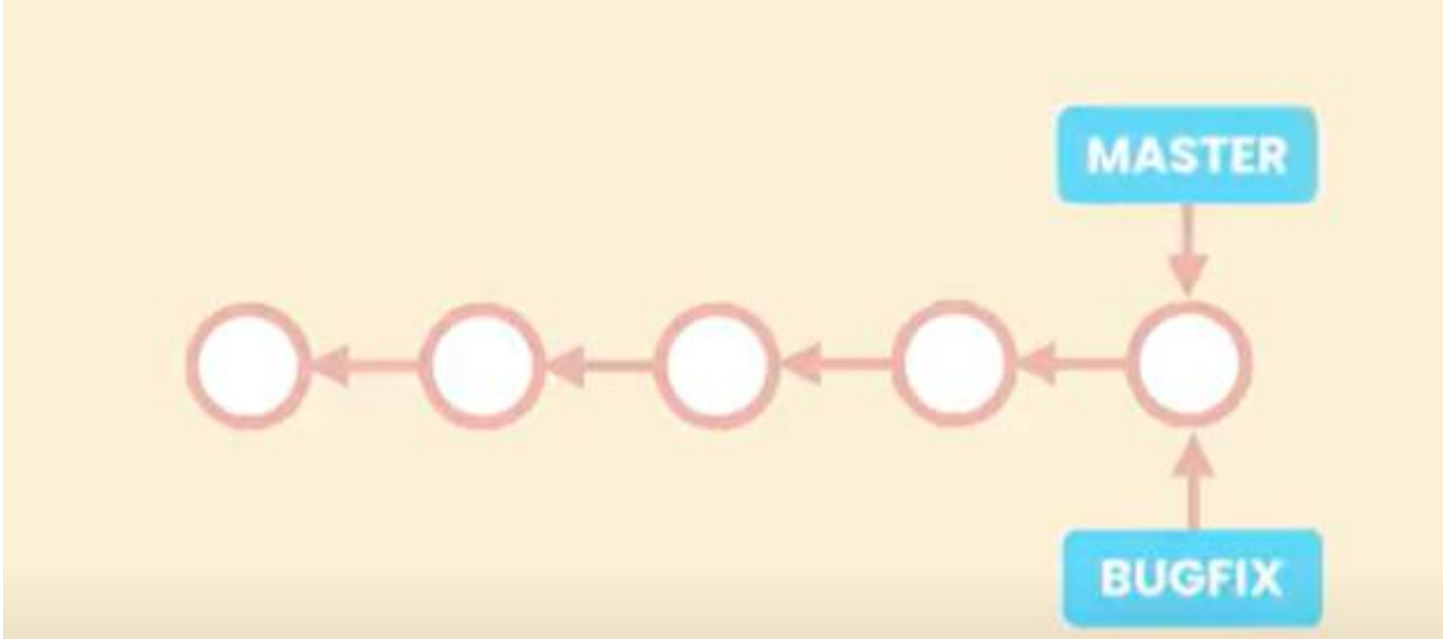


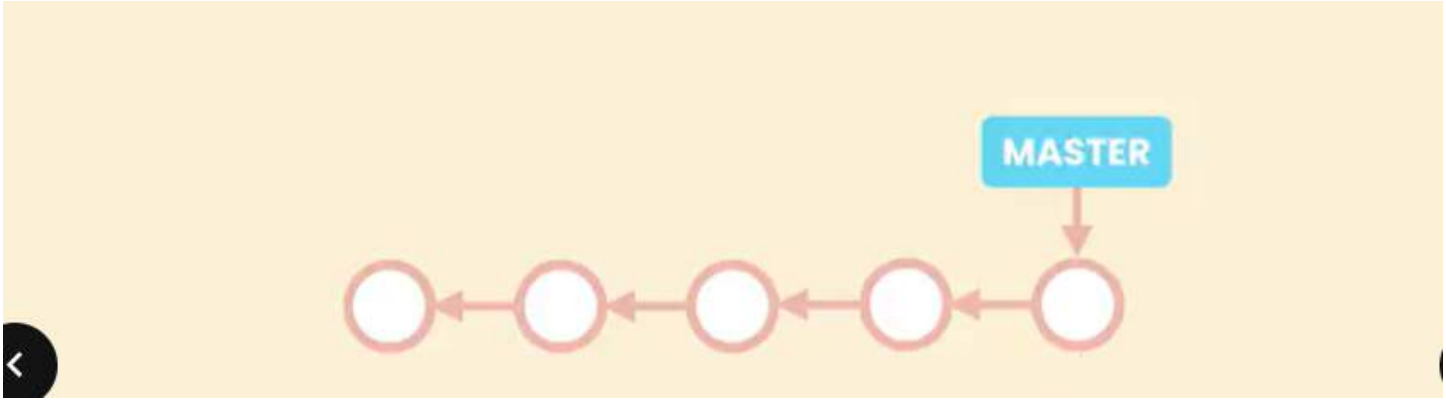
Code in this directory is at version 1 . we take a copy of this directory and call it bug fix.At this point in time the now some change sin bugfix directory so we get version 1.2 we want changes bring back to Master .

How we do this we can simply rename bugfix to Master .so we can say this is our new master directory

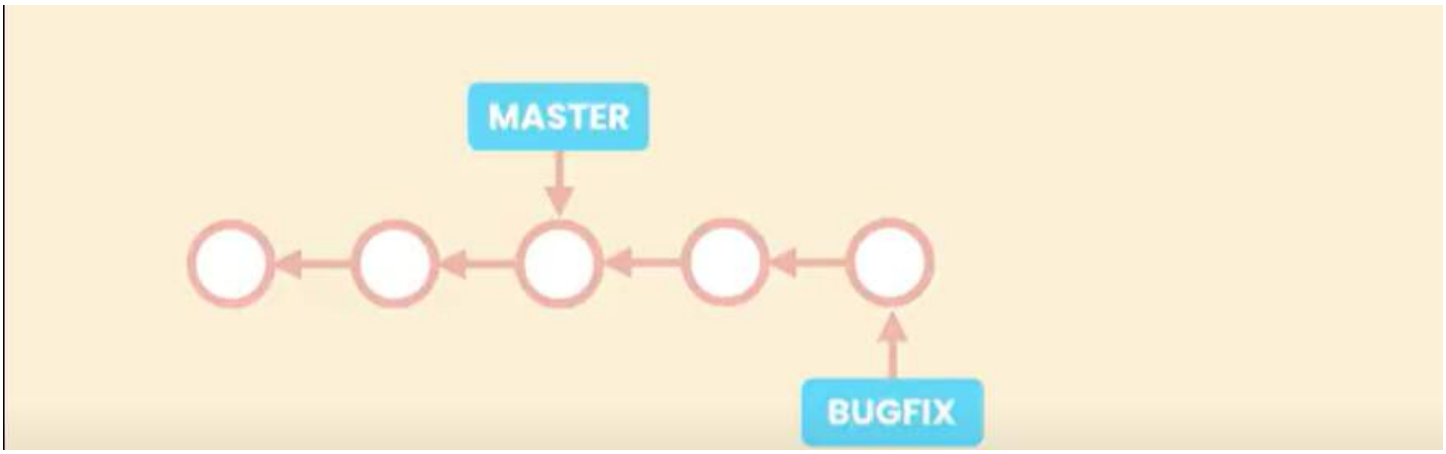
- If two branches have not diverged and there is a direct linear path from the target branch to the source branch git runs a fast forward merge .It simply brings the pointer of the source branch forward.we can simply remove bug fix the pointer indicates Master.





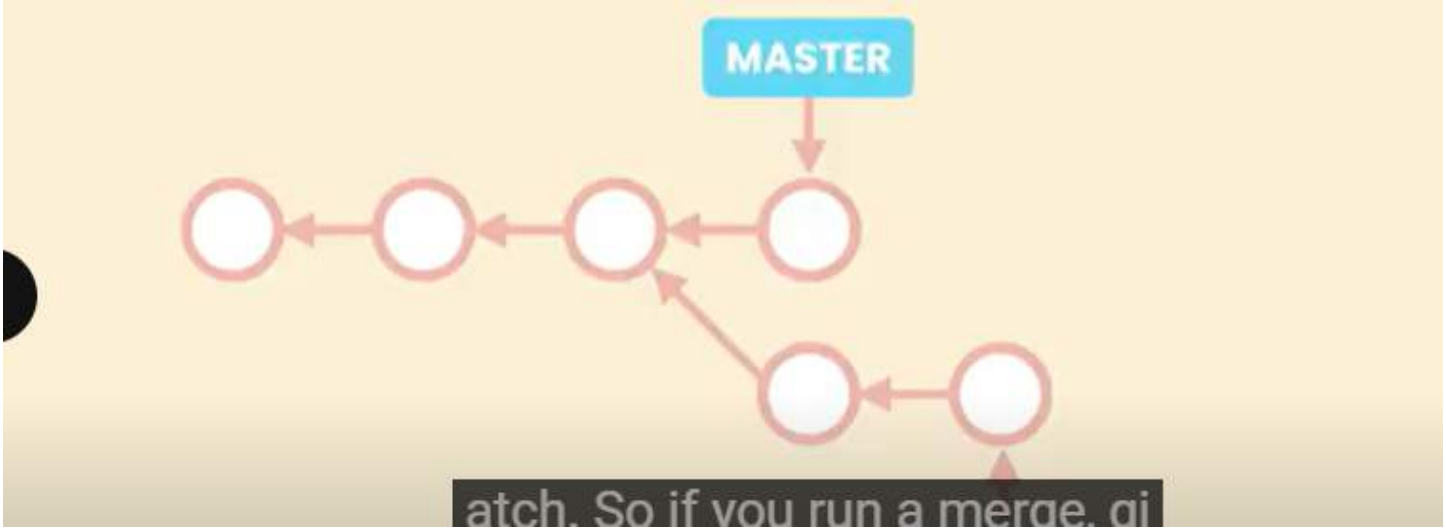


3 way Merge

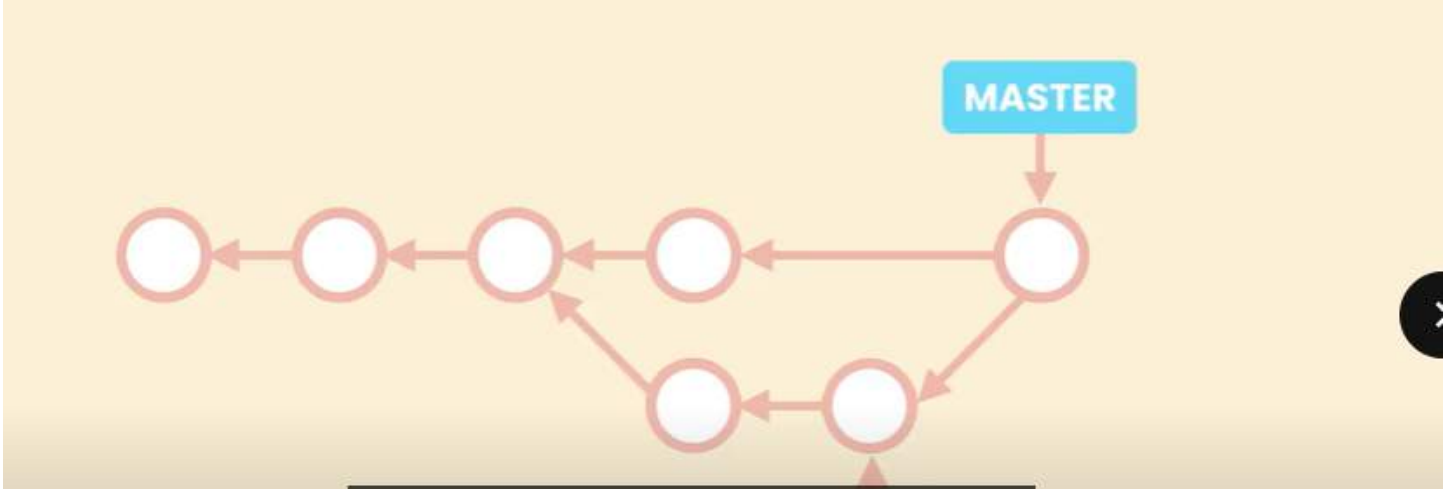


in this bugfix branch is 2 head commit than master branch

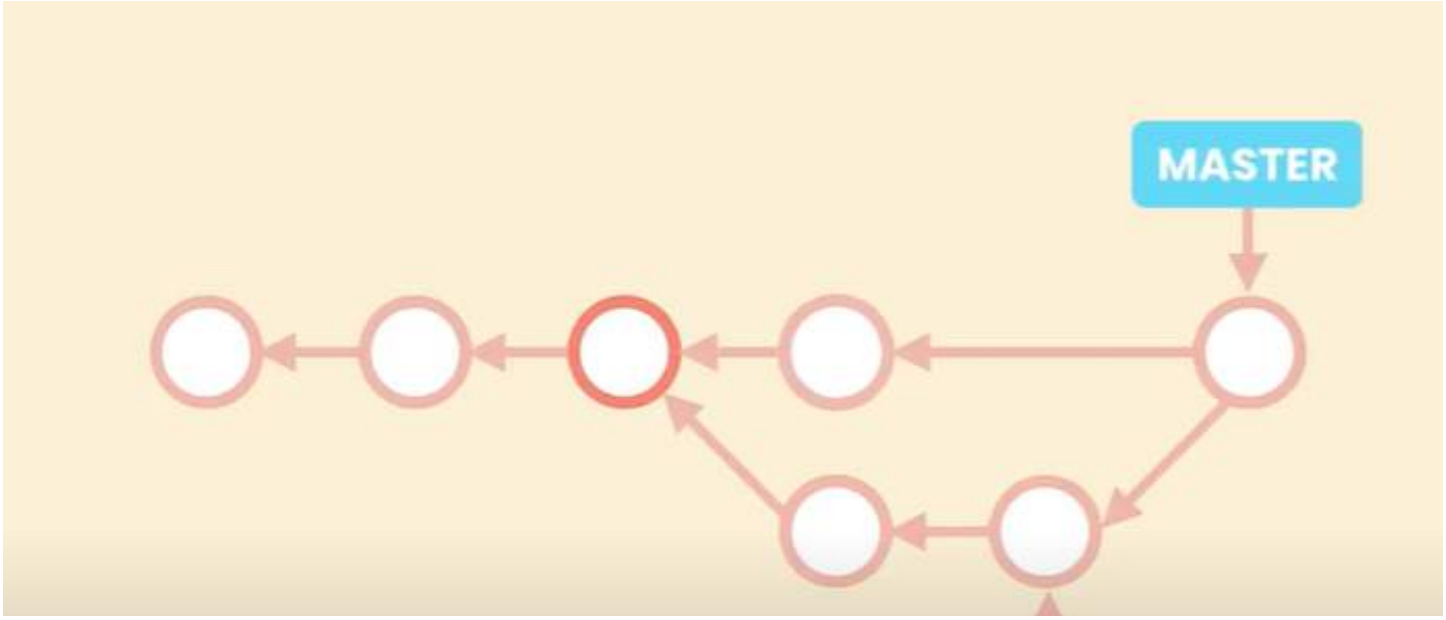
before we merge it with Master Lets go back to Master and add an aditional commit.now our branches are diverged so some changes in the Master that dont exist in the bugfix branch



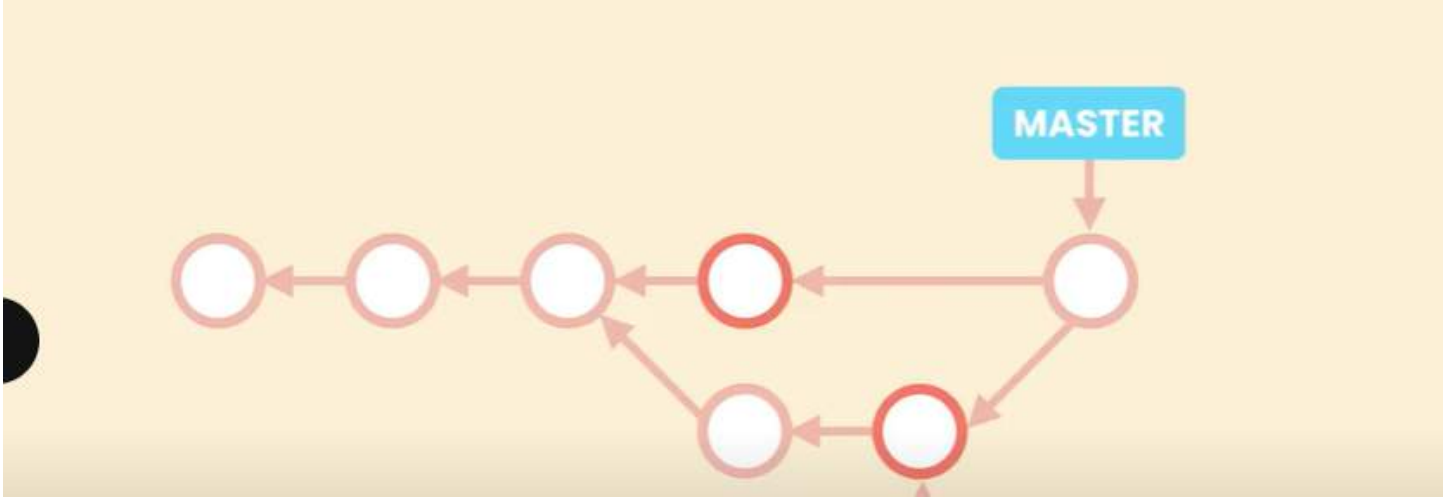
BUGFIX



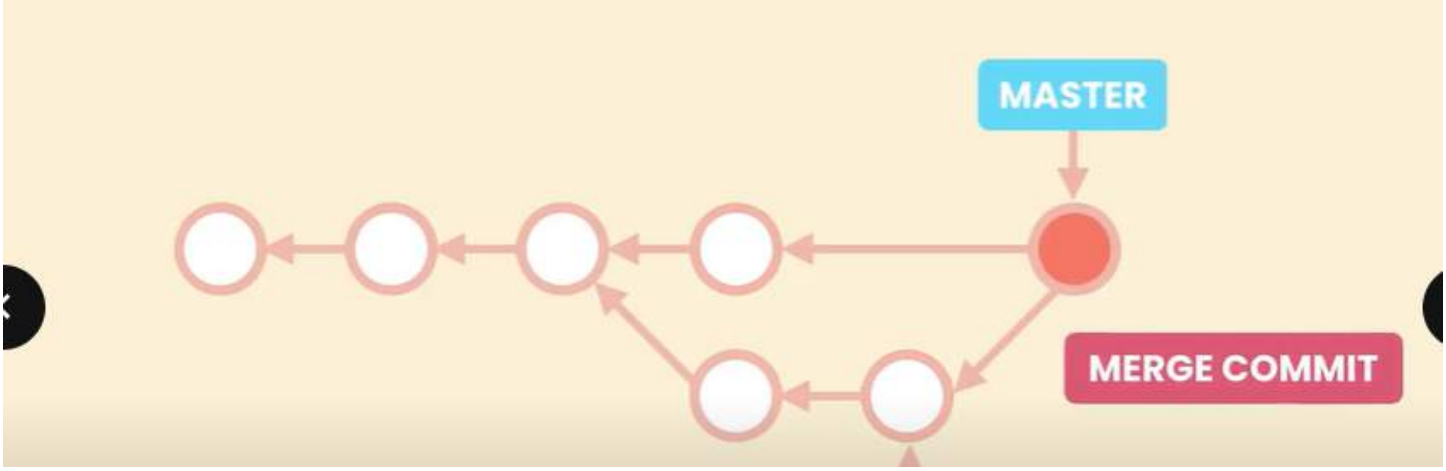
BUGFIX



BUGFIX



BUGFIX



BUGFIX

if you run a merge git can't move the master. pointer forward have it point to the same commit as bug fix because otherwise we will loose the latest commit in the master branch. So when we run merge git creates a new commit that combines the changes from these two branches.

The reason this is called 3-way merge because this new commit is based on three different commits the common ancestor of the branches.

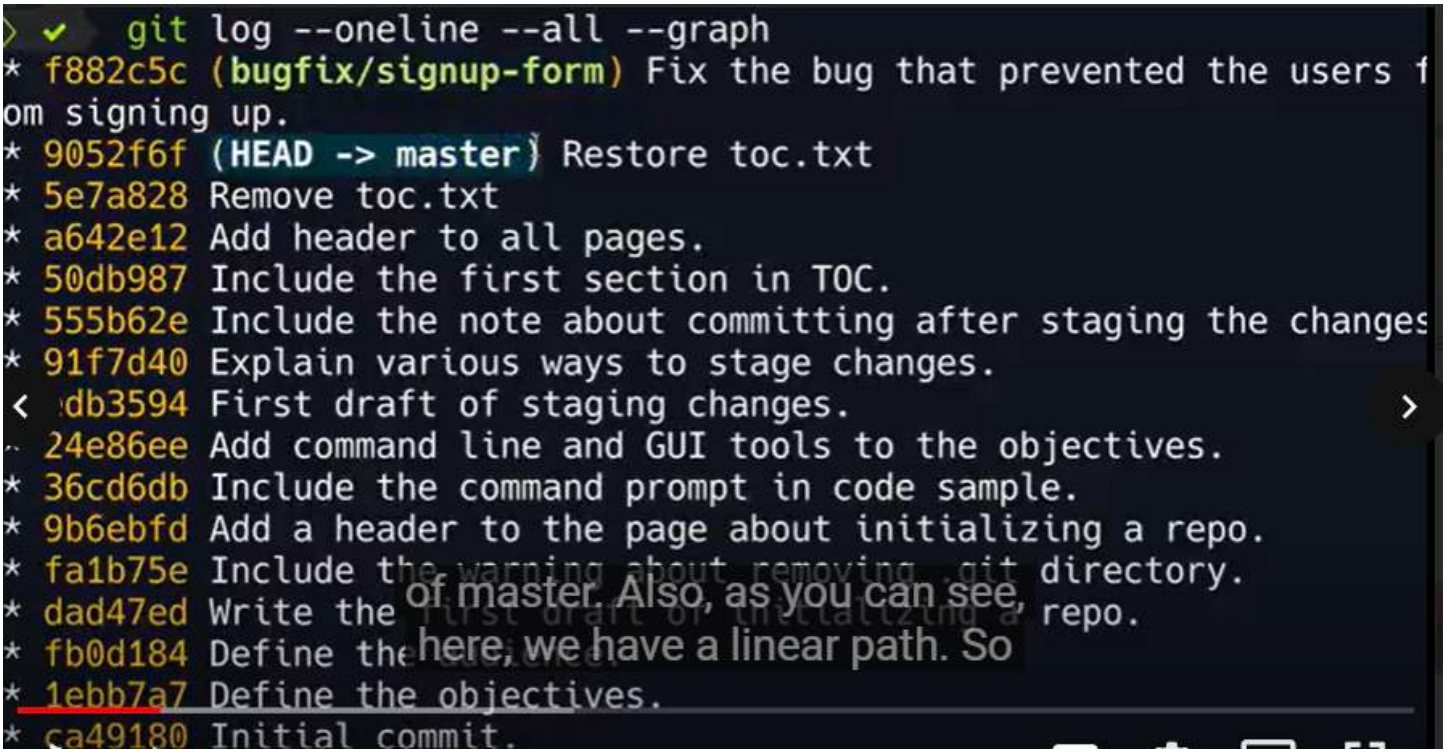
which include the before codes and the tips of our branches which contains the after code. So git looks at three different snapshots and after snapshots and based on these it will figure out how we should combine the changes so we create this new commit which is called merge commit

fast forward merges....if branches have not diverged.

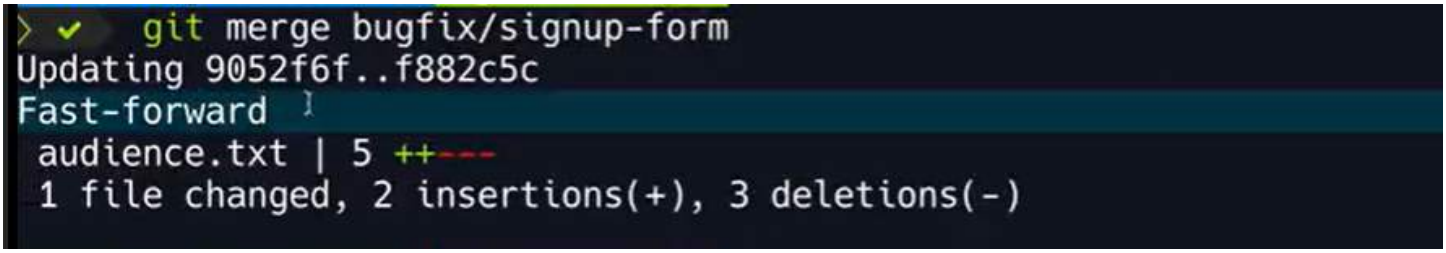
3-way merges....if branches have diverged

Example of fast forward merges

git log --oneline --all --graph



git merge bugfix/signup-form



git log –oneline --all --graph

```
* f882c5c (HEAD -> master, bugfix/signup-form) Fix the bug that preve
ted the users from signing up.
* 9052f6f Restore toc.txt
* 5e7a828 Remove toc.txt
* a642e12 Add header to all pages.
* 50db987 Include the first section in TOC.
* 555b62e Include the note about committing after staging the changes
* 91f7d40 Explain various ways to stage changes.
* edb3594 First draft of staging changes.
* 4e86ee Add command line and GUI tools to the objectives.
* 36cd6db Include the command prompt in code sample.
* 9b6ebfd Add a header to the page about initializing a repo.
* fa1b75e Include the warning about removing .git directory.
* dad47ed Write the first draft of initializing a repo.
* fb0d184 Define the audience.
* 1ebb7a7 Define the objectives.
* ca49180 Initial commit.
```

changes. Now, let's look at our history one more time. So now

- git branch bugfix(but after that we have to switch that branch so we use another way)
- git switch -C bugfix/login-form

```
~/Projects/Venus  git  master
❯ ✓  git switch -C bugfix/login-form
Switched to a new branch 'bugfix/login-form'

~/Projects/Venus  git  bugfix/login-form
```

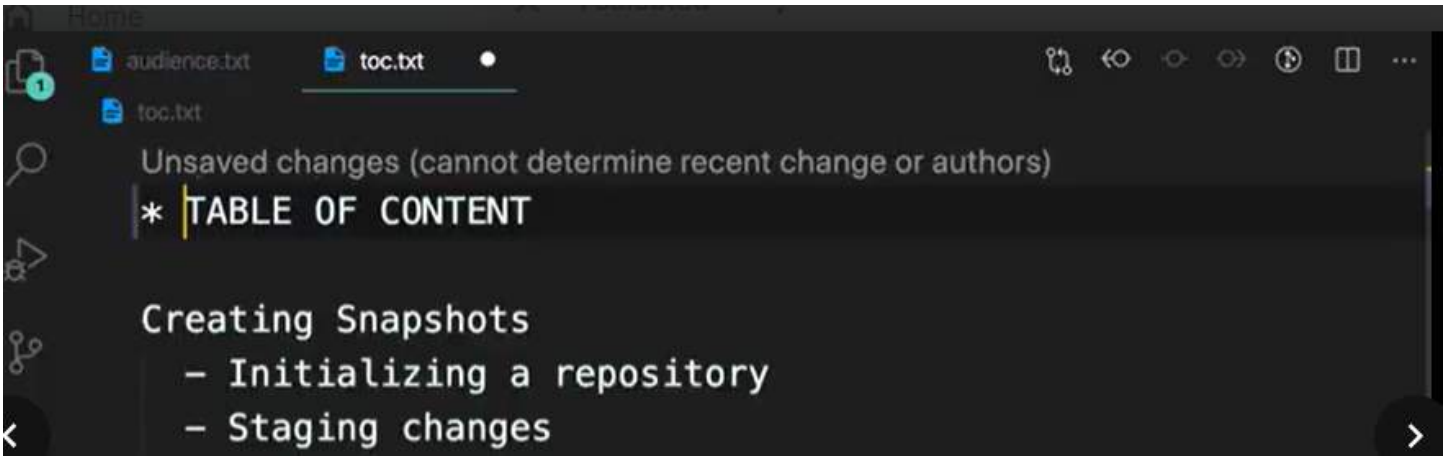
- code toc.txt

audience.txt toc.txt

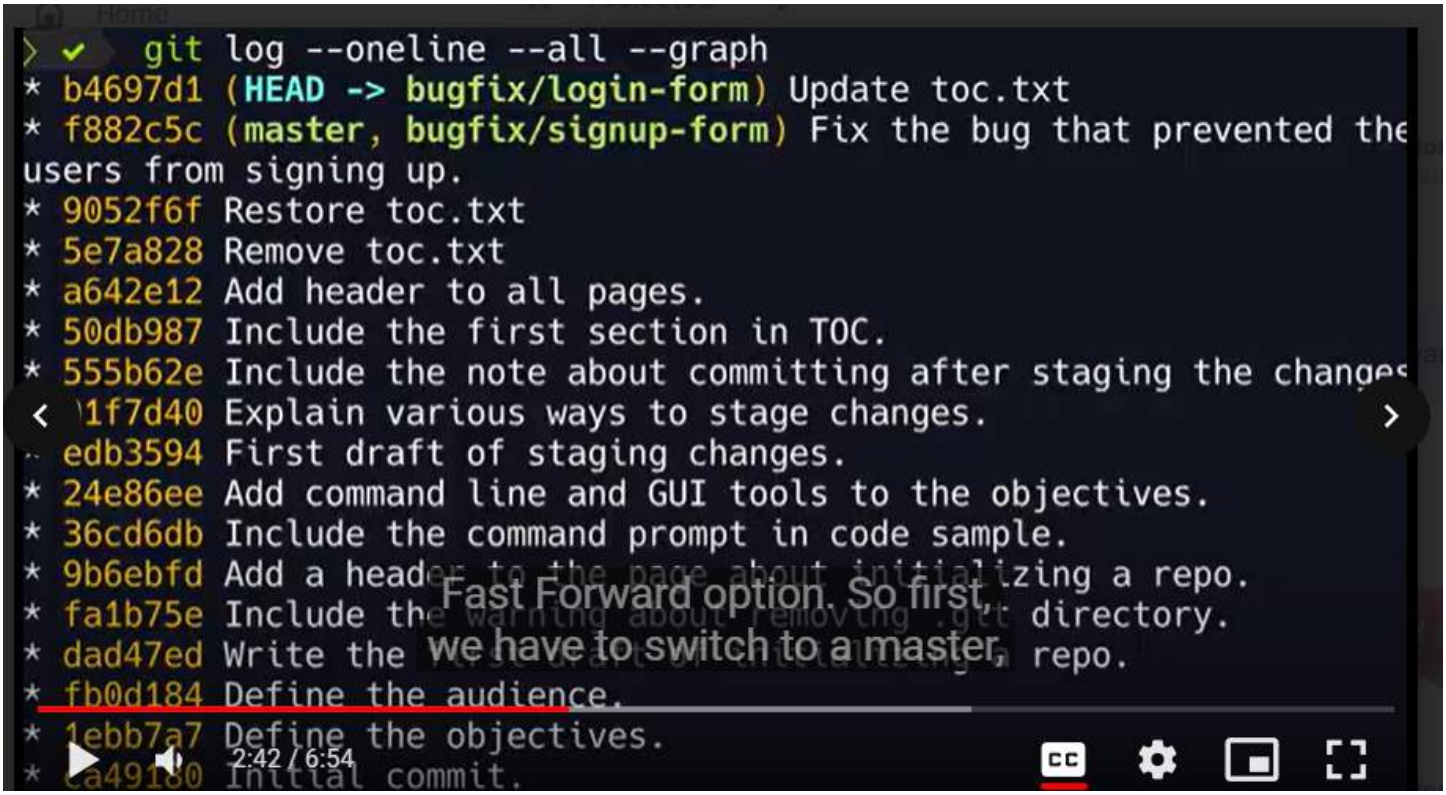
toc.txt

You, a day ago | 1 author (You)

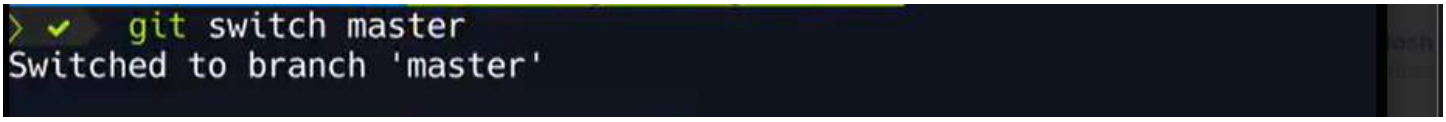
```
TABLE OF CONTENT

Creating Snapshots
  - Initializing a repository
  - Staging changes
```



- git add .
- git commit -m “update toc.txt”
- git log --oneline --all --graph



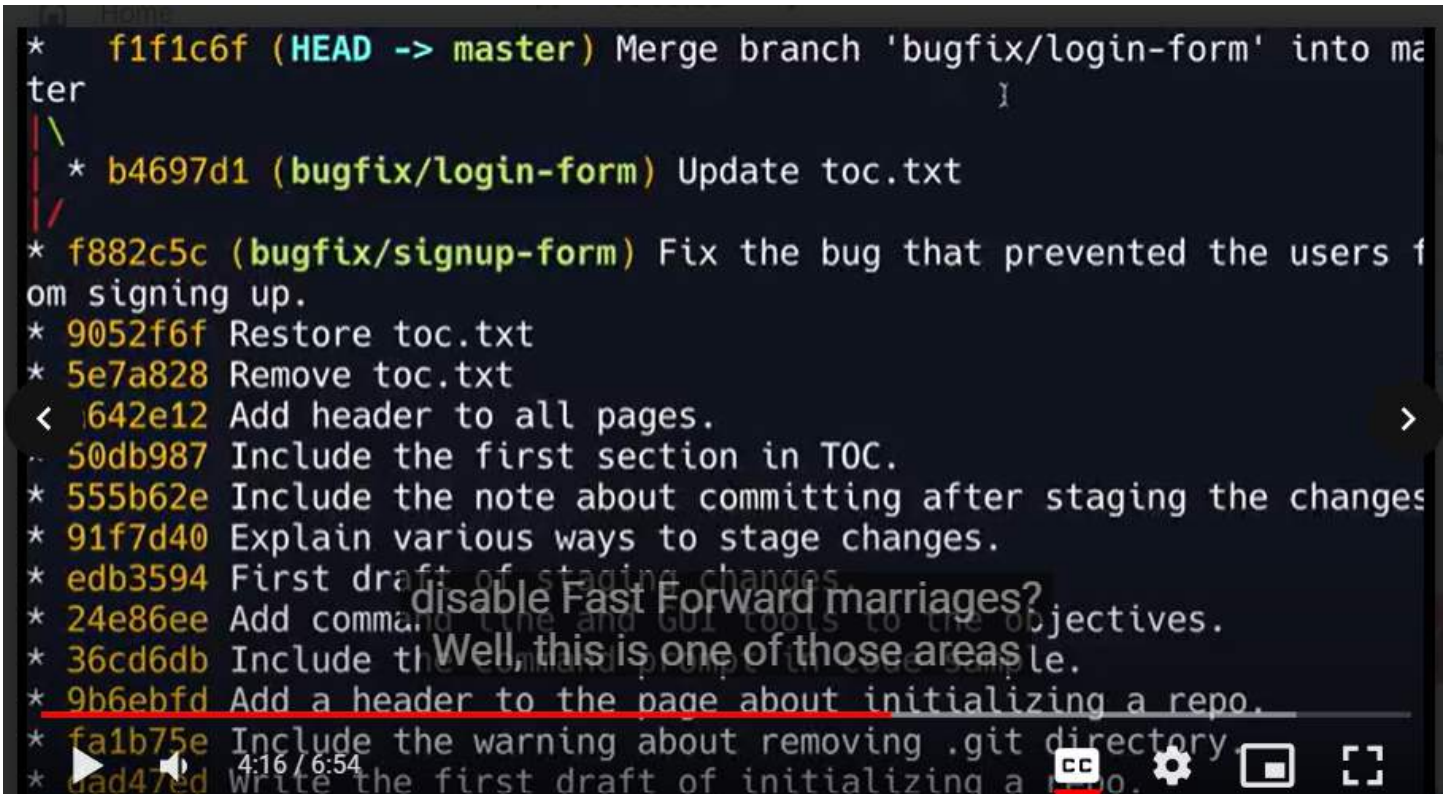
- git switch master



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

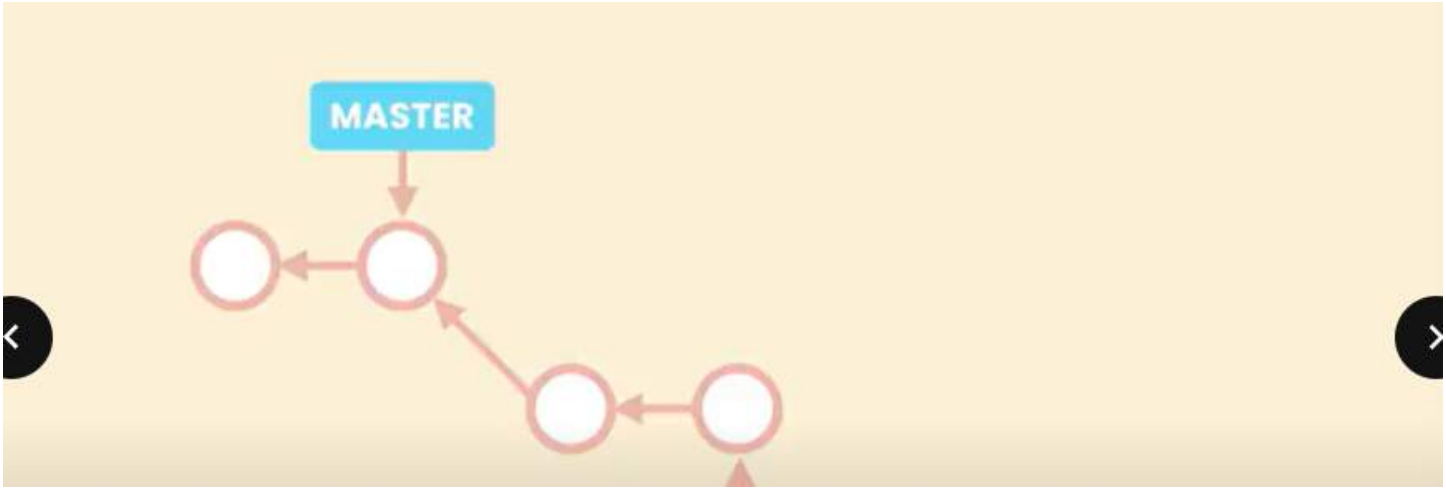
If fast forward is possible dont do it create a merge commit that combines all the changes in this target branch and bring them into Master

- git log --oneline --all --graph



- (with shakeeb bhai)

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



Feature

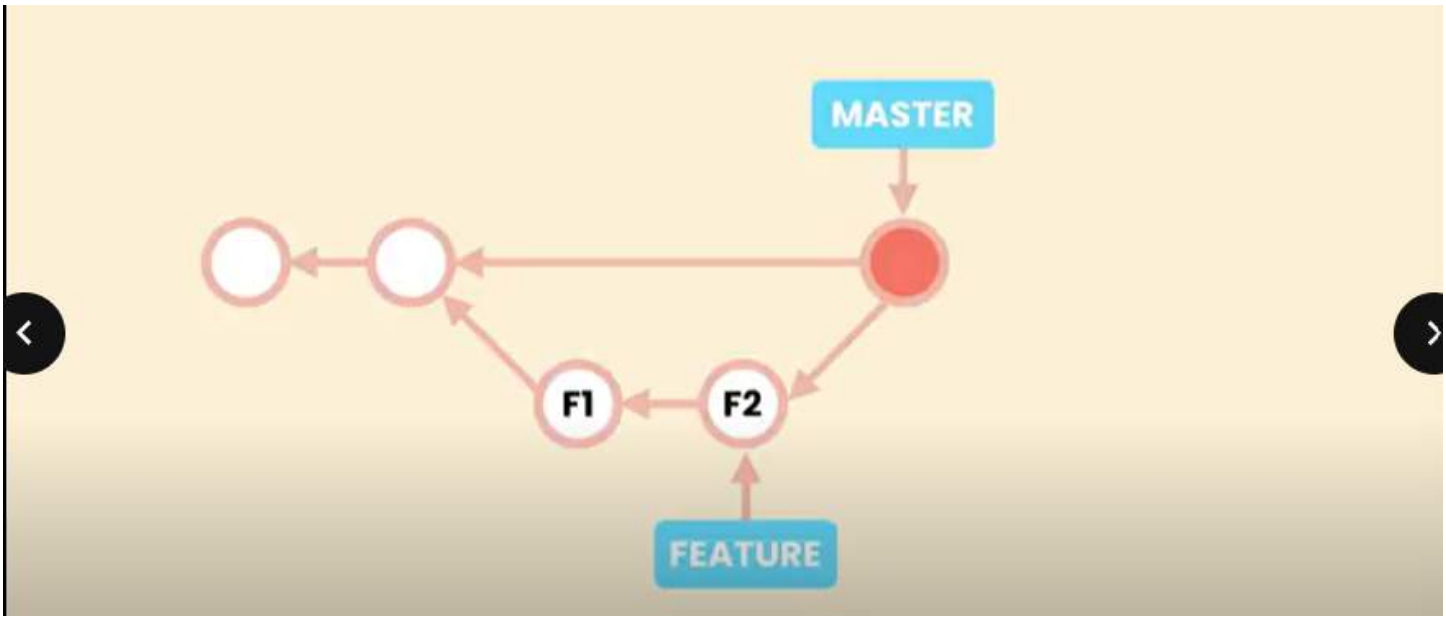
fast forward merge

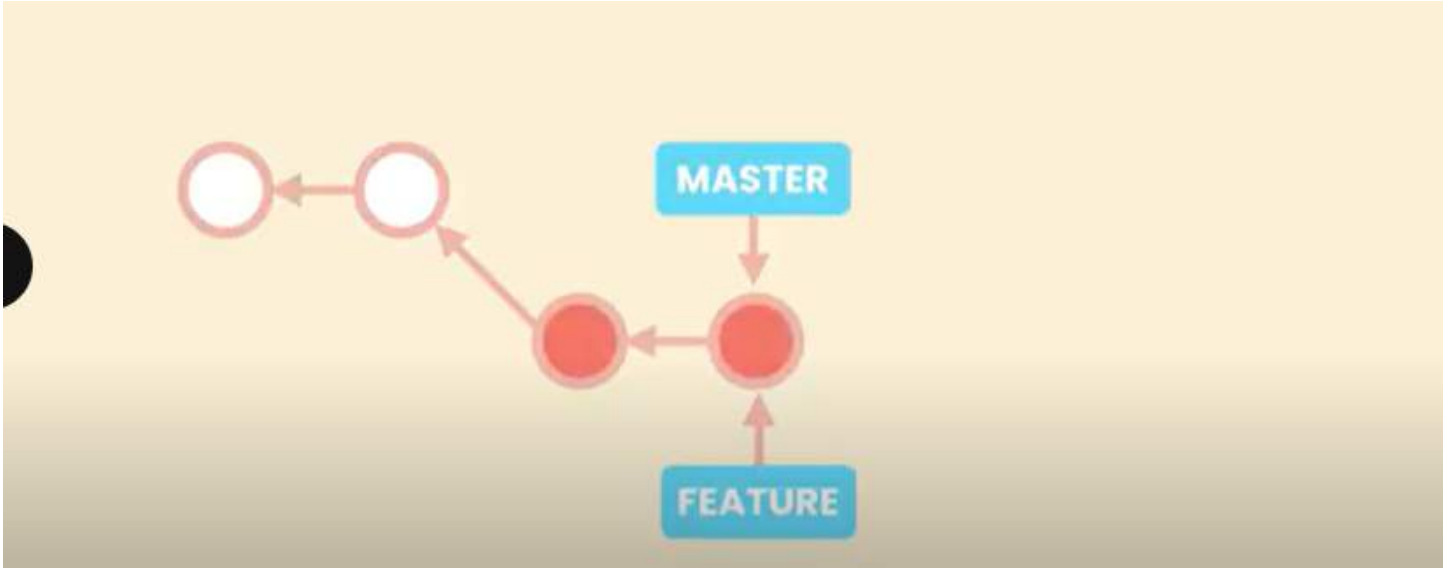
.....If you are not using fast forward merge what happen we have a merge commit that combines all the changes in the feature branch this new merge commit combines all the changes in F! and F2 lets see we work on this feature branch than we told to remove this feature from the codebase . We can easily revert the last commit in the master branch . we will discuss about reverting in future.when we talk about rewriting history.

By reverting a commit we create commit that is the opposite of that commit. So essentially it undoes everything happened in another commit .So If we use a merge commit there is a single commit that we have to revert.

In contrast we use fast forward option Master Pointer is going to move over feature branch and now u want to take out this feature from

the code base we have more commits that we have to revert more commits this can be a bit more complex.So once again both this argument are valid.



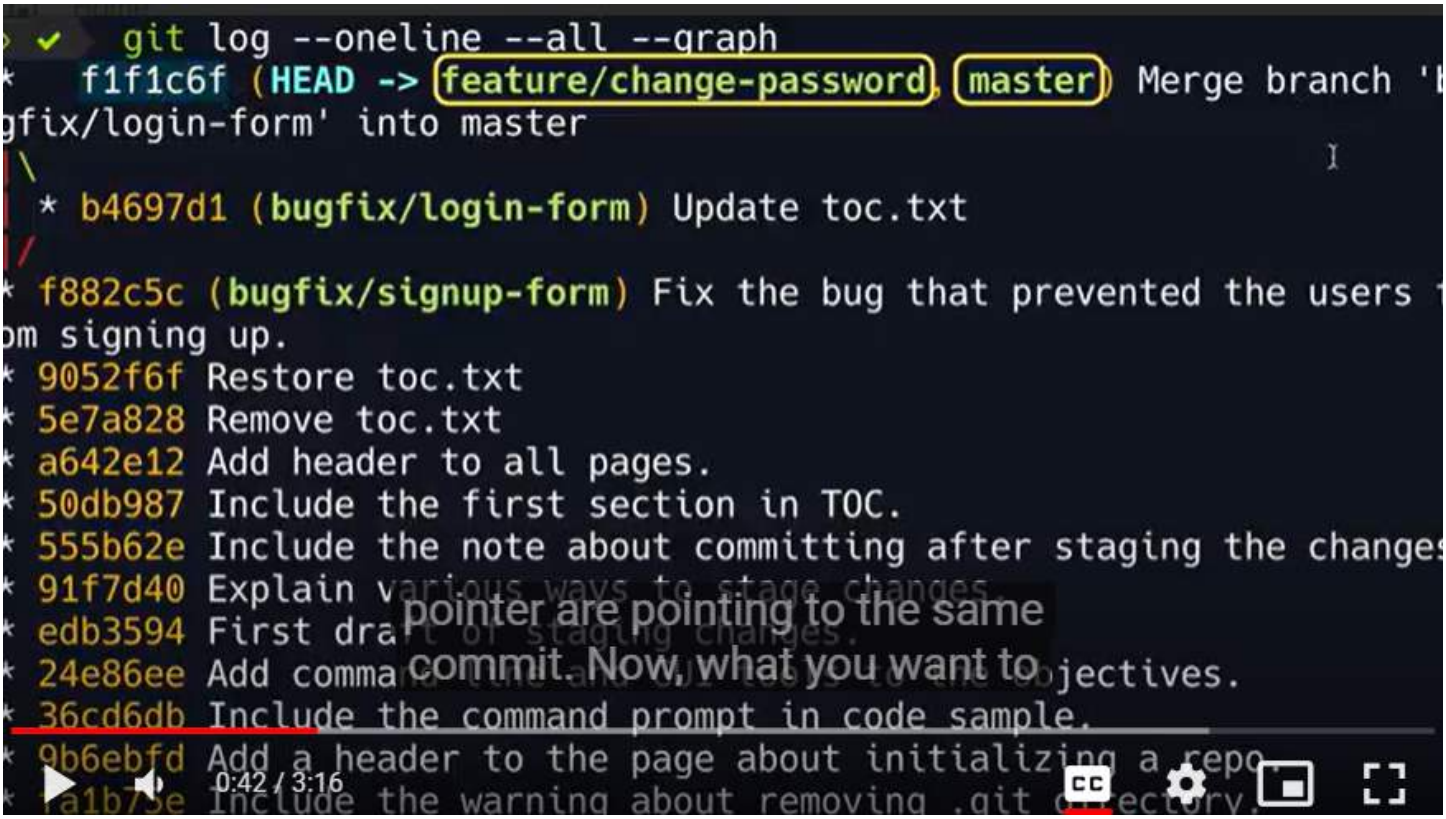


3 way -Merge

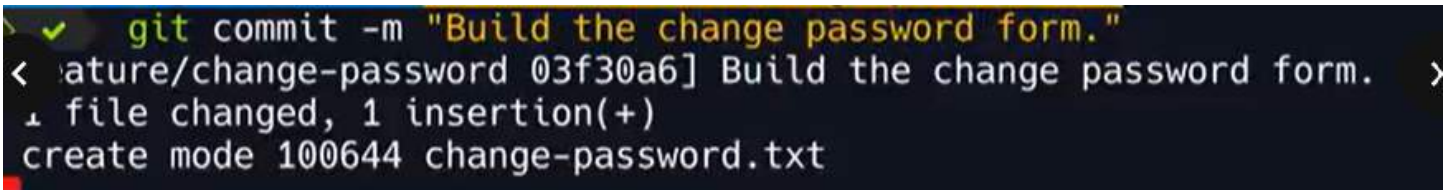
git switch -C feature/change-password

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

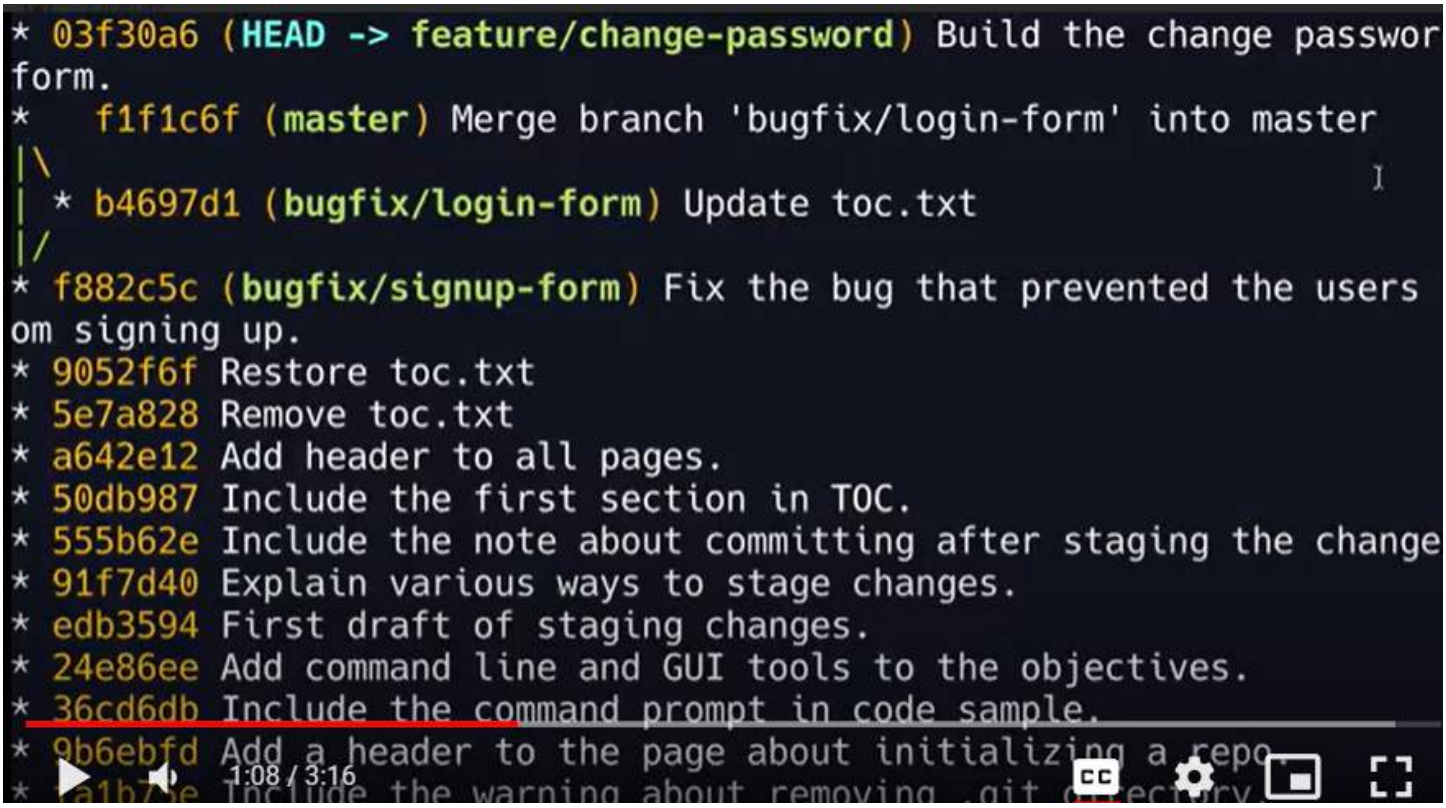
git log --oneline --all --graph



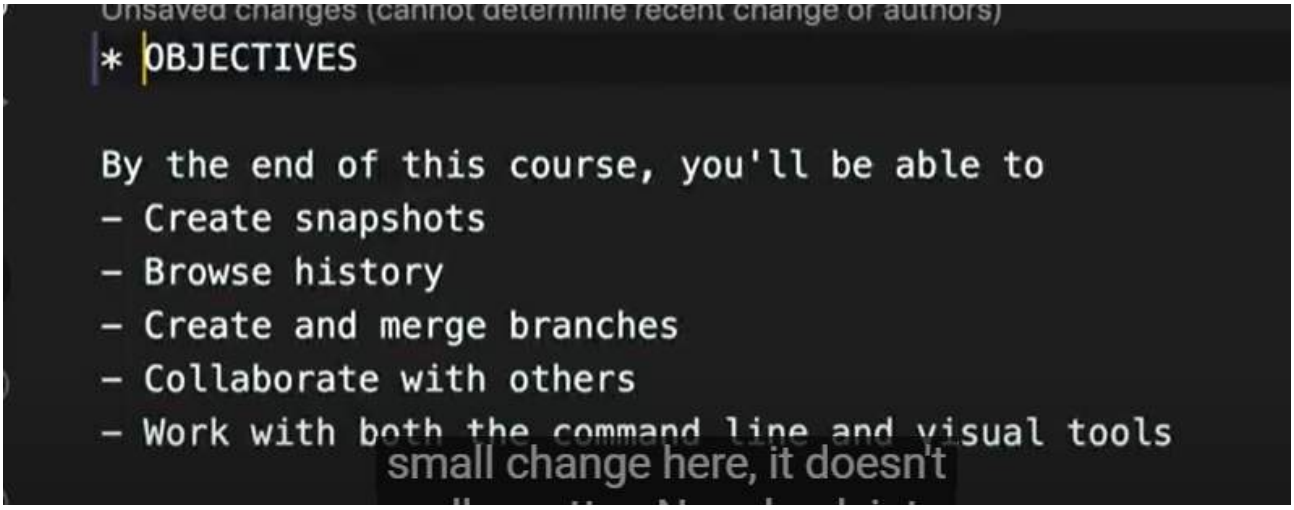
- change password pointer and master merge pointer on the same point(commit)
- HOW TO BRANCHES CAN DIVERGE
- echo hello > change-password.txt
- git add .
- git commit -m “Build the change password form.”



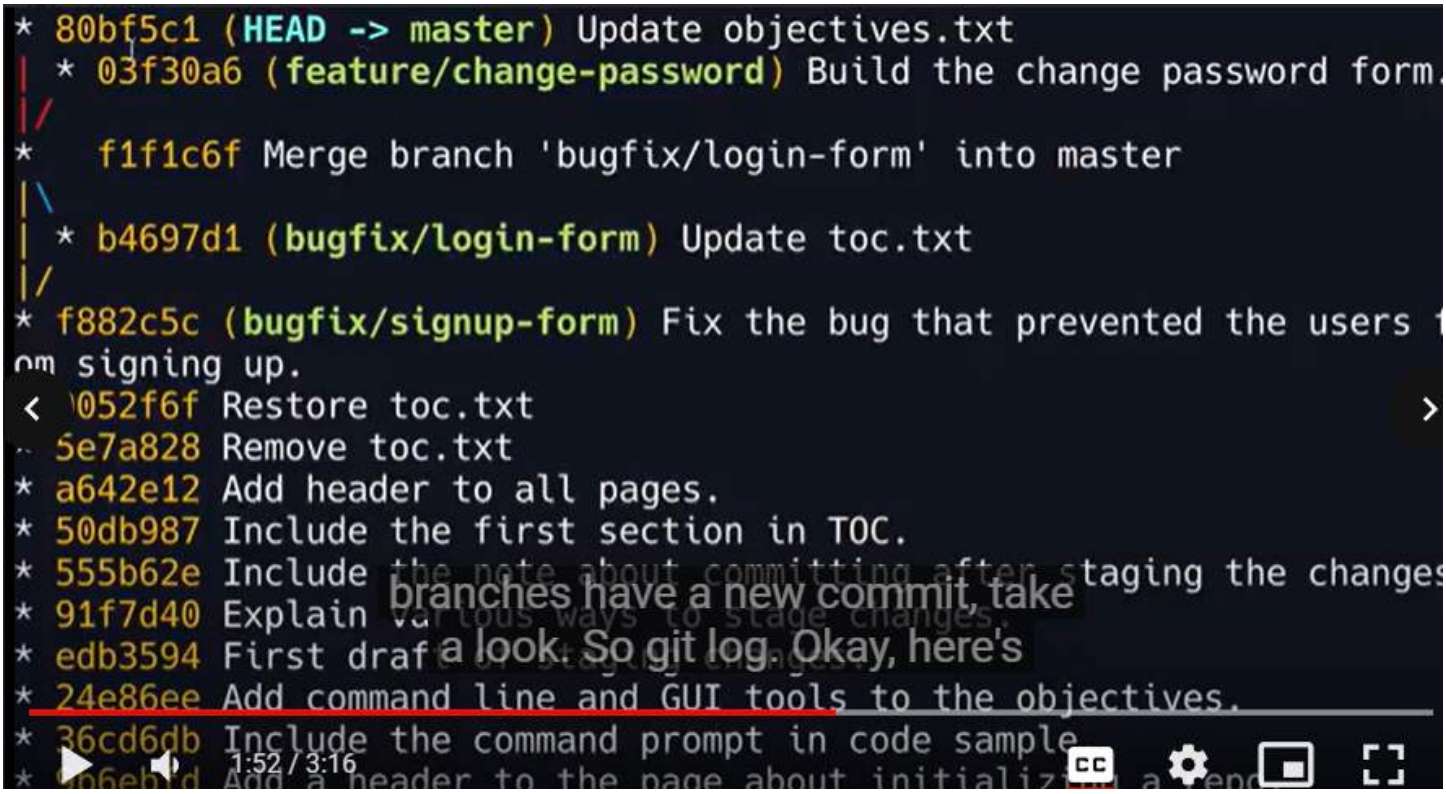
- git log --oneline --all --graph



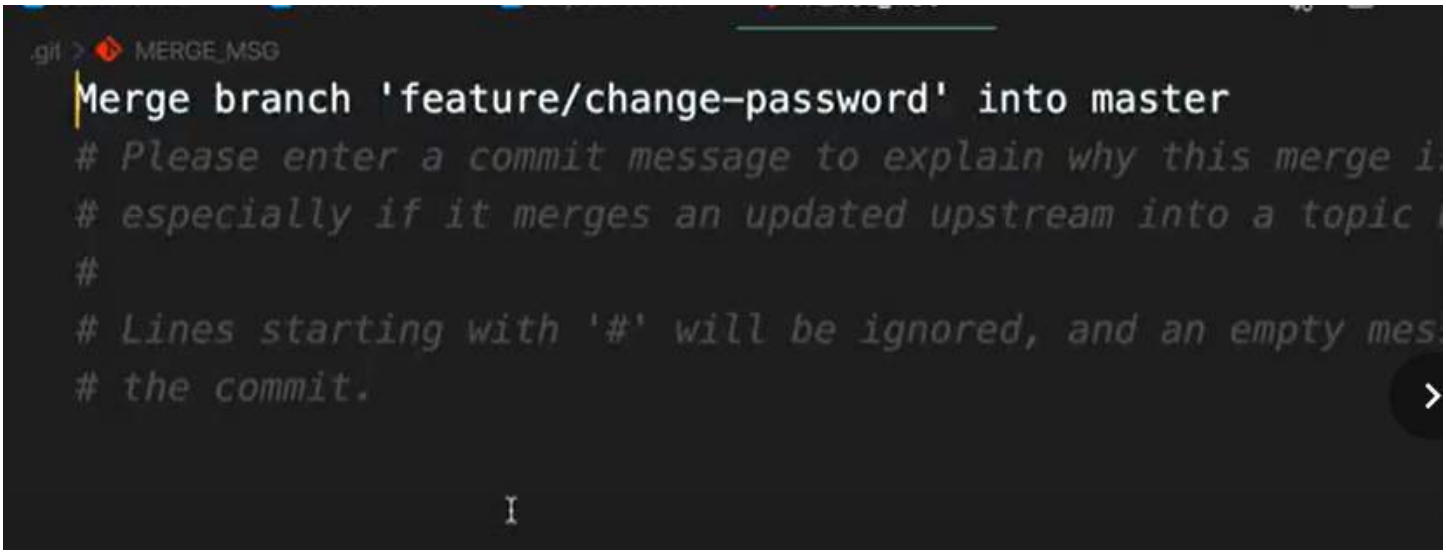
- for diverge branch go back to master branch
- git switch master
- code objectives.txt

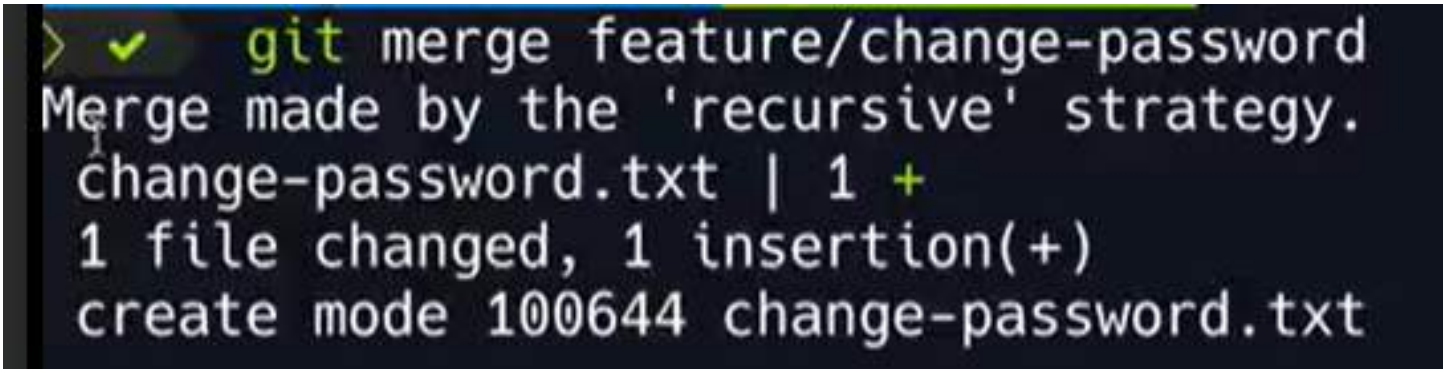


- git add .
- git commit -m “update objectives.txt”
- git log --oneline --all --graph

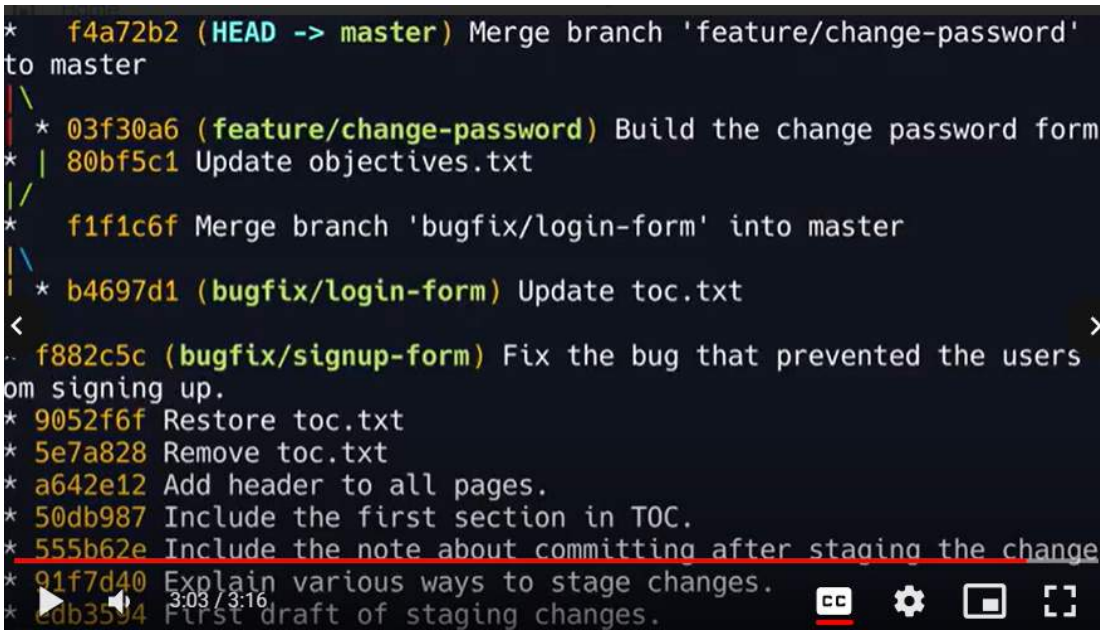


- u can see branches are diverged.this(f1f1c6f) is the commit we added a new commit in this branch and go back to master and added a new commit to Master .Now we do not have a direct linear path from change password to master .if we stuck in change-password we cant go to the master.
- here is on the master branch
- git merge feature/change-password





- 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

1. change1, change2
2. change,Delete
3. Add1,Add2

When we merge branches there is a conflicts

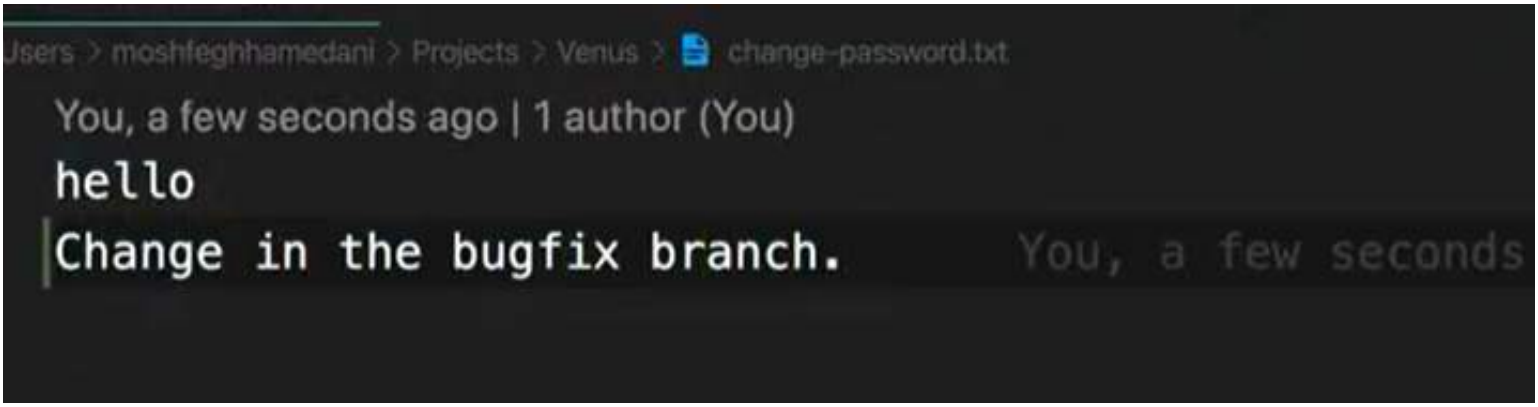
1. Conflicts happen when the same line of code has been changed in different ways in two branches
2. If given file is changed in one branch but its deleted in another branch
3. when the same file is added twice in two diffrent branches .But the content of this file is different

In these cases git cannot figure out how to merge the changes .So it will stop the merge process.So in this case we need to jump in and tell Git how we want to proceed.

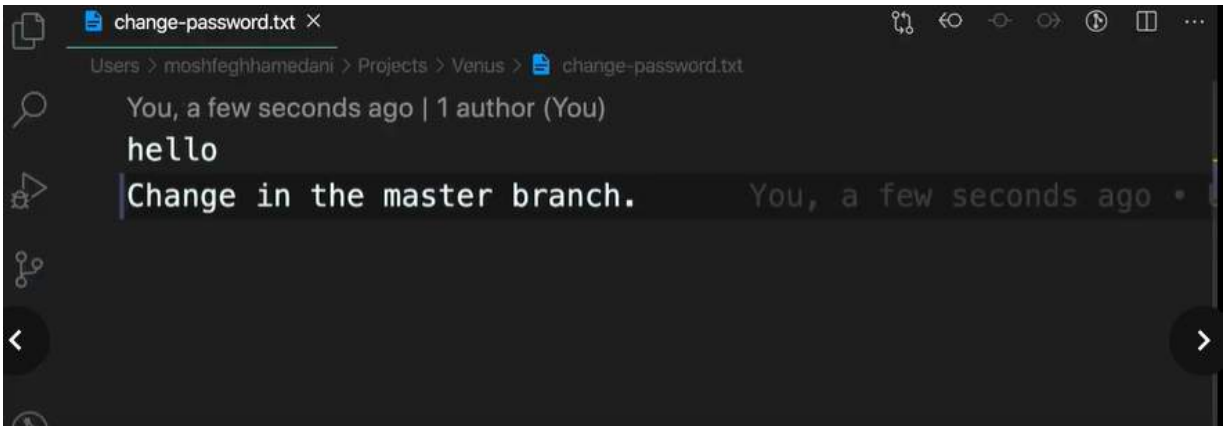
Examples

On the master

- git switch -C bugfix/change-password
- code change-password.txt



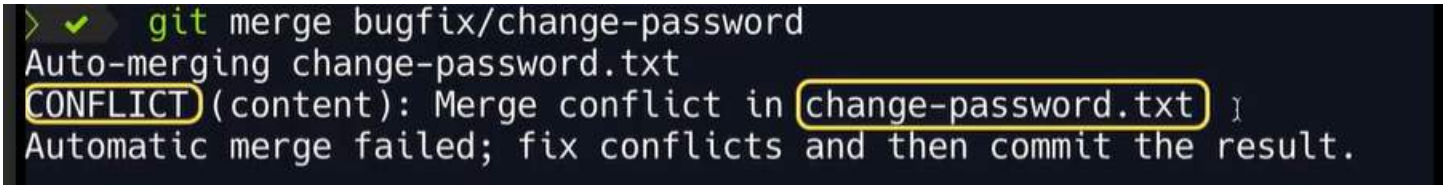
- git add .
- git commit -m “update change-password.txt”
- Now switch to master branch
- git switch master
- code change-password.txt



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

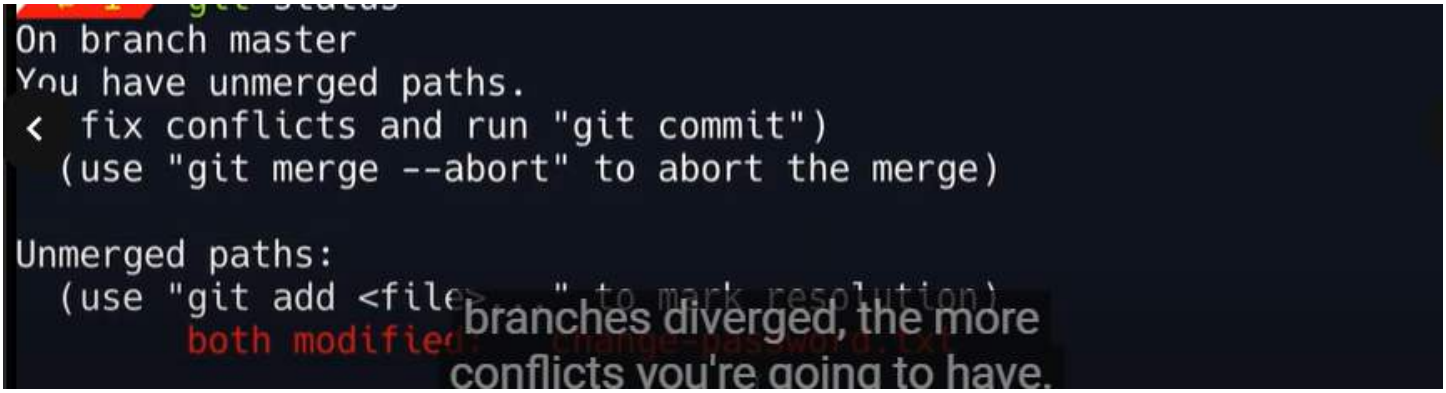
Both changes merge in diffrent ways

git merge bugfix/change-password



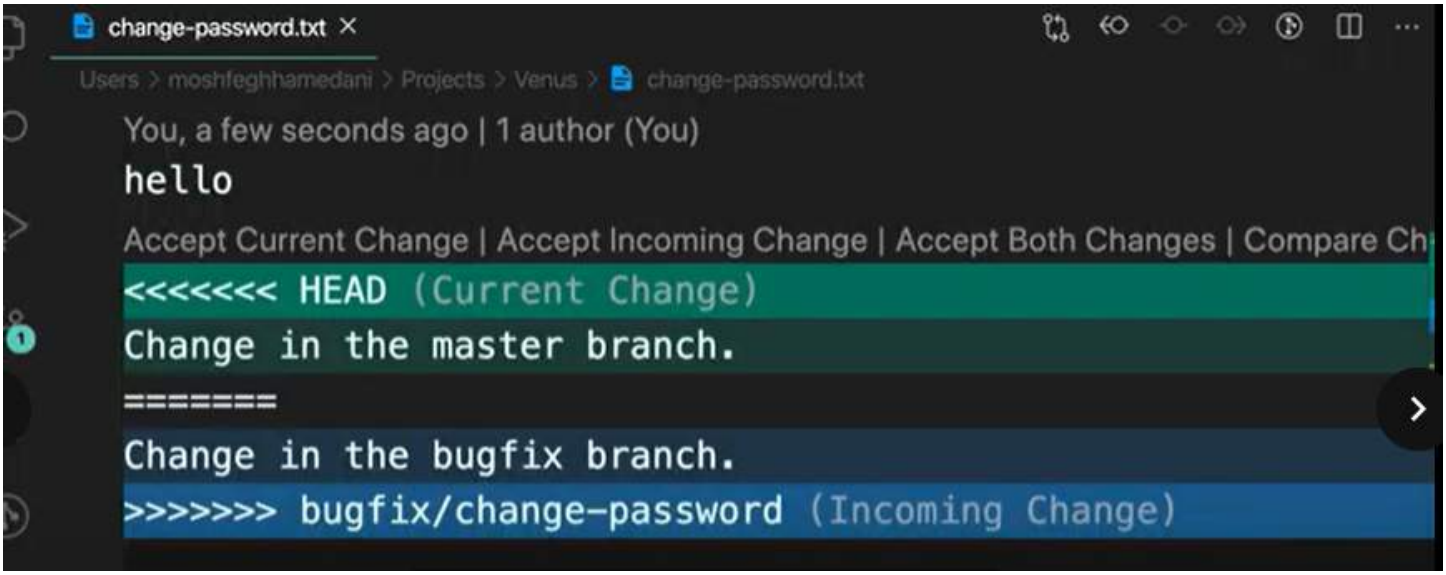
now conflict .I know we have to jump in and manually combine the changes.we are in the middle of a merge process. so lets run

git status



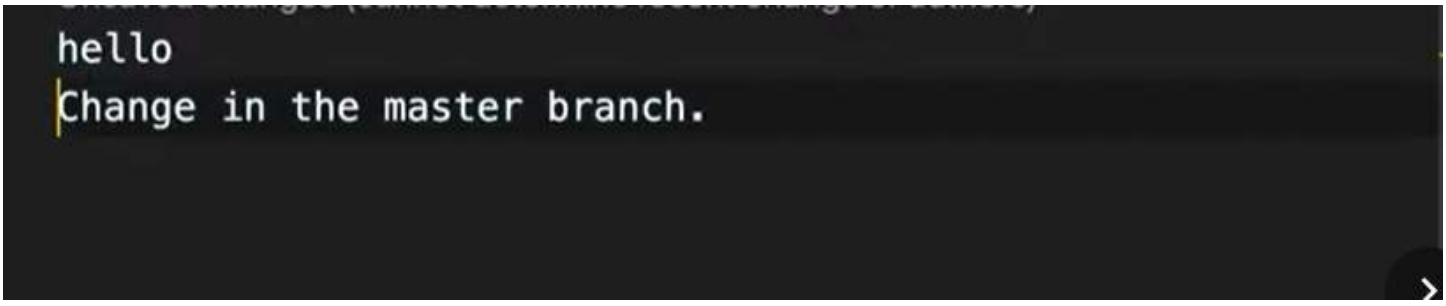
this is very simple scenario in real world we have 10 files listed here basically the more branches diverged the more conflict you are going to have .

- code change-password.txt



there is a marker change in one branch and another branch

HEAD current branch we have some change and in other branch we have some change.in real situation you have multiple conflicts in a file so for different chunks of code you are going to see these in these markers. so what we can do here in VSCode we can accept in current change



2nd way accept incoming changes

```
hello
Change in the bugfix branch.
```

3rd option accept both changes

- we can also edit manually merge

```
You, a few seconds ago | 1 author (You)
hello
Change in the bugfix branch.
Change in the master branch.      You, a few seconds ago
```

- 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 status
On branch master
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)
<
Changes to be committed:
        modified:   change-password.txt
```

- git commit

```
Users > moshfeghhamedani > Projects > Venus > .git > COMMIT_EDITMSG
Merge branch 'bugfix/change-password' into master

# Conflicts:
# change-password.txt
#
# It looks like you may be committing a merge.
# If this is not correct, please remove the file
```

Aborting A Merge

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

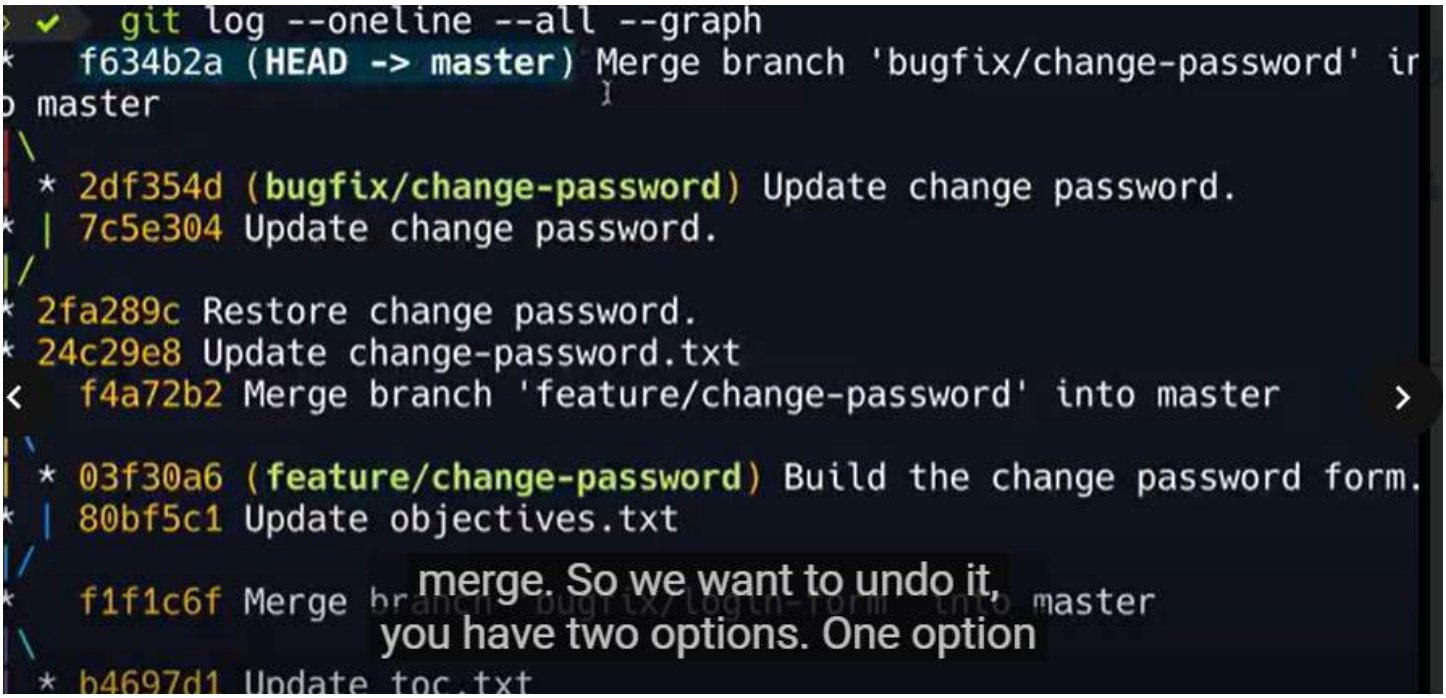
git merge --abort

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.

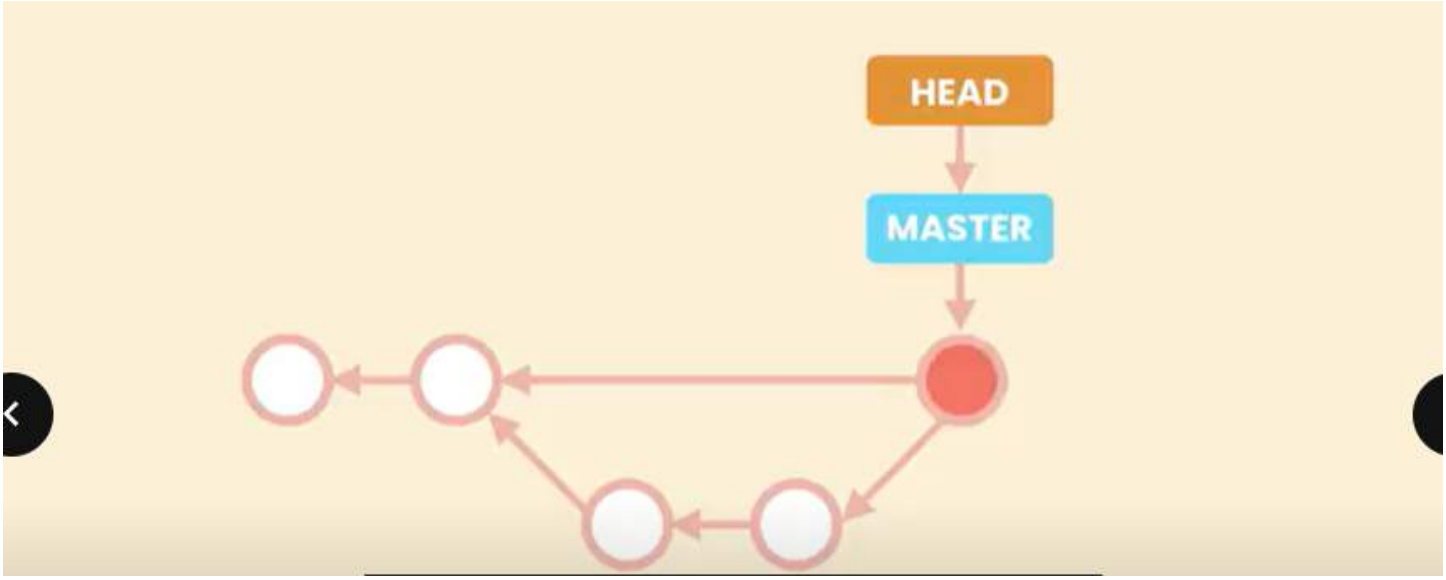
- git log --oneline --all --graph



- on the top we have merge commit Lets assume that this is a faulty merge. So we want to undo it you have two option

1. to remove this commit(remove means rewriting history we have to be very careful)(its ok when it is in local repo but if its in remote then we shouldnt rewrite our history.)
2. we should revert it instead of removing commit we should revert it .which means we are going to create a new commit that will cancel all the changes in this commit. In this we are going to look at both the solution.

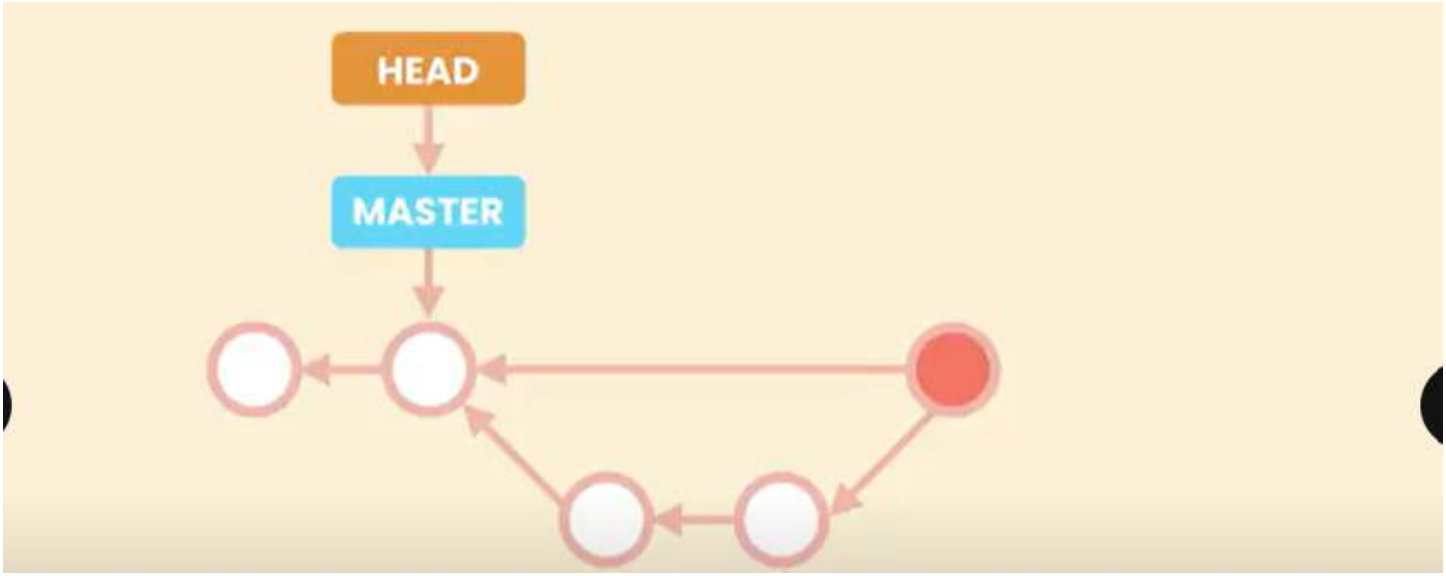
- [ ] First we have to see how to remove the last commit .here is the picture of our history both the HEAD & MASTER pointers are pointing to the last commit which is a MERGE commit



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



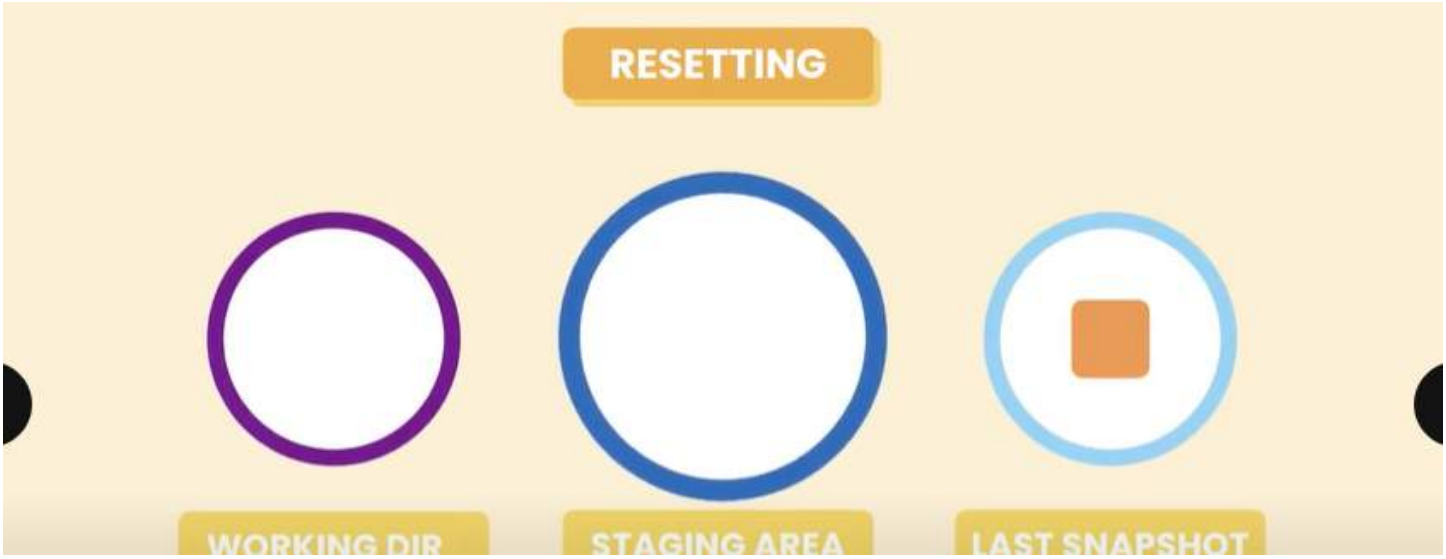
FEATURES

- git reset --hard HEAD ~1

RESETTING

- Soft
- Mixed

- hard



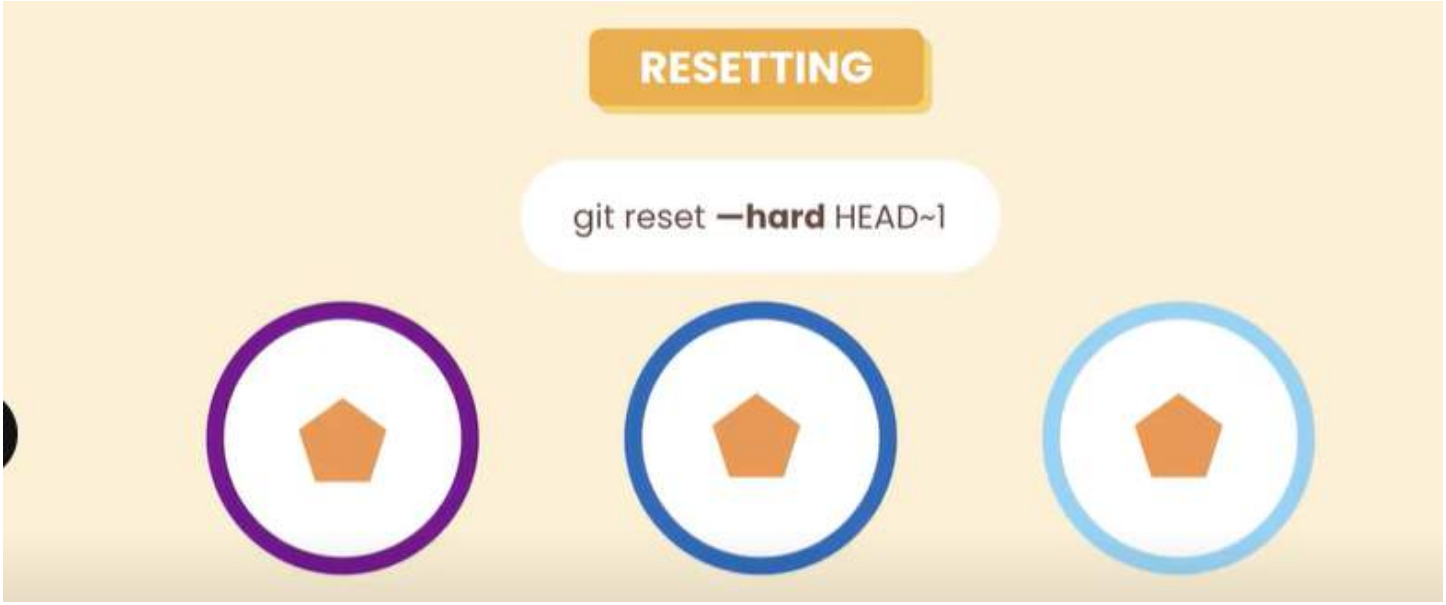
here we have working directory, staging area and last snapshot .Let me reset the head using soft option

- git reset --soft HEAD~1
- git pointer point to a different snapshot But our staging area and working directory are not affected .

▢ Mixed

- If we used the mixed option the default option so we dont have to specify it git is going to get new snapshot and put it in the staging area as well .

- 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
rom signing up.
```

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

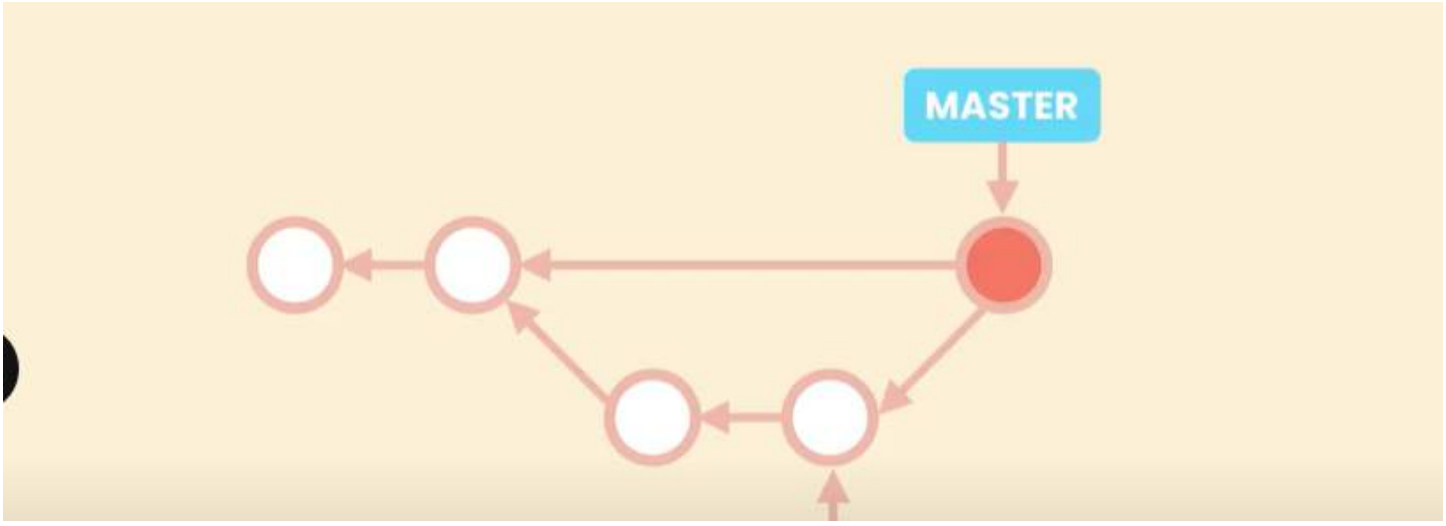
```
git reset --hard f634b2a
HEAD is now at f634b2a Merge branch 'bugfix/change-password' into mas
ter
```

```
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
```

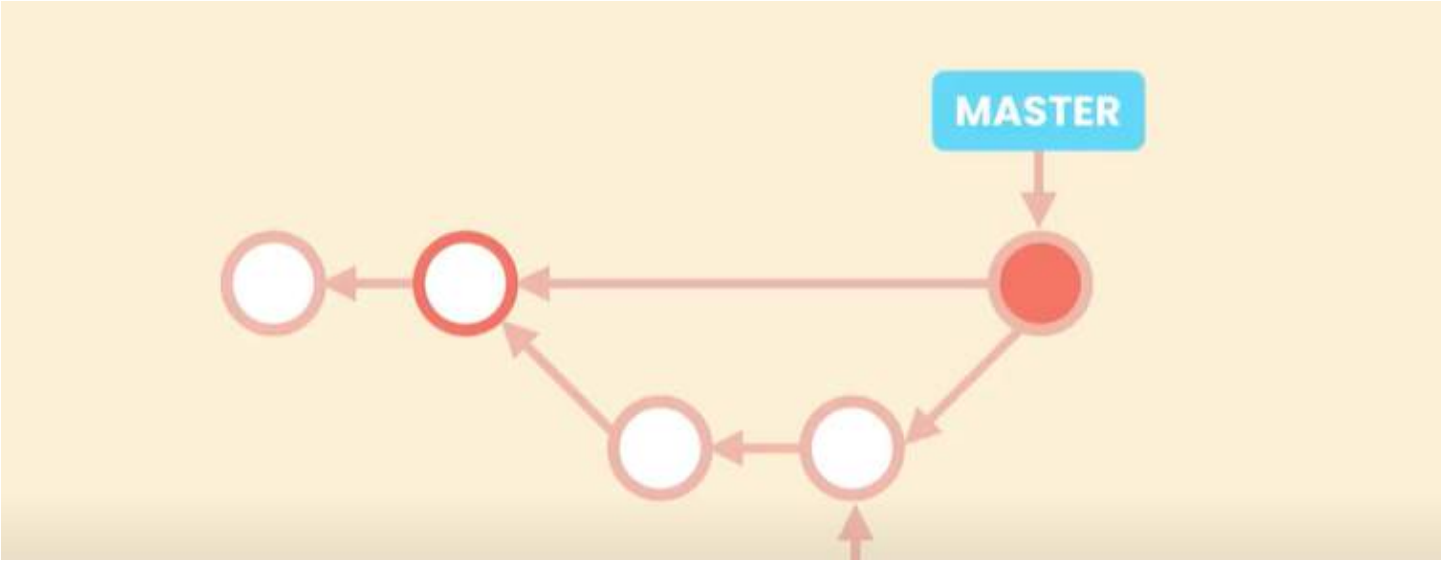
- Second option Revert the commit
- git revert HEAD

```
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 ,

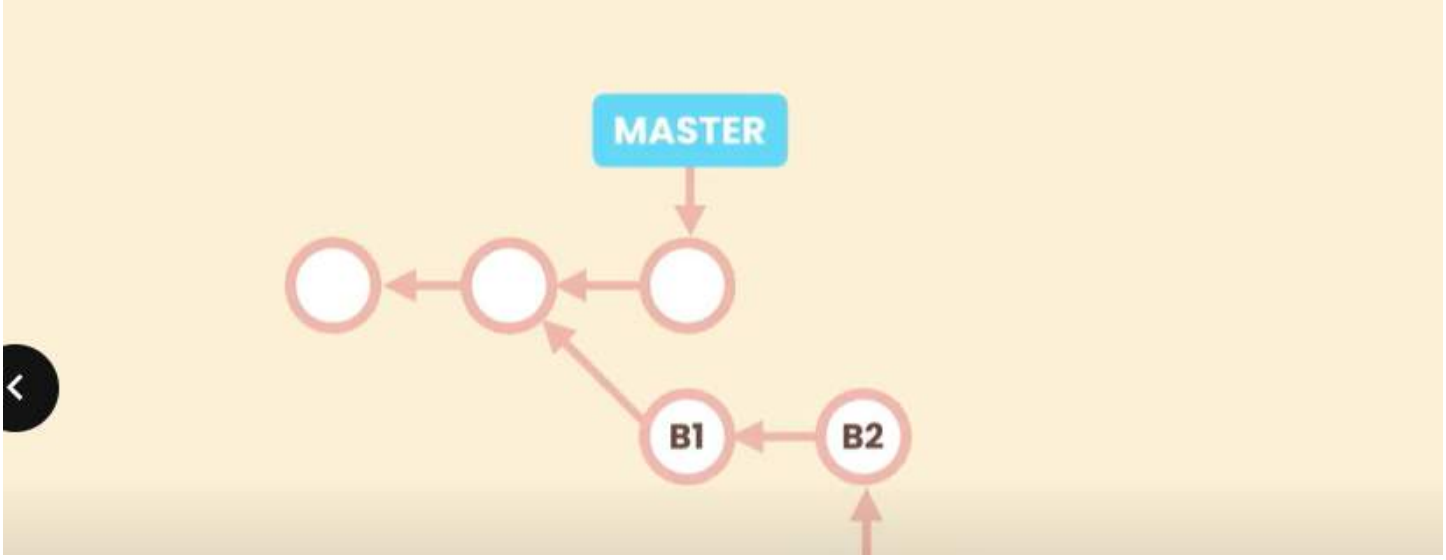
- git revert -m 1 HEAD(target the last commit)
-

```
Revert "Merge branch 'bugfix/change-password' into master"

This reverts commit f634b2afbc815f489641ae7ade2ea9061ee383c0
changes made to 7c5e304e2ab03a94e0bbf4e86bf41a934a7ccbf6.

# Please enter the commit message for your changes. Lines
# with '#' will be ignored, and an empty message aborts the
#
```

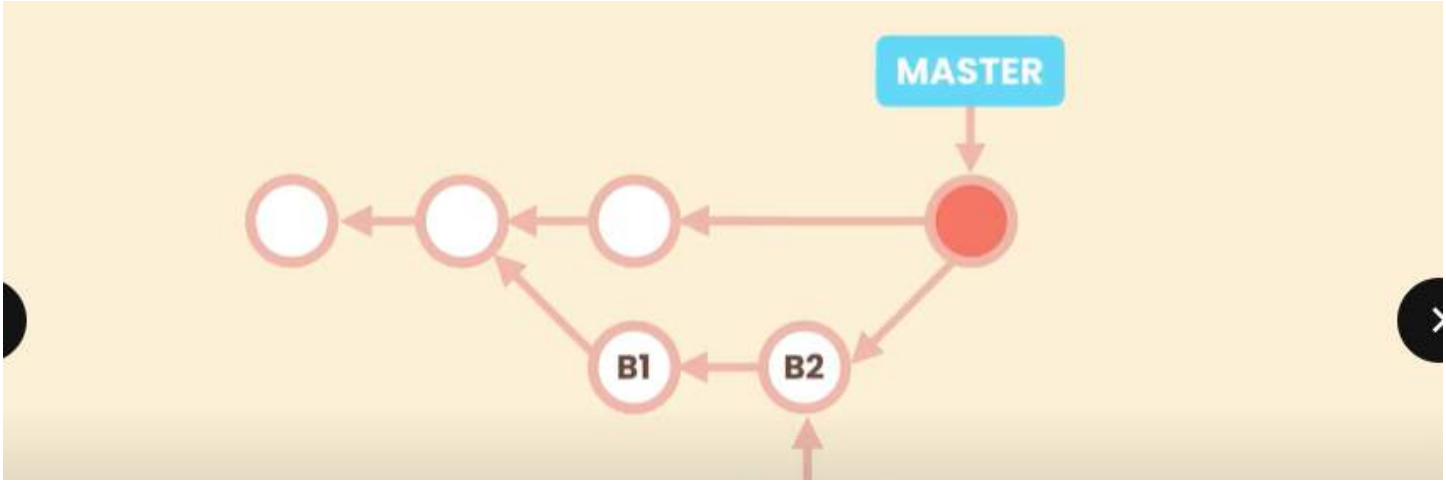
git log --oneline --all --graph

```
* 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
```

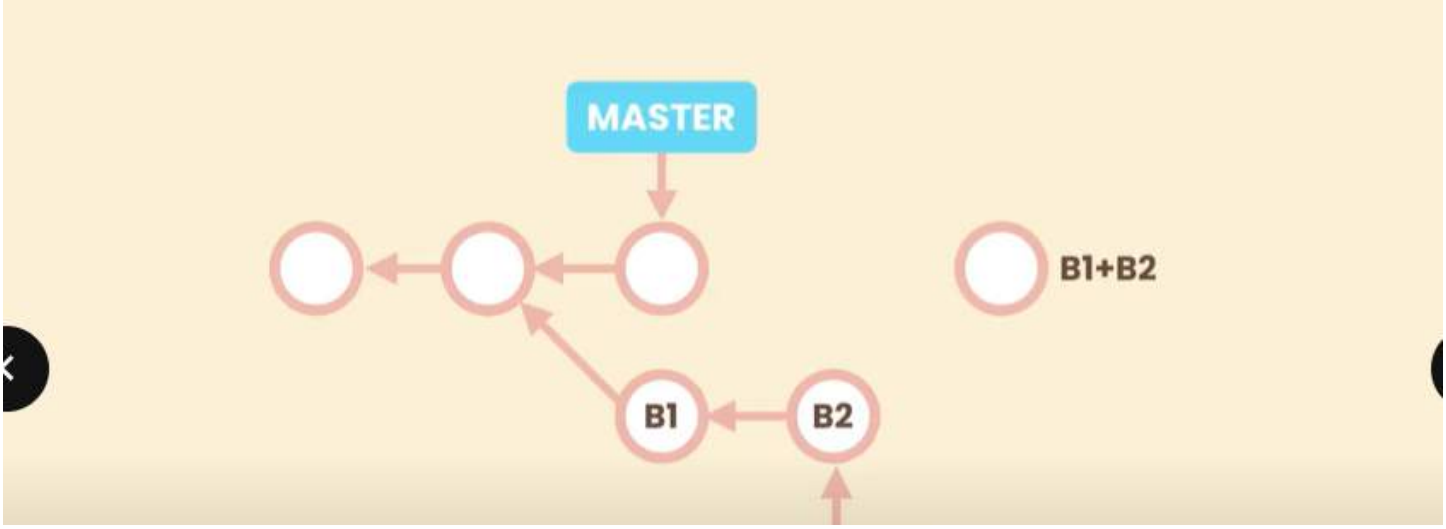
SQUASH MERGING(15)



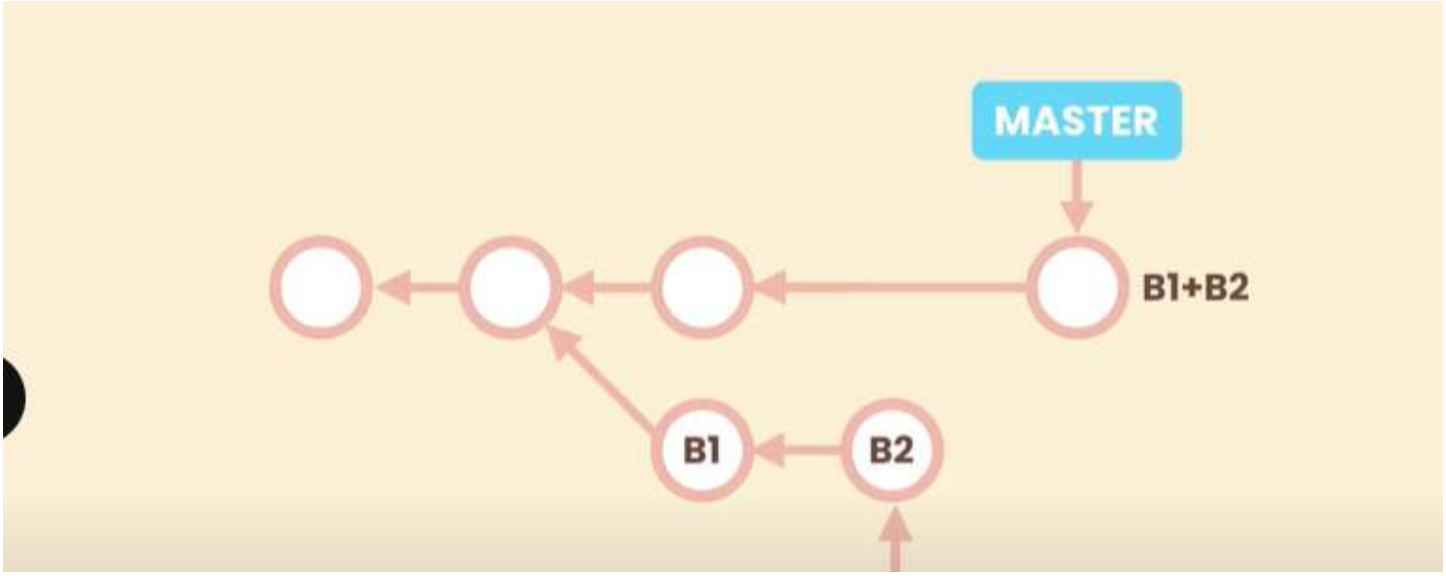
BUGFIX



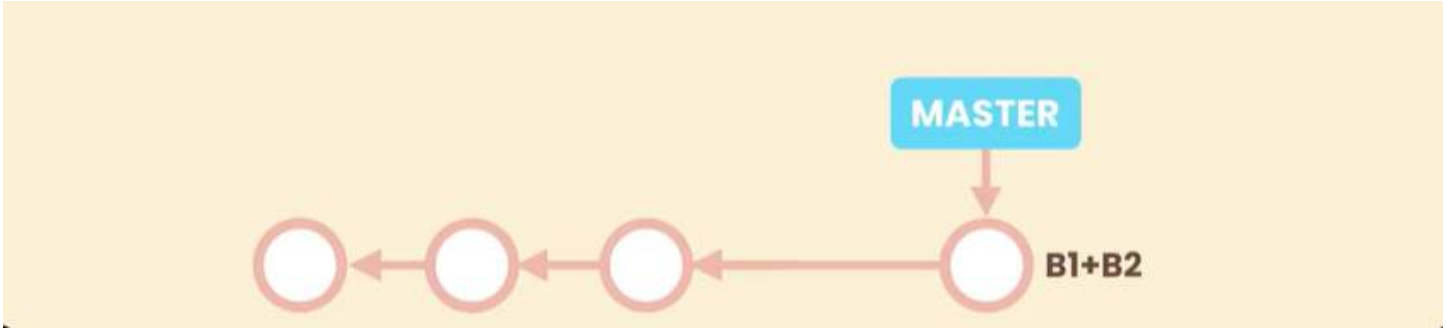
Bugfix



BUGFIX



BUGFIX



We have a bug fix branch B1 & B2 are the part of history of Master.But what if these Commit are not good quality commits, may be they are too fine grained or they don't represent a single logical chain set may be.

WE have mixed different things in each Commit. So These Commits are not good quality Commits. Perhaps we added this commit at checkpoints along the way. So we don't really care about the history of the bugfix branch.

So we dont really care about the history of the bigfix branch.So we dont want these commits to pollute the history of SQUASH MERGING.So lets undo the merge and we are going to create a new commit that combines all the changesin the bug fix branch. So we have a single logical chain set that represent all the changes for fixing this bug .

Now we can apply this commit on top of the master .

This is what we call squash merging now u need to pay attention .New commit is not a new commit Because it doesnot have two parents.

It doesnot have reference to . It just a regular commit that we have added on top of master.

Now we're done with the bug fix branch so we can delete it. And left with a simple clean and linear history.

This is the benefit of SQUASH MERGING But it doesnt mean you should use this technique all time. U should use this techniques all time.

U should use it ith small, short lived branches with bad history like bug fixes or features that you can implement in a few hours or in a day.

Now lets see a SQUASH MERGING in action.

- 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

```
* 827885a (HEAD -> bugfix/photo-upload) Update toc.txt
* 243d308 Update audience.txt
* 3a44949 (master) Revert "Merge branch 'bugfix/change-password'
master"
*   f634b2a Merge branch 'bugfix/change-password' into master
|\
| * 2df354d (bugfix/change-password) Update change password.
* | 7c5e304 Update change password.
|/
* fa289c Restore change password.
* 24c29e8 Update change-password.txt
*   f4a72b2 Merge branch 'feature/change-password' into master
|\
| * 03f30a6 (feature/change-password) Build the change password f
* | 80bf5c1 Update objectives.txt
|/
*   f1f1c6f Merge branch 'bugfix/login-form' into master
|\
```

our bug fix branch is two

- git switch master
- git merge --squash bugfix/photo-upload
- git status -s

```
> ✔  git merge --squash bugfix/photo-upload
Updating 3a44949..827885a
Fast-forward
Squash commit -- not updating HEAD
 audience.txt | 1 +
 toc.txt      | 2 +-
 2 files changed, 2 insertions(+), 1 deletion(-)
```

```
> ✔  git status -s
M  audience.txt
M  toc.txt
```

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

git log --oneline --all --graph

```
* b697ca8 (HEAD -> master) Fix the bug on the photo upload page.
| * 827885a (bugfix/photo-upload) Update toc.txt
| * 243d308 Update audience.txt
|/
* 3a44949 Revert "Merge branch 'bugfix/change-password' 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
|/
```

's go. Let's bring up the hi
tory. So here on the master, we

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

```
> ✔  git branch --merged
  bugfix/change-password
  bugfix/signup-form
  feature/change-password
* master
```

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)

```
> ✔  git branch -d bugfix/photo-upload
error: The branch 'bugfix/photo-upload' is not fully merged.
If you are sure you want to delete it, run 'git branch -D bugfix/phot
-upload'.
```

because git doesnot say this as a real merge

so we enforced a deletion

```
1  git branch -D bugfix/photo-upload
eted branch bugfix/photo-upload (was 827885a).
```

git log --oneline --all --graph

```
* b697ca8 (HEAD -> master) Fix the bug on the photo upload page.
* 3a44949 Revert "Merge branch 'bugfix/change-password' 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
```

So we have a simple linear history we have a single commit that combines

all the changes for fixing the bugs on the photo upload page .

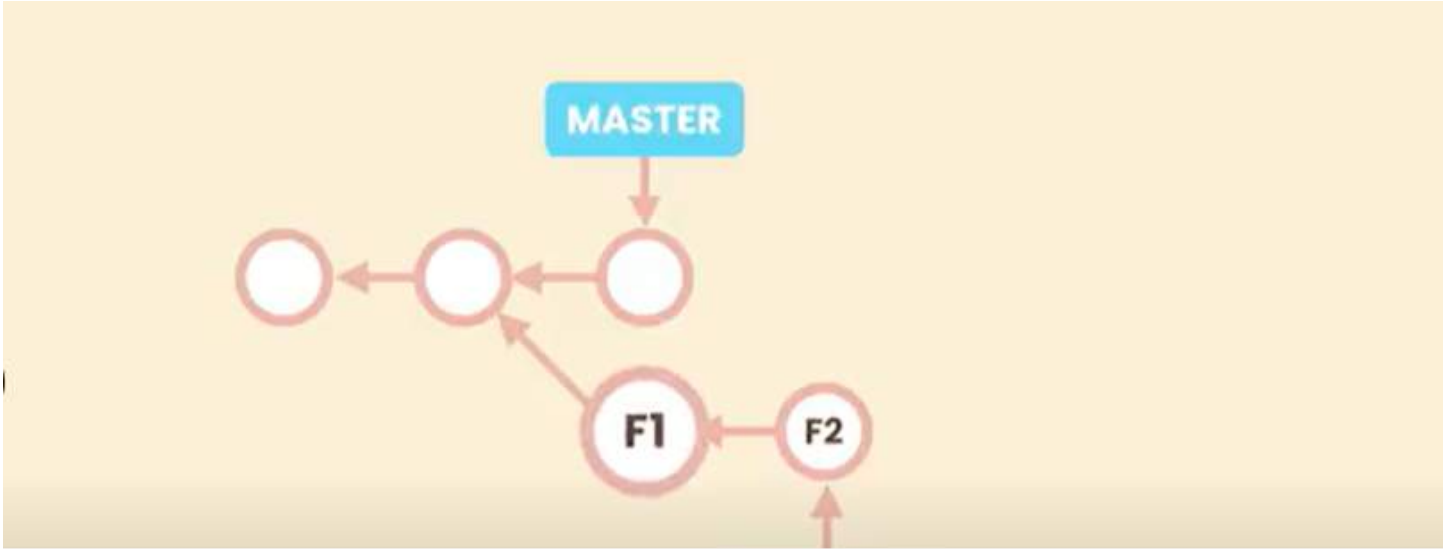
In the last when we r playing a squash merge on top of Master we might end up with conflicts, the process for resolving conflicts is exactly like before so we used merge to resolve the conflicts and then make a commit.

REBASING:

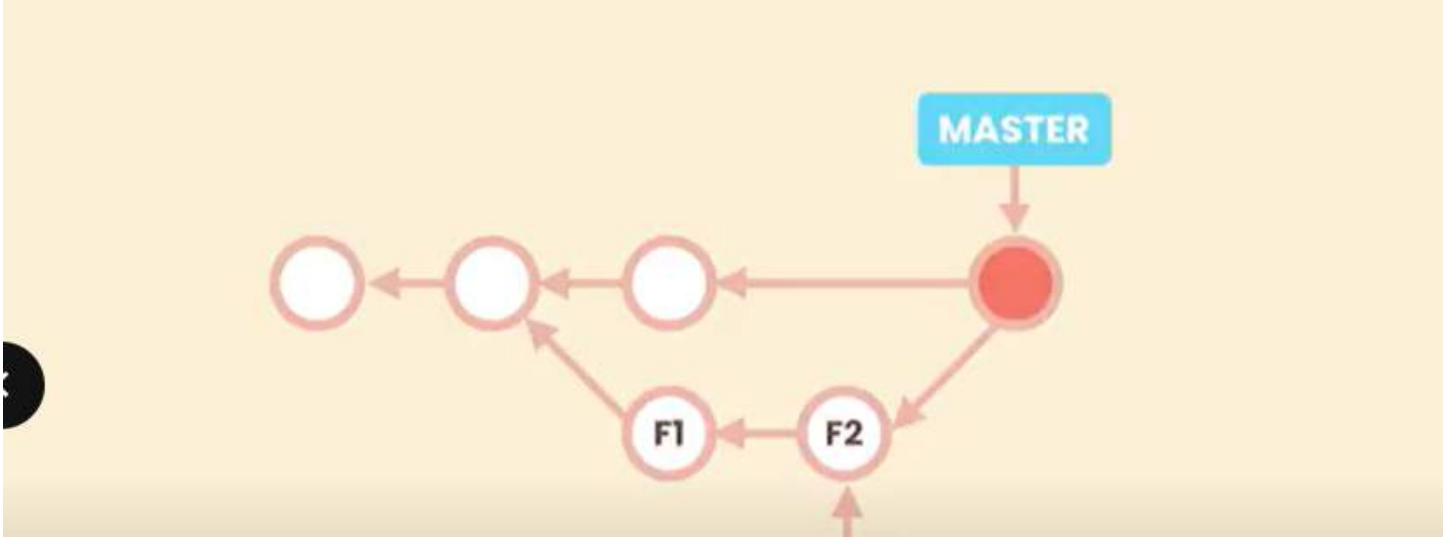
Rewrite history (do it in local repo)

Rebasing is another technique for changing of one branch into another.So we have a FEATURE BRANCH with two commits when we are done we can do a merge.

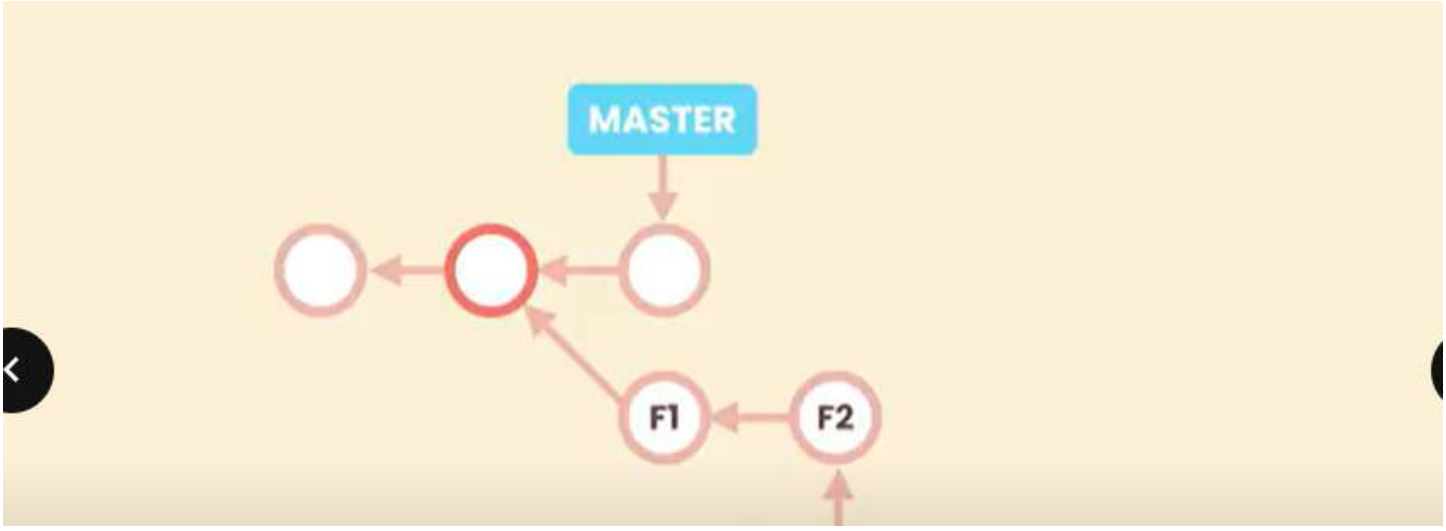
Non Linear History .But History is not complicated but we have more branches and we are working on more complex scenarios.Our History might get really hard to read . so here we can use a different techniques result in linear history.



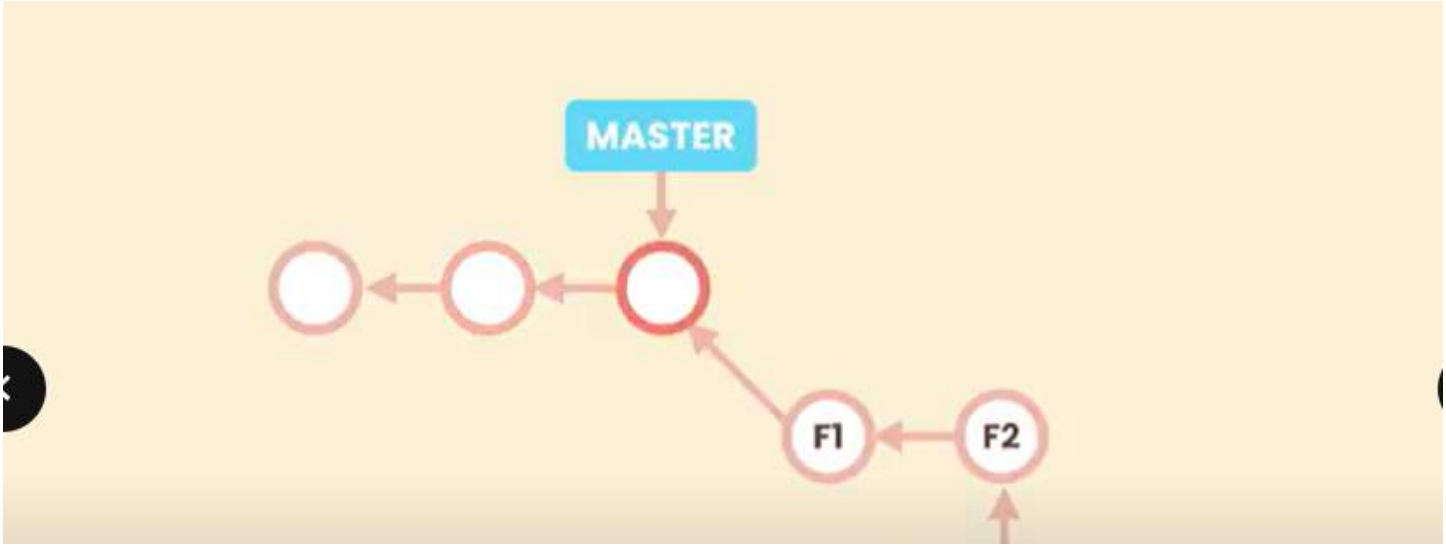
FEATURE



FEATURE



FEATURE

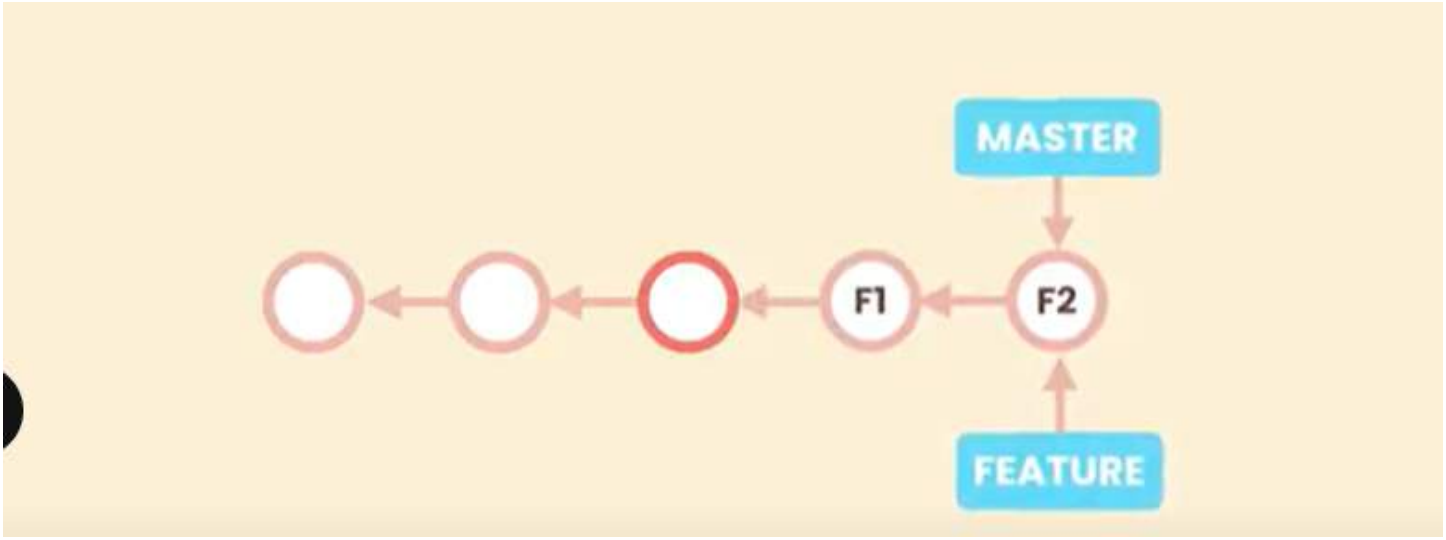


FEATURE

Now using the rebase command change the base of our feature branch so we can base it on the latest commit on Master.

Whats the point of this ? Well look at the shape of our history .

- We have a direct linear path from feature to master. So if u want to merge the feature branch into Master .We can do a fast forward merge and now we have a linear history,This is Called rebasing.

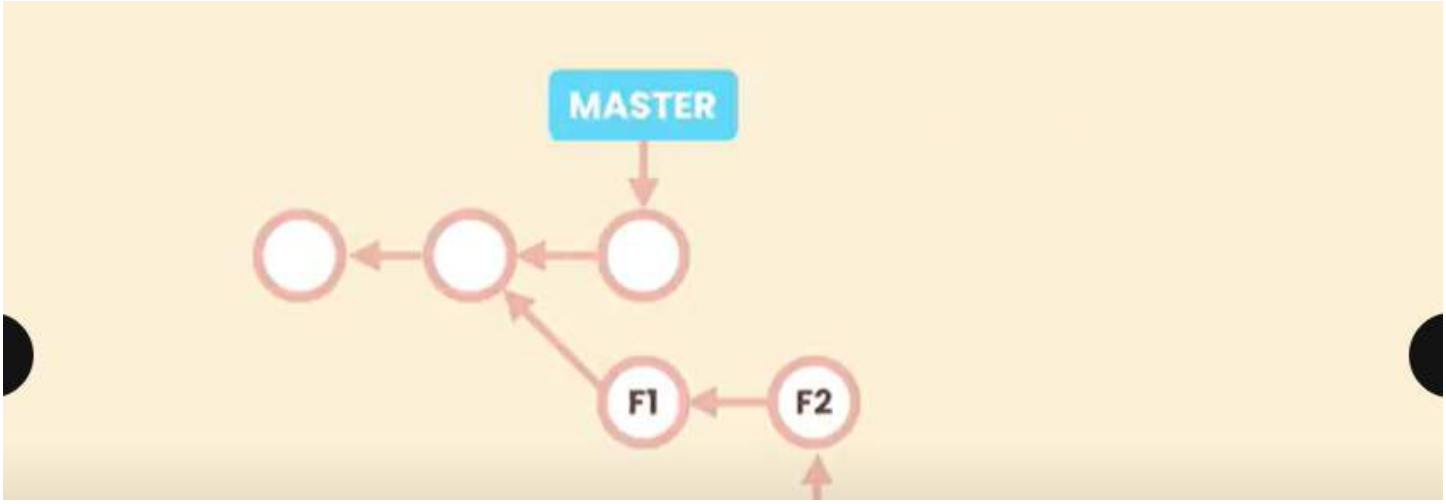


Its look like a good idea but be cautious with rebasing because rebasing REWRITE HISTORY.

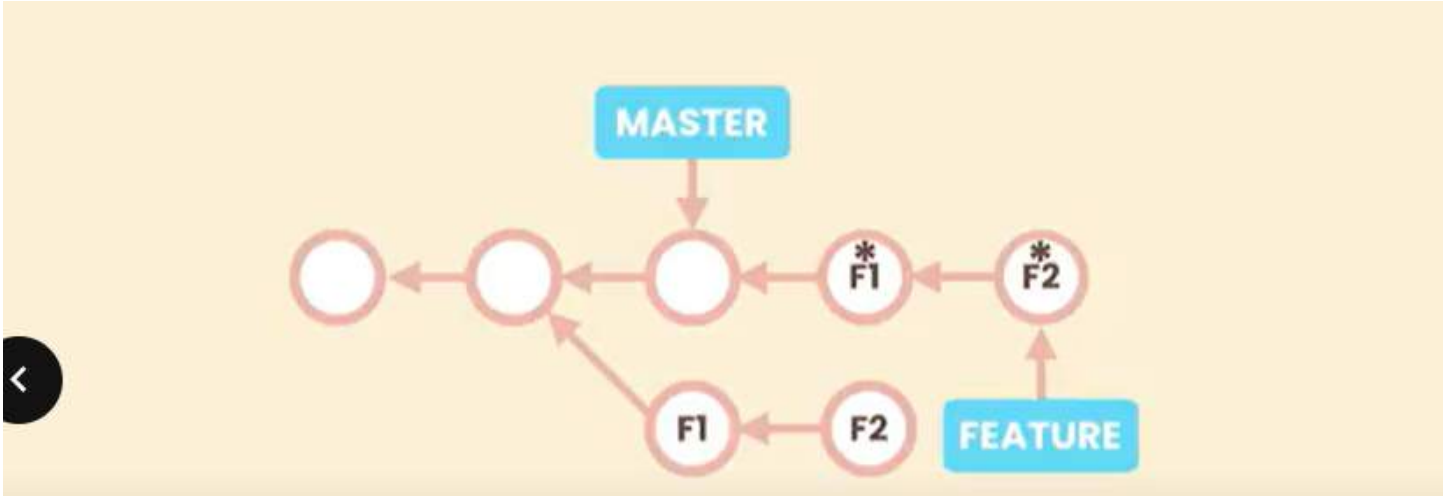
1. that means u should use rebasing only for branches or commits that are local in your repository.
2. If u shared this commit with other people in ur team if u have pushed or changes y shouldnt used rebasing.

Otherwise u r going to make a big mess.

Let me explain why REBASING reWRITE HISTORY this is the state before rebasing



FEATURE



when u r rebasing the FEATURE branch git is not going to change the base or parent F1 because commits and git are immutable they cannot be changed .so git is going to look at this commit is going to create a new commit thats look likes F1 & F2 we dont have any branches or anyother commit.

pointed to its So at some point in the future. Git is going to remove this commit.So we are essentially rewriting history. These two commit as before. So if we have shared F1 &F2 with other created a new commit on top of F2 Mow after rebasing there gistory is going to get screwed. Lets Create a new feature branch called Shopping Cart

```
git switch -C feature/shopping-cart
Switched to a new branch 'feature/shopping-cart'

~/Projects/Venus  git  feature/shopping-cart
echo hello > cart.txt

~/Projects/Venus  git  feature/shopping-cart
git add .

~/Projects/Venus  git  feature/shopping-cart
git commit -m "Add cart.txt"
[feature/shopping-cart 74c35bb] Add cart.txt
```

git log --oneline --graph

```
git log --oneline --all --graph
* 74c35bb (HEAD -> feature/shopping-cart) Add cart.txt
* b697ca8 (master) Fix the bug on the photo upload page.
* 3a44949 Revert "Merge branch 'bugfix/change-password' into master"
*   f634b2a Merge branch 'bugfix/change-password' into master
|\
| * 2df354d (bugfix/change-password) Update change password.
* | 7c5e304 Update change password.
|/
* fa289c 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
```

ur history. So our shopping
art branch is one commit ahead

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

~/Projects/Venus  git  master
echo hello >> toc.txt

~/Projects/Venus  git  master
git commit -am "Update toc.txt"
[master 77a0f2c] Update toc.txt
 1 file changed, 1 insertion(+)
```

```
* 77a0f2c (HEAD -> master) Update toc.txt
| * 74c35bb (feature/shopping-cart) Add cart.txt
|/
* b697ca8 Fix the bug on the photo upload page.
* 3a44949 Revert "Merge branch 'bugfix/change-password' 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
```

ore time. Look, our branches
ave diverged. This is the base

```
> ✔  git switch feature/shopping-cart
Switched to branch 'feature/shopping-cart'
```

```
> ✔  git rebase master
Successfully rebased and updated refs/heads/feature/shopping-cart.
```

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

git log --oneline --all --graph

```
* 901fe0d (HEAD -> feature/shopping-cart) Add cart.txt
* 77a0f2c (master) Update toc.txt
* b697ca8 Fix the bug on the photo upload page.
* 3a44949 Revert "Merge branch 'bugfix/change-password' into master"
*   f634b2a Merge branch 'bugfix/change-password' into master
|\
| * 2df354d (bugfix/change-password) Update change password.
* | 7c5e304 Update change password.
|/
fa289c 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
```

ast forward merge to bring the
aster pointer upward. So let's

now fast forward merge so

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

~/Projects/Venus  git  master
git merge feature/shopping-cart
Updating 77a0f2c..901fe0d
Fast-forward
 cart.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 cart.txt
```

git log --oneline --all --graph

```
* 901fe0d (HEAD -> master, feature/shopping-cart) Add cart.txt
* 77a0f2c Update toc.txt
* b697ca8 Fix the bug on the photo upload page.
* 3a44949 Revert "Merge branch 'bugfix/change-password' into master"
*   f634b2a Merge branch 'bugfix/change-password' into master
|\
| * 2df354d (bugfix/change-password) Update change password.
* | 7c5e304 Update change password.
|/
* fa289c 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
```

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

IF CONFLICTS WHAT HAPPENED

```
echo ocean > toc.txt

~/Projects/Venus  git  master !
git commit -am "Update toc.txt"
[master add47d7] Update toc.txt
 1 file changed, 1 insertion(+), 6 deletions(-)
```

```
git switch feature/shopping-cart
Switched to branch 'feature/shopping-cart'

~/Projects/Venus  git  feature/shopping-cart
echo mountain > toc.txt

~/Projects/Venus  git  feature/shopping-cart
git commit -am "Write mountain to toc"
[feature/shopping-cart 5670ecc] Write mountain to toc
 1 file changed, 1 insertion(+), 6 deletions(-)
```

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

git log --oneline --all --graph

```
* 5670ecc (HEAD -> feature/shopping-cart) Write mountain to toc
| * add47d7 (master) Update toc.txt
|/
* 901fe0d Add cart.txt
* 77a0f2c Update toc.txt
* b697ca8 Fix the bug on the photo upload page.
* 3a44949 Revert "Merge branch 'bugfix/change-password' 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.
```

git log, look, our branches have diverged one more time, we have

```
git switch feature/shopping-cart
Already on 'feature/shopping-cart'

~/Projects/Venus  git  feature/shopping-cart
git rebase master
Auto-merging toc.txt
CONFLICT (content): Merge conflict in toc.txt
error: could not apply 5670ecc... Write mountain to toc
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git reb
```

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

git mergetool

```
1  git mergetool
Merging:
toc.txt

Normal merge conflict for 'toc.txt':
  {local}: modified file
  {remote}: modified file
```

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" to track)
```

A mergetool in this case pmerge create this file automatically as a backup (so look the content 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 dont want this file in our repo so we remove it

git clean -fd

for future to eliminate these automatically file for pmerge tool we used

git config --global mergetool.keepbackup false

collaboration

1. **Collaboration Workflow**
2. **Pushing,fetching and pulling**
3. **Pull Requests, issues and milestone**
4. **Contributing to open-source projects**