**11)Git Branching Understanding the concept of Creating Branches**

Master Branch is the main branch. (master branch)

We develop a branch when we are developing a new feature and multiple developers are working on it.

The below is the main branch the head is pointed at master branch

**Diagram

Description automatically generated with low confidence**

Now we did another commit

The below is the main branch the head is pointed at master branch after new commit

**Diagram

Description automatically generated with medium confidence**

To create a branch

git checkout -b loginfeature (Creates a new branch named loginfeature)

The head is pointing at loginfeature branch

**Chart

Description automatically generated with low confidence**

Now we did another commit

Below the head is pointed at loginfeature branch after new commit but master remained same

**Timeline

Description automatically generated**

Now we did another commit

Below the head is pointed at loginfeature branch after new commit

**A picture containing diagram

Description automatically generated**

To move back to master

git checkout master

Now the head is pointing at master branch

**A picture containing timeline

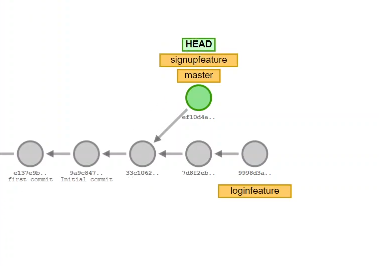
Description automatically generated**

Now we do a new commit to the master branch

**Timeline

Description automatically generated with medium confidence**

Now we create a new branch called signupfeature

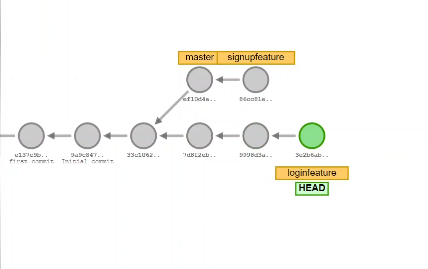
****

We do new commit to signup feature

To do another coomit to loginfeature

We do git checkout loginfeature head will move to loginfeature branch and do commit in that branch.

We can also create a branch by (git branch new\_branch)

**** **Diagram

Description automatically generatedDiagram

Description automatically generated with medium confidence**

At the end we can combine all branches together

**12)Practical Implementation of branching in the Project**

git add.

This adds all the files in the working area to staging area

git log –oneline

It returns the hash code of commit where the head is master branch or other branches and commit message in oneline (like its namesake)

When we do git log when we have multiple branches

The log gives us where the head is pointed at the point of various commits

git branch

It gives us the list of all branches in the git

**Summary**

Let’s say we made a commit, it will be in the master branch. 1

Now let’s create a child1 branch and do a commit(child) in child branch. 2

Now let’s shift back to master and do another commit(final). 3

git status 1 gives only initial commit

git status 2 gives initial and child commit

git status 3 gives initial and final commit

**13)Rename and Delete Branch using Git Branch**

In git folder in objects subfolder, we can see all the commits.

The branches are called references.

They are (the branches) stored in refs subfolders.

In the ref’s subfolder there is a heads subfolder, in it there will be files with names of all branches.

cat .git/refs/heads/master (It gives hash id for that branch)

ac041869c447690d552a84c1e6286e920428f3d6

cat .git/HEAD (It tells us in which branch the head is located)

When we type git branch it lists all branches and the branch the head is located in is in green color.

**Renaming Branch**

To rename a branch name

We must go that branch that needs to be renamed

git checkout to\_be\_renamed\_branch

then type

git branch -m new\_branch\_name (This renames the branch name)

**Deleting Branch**

To rename a branch name

We must go that branch that does not need to be deleted

git checkout master\_branch

then type

git branch -d new\_branch\_name (This deletes the branch name)

the above command works if we have not done any commits in the branch that needs to be deleted.

If we have done any commits the above command does not work and git says it can’t delete until the changes are merged to master branch. But if we still want to delete it we can do

git branch -D new\_branch\_name

**14)Merging Two Branches**

**Diagram

Description automatically generated**

It means head is in the master branch

Diagram

Description automatically generated

It means head is in the login\_feature branch

For merging we need two things

Target (master)

Source (login feature)

**Fast-Forward Merge**

To merge login feature to master

git merge login\_feature

Diagram

Description automatically generated

Diagram

Description automatically generated

This is a fast-forward merge.

This is called fast-forward because after branching we have not made any new commits to master branch.

**Recursive-Strat Merge**

But if we do any commits in master then it is called recursive strat merge.

The point at the recursive merge is a new commit.

A picture containing text, electronics

Description automatically generated

Now when we do git merge dummy

A picture containing text, electronics

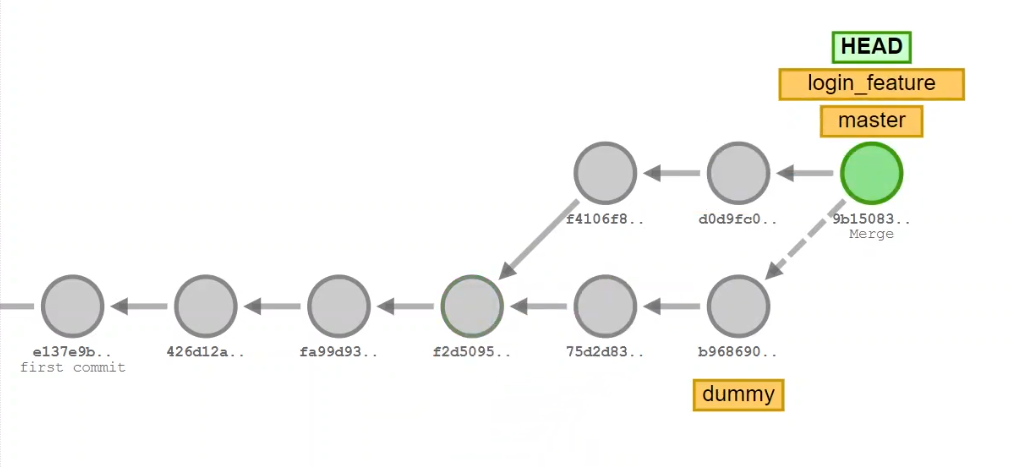
Description automatically generated

Now when we move to login feature



And merge again it will do a fast forward merge

git merge master



Text

Description automatically generated

**15)Resolve Conflicts while merging two branches**

Suppose there is a file we create a new branch and make a change it to it.

We move to master again and make some other change to the file,

Now when we try to merge these two branches how to merge them?

git merge new\_branch (It doesn’t work because there will be a conflict)

git status

It will give us the file where the conflict is happening

Now when we open that file by vi filename.txt

Created New File Index

<<<<<<<<HEAD

Created New File Index

========

Changes done in the feature branch

>>>>>>>> new\_branch

(If we want changes in master branch only)

Created New File Index

Created New File Index

git merge – abort(to abort merging) (To stop merging)

**16)Git Rebase**

* Rebase is one of the most powerful Git commands.
* Rebasing is often used as an alternative to merging.
* Rebasing a branch updates one branch with another by applying the commits of one branch on top of the commits of another branch.
* Git Rebase is used to clean up our local commit history
* Rebase is an advanced command which is used rarely.
* Merge preserve history
* Rebase doesn't preserve history

**Do not Use Rebase when**

* The branch is public when it is shared to all the developers.
* Most of the teams prefer to merge over rebase.

**Common places where we use rebases**

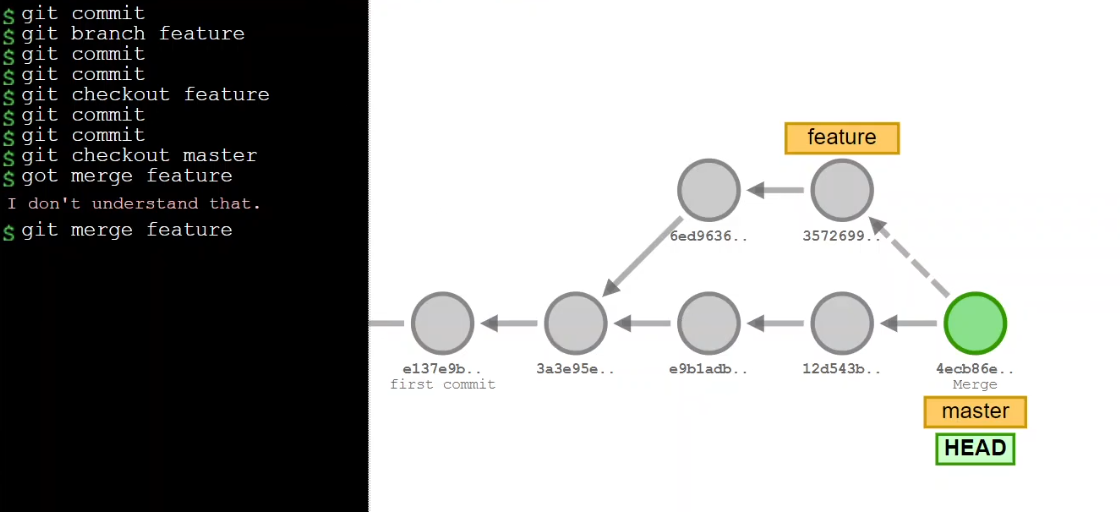
* Cleaning up your commits before sharing your branch.
* Pulling changes from another branch without merge

A picture containing text, device, black, meter

Description automatically generated

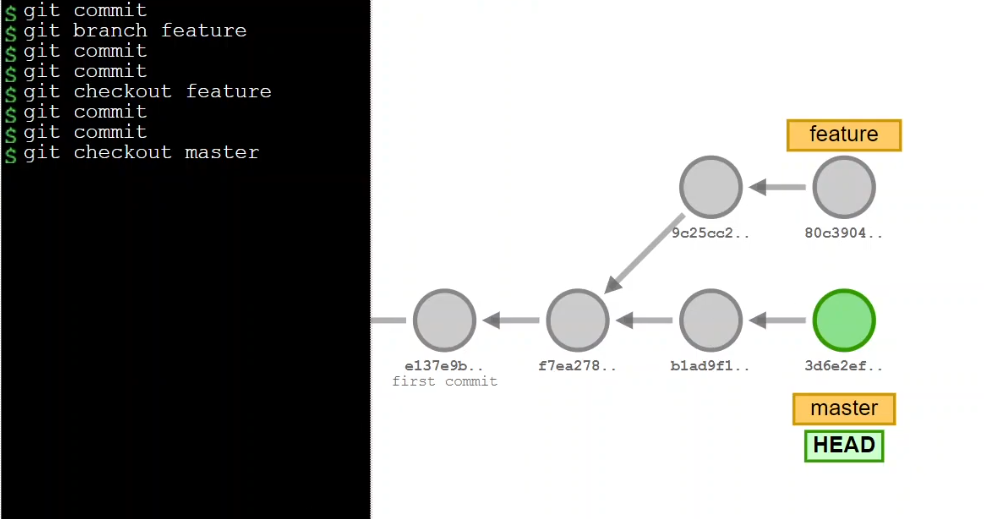
Git Merge mainatains history of the commits

Git Rebase doesn’t store the history in the above(right) we can see rebase does commits p4 and p5 then a4, a5, a6 onto them so all commits are in a6.



The above is merging

The below is rebase process my steps

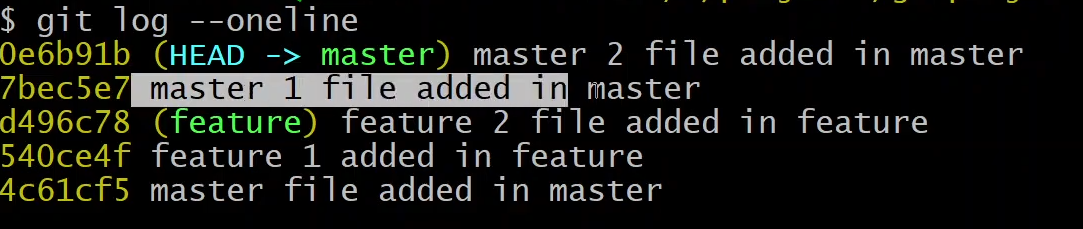


Diagram

Description automatically generated with medium confidence

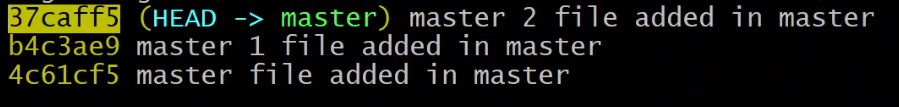
We can see that the last two commits of master are added on to the feature branch.

**17)Implementing Git Rebase**

****

In Rebase the master files are added on top of the feature files (they are added last).

We can also observe that the hash code of master file added 0e6b91b is different from



37caff5 the hash when master commit is initially done.

**Comparing to Merge**

**Text

Description automatically generated**

**Text

Description automatically generated**

But in merge we can observe that 2864dc6 THE Hash Code remains same.

**18)Interactive Rebasing**

Interactive rebasing is the process of combining multiple commits into a single commit.

Let’s say we commit a homepage in master.

Then in branch named feature we do three commits

Started working on new feature (a)

still working on the feature (b)

fix logic issues (c)

Completed New Feature (d)

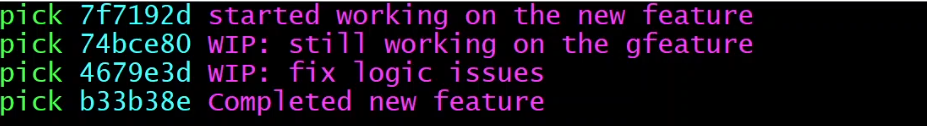
Text

Description automatically generated

We can combine a, b, c, d commits into master commit using interactive rebase

git rebase -i master

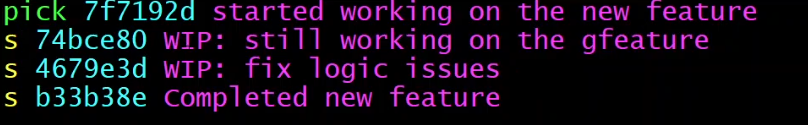
Now git will show all commits to be merged into master



Most recent on top

The below are the option available for interactive rebasing



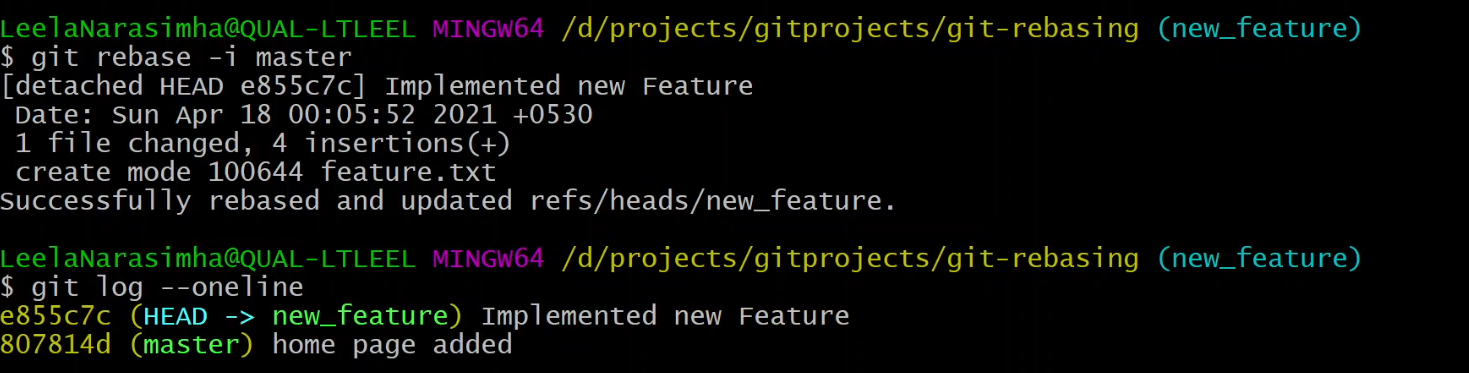


Now bottom 3 commits pushed into top commit

This rebase is to be done from a branch other than the branch that is being rebased so we move to master

git rebase -i master

It will ask us to give a new commit message (We can also remove previous commit messages)



Now we can see that only commit is there that is implemented the new feature and its hash code is different to the 4 earlier commits.

**19)Modify or Change the Last or Latest Commit**

In the last commit we added some files

But later we found another file is to be added into that commit, so instead of doing another commit we can alter the previous commit using amend command.

git commit --amend

It will ask us the recent commit and ask us if we want to change the commit message(we can If we want to change)

Changing the Last Commit: git commit --amend

The git commit --amend command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. But, amending does not just alter the most recent commit, it replaces it entirely, meaning the amended commit will be a new entity with its own ref. To Git, it will look like a brand new commit, which is visualized with an asterisk (\*) in the diagram below. There are a few common scenarios for using git commit --amend. We'll cover usage examples in the following sections.

Change most recent Git commit message

git commit --amend

Let's say you just committed and you made a mistake in your commit log message. Running this command when there is nothing staged lets you edit the previous commit’s message without altering its snapshot.

Premature commits happen all the time in the course of your everyday development. It’s easy to forget to stage a file or to format your commit message the wrong way. The --amend flag is a convenient way to fix these minor mistakes.

git commit --amend -m "an updated commit message"

Adding the -m option allows you to pass in a new message from the command line without being prompted to open an editor.

Changing committed files

The following example demonstrates a common scenario in Git-based development. Let's say we've edited a few files that we would like to commit in a single snapshot, but then we forget to add one of the files the first time around. Fixing the error is simply a matter of staging the other file and committing with the --amend flag:

# Edit hello.py and main.py

git add hello.py

git commit

# Realize you forgot to add the changes from main.py

git add main.py

git commit --amend --no-edit

The --no-edit flag will allow you to make the amendment to your commit without changing its commit message. The resulting commit will replace the incomplete one, and it will look like we committed the changes to hello.py and main.py in a single snapshot.

**20)Git Cherry Pick**

Cherry Pick is used if you want to apply a particular commit from one branch into another branch.

Cherry pick is mainly used if you don't want to merge the whole branch and you want some of the commits.

Cherry pick is just like rebase it is, and it is an advanced concept and a powerful command.

Cherry-pick is a useful tool, but it is not always a good option.

It can cause duplicate commits.

Mainly, cherry pick is used for the bug fixes where you want to place that bugfix commit in all the version branches.

It is also used when we accidentally made a commit in the wrong branch.

A picture containing chart

Description automatically generated

The red color commit in branch is copied to master branch in master branch in a different hash.

Text

Description automatically generated

We want the 6a6628c commit from the new branch to master branch.

So first we shift to master branch

git checkout master

git cherry-pick 6a6628c

By doing the above commit the commit from the branch is duplicated onto the master branch.