**SDET start**

**What is GIT**

GIT is version control system. It is distributed version control system.

To track the changes

Maintain different version

Code sharing for review

**SDET end**

First create a repo (for ease with the same name as your folder) in the github

Go to your folder --> right click 🡪bash here

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/Santi2020/testrepo.git

git push -u origin main

git remote add origin https://github.com/Santi2020/testrepo.git

git branch -M main

git push -u origin main

**To change the remote origin**

TISHAN1KA@DESKTOP-1QBSVOI MINGW64 ~/eclipse-workspace/Safeway (master)

$ git remote --verbose

origin https://github.com/Santi2020/Safeway.git (fetch)

origin https://github.com/Santi2020/Safeway.git (push)

**SDET New start here**

**What is Git?**

It is a version control tool. Helps to track any changes in the code. It allows all the developers to have access to all the code.

Git is a DevOps tool used for source code management. It is a free and open-source version control system used to handle small to very large projects efficiently. Git is used to tracking changes in the source code, enabling multiple developers to work together on non-linear development.

Git is a distributed version control system.

Using git we can

* Maintain version
* Track changes
* Share between the team

**Git architecture**

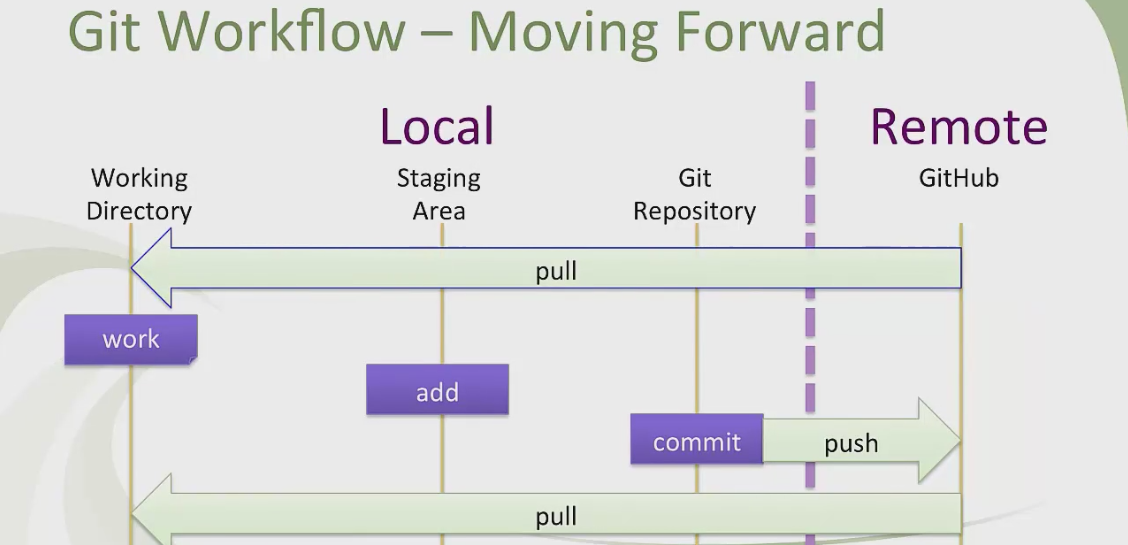
Working dir 🡺workspace 🡺Untracked files

Staging 🡺Tracked files

Git : Local repository 🡺 committed files

GITHub : Remote repository 🡺Remote files

Working dir 🡺 Staging 🡺 Local repository {Git) 🡺 Remote repository / GitHub



**Installing Git**

Google for Git download

Download the Git software

Go the downloaded folder 🡺 right click 🡺run as administrator 🡺 follow the instruction by clicking next

To verify 🡺Go to search🡺 Git

To check the version 🡺On command prompt 🡺 git -- version

**To Launch Git**

Go to your project folder🡺 Right click 🡺 Git Bash Here

Dir : to view your file (this is your project files in local system)

GIT commands

Configuration commands that is applicable for all repositories( all projects)

* **git config --global user.email “myemail@gmail.com” :** one time command
* **git config --global user.name "your\_name" :** one time command required to specify your name

**To query**

* git config –list : to list all the configrations
* git config user.name : to see the name in config
* git config user.email : to see the email in config
* **git init** : To initialze the repository / creates empty repository
* **git status** : Get the status
* **git add -A** : adds all files to the staging area
* **git add** . : adds all files to the staging area
* **git add pom.xml**: adds specified files to the repository
* **git commit -m "this is first commit"**: To commit the changes to the repository
* **git commit -a -m “adding modified files” :** Adds and commit in single command
* **git log:** Shows commit history **,** displays the details of each commit ex: unique commit code, name, email, files, comment
* **git diff :** is used to compare the difference between the files

compare between

1. working dir and staging
2. staging and local repo
3. local repo and remote repo

Req 1: To see the diff in file content bet working dir and staging area ( comparing before commit)

|  |
| --- |
| TISHAN1KA@DESKTOP-1QBSVOI MINGW64 /c/santoshi/Sample\_Git\_testing (master)  $ git diff index.txt  warning: LF will be replaced by CRLF in index.txt.  The file will have its original line endings in your working directory  **diff --git a/index.txt b/index.txt**  **index 89e0dc8..b13e93d 100644**  **--- a/index.txt**  **+++ b/index.txt**  @@ -1,2 +1,2 @@  -nimals  +rdsnimals  a/index.txt : represents source (staging area)  b/index.txt : represents destination (working dir)  89e0dc8 : hash of the file content from source/staging  b13e93d :hash of the file content from destination/ workspace  **100644 : Git file mode**  **100 (1st 3 digit) represents the file type**  **644 (next 3 digits) represents the file permission (rw-r-r)**  **4 – r**  **2 - w**  1 - e  (group- owner)  **--- a/index.txt : source file is missing some lines**  **+++ b/index.txt : new lines added to destibation/working space**  @@ -1,2 +1,2 @@ : -1 means one line missing in staging    -nimals  +rdsnimals  Give the difference from both files  If any line prefixed with space means it is unchanged  If any line prefixed with + it is added in destination copy  If any line prefixed with – it is removed from destination copy |

\*Last Commit is referred as HEAD

git diff Head index.txt

**Req 2: To see the diff in file content bet working dir and last commit**

|  |
| --- |
| git diff Head index.txt |

Req 3: To see the diff in file content bet staged copy and last commit

|  |
| --- |
| git diff --staged head index.txt  or  git diff --cached head index.txt  git diff --staged head filename  or  git diff --cached head filename |

**Req 4: To see the diff in file content bet specific commit and working directory copy**

To view all the commits

**git log --oneline**

The above commands displays commit Ids and the comment

**git diff 59f9950 index.txt**

In the above **59f9950** is the commit id to be compared

**Req 5: To see the diff in file content bet specific commit and staging area copy**

git diff –staged **59f9950**

**Req 6: To see the diff in file content bet specific commits**

git diff 59f9950 496a213 index.txt

**Req 7: To see the diff in file content bet two branches (Ex master and test)**

git diff master test : Shows all the difference in master and test branch

**Req 8: To see the diff in file content bet local and remote repositories**

git diff master origin/master/url\_repository\_url

Create a new folder and create 3 files

|  |
| --- |
| git init : initialize |
| git add . : add all files to the stage |
| git status : |
| git commit -m “This is my 1st commit” |
| ls : display all file in working space |
| git ls-files : to diplay all files in staging |
| git rm file1.txt : removes file1.txt from working directory and staging |
| git rm -r . : removes all files from working directory and staging |
| git rm –cached file2.txt : Remove files from only staging |
| rm file3.txt : Remove files from working directory only |
| git checkout -- file1.txt : to undo the changes in working dir |
| git add file2.txt ; git commit -m "file2 added to the  repository" |
| git reset index.txt  Undo commits at repo level  git <mode> <commit id>  git reset --mixed 3e9df31 |
| git reset --soft 92f8c16 |
| git reset --hard 92f8c16 |
| git branch : to view available branches |
| git branch branchname : to create new branch |
| To switch from one branch to another  git checkout branchname |
| Creating and switching to a branch  git checkout -b branchname |
| git merge feature : to merge the file from feature(branch) |
| To view all the commits  git log –oneline |
| To view all the commits and graphs of the branch  git log –oneline --graph |
| git branch -d feature : delete a branch |
| git log --oneline branchname  git log –oneline master  git log –oneline feature |
| git rebase master |
|  |
|  |
|  |
| **GITHUB Commands** |
| git push origin main : pust from local repo to remote repo |
| git pull origin master : pull from remote to local |
|  |

Git table

|  |
| --- |
| **Fast forward Merge example**  In fast-forward merge changes are done only in the child branch. Master branch is untouched.  git init  touch a.txt b.txt  git add a.txt;  git commit -m “C1M”  git add b.txt;  git commit -m “C2M”  git log –oneline 🡪 gets all commits in one line  query how many branches  git branch  create a new branch  git branch feature  switch to feature branch  git checkout feature  create 2 files  touch x.txt y.txt  add and commit x.txt  git add x.txt ; git commit -m “C1Feature”  add and commit y.txt  git add y.txt ; git commit -m “C2Feature”  switch to master branch  git checkout master  To merge feature branch into master  git merge |

|  |
| --- |
| Three way merge  Create 2 files in master  Touch a.txt b.txt  Git add a.txt; git commit -m “C1M”  Git add b.txt; git commit -m “C2M”  Create feature branch  Git branch feature  Create 2 new files  Touch x.txt y.txt  Git add x.txt; git commit -m “C1F”  Git add y.txt; git commit -m “C2F”  Switch to master  Git checkout master  Create 1 new file  Touch c.txt  git merge feature  (if no conflict then it onens a |

|  |
| --- |
| Three way merge with confilict  After creating branch if master and feature branches modify the same file and try to merge, there there will be confict and an error occures. This conflict has to be resolved manually  Switch to master  Git checkout master  Merge  Git merge feature   * this will give merge conflict error   Cat file.txt  Make necessary changes, by adding and deleting lines  Save the file  git add file.txt; git commit -m “MC”  now successfully merged  git log –oneline  git log –oneline –graph  After merging successfully you may delete the feature branch  git branch -d feature |

|  |
| --- |
| Rebase (internall uses fast forward commit)  Create a.txt b.txt  Add a.txt commit  Add b.txt commit  Create branch feature  Git checkout feature  Creat x.txt, y.txtx  Add x.txt commit  Add y.txt commit  Git checkout master  Create c.txt  Add c.txt commit  For rebasing switch to child branch  git checkout feature  git rebase master  switch to master  git checkout master  next merge  git merge feature  git log –oneline --graph |

|  |
| --- |
| **Clone the repo /push into the repo**  Create a folder where you want to clone the project  Go to git bash  git clone “https://github.com/Santi2020/testrepo.git”  touch file2  vi file1  if you add new files or make changes  git add .  git commit -m “updates file 1 and created file 2”  git log –oneline  git push origin main |

|  |
| --- |
| Pull  Go to the folder  Git bash  git pull origin main |

|  |
| --- |
| UI options in Eclipse and Intellij  CLONE a project  Right click on project explorer 🡪 import🡪projects from git  🡪next 🡪 clone URL🡪 provide remote repo URL 🡪next🡪 select the branch 🡪 select directory 🡪 import as general project 🡪provide a project name 🡪 finish  PUSH the changes  Right click on the file 🡪 Team 🡪 Add to index 🡪(file added to staging)  Right click on the file 🡪 Team 🡪 commit 🡪 which open a window to provide commit message 🡪 click commit (for local repo)   * commit and push (for remote repo) 🡪 provide username and password credentials   PULL the changes  Right click on the file 🡪 Team 🡪Pull |
|  |

**Remove command**

Req 1 : Remove files from both staging and working directory

git rm file1.txt : removes files from working directory and staging

Req 2 : Remove files from only staging

git rm –cached file2.txt

Req 2 : Remove files from working directory only

rm file3.txt

**Checkout command**

We can use checkout command to discard unstaged changes in the tracked files or the working directory

Meaning

Only for working directory

To discard un-staged changes(changes not added to the staging area)

In the tracked files (the files which are already added to the staging area/commit)

git checkout -- file1.txt

**Reset command**

This is opposite of git add command.

To remove changes from staging area :

* To undo commits at repository level : Moves the HEAD to the specified commitand all the remaining recent commits are removed
* Modes will decide wheather these changes are going to remove from staging area and working directory or not

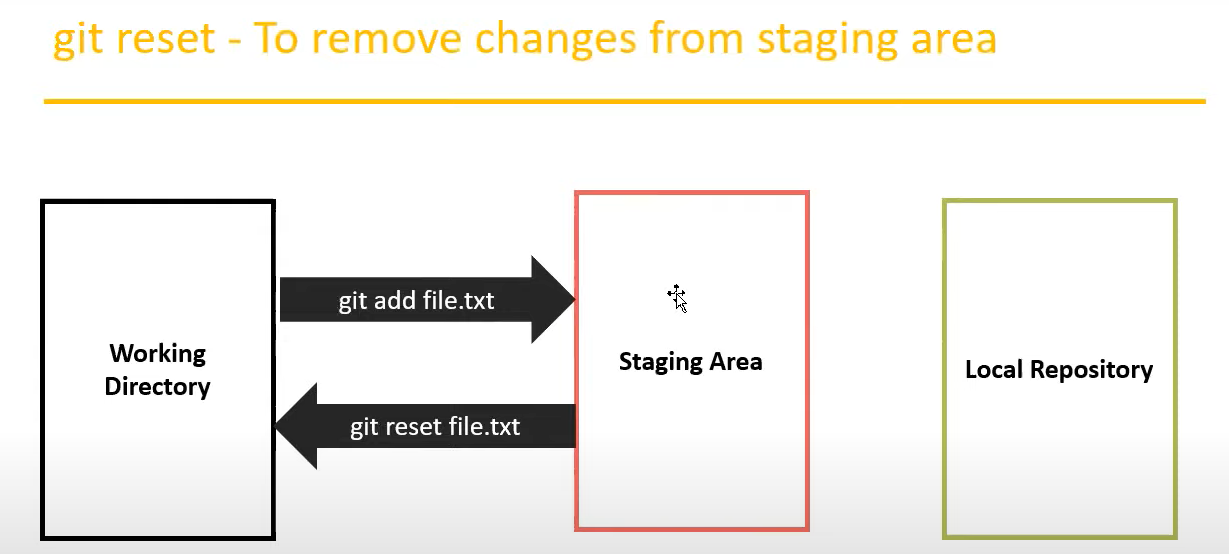
git reset <mode> <commit mode>

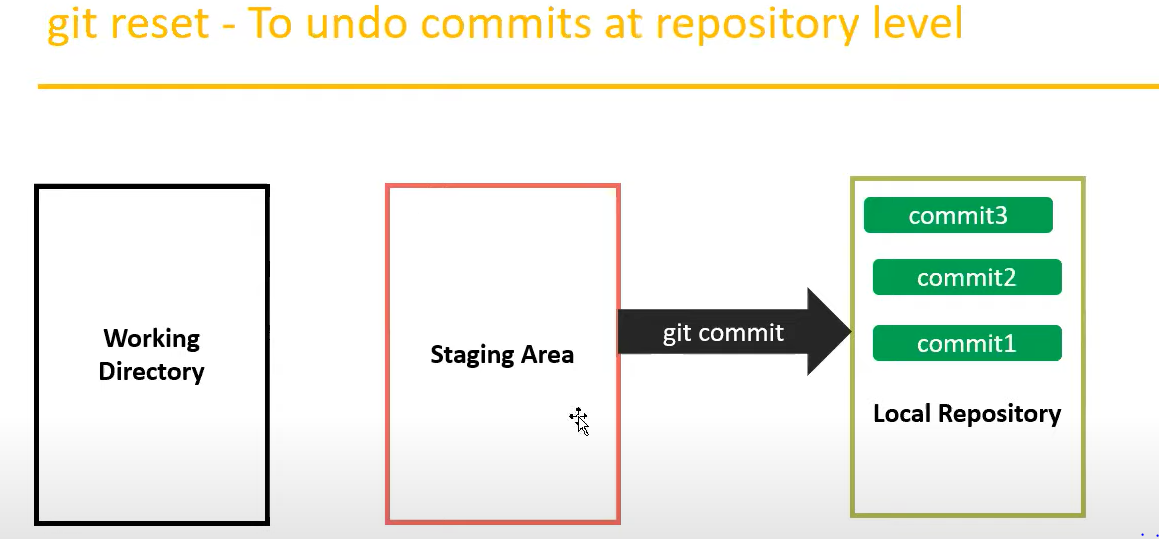
Modes

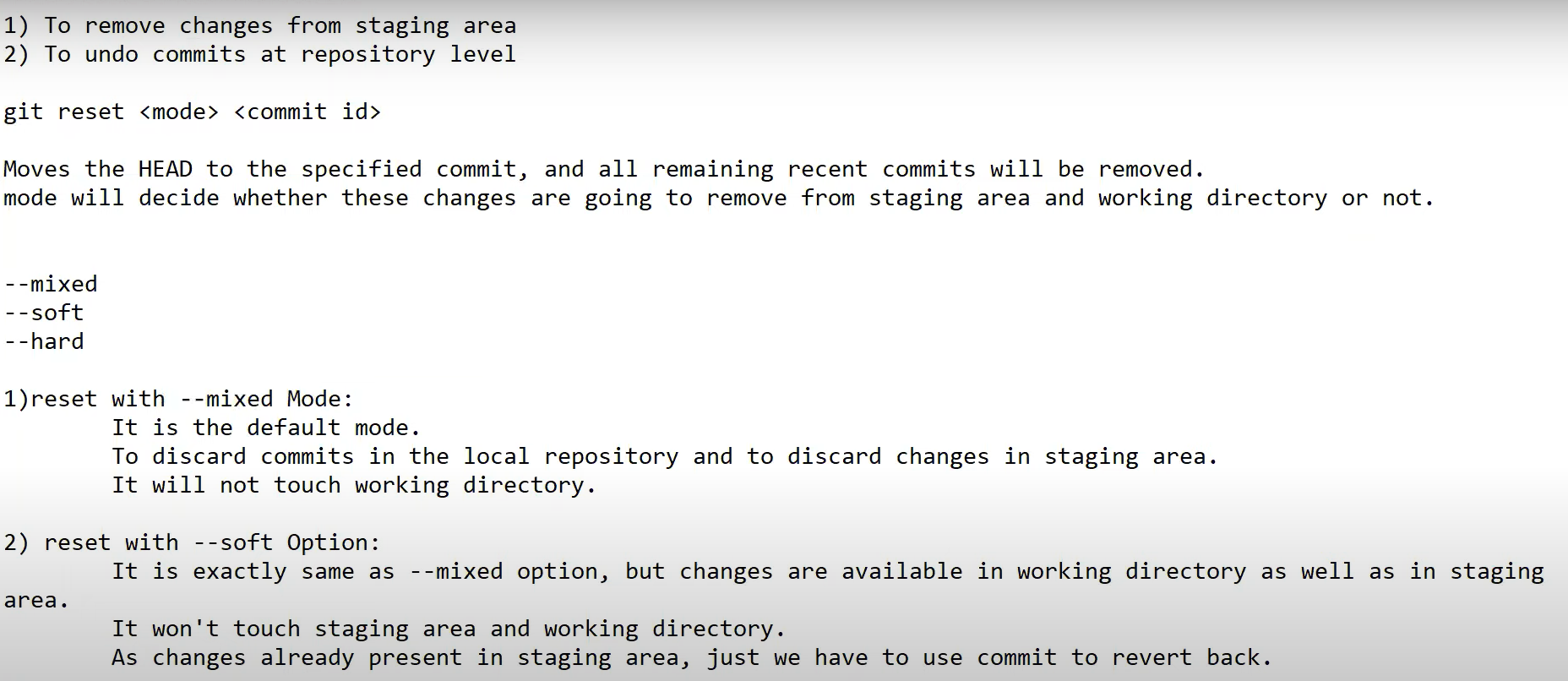
--mixed : git reset --mixed 3e9df31

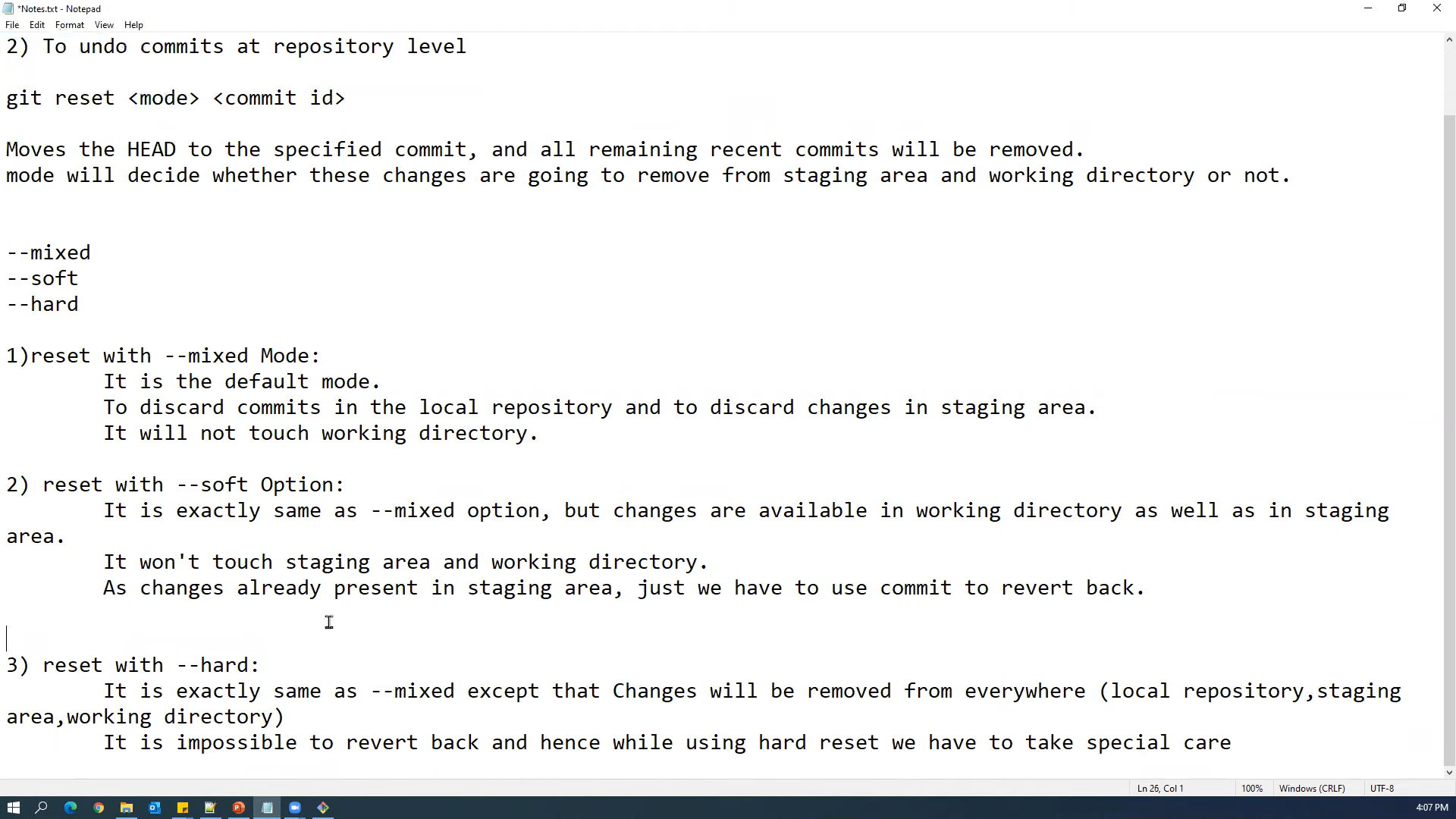
--soft : git reset --soft 92f8c16

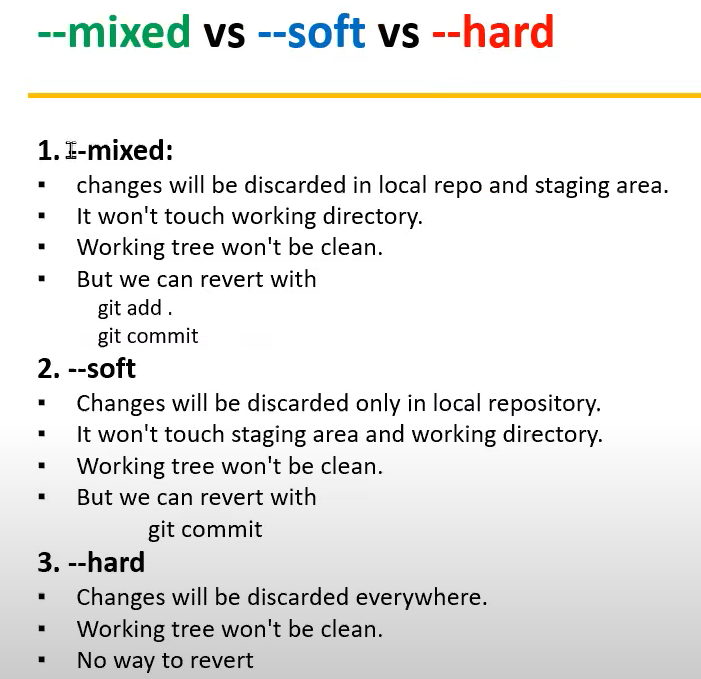
--hard : git reset --hard 92f8c16

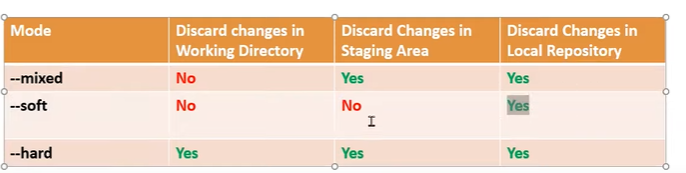












**Branch commands**

**Git branch : to view available branch**

git branch branchname : to create new branch and inherit all

**Switch from one branch to another**

Git checkout branchname

**To delete a branch**

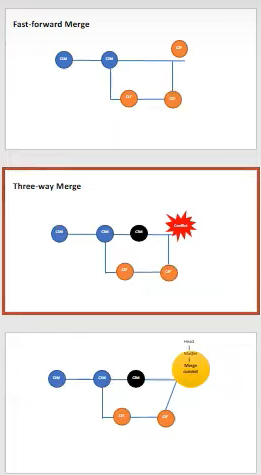
git branch -d feature

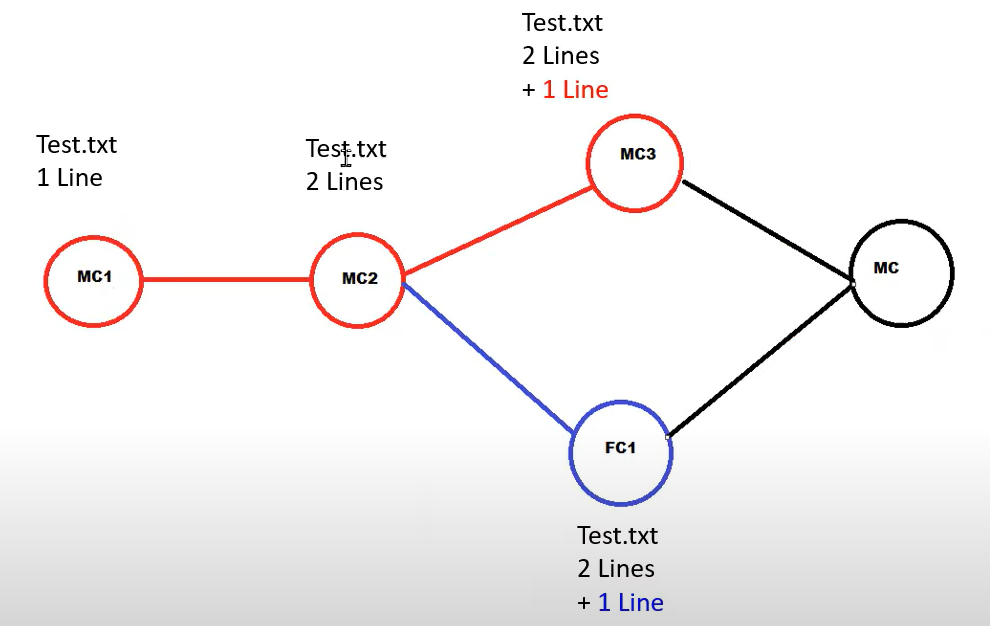
**Merge commands**

* **Fast forward merge**

git merge feature : to merge the file from feature

* **Three way merge**





**Git Rebase commands**

Rebase merges the file by creating new commits

git rebase master : to merge the file from feature into master

steps

TISHAN1KA@DESKTOP-1QBSVOI MINGW64 /c/santoshi/Sample\_Git\_testing/sampleproject9 (master)

$ git init

$ touch a.txt b.txt

$ git add a.txt; git commit -m "c1m"

$ git add b.txt; git commit -m "c2m"

$ git log --oneline

$ git branch feature

$ git checkout feature

$ git branch

$ touch x.txt y.txt

$ git add x.txt; git commit -m "c1f"

$ git add y.txt; git commit -m "c2f"

$ git log --oneline

$ git checkout master

$ touch c.txt

$ git add c.txt; git commit -m "c3m"

$ git log --oneline master

$ git log --oneline feature

$ git checkout feature

$ git rebase master

$ git log --oneline

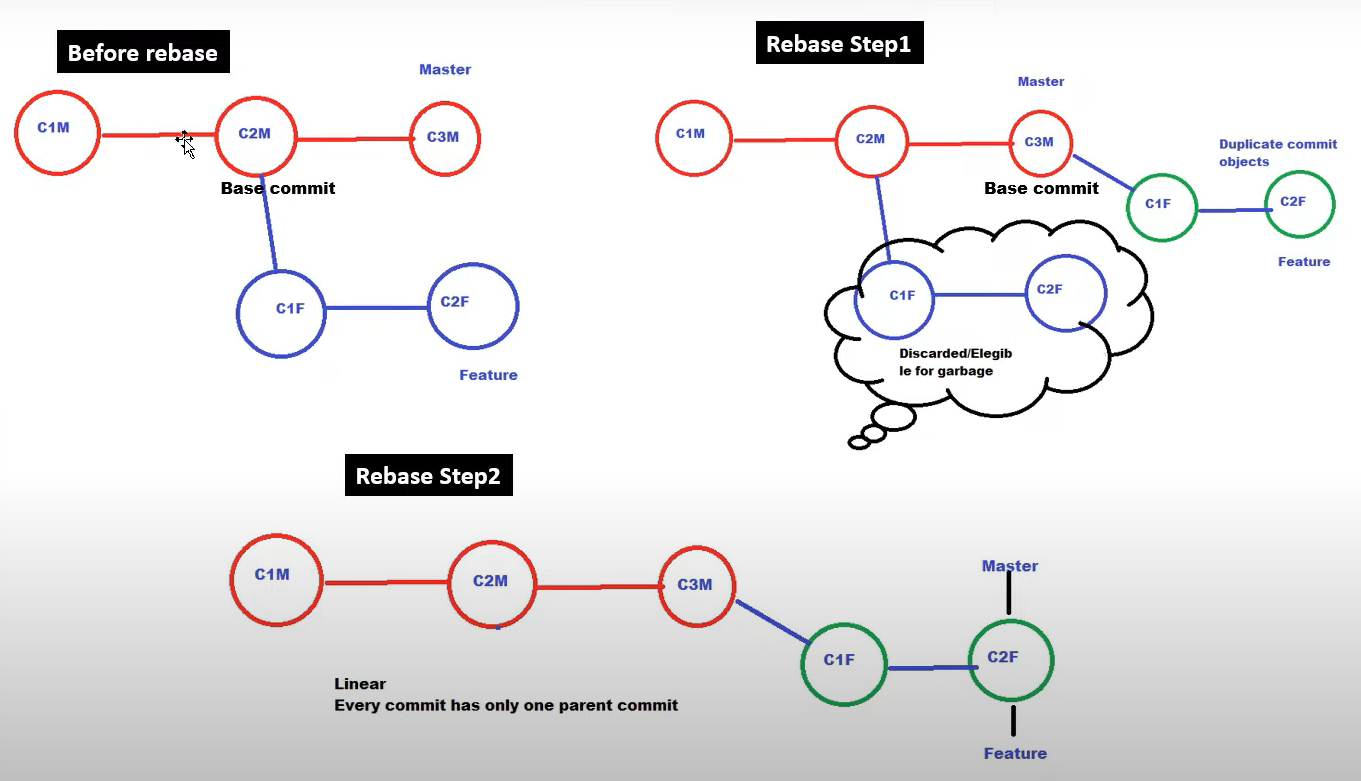
$ git checkout master

$ git log --oneline

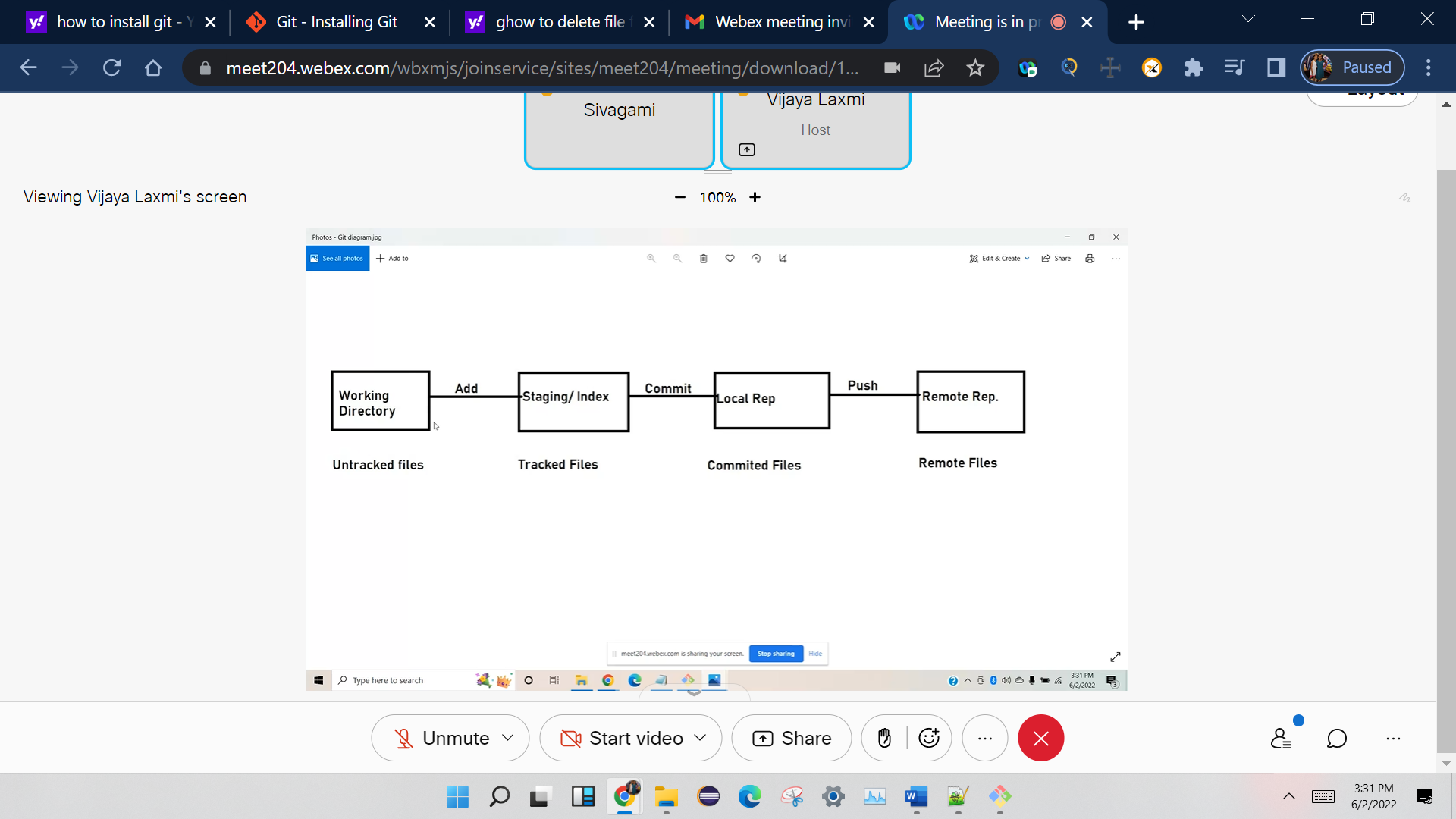
$ git merge feature

$ git log --oneline -graph

$ git log --oneline --graph



**^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^**



Unix commands

Vim filename

Esc :wq 🡺 save and close the file

**Whenever any file is added or modified**

git add a.txt b.txt

git commit -m comment

**or**

**To perform add and commit in single command**

git commit -a -m "both files are modified"

**I delete example2.xml file**

git status

git add **example2**.xml

git restore --staged **example2**.xml

git commit -a

After add new files to the repos 🡺 when git commit is executed 🡺 it will ask for comment and opens file.

To view all the commits

git log --oneline

* **git init** : To initialze the repository

o/p: Initialized empty Git repository in C:/Users/TISHAN1KA/eclipse-workspace/CucumberFramework/.git/

* **git status** : Get the status

output :

On branch master

No commits yet

Untracked files:

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

.classpath

.gitignore

.project

.settings/

old\_pom.xml

pom.xml

src/

vijaya.xml

TISHAN1KA@DESKTOP-1QBSVOI MINGW64 ~/eclipse-workspace/CucumberFramework (master)

**$ git log**

output

commit c4d7808acddb491885367da1c4ca9515732892e4 (HEAD -> master)

Author: santoshi <bandi.santoshi45.com>

Date: Thu Jun 2 09:57:14 2022 -0700

restoring the deleted file

commit 44354c208089d3ca559055cc5ac57b864cb59b48

Author: santoshi <bandi.santoshi45.com>

Date: Thu Jun 2 09:31:57 2022 -0700

this is first commit

git init

git add README.md

git commit -m "first commit"

git branch -M main

git remote add origin https://github.com/Santi2020/testrepo.git

git push -u origin main

git remote add origin https://github.com/Santi2020/testrepo.git

git branch -M main

git push -u origin main

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)

clone Clone a repository into a new directory

init Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)

add Add file contents to the index

mv Move or rename a file, a directory, or a symlink

restore Restore working tree files

rm Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)

bisect Use binary search to find the commit that introduced a bug

diff Show changes between commits, commit and working tree, etc

grep Print lines matching a pattern

log Show commit logs

show Show various types of objects

status Show the working tree status

grow, mark and tweak your common history

branch List, create, or delete branches

commit Record changes to the repository

merge Join two or more development histories together

rebase Reapply commits on top of another base tip

reset Reset current HEAD to the specified state

switch Switch branches

tag Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)

fetch Download objects and refs from another repository

pull Fetch from and integrate with another repository or a local branch

push Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some

concept guides. See 'git help <command>' or 'git help <concept>'

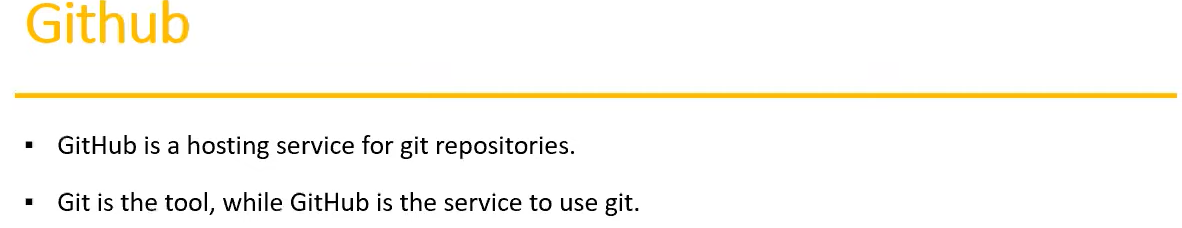
to read about a specific subcommand or concept.

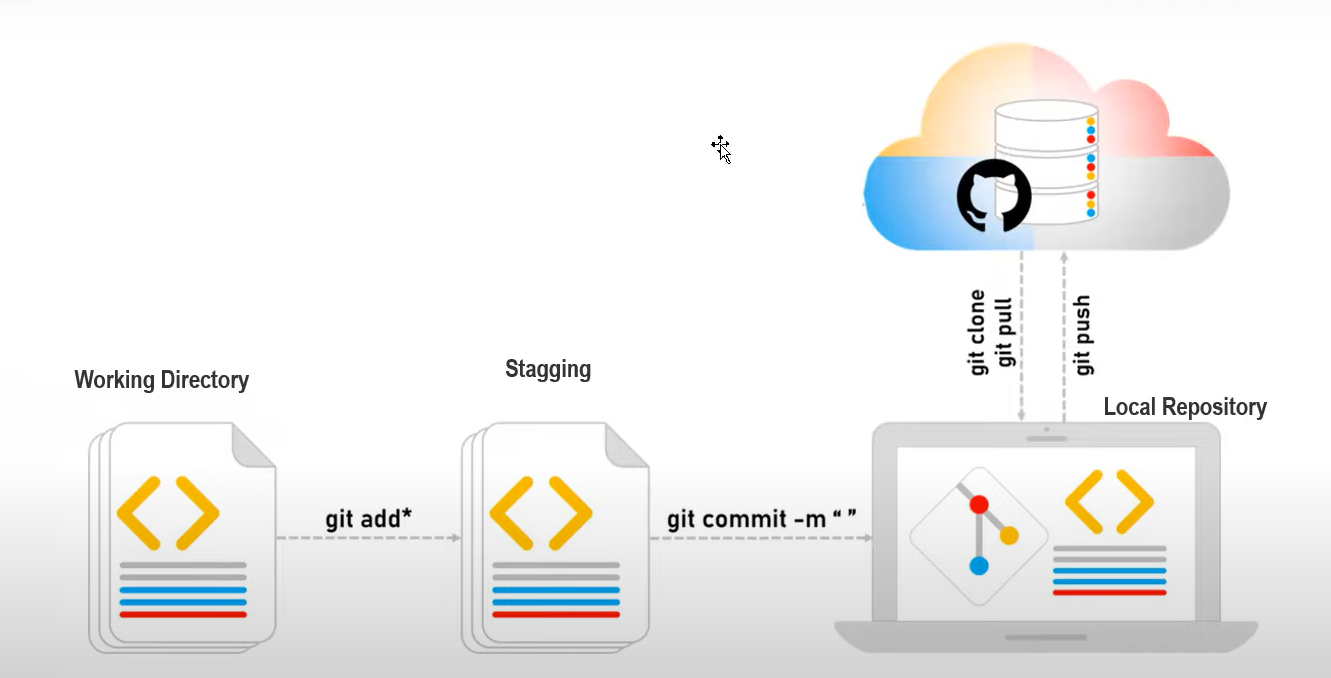
See 'git help git' for an overview of the system.

**GITHUB**

**Remote repository url**

[**https://github.com/Santi2020/testrepo.git**](https://github.com/Santi2020/testrepo.git)





git clone 'https://github.com/Santi2020/testrepo.git'

ls

cd testrepo

make changes to myfile

create another file

add both to staging

commit the files

**git push origin main**

**Git commands using GUI –Eclipse and intellij**

[**https://github.com/Santi2020/GUItest.git**](https://github.com/Santi2020/GUItest.git)

**Cloning steps for eclipse**

**Right in the project explorer -> import -> git -> projects from git =>clone url -> provide repo URL ->next ->next ->next -> import as general project ->next -> finish**

**Pushing steps from eclipse**

**To add file to staging**

**Right click the file -> team -> add to index**

**To commit**

**Right click the file -> team ->commit 🡪 show one more window. enter the comments 🡪 commit and push 🡪 provide your credentials**

**To pull**

**Right click the file -> team ->pull ->ok -> close**

**Steps for intellij**

**Launch intellij**

**Select get from VCS**

**Select**

**Provide url**

**Choose the directory**

**Clone**

**Make changes to the file**

**Right click the file -> git ->add :: File added to staging**

**Go to Git menu**

**Commit**

**Enter comment**

**Click commit**

**To Push**

**Select the file=> git menu ->push -> login via git**

**To Pull**

**Go to github make changes and commit**

**Select the file=> git menu ->pull -> login via git**

**Importing sivagamis project**

**Goto desired workspace**

**Click on open perspective**

**Select git->open**

**Click on clone**

**Provide URL=>Next🡺 finish**

**Next select the project 🡪right click🡪 import projects**

**Again click on open perspective🡪 select java(default)**

**Top right hand corner(open perspective) 🡺Search icon 🡪git**

**^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^**

**Guru99.com**

[**https://career.guru99.com/top-40-interview-questions-on-git/**](https://career.guru99.com/top-40-interview-questions-on-git/)

**1) What is GIT?**

GIT is a distributed version control system and source code management (SCM) system with an emphasis to handle small and large projects with speed and efficiency.

**2) What is a repository in GIT?**

A repository contains a directory named .git, where git keeps all of its metadata for the repository. The content of the .git directory are private to git.

**3) What is the command you can use to write a commit message**

The command that is used to write a commit message is “git commit –a”.  The –a on the command line instructs git to commit the new content of all tracked files that have been modified. You can use “git add<file>” before git commit –a if new files need to be committed for the first time.

**4) What is the difference between GIT and SVN?**

The difference between GIT and SVN is

a) Git is less preferred for handling extremely large files or frequently changing binary files while SVN can handle multiple projects stored in the same repository.

b) GIT does not support ‘commits’ across multiple branches or tags.  Subversion allows the creation of folders at any location in the repository layout.

c) Gits are unchangeable, while Subversion allows committers to treat a tag as a branch and to create multiple revisions under a tag root.

**5) What are the advantages of using GIT?**

a) Data redundancy and replication

b) High availability

c) Only one.git directory per repository

d) Superior disk utilization and network performance

e) Collaboration friendly

f) Any sort of projects can use GIT

**6) What language is used in GIT?**

GIT is fast, and ‘C’ language makes this possible by reducing the overhead of runtimes associated with higher languages.

**7) What is the function of ‘GIT PUSH’ in GIT?**

‘GIT PUSH’ updates remote refs along with associated objects.

**8) Why GIT better than Subversion?**

GIT is an open source version control system; it will allow you to run ‘versions’ of a project, which show the changes that were made to the code overtime also it allows you keep the backtrack if necessary and undo those changes.  Multiple developers can checkout, and upload changes and each change can then be attributed to a specific developer.

**9) What is “Staging Area” or “Index” in GIT?**

Before completing the commits, it can be formatted and reviewed in an intermediate area known as ‘Staging Area’ or ‘Index’.

**10) What is GIT stash?**

GIT stash takes the current state of the working directory and index and puts in on the stack for later and gives you back a clean working directory.  So in case if you are in the middle of something and need to jump over to the other job, and at the same time you don’t want to lose your current edits then you can use GIT stash.

**11) What is GIT stash drop?**

When you are done with the stashed item or want to remove it from the list, run the git ‘stash drop’ command.  It will remove the last added stash item by default, and it can also remove a specific item if you include as an argument.

**12) How will you know in GIT if a branch has been already merged into master?**

Git branch --merged lists the branches that have been merged into the current branch

Git branch—-no merged lists the branches that have not been merged

**13) What is the function of git clone?**

The git clone command creates a copy of an existing Git repository.  To get the copy of a central repository, ‘cloning’  is the most common way used by programmers.

**14) What is the function of ‘git config’?**

The ‘git config’ command is a convenient way to set configuration options for your Git installation.  Behaviour of a repository, user info, preferences etc. can be defined through this command.

**15) What does commit object contain?**

a)      A set of files, representing the state of a project at a given point of time

b)      Reference to parent commit objects

c)       An SHAI name, a 40 character string that uniquely identifies the commit object.

**16) How can you create a repository in Git?**

In Git, to create a repository, create a directory for the project if it does not exist, and then run command “git init”. By running this command .git directory will be created in the project directory, the directory does not need to be empty.

**17) What is ‘head’ in git and how many heads can be created in a repository?**

A ‘head’ is simply a reference to a commit object. In every repository, there is a default head referred as “Master”.  A repository can contain any number of heads.

**18)   What is the purpose of branching in GIT?**

The purpose of branching in GIT is that you can create your own branch and jump between those branches. It will allow you to go to your previous work keeping your recent work intact.

**19) What is the common branching pattern in GIT?**

The common way of creating branch in GIT is to maintain one as “Main“

branch and create another branch to implement new features. This pattern is particularly useful when there are multiple developers working on a single project.

**20) How can you bring a new feature in the main branch?**

To bring a new feature in the main branch, you can use a command “git merge” or “git pull command”.

**21) What is a ‘conflict’ in git?**

A ‘conflict’ arises when the commit that has to be merged has some change in one place, and the current commit also has a change at the same place. Git will not be able to predict which change should take precedence.

**22) How can conflict in git resolved?**

To resolve the conflict in git, edit the files to fix the conflicting changes and then add the resolved files by running “git add” after that to commit the repaired merge,  run “git commit”.  Git remembers that you are in the middle of a merger, so it sets the parents of the commit correctly.

**23) To delete a branch what is the command that is used?**

Once your development branch is merged into the main branch, you don’t need

development branch.  To delete a branch use, the command “git branch –d [head]”.

**24) What is another option for merging in git?**

“Rebasing” is an alternative to merging in git.

**25) What is the syntax for “Rebasing” in Git?**

The syntax used for rebase is “git rebase [new-commit] “

**26) What is the difference between ‘git remote’ and ‘git clone’?**

‘git remote add’  just creates an entry in your git config that specifies a name for a particular URL.  While, ‘git clone’ creates a new git repository by copying and existing one located at the URI.

**27) What is GIT version control?**

With the help of GIT version control, you can track the history of a collection of files and includes the functionality to revert the collection of files to another version.  Each version captures a snapshot of the file system at a certain point of time. A collection of files and their complete history are stored in a repository.

**28) Mention some of the best graphical GIT client for LINUX?**

Some of the best GIT client for LINUX is

a) Git Cola

b) Git-g

c) Smart git

d) Giggle

e) Git GUI

f) qGit

**29) What is Subgit? Why to use Subgit?**

‘Subgit’ is a tool for a smooth, stress-free SVN to Git migration.  Subgit is a solution for a company -wide migration from SVN to Git that is:

a) It is much better than git-svn

b) No requirement to change the infrastructure that is already placed

c) Allows to use all git and all sub-version features

d) Provides genuine stress –free migration experience.

**30) What is the function of ‘git diff ’ in git?**

‘git diff ’ shows the changes between commits, commit and working tree etc.

**31) What is ‘git status’ is used for?**

As ‘Git Status’ shows you the difference between the working directory and the index, it is helpful in understanding a git more comprehensively.

**32) What is the difference between the ‘git diff ’and ‘git status’?**

‘git diff’ is similar to ‘git status’, but it shows the differences between various commits and also between the working directory and index.

**33) What is the function of ‘git checkout’ in git?**

A ‘git checkout’ command is used to update directories or specific files in your working tree with those from another branch without merging it in the whole branch.

**34) What is the function of ‘git rm’?**

To remove the file from the staging area and also off your disk ‘git rm’ is used.

**35) What is the function of ‘git stash apply’?**

When you want to continue working where you have left your work, ‘git stash apply’ command is used to bring back the saved changes onto the working directory.

**36) What is the use of ‘git log’?**

To find specific commits in your project history- by author, date, content or history ‘git log’ is used.

**37) What is ‘git add’ is used for?**

‘git add’ adds file changes in your existing directory to your index.

**38) What is the function of ‘git reset’?**

The function of ‘Git Reset’ is to reset your index as well as the working directory to the state of your last commit.

**39) What is git Is-tree?**

‘git Is-tree’ represents a tree object including the mode and the name of each item and the SHA-1 value of the blob or the tree.

**40) How git instaweb is used?**

‘Git Instaweb’ automatically directs a web browser and runs webserver with an interface into your local repository.

**41) What does ‘hooks’ consist of in git?**

This directory consists of Shell scripts which are activated after running the corresponding Git commands.  For example, git will try to execute the post-commit script after you run a commit.

**42) Explain what is commit message?**

Commit message is a feature of git which appears when you commit a change. Git provides you a text editor where you can enter the modifications made in commits.

**43) How can you fix a broken commit?**

To fix any broken commit, you will use the command “git commit—amend”. By running this command, you can fix the broken commit message in the editor.

**44) Why is it advisable to create an additional commit rather than amending an existing commit?**

There are couple of reason

a)  The amend operation will destroy the state that was previously saved in a commit.  If it’s just the commit message being changed then that’s not an issue.  But if the contents are being amended then chances of eliminating something important remains more.

b) Abusing “git commit- amend” can cause a small commit to grow and acquire unrelated changes.

**45) What is ‘bare repository’ in GIT?**

To co-ordinate with the distributed development and developers team, especially when you are working on a project from multiple computers ‘Bare Repository’ is used. A bare repository comprises of a version history of your code.

**46) Name a few Git repository hosting services**

* Pikacode
* Visual Studio Online
* GitHub
* GitEnterprise
* SourceForge.net

**^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^**

[Software Testing Help](https://www.softwaretestinghelp.com/)

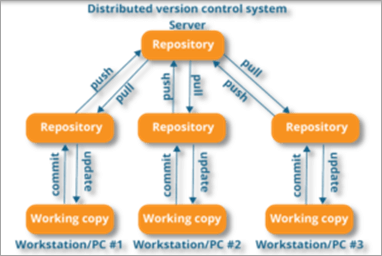
**Q #1) What is Git?**

**Answer:** Git is a Distributed Version Control tool. It is compatible with distributed non-linear workflows as it offers data assurance for building good quality software.

Git is free and open-source. It can be used for almost any kind of project, be it small or big in size. Git is known for its great speed and efficiency. Git repositories are very easy to find and access. Due to its certain features, Git is highly flexible, secure and compatible with your system.

**Q #2) What is a distributed Version Control System?**

**Answer:** A distributed VCS is a system that does not depend upon a central server to keep a project file and all its versions. In distributed VCS, each collaborator or developer gets a local copy of the main repository and this is called a clone.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/distributed-version-control-system.png)

[image[*source*](https://www.edureka.co/)]

As you can see in the above diagram, every collaborator maintains a local repository on their local machines. They can commit and update the local repositories without any issues.

Using a pull operation, a developer can update his local repository with the latest changes from the central server. Using the push operation, they can send their changes from the local repository to the central server.

**Q #3) Who created Git?**

**Answer:** Git was created by Linus Torvalds in 2005 on the road to develop Linux Kernel.

**Q #4) Which language is used in Git?**

**Answer:** C is the underlying programming language in which Git is written. C language makes Git fast by evading runtime overheads linked with other high-level programming languages.

**Q #5) What are the advantages/main features of Git?**

**Answer: Enlisted below are the various features of Git.**

**(i) Free & Open Source:**

Git is issued under GPL’s (General Public License) open source license. You need not pay anything to use Git.

It is absolutely free. As it is open-source, you can modify the source code according to your needs.

**(ii) Speed:**

As you are not required to connect to any network for executing all the actions, it performs all the tasks quickly. Obtaining version history from a locally stored repository can be one hundred times speedier than obtaining it from the remote server.

Git is written in C, which is the underlying programming language that evades runtime overheads linked with other high-level languages.

**(iii) Scalable:**

Git is highly scalable. So, if the number of collaborators increases in the coming time, then Git can easily accommodate this change.

Despite the fact that Git represents an entire repository, the data kept on the client’s side is very small as Git compacts the entire vast data through a lossless compression technique.

**(iv) Reliable:**

As every collaborator has its own local repository, on the instances of a system crash, the lost data can be recuperated from any of the local repositories. At all times, you will have a backup of all your files.

**(v) Secure:**

Git utilizes the SHA1 (Secure Hash Function) to name and identify objects inside its repository. Each artifact and commit are check-summed and recovered through its checksum during checkout.

The Git history is saved in a manner in which the ID of a specific version (a commit in terms of Git) relies on the total development history running up to that commit. Once a file version is pushed to Git, then there is no way to change it without being noticed.

**(vi) Economical:**

In the case of a centralized version control system, the central server must be strong enough to attend requests of the entire team. This is not a problem for smaller teams, however as the team expands, the hardware limitations of the server can be an impediment for performance.

In the case of distributed version control systems like Git, the team members don’t require interaction with the server expect when they are required to push or pull changes. All the heavy lifting occurs at the client end, thus the server hardware can be kept quite simple certainly.

**(vii) Supports Non-linear Development:**

Git provides rapid branching & merging and contains particular tools for envisaging and traversing a non-linear development history. A basic notion in Git is that a change will be merged more frequently than it is written as it is sent across different reviewers.

Git Branches are extremely lightweight. A branch in Git refers only to a single commit. The complete branch structure can be created, with the help of parent commits.

**(viii) Easy Branching:**

Branch management through Git is very straightforward and easy. It requires just a few jiffies to create, delete, and merge branches. Feature branches give an insulated environment to each change to your codebase.

When a developer requires to begin working on something, irrespective of the size of work, they create a new branch. This makes sure that the master branch constantly holds a production-quality code.

**(ix) Distributed Development:**

Git provides every developer a local copy of the whole development history, plus the changes get cloned from one such repository to another. These changes are introduced as added development branches and can be merged in the same manner as a locally developed branch.

**(x) Compatibility along with present Systems or Protocol:**

Repositories can be published through HTTP, FTP or a Git protocol on top of either a plain socket or ssh.

**Q #6) How do you create a Repository in Git?**

**Answer:** To create a repository, you need to create a directory for the project if it does not already exist, and then simply execute the command “**git init**”. By executing this command, a .git directory will be created inside the project directory i.e. now your project directory has turned into a Git repository.

**Q #7) What is a .git Directory?**

**Answer:** The moment you create a repository, you will find a .git directory present inside it. This .git directory contains all the metadata of the repository and maintains a track of all the changes made to the files in your repository, by keeping a commit history.

All the information regarding commits, hooks, refs, object databases, remote repository addresses, etc. are kept inside this folder. This is the most crucial part of Git. When you clone any Git repository on your local machine, this .git is the directory that actually gets copied.

**Q #8) What happens if the .git directory gets deleted?**

**Answer:** If the .git/ directory gets deleted, then you will lose track of your project’s history. The repository will no longer be under version control.

**Q #9) Which command is used for writing a Commit Message in Git?**

**Answer:** The command used for passing on a message to a git commit is git commit -m “commit message”. The flag m is used to pass a commit message.

**Q #10) What is the bare Git repository? How is it different from a standard/non-bare Git repository?**

**Answer:** Repositories that are created through git init command are the standard/non-bare Git repositories.

In the top-level folder of such repository, you will find two things:

1. A .git subdirectory keeping all metadata and track of the history of your repo.
2. A working tree.

The repositories which are created using git init –bare command are known as bare Git repositories. They are mainly used for sharing. They do not contain any working tree. They keep the git revision history of your repository in the root folder rather than having it inside the .git subfolder.

It just contains bare repository data. This is how a bare Git repository is different from a standard Git repository. Also, a bare repository does not have a default remote origin repository as it serves as an origin repository for multiple remote users.

Since a bare repository does not contain any workspace, the git push and git pull commands do not work over a bare repo. You are not required to commit any changes to a bare repo.

**Q #11) Mention some Git Repository Hosting Services.**

**Answer:**

* Github
* Pikacode
* Gitlab
* Microsoft VSTS
* BitBucket
* GitEnterprise
* SourceForge
* LaunchPad
* Perforce
* Beanstalk
* Assembla

**Q #12) Name some Basic Operations in Git.**

**Answer: Some basic operation in Git include:**

* Initialize
* Add
* Commit
* Push
* Pull

**Q #13) Name some Advanced Operations in Git.**

**Answer: Some advanced operations in Git are:**

* Branching
* Merging
* Rebasing

**Q #14) How will you distinguish between Git and SVN?**

**Answer:**Git is a distributed version control whereas SVN is centralized. This leads to many differences between the two in terms of their features and functionalities.

|  | **Git** | **SVN** |
| --- | --- | --- |
| **Server Architecture** | The computer on which your Git has installed acts as both client and server. Each developer has a local copy of the complete version history of the project on their individual computers. Git changes occur locally.  Hence, the developer is not required to be connected to the network at all times. Only for push and pull operations, developers would need internet connection to connect to remote server. | SVN has a separate client and server. It is not locally available. You will be required to be connected to the network to perform any action.  Also, in SVN, since everything is centralized, so in case the central server gets crashed or corrupted, it will result in entire data loss for the project. |
| **Branching** | Git is mostly preferred by developers due to its effective branching model. Git branches are lightweight but powerful.  They are only references to a particular commit. You can create, delete or modify a branch anytime with no impact on other commits. So, fork, branch and merge are easy with Git. | SVN has a complicated branching and merging model and its time-consuming to manage.  In SVN, branches are generated as directories within the repository. This directory structure is mainly problematic. When the branch is ready, you need to commit back to the trunk. Since you are not the only one who is merging the changes, so the version of the truck may not be regarded as developers’ branches. This can lead to conflicts, missing files and jumbled changes in your branch. |
| **Access Control** | Git presumes that all the contributors will be having the same permissions. | SVN permits you to define read/write access controls at each and directory level. |
| **Auditability** | In Git, the changes are tracked at the repository level. Git does not bother too much about maintaining the precise history of changes made in your repository. The distributed nature of Git lets any collaborator change any part of their local repo’s history. With Git, it’s difficult to figure a true history of changes in your codebase.  For example, you will lose history after rename in Git. | In SVN, the changes are tracked at the file level.  SVN maintains a pretty consistent and precise change history. You can recover exactly the same data as it was at any instant in the past.  SVN history is permanent and always definite. |
| **Storage Requirements** | Git and SVN store the data in the same manner. The disk space usage is equal for both of them. The only difference comes into picture in case of binary files. Git is not friendly to binary files.  It can’t handle the storage of large binary files. | SVN has an xDelta compression algorithm that works for both binary and text files.  So, SVN can handle storing large binary files in comparatively lesser space than Git. |
| **Usability** | Both Git and SVN use command line as primary UI.  Git is largely used by developers/technical users. | SVN is largely used by non-technical users as it's easier to learn. |
| **Content** | Cryptographic SHA-1 Hash. | No hashed content. |
| **Global Revision Number** | Not available | Available |

**Q #15) How will you differentiate between Git and GitHub?**

**Answer:**Git is a high-quality version control system. It is distributed in nature and is employed to track changes in source code throughout software development. It has a unique branching model that helps in synchronizing work among developers and tracking changes in any files.

The primary goals of Git are speed, data integrity, providing support to distributed, non-linear workflows. Git is installed and maintained on the local machine, instead of the cloud.

GitHub is a cloud-based Git repository hosting service that brings teams together. It gives you a web-based GUI as well as provides access control and many collaboration features, fundamental task management tools for each project.

Also, GitHub is an open-source i.e. code is kept on a centralized server and can be accessed by everyone.

**Q #16) What is a conflict in Git and how to resolve it?**

**Answer:**Git has an automatic merging feature that handles the merge commits on its own, provided the code changes have occurred on different lines and in different files.

But, in case of competing for commits where there are changes in the same lines of code of a file or a file has been deleted in one branch but exists and modified in another, Git is unable to automatically resolve differences and thus raises merge conflict.

In such cases, it requires your help to decide which code to include and which code to discard in the final merge.

A merge conflict can occur during merging a branch, rebasing a branch, or cherry-picking a commit. Once a conflict is detected, Git highlights the conflicted area and asks you to resolve it. Once the conflict is resolved, you can proceed with the merge.

**Follow the below steps to resolve a competing line change merge conflict:**

1. Open Git Bash (Git command line).
2. Use **cd <repository-name>** command to go to the local Git repository which is having the merge conflict.
3. Use the **git status** command to produce the list of files affected by the merge conflict.
4. Open the text editor that you use and traverse to the file that has merge conflicts.
5. To see the start of the merge conflict in your file, look the document for the conflict marker <<<<<<<. At the point when you open the file, you’ll observe the modifications from the HEAD or base branch after the line <<<<<<< HEAD. Then, you’ll observe =======, which partitions your modifications are from the modifications in the other branch, trailed by >>>>>>> BRANCH-NAME.
6. Choose in the event that you need to keep just your branch’s changes, just keep the other branch’s changes, or make a fresh change, that may include changes from the two branches. Erase the conflict markers <<<<<<<, =======, >>>>>>> and do the changes that you need in the final merge.
7. Use **git adds.** command to add or stage your changes.
8. Finally, use the **git commit -m “message”** command to commit your changes with a comment.

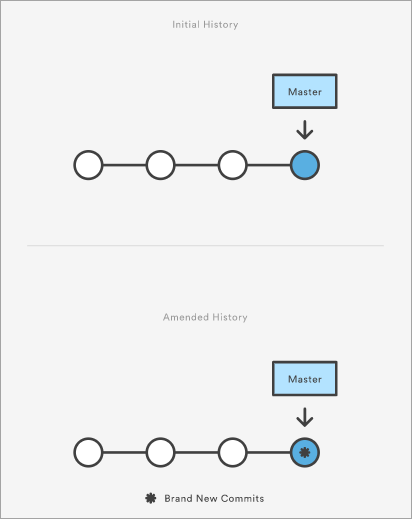
**To resolve the removed file merge conflict, you need to follow the below steps:**

1. Open Git Bash (Git command line).
2. Use **cd <repository-name>** command to go to the local Git repository that has the merge conflict.
3. Use the **git status** command to produce the list of files affected by the merge conflict.
4. Open the text editor that you use and traverse to the file that has merge conflicts.
5. Choose if you wish to keep the removed file. You can check the latest changes done in the removed file in your text editor.
6. Use **git add <filename>** command to add the removed file back to the repository. Or, Use **git rm <filename>** command to remove the file from your repository.
7. Finally, use the **git commit -m “message”** command to commit your changes with a comment.

**Q #17) How will you fix a Broken Commit?**

**Answer:** To fix a broken commit or to change the last commit, the most convenient method is to use the command “git commit -amend’.

It allows you to combine staged changes with the previous commit as an alternative for creating an entirely new commit. This replaces the most recent commit with the amended commit.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/broken-commit.png)

[image [*source*](https://www.atlassian.com/)]

Through this command, you can also edit the previous commit message without changing its snapshot.

**Q #18) What is the use of git instaweb?**

**Answer:** It is a script through which you can instantly browse your working Git repository in a web browser.

This script sets up gitweb and a webserver to browse the local repository. It automatically directs a web browser and runs a web server through an interface into your local repository.

**Q #19) What is git is-tree?**

**Answer:** ‘git is-tree’ signifies a tree object comprising the mode and the name of all items along with the SHA-1 value of the blob or the tree.

**Q #20) Is there a way to revert a git commit that’s already been pushed and made public?**

**Answer:** Yes, to fix or revert a bad commit, there are two approaches that can be used based upon the scenario.

**They are:**

1. The very obvious way is to make a fresh commit where you remove the bad file or fix the errors in it. Once done, you can push it to a remote repository.
2. Another approach is to create a new commit to undo all changes that were done in the previous bad commit. This can be done through git revert command – “git revert <name of bad commit>”

**Q #21) How will you differentiate between git pull and git fetch?**

**Answer: Git pull**command pulls all new commits from a specific branch in the central repository and makes the target branch in your local repository up-to-date.

**Git fetch** also aims at the same thing, however, its underlying functionality is a bit different. When you do a git fetch, all the new commits from a specific branch will be pulled in your central repository and these changes will be stored in a new branch in your local repository. This is called a fetched branch.

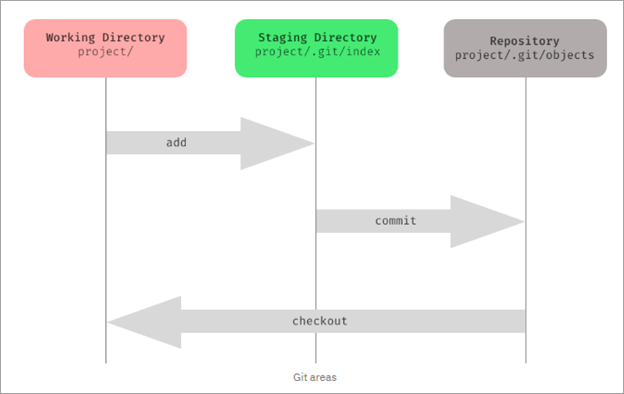
If you wish to see these changes in your target branch, then you need to perform a git merge after git fetch. The target branch will be updated with the latest changes only after merging it with the fetched branch.

So, a git pull brings the local branch up-to-date with its remote version, whereas a git fetch does not directly change your own local branch or working copy under refs/heads. Git fetch can be used to update your remote-tracking branches under **refs/remotes/<remote>/.**

In simple words, **git pull is equal to git fetch followed by a git merge**.

**Q #22) What is the use of Staging area or Indexing in Git?**

**Answer:** From Git’s perspective, there are three areas where the file changes can be kept i.e. working directory, staging area, and repository.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/of-staging-area-or-indexing-in-Git.png)

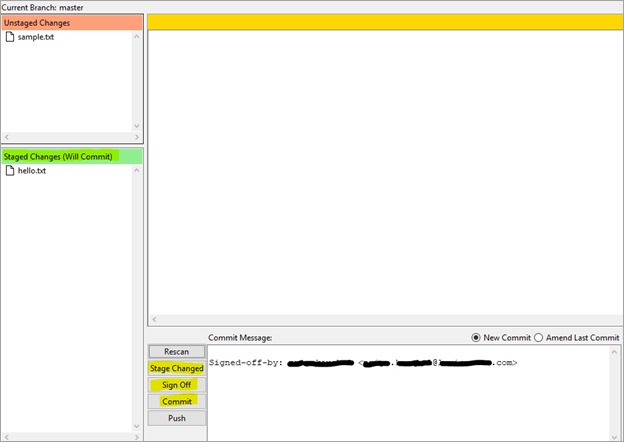
First, you make changes in your project’s working directory stored on your computer file system. All the changes remain here until you add them to an intermediate area called staging area.

You can stage the changes by executing git add . command. This staging area gives you a preview of your next commit and basically lets you fine-tune your commits. You can add or remove changes in the staging area until you are satisfied with the version you are going to commit.

Once you verify your changes and sign off the stage changed, then you can finally commit the changes. Upon commit, they go the local repository i.e. into .git/objects directory.

If you use Git GUI, then you will see the option to stage your changes. In the below screenshot, the file sample.txt is under unstaged changes area which means that it’s in your working directory.

You can select a file and click on ‘stage changed’, then it will be moved in the staging area. **For example**, the file hello.txt is present in stage changed (will commit) area. You can verify your changes and then do a sign-off, followed by a commit.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/Staging.png)

Staging is also referred to as indexing because git maintains an index file to keep track of your file changes across these three areas. The files which are staged are currently in your index.

When you add changes to the staging area, then the information in the index gets updated. When you do a commit, its actually what’s in the index that gets committed, and not what’s in the working directory. You can use the **git status** command to see what’s in the index.

**Q #23) What is Git Stash?**

**Answer:** GIT stash captures the current state of the working directory and index and keeps it on the stack for future use. It reverts the uncommitted changes (both staged and unstaged) from your working directory and returns you a clean working tree.

You can work on something else now, and when you come back, you can re-apply these changes. So, if you want to switch from one context to another without losing your current changes, then you can use stashing.

It is helpful in quick context switching, where you are in a mid-way of a code change that you don’t want to commit or undo it right now and you have got something else to work on. The command to use is git stash.

**Q #24) What is the Git Stash drop?**

**Answer:** When you no longer require a specific stash, you can remove it by executing **git stash drop <stash\_id> command**. If you want to remove all the stashes in one go from the repository then you can run **git stash clear command**.

**Q #25) What is Git stash apply? How is it different from Git stash pop?**

**Answer:** Both the commands are used to reapply your stashed changes and start working from where you had left.

In **git stash apply**command, the changes will be re-applied to your working copy and will also be kept in the stash. This command can be used when you want to apply the same stashed changes to multiple branches.

In **git stash pop** command, the changes are removed from the stash and are re-applied to the working copy.

**Q #26) What is the use of git clone command?**

**Answer:** The **git clone** command creates a copy of the existing central Git repository into your local machine.

**Q #27) When is the git config command used?**

**Answer:** The **git config**command is used to set configuration options for your Git installation.

**For example,** after you download Git, you need to use below the config commands to setup username and commit email address in Git respectively:

$ git config –global user.name “<username>”

$ git config –global user.email “<email id>”

So, mainly, things like the behavior of the repository, user information and preferences can be set up with the help of this command.

**Q #28) How will you identify if the branch is already merged into master?**

**Answer:**

**By executing the below commands, you can get to know the branch merge status:**

1. **git branch –merged master:** This will list out all the branches that have been renamed into master.
2. **git branch –merged:**This will list out all the branches that have been merged into HEAD.
3. **git branch –no-merged:**This will list out all the branches that are not yet merged.

By default, this command tells the merge status of local branches only. If you want to know about both local and remote branch merge status, then you can use -a flag. If you want to check only for remote branches, then you can use -r flag.

**Q #29) What are Hooks in Git?**

**Answer:** Git hooks are certain scripts that Git runs before or after an event like commit, push, update or receive. You will find the ‘hooks’ folder inside .git directory in your local repository. You will find the build-in scripts here pre-commit, post-commit, pre-push, post push.

These scripts get executed locally before or after the occurrence of an event. You can also modify these scripts according to your needs and Git will execute the script when that particular event occurs.

**Q #30) What is the use of git fork? How is forking different from cloning?**

**Answer:** To fork a project means to create a remote, server-side copy of the original repository. You can rename this copy, and start doing a new project around this without affecting the original project. The fork is not the core concept of Git.

The fork operation is used by Git workflow and this idea exists longer for free and open-source software like GitHub. Generally, once you have forked the project, you will rarely contribute to the parent project again.

**For example,** OpenBSD is a Unix-like open-source Operating system that was developed by forking NetBSD which is another Unix-like open-source OS.

However, in the fork, a direct connection exists between your forked copy and original repository. At any time, you can contribute back to the original project by using the pull requests.

In the forked copy, all the main data like codes and files get copied from the original repository, however, branches, pull requests and other features do not get copied. Forking is an ideal way for open source collaboration.

Cloning is essentially a Git concept. A clone is a local copy of any remote repository. When we clone a repository, the entire source repository along with its history and branches gets copied to our local machine.

Unlike forking, there is no direct connection between the cloned repository and the original remote repository. If you want to do pull requests and continue back to the original project, then you should get yourself added as a collaborator in the original repository.

Cloning is also a great way for creating a backup of the original repository as the cloned copy also has all the commit history.

**Q #31) How will you find out what all files have been changed in a particular Git commit?**

**Answer:** By using the hash value of the particular commit, you can execute the below command to get the list of files that have been changed in a particular commit:

**git diff-tree -r {hash}**

This will list down all the files that have been modified, and also the files that have been added. The -r flag is used to list individual files along with their path instead of collapsing them in their root directory names only.

**You can also use the below command:**

**git diff-tree –no-commit-id –name-only -r {hash}**

–no-commit-id will retrain the commit hash numbers to come in the output. Whereas, -name will exclude the file paths and only give the file names in the output.

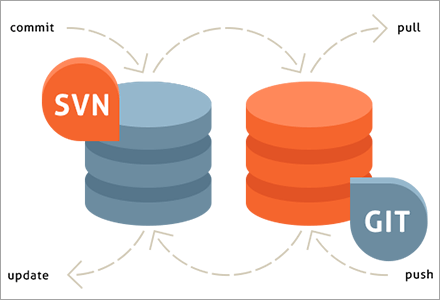
**Q #32) What is the difference between git checkout [branch name] and git checkout -b [branch name]?**

**Answer:** The command **git checkout [branch name]** will switch from one branch to another.

The command **git checkout -b [branch name]** will create a new branch and also switch to it.

**Q #33) What is SubGit?**

**Answer:** SubGit is a tool that is used for SVN to Git Migration. It is developed by a company called TMate. It converts the SVN repositories to Git and lets you do work on both the systems concurrently. It auto-syncs the SVN with Git.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/SubGit.png)

[image [*source*](https://subgit.com/)]

You can create an SVN||Git mirror using this tool. SubGit should be installed on your Git server. It will detect all the settings of your remote SVN repository, including SVN revisions, branches, and tags, and converts them into Git commits.

It also preserves the history including tracking merge data.

**Q #34) Can you recover a deleted branch in Git?**

**Answer:** Yes, you can. To recover a deleted branch, you should know the SHA off the top of your head. SHA or hash is a unique ID that Git creates with every operation.

When you delete a branch, you get the SHA displayed on the terminal:

**Deleted branch <your-branch-name> (was <sha>)**

You can use the below command to recover the deleted branch:

**git checkout -b <your-branch-name> <sha>**

If you don’t know the SHA for the commit at the tip of your branch then you can first use the **git reflog** command to know the SHA value and then apply the above checkout command to restore your branch.

**Q #35) What is** **git diff command? How is it different from git status?**

**Answer:** **Git diff** is a multi-use command that can be executed to show the differences between two arbitrary commits, changes between the working tree & a commit, changes between working tree & an index, changes between two files, changes between index & a tree, etc.

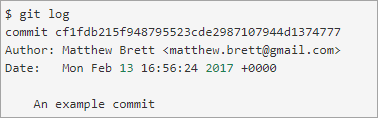
The **git status** command is used to inspect a repository. It shows the state of the working directory and staging area. It will list down the files that have been staged, which haven’t been staged and the files that are untracked.

**Q #36) What does a Commit object contain?**

**Answer:** The commit object contains the top-level tree object hash, parent commits hash(if any), author and committer information, commit date and commit message.

You can view this through the **git log** command.

**Example:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/commit-object.png)

[image [*source*](https://matthew-brett.github.io/)]

**Q #37) What is git cherry-pick? What are the scenarios in which git cherry-pick can be used?**

***Answer:*Git cherry-pick** is a powerful command to apply the changes introduced by one or more existing commits. It allows you to pick a commit from one branch and apply it to another.

**git cherry-pick commitSha** is the command used for cherry-picking. commitSha is the commit reference.

This command can be used for undoing changes. For instance, if by mistake you have made a commit to a wrong branch, then you can check out the correct branch and cherry-pick the commit to where it should belong.

It can also be used in team collaboration. There can be scenarios where the same code needs to be shared between two components of the product. In this case, if one developer has already written that code, then the other one can cherry-pick the same.

Cherry-picking is also useful in bug hotfixes where a patch commit can be cherry-picked directly into the master branch to fix the issue as soon as possible.

**Q #38) What is ‘git reset’ is used for? What is the default mode of this command?**

***Answer:*Git reset** is a powerful command for undoing local changes to the state of a Git repo. This command resets the current HEAD to the specified stage.

It resets both the index and the working directory to the state of your last commit. Git reset has three modes i.e. soft, hard and mixed. The default mode of operation is mixed.

**Q #39) What is the difference between ‘HEAD’, ‘working tree’ and ‘index’?**

**Answer:** The working tree or workspace is the directory containing the source files that you are currently working on.

The index is the staging area in Git where the commits are prepared. It lies between the commit and your working tree. Git index is one large binary file that enlists all files in the current branch, their names, sha1 checksums, and timestamps.

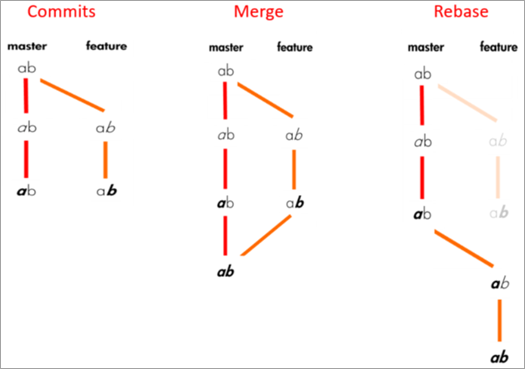
This file is present at <baseOfRepo>/.git/index. HEAD is the reference or pointer to the latest commit in the current checkout branch.

**Q #40) What’s the difference between rebase and merge? When should you rebase and when should you merge?**

**Answer:** Both rebase and merge commands are used to integrate changes from one branch to another but in a different manner.

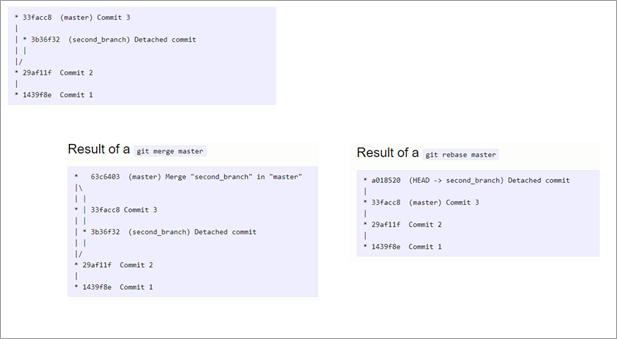
As seen in the below two images, suppose you have commits (this is before merge/rebase). After the merge, you will get the result as a combination of commits. It binds together the histories of both the branches and creates a new ‘merge commit’ in the feature branch.

On the other hand, rebase will move the whole feature branch to begin at the tip of the master branch.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/rebase-and-merge.png)

[image[*source*](https://medium.com/)]

**Commits will look like:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/commits.png)

Rebasing is not recommended for public branches as it creates inconsistent repositories. However, rebasing is a good option for private branches/individual developers. It is not very suitable for branch-per-feature mode. But if you have a branch-per-developer model, then rebasing is of no harm.

Also, rebase is a destructive operation, so your development team should be skilled enough to apply it correctly. Otherwise, committed work can be lost.

Furthermore, reverting a merge is easier than reverting a rebase. So, if you know that there can be possibilities for revert required, then you should use the merge.

Merge perseveres history as it is whereas rebase rewrites history. Thus, if you want to see the history completely as it occurred then you should use merge.

**Q #41) What is the syntax for rebasing?**

**Answer:** The syntax for rebase command is **git rebase [new-commit]**

**Q #42) How will you remove a file from Git without actually removing it from your local filesystem?**

**Answer:** **You can use the ‘cached’ option for this:**

**git rm -rf –cached $FILES**

This command will remove the files from your repository without deleting them from your disk.

**Q #43) What is the common branching pattern in Git?**

**Answer:**The common branching pattern is based on the git-flow. It has two main branches i.e. master and development.

* The master branch contains the production code. All the development code is merged into the master branch at some point in time.
* The development branch contains the pre-production code. When the features are completed, they get merged to the master branch, generally through a CI/CD pipeline.

**This model also has some supporting branches that are utilized during the development cycle:**

* **Feature Branches/Topic Branches:** They are used to develop new features for upcoming releases. It may branch off from the develop branch and must be merged back into the develop branch. Generally, these branches exist only in developer repositories, and not in origin.
* **Hotfix Branches:** They are used for unplanned production release when there is a need to fix any critical bug immediately in the live prod version. They may branch off from master and must be merged back into develop and master.
* **Release Branches:** They are used for the preparation of new production release. The release branch lets you do minor bug fixes and prepare metadata for release. They may branch off from development and must be merged back into master and develop.

GIT and GITHUB –Raghav

Git Setup on Local System:

Step 1 : check if git is already installed

git --version

Step 2 : download Git installer from https://git-scm.com/

Step 3 : Run installer and install git

Step 4 : Check if git is installed git --version

Create GitHub Account:

Step 1 : Goto https://github.com/ and sign up

Step 2 : Login to GitHub

Step 3 : Create a new Repository

Git Commands

Step 1 : Create a new folder and open Git Bash/CMD and goto the folder location

Step 2 : Run Commands

git config --global user.email "yourGitHub@email.com"

git config --global user.name "MYname"

verify email

git config --global user.email

git config --global user.name

Step 3 : Initialize Git git init

Step 4 : Add some files in the folder

Step 5 : Run commands git status

git add

git commit -m “…..”

Step 6 : Add the remote repo url

git remote add origin url\_of\_hithub\_repo

git remote show origin

Step 7 : Push the changes to repository

git push -u origin master

Step 8 : Check output of following commands

git log

git --help

Git Branches:

1. What are branches

2. How to create branch

3. How to checkout branch

4. How to merge branch to master

5. How to delete branch (local and remote)

Step 1 : Create branch git branch “branch name”

Step 2 : Checkout branch git checkout “branch name”

Step 3 : Make some changes in your project

Do add, commit, push git push -u origin newbranch

Check the branch is visible on GitHub repository

Step 4 : On local checkout master branch - git checkout master

Step 5 : Merge new branch in master branch git merge “branch name”

Step 6 : Push the changes - git push -u origin master

Step 7 : Delete branch

git branch -d “branch name” // will delete from local

git push origin --delete “branch name” // will delete from remote

Git Tags:

1. What are tags / releases

2. Why should i create TAGs

3. When to create TAGs

4. How to create TAGs in git

create | show | publish | delete

Step 1: Checkout the branch where you want to create the tag

git checkout "branch name"

example : git checkout master

Step 2: Create tag with some name

git tag "tag name"

example : git tag v1.0

Step 3: Display or Show tags git tag

git show v1.0

git tag -l “v1.\*”

Step 4: Push tags to remote git push origin v1.0

Step 5: Delete tags (if required only)

to delete tags from local : git tag -d v1.0

git tag --delete v1.0

to delete tags from remote :git push origin -d v1.0

git push origin --delete v1.0

git push origin :v1.0

to delete multiple tags at once:

git tag -d v1.0 v1.1 (local)

git push origin -d v1.0 v1.1 (remote)

Checking out TAGS

We cannot checkout tags in git

We can create a branch from a tag and checkout the branch

git checkout -b "branch name" "tag name"

example : git checkout -b ReleaseVer1 v1.0

Creating TAGS from past commits

git tag "tag name" "reference of commit"

example : git tag v1.2 5fcdb03

To generate Access Token

1. Login to your GitHub account

2. Verify your email address, if it hasn't been verified yet.

3. In the upper-right corner of any page, click your profile photo, then click Settings.

4. In the left sidebar, click Developer settings.

5. In the left sidebar, click Personal access tokens.

6. Click Generate new token.

7. Give your token a descriptive name.

8. To give your token an expiration, select the Expiration drop-down menu, then click a default or use the calendar picker

9. Select the scopes or permissions, you'd like to grant this token. To use your token to access repositories from the command line, select repo.

10. Click Generate token

Never Stop Learning

**To copy file from one branch to another branch**

git checkout feature2 raghu/santoshi/tisha/anika/ace # git checkout feature