**A)What is Git**

1)Git is a Distributed Version Control system(DVCS). It lets you track changes made to a file and allows you to revert back to any particular change that you wish.

2)It is a distributed architecture, it **does not rely on a central server** to store all the versions of a project’s files.

3)Instead, every developer “clones” a copy of a repository with “Local repository” and has the full history of the project available on his hard drive. So when there is a server outage all you need to do to recover is one of your teammate’s local Git repository.

4)There is a central cloud repository where developers can commit changes and share them with other teammates.

***Language***

---Git uses ‘C’ language. GIT is fast, and ‘C’ language makes this possible by reducing the overhead of run times associated with high-level languages.

***Repository in Git***

1)Repository in Git is a place where Git stores all the files. Git can store the files either on the local repository or on the remote repository

2)To create a repository, create a directory for the project if it does not exist, then run the

command “ **git init** ”. By running this command .git directory will be created in the project

directory.

**Bare repository(central repo)**

A “bare” repository in Git contains information about the version control and no working files (no tree) and it doesn’t contain the special .git sub-directory. Instead, it contains all the contents (config description HEAD hooks/ info/ objects/ refs/) of the .git sub-directory directly in the main directory itself, whereas (non-bear repo) the working directory

consists of :can do modifications

1. A .git subdirectory with all the Git related revision history of your repository.

2. A working tree, or checked out copies of your project files.

***Various Git repository***

Github

Gitlab

Bitbucket

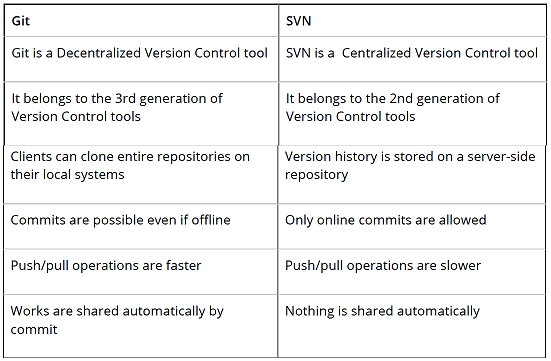
SourceForge

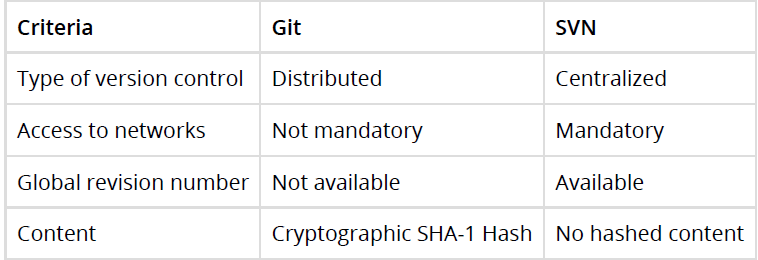
GitEnterprise

**SVN to Git migration**

SubGit is a tool for SVN to Git migration. It can create a writable Git mirror of a local or remote Subversion repository and use both Subversion and Git as long as you like. Now you can also include some advantages like you can do a fast one-time import from Subversion to Git or use SubGit within Atlassian Bitbucket Server. We can use SubGit to create a bi-directional Git-SVN mirror of an existing Subversion repository. You can push to Git or commit to Subversion as per your convenience. Synchronization will be done by SubGit

**Difference between Git and SVN**





**The difference between GIT and SVN**

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.

**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(multiple people can work) friendly

f) Any sort of projects can use GIT

**What is a distributed VCS?**

These are the systems that don’t rely on a central server to store a project file and all its

versions.

In Distributed VCS, every contributor can get a local copy or “clone” of the main repository.

Eevery programmer can maintain a local repository which is actually the copy or clone of the central repository which is present on their hard drive. They can commit and update their local repository without any hassles. With an operation called “pull”, they can update their local repositories with new data from the central server and “pull” operation affects changes to the main repository from their local repository.

***Benefits*** ***VCS***

With the Version Control System(VCS), all the team members are allowed to work freely on any file at any time. VCS gives you the flexibility to merge all the changes into a common version.

All the previous versions and variants are neatly packed up inside the VCS. You can

request any version at any time as per your requirement and you’ll have a snapshot of the complete project right at hand.

Whenever you save a new version of your project, your VCS requires you to provide a

short description of the changes that you have made. Additionally, you can see what

changes are made in the file’s content. This helps you to know what changes have been

made in the project and by whom.

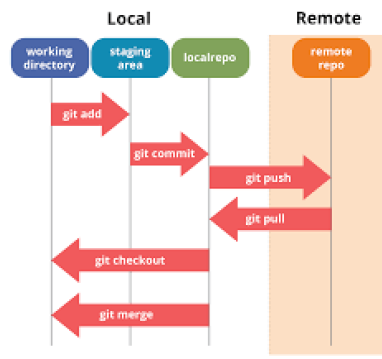
A distributed VCS like Git allows all the team members to have a complete history of the

project so if there is a breakdown in the central server you can use any of your teammate’s local Git repository.

**Difference between Git and Github?**

Git is a version control system of distributed nature that is used to track changes in source code during software development. It aids in coordinating work among programmers, but it can be used to track changes in any set of files. The main objectives of Git are speed, data integrity, and support for distributed, non-linear workflows.

GitHub is a Git repository hosting service, plus it adds many of its own features. GitHub provides a Web-based graphical interface. It also provides access control and several collaboration features, basic task management tools for every project.



**1.Working tree**

The working tree/working directory/workspace is the directory tree of (source) files that

you are able to see and edit.

**2.Index or ‘staging area’**

The index/staging area is a single, large, binary file in <baseOfRepo>/.git/index, which lists

all files in the current branch, their SHA-1 checksums, timestamps, and the file name – it is not another directory which contains a copy of files in it.

That before completing the commits, it can be formatted and reviewed in an intermediate area known as ‘Staging Area’ or ‘Index’. From the diagram it is evident that every change is first verified in the staging area I have termed it as “stage file” and then that change is committed to the repository

**3.HEAD**

HEAD is used to refer to the last commit in the currently checked-out branch

**Commit**

The command that is used to write a commit message is “ **git commit -a** ”.

Now explain about -a flag by saying -a on the command line instructs git to commit the new content of all tracked files that have been modified. Also, mention you can use “ **git add <file>** ” before git commit -a if new files need to be committed for the first time.

**commit object contain**

Commit object contains the following components, you should mention all the three points presented below:

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

2. Reference to parent commit objects

3. An SHA-1 name, a 40 character string that uniquely identifies the commit object

**Fix a broken commit**

use the command “ **git commit --amend** ”. When you run this command, you can fix the broken commit message in the editor

**Different ways you can refer to a commit**

In Git each commit has a unique hash. These hashes are used to identify the corresponding commits in various scenarios, for example, while trying to checkout a

particular state of the code using the **git checkout {hash}** command.

Along with this, Git maintains a number of aliases to certain commits, known as refs. Also,

every tag that is created in the repository effectively becomes a ref and that is exactly why you can use tags instead of committing hashes in various git commands. Git also

maintains a number of special aliases that are changed based on the state of the repository, such as **HEAD, FETCH\_HEAD, MERGE\_HEAD**, etc.

In Git, commits are allowed to be referred to as relative to one another. In the case of

merge commits, where the commit has two parents, ^ can be used to select one of the

two parents, for example, **HEAD^2** can be used to follow the second parent.

And finally, refspecs are used to map local and remote branches together. However, these

can also be used to refer to commits that reside on remote branches allowing one to

control and manipulate them from a local git environment.

**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.

**How to revert previous commit in git**

Say you have this, where C is your HEAD and (F) is the state of your files.

(F)

A-B-C

↑

master

To nuke changes in the commit:

**git reset --hard HEAD~1**

Now B is the HEAD. Because you used --hard, your files are reset to their state at commit B.

To undo the commit but keep your changes:

**git reset HEAD~1**

Now we tell Git to move the HEAD pointer back one commit (B) and leave the files as they are and git status shows the changes you had checked into C.

To undo your commit but leave your files and your index

**git reset --soft HEAD~1**

When you do git status , you'll see that the same files are in the index as before.

**Scenarios**

**1.Conflict in git**

Git can handle on its own most merges by using its automatic merging features. There arises a conflict when two separate branches have made edits to the same line in a file, or when a file has been deleted in one branch but edited in the other. Conflicts are most likely to happen when working in a team environment.

The following steps will resolve conflict in Git-

1. Identify the files that have caused the conflict.

2. Make the necessary changes in the files so that conflict does not arise again.

3. Add these files by the command git add.

4. Finally to commit the changed file using the command git commit

**2.Revert a commit that has already been pushed and made public**

There can be two approaches to tackle this question and make sure that you include both

because any of the below options can be used depending on the situation:

Remove or fix the bad file in a new commit and then push it to the remote repository. This is the most obvious way to fix an error. Once you have made necessary changes to the file, then commit it to the remote repository using the command: git commit -m “commit message”

Also, you can create a new commit that undoes all changes that were made in the bad

commit. To do this use the command

**git revert <name of bad commit>**

**3.What work is restored when the deleted branch is recovered?**

The files which were stashed and saved in the stash index list will be recovered back. Any

untracked files will be lost. Also, it is a good idea to always stage and commit your work or stash them.

If you want to fetch the log references of a particular branch or tag then run the command –

**git reflog <ref\_name**>

**4.Find a list of files that have changed in a particular commit**

For this answer instead of just telling the command, explain what exactly this command will do.

To get a list file that has changed in a particular commit use the below command:

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

Given the commit hash, this will list all the files that were changed or added in that commit. The -r flag makes the command list individual files, rather than collapsing them into root directory names only.

You can also include the below-mentioned point, although it is totally optional but will help in impressing the interviewer.

The output will also include some extra information, which can be easily suppressed by

including two flags:

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

Here –no-commit-id will suppress the commit hashes from appearing in the output, and –nameonly will only print the file names, instead of their paths

**5.How will you know in Git if a branch has already been merged**

**into master**

The answer is pretty direct. To know if a branch has been merged into master or not you can use the below commands:

**git branch --merged** – It lists the branches that have been merged into the current branch.

**git branch --no-merged** – It lists the branches that have not been merged.

**6.In Git, how would you return a commit that has just been**

**pushed and made open**

One or more commits can be reverted through the use of git revert. This command, in a true sense, creates a new commit with patches that cancel out the changes introduced in specific commits. If in case the commit that needs to be reverted has already been published or changing the repository history is not an option then in such cases, git revert can be used to revert commits. If you run the following command then it will revert the last two commits:

**git revert HEAD~2..HEAD**

**8.How to remove a file from git without removing it from your file**

**system?**

One has to be careful during a git add, else you may end up adding files that you didn’t want to commit. However, git rm will remove it from both your staging area (index), as well as your file system (working tree), which may not be what you want.

Instead, use git reset:

**git reset filename** # or

**echo filename >> .gitignore** # add it to .gitignore to avoid re-adding it

This means that git reset <paths> is exactly the opposite of git add <paths>.

**9.How do you squash the last N commits into a single commit?**

**There are two options to squash the last N commits into a single commit include both of the below-mentioned options in your answer**

If you want to write the new commit message from scratch use the following command

**git reset –soft HEAD~N && git commit** If you want to start editing the new commit message with a concatenation of the existing commit messages then you need to extract those messages and pass them to Git commit for that I will use

**git reset –soft HEAD~N && git commit –edit -m”$(git log –format=%B –reverse** [**.HEAD@{N}**](mailto:.HEAD@%7bN%7d)**)”**

Squashing multiple commits into a single commit will overwrite history, and should be done with caution. However, this is useful when working in feature branches. To squash the last N commits of the current branch, run the following command (with {N} replaced with the number of commits that you want to squash):

**git rebase -i HEAD~{N}**

Upon running this command, an editor will open with a list of these N commit messages, one per line. Each of these lines will begin with the word “pick”. Replacing “pick” with “squash” or “s” will tell Git to combine the commit with the commit before it. To combine all N commits into one, set every commit in the list to be squash except the first one. Upon exiting the editor, and if no conflict arises, git rebase will allow you to create a new commit message for the new combined commit.

**10.How do you configure a Git repository to run code sanity checking tools right before making commits, and preventing them if the test fails?**

I will suggest you to first give a small introduction to sanity checking. Sanity or smoke test determines whether it is possible and reasonable to continue testing. Now explain how to achieve this.

This can be done with a simple script related to the pre-commit hook of the repository. The precommit hook is triggered right before a commit is made, even before you are required to enter a commit message. In this script, one can run other tools, such as linters and perform sanity checks on the changes being committed into the repository.

Finally, give an example, you can refer the below script:

**#!/bin/sh**

**files=$(git diff –cached –name-only –diff-filter=ACM | grep ‘.go$’)**

**if [ -z files ]; then**

**exit 0**

**fi**

**unfmtd=$(gofmt -l $files)**

**if [ -z unfmtd ]; then**

**exit 0**

**fi**

**echo “Some .go files are not fmt’d”**

**exit 1**

This script checks to see if any .go file that is about to be committed needs to be passed through the standard Go source code formatting tool gofmt. By exiting with a non-zero status, the script effectively prevents the commit from being applied to the repository.

**11.What is ‘head’ in git and how many heads can be created in a**

**repository**

**The reference to a commit object** is called as the ‘head’. Every repository has a ‘Master’ which is the default head. There can be multiple heads in a repository.

**git config**

Git uses your username **to associate commits with an identity**. The git config command can be used to change your Git configuration, including your username.

Now explain with an example.

Suppose you want to give a username and email id to associate a commit with an identity so that you can know who has made a particular commit. For that I will use:

**git config –global user.name “Your Name”** : This command will add a username.

**git config –global user.email “Your E-mail Address”** : This command will add an email id.

**git stash**

Often, when you’ve been working on part of your project, things are in a messy state and you want to switch branches for some time to work on something else. The problem is, you don’t want to do a commit of half-done work just so you can get back to this point later. The answer to this issue is Git stash. Stashing takes your working directory that is, your modified tracked files and staged changes and saves it on a stack of unfinished changes that you can reapply at any time.

**git stash apply**

If you want to continue working where you had left your work then **‘ git stash apply** ‘ command is used to bring back the saved changes onto your current working directory.

**git stash drop**

Git ‘stash drop’ command is used to remove the stashed item. It will remove the last added stash item by default, and it can also remove a specific item if you include it as an argument.

Now give an example.

If you want to remove a particular stash item from the list of stashed items you can use the below commands:

git stash list: It will display the list of stashed items like:

stash@{0}: WIP on master: 049d078 added the index file

stash@{1}: WIP on master: c264051 Revert “added file\_size”

stash@{2}: WIP on master: 21d80a5 added number to log

If you want to remove an item named stash@{0} use command git stash drop stash@{0}.

Consider

$ git status

On branch master Changes to be committed: new file: style.css

Changes not staged for commit: modified: index.html

$ git stash

Saved working directory and index state WIP on master: 5002d47 our new homepage

HEAD is now at 5002d47 our new homepage

$ git status

On branch master nothing to commit, working tree clean

The one place we could use stashing is if we discover we forgot something in our last commit and have already started working on the next one in the same branch:

$ git stash save

$ git add -u

$ git commit --ammend

$ git stash pop

**git cherry-pick**

The command git cherry-pick is normally used to introduce particular commits from one branch within a repository onto a different branch. Another common use is to forward- or back-port commits from a maintenance branch to a development branch. This is in contrast with other ways such as merge and rebase which normally apply many commits onto another branch.

Consider:

**git cherry-pick <commit-hash>**

**git reflog**

The ‘reflog’ command keeps a track of every single change made in the references (branches or tags) of a repository and keeps a log history of the branches and tags that were either created locally or checked out. Reference logs such as the commit snapshot of when the branch was created or cloned, checked-out, renamed, or any commits made on the branch are maintained by Git and listed by the ‘reflog’ command.

Note: The branch will be recoverable from your working directory only if the branch ever

existed in your local repository i.e. the branch was either created locally or checked-out from a remote repository in your local repository for Git to store its reference history logs.

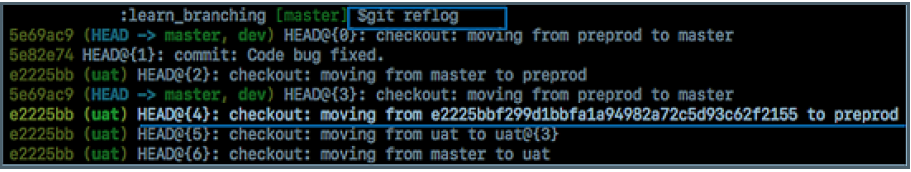
This command must be executed in the repository that had the lost branch. If you consider the remote repository situation, then you have to execute the reflog command on the developer’s machine who had the branch.

command: git reflog

**How to recover a deleted branch using git reflog**

Step 1: History logs of all the references

Get a list of all the local recorded history logs for all the references (‘master’, ‘uat’ and ‘prepod’) in this repository.

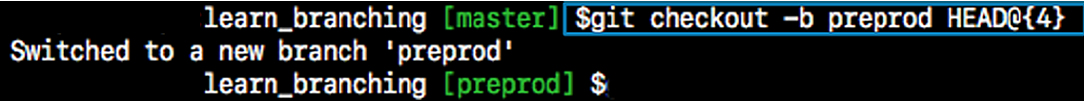


Step 2: Identify the history stamp

As you can see from the above snapshot, the highlighted commit id: e2225bb along with the HEAD pointer index:4 is the one when ‘preprod’ branch was created from the current HEAD pointer pointing to your latest work.

Step 3: Recover

If you want to recover back the ‘preprod‘ branch then use the command ‘git checkout’ passing the HEAD pointer reference with the index id – 4. This is the pointer reference when ‘preprod’ branch was created long commit id highlighted in the output screenshot.



**Git bisect** , **determine the source of a (regression) bug**

Git bisect is used to find the commit that introduced a bug by using binary search. The

command for Git bisect is

**git bisect <subcommand> <options>**

Now since you have mentioned the command above explain to them what this command

will do.

This command uses a binary search algorithm to find which commit in your project’s

history introduced a bug. You use it by first telling it a “bad” commit that is known to

contain the bug, and a “good” commit that is known to be before the bug was introduced.

Then Git bisect picks a commit between those two endpoints and asks you whether the

selected commit is “good” or “bad”. It continues narrowing down the range until it finds

the exact commit that introduced the change.

Git provides a rather efficient mechanism to find bad commits. Instead of making the user try out every single commit to find out the first one that introduced some particular issue into the code, git bisect allows the user to perform a sort of binary search on the entire history of a repository.

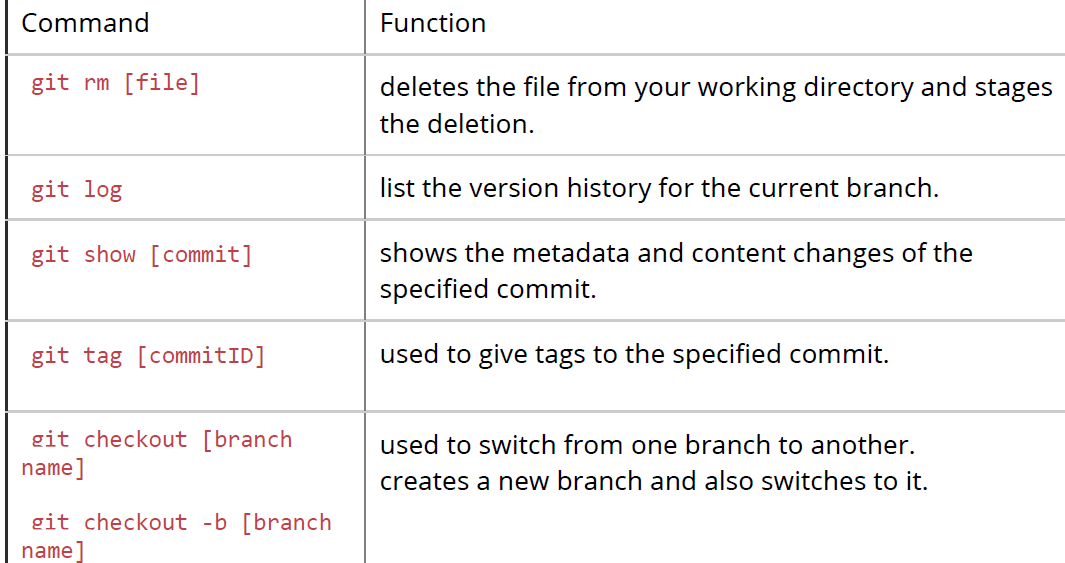
By issuing the command git bisect start, the repository enters bisect mode. After this, all you have to do is identify a bad and a good commit:

**git bisect bad** # marks the current version as bad

**git bisect good {hash or tag}**# marks the given hash or tag as good, ideally of some earlier commit.

Once this is done, Git will then have a range of commits that it needs to explore. At every step, it will checkout a certain commit from this range, and require you to identify it as good or bad.

After which the range will be effectively halved, and the whole search will require a lot less number of steps than the actual number of commits involved in the range. Once the first bad commit has been found, or the bisect mode needs to be ended, the following command can be used to exit the mode and reset the bisection state: **git bisect reset**



**GIT PUSH**

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

**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

**git diff**

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

**git status**

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

**git init**

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

**git checkout**

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.

**git rm**

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

**git log**

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

**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

**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

**git instaweb**

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

Here are the Git commands to be covered:

git config, git init, git clone ,git add, git commit, git diff, git reset, git status, git rm, git log, git show, git tag, git branch, git checkout, git merge, git remote, git push, git pull, git stash

**Differences**

1.rebasing and merge

In Git, the rebase command is used to integrate changes from one branch into another. It is an alternative to the “merge” command. The difference between rebasing and merge is that rebase rewrites the commit history in order to produce **a straight, linear succession of commits**.

Merging is Git’s way of putting a forked history back together again. The git merge

command helps you take the independent lines of development created by git branch and

integrate them into a single branch.

revert and reset

Git reset is a powerful command that is used to undo local changes to the state of a Git

repository. Git reset operates on “The Three Trees of Git” which are, Commit History(HEAD ), the Staging Index, and the Working Directory.

Revert command in Git creates a new commit that undoes the changes from the previous

commit. This command adds a new history to the project. It does not modify the existing

history.

"git pull" and "git fetch

In the simplest terms, git pull does a git fetch followed by a git merge. difference

When you use pull, Git tries to automatically do your work for you. It is context sensitive, so Git will merge any pulled commits into the branch you are currently working in. pull automatically merges the commits without letting you review them first. If you don’t closely manage your branches, you may run into frequent conflicts.

When you fetch, Git gathers any commits from the target branch that do not exist in

your current branch and stores them in your local repository. However, it does not

merge them with your current branch. This is particularly useful if you need to keep your

repository up to date, but are working on something that might break if you update your

files. To integrate the commits into your master branch, you use merge.

‘git diff ’and ‘git status’

‘ git diff ’ depicts the changes between commits, commit and working tree, etc. whereas ‘ git status ’ shows you the difference between the working directory and the index, it is helpful in understanding a git more comprehensively. ‘git diff’ is similar to ‘git status’, the only difference is that it shows the differences between various commits and also between the working directory and index.

‘git remote’ and ‘git clone

‘git remote add’ creates an entry in your git config that specifies a name for a particular URL whereas

‘git clone’ creates a new git repository by copying an existing one located at the URL

pull request and a branch

A branch is just a separate version of the code.

A pull request is when someone take the repository, makes their own branch, does some

changes, then tries to merge that branch in (put their changes in the other person's code

repository).

fork, branch, clone

A fork is a copy of a repository. Normally you fork a repository so that you are able to

freely experiment with changes without affecting the original project. Most commonly,

forks are used to either propose changes to someone else’s project or to use someone

else’s project as a starting point for your own idea.

git cloning means pointing to an existing repository and make a copy of that repository in

a new directory, at some other location. The original repository can be located on the local file system or on remote machine accessible supported protocols. The git clone command is used to create a copy of an existing Git repository. When you clone, you are actually copying the entire source repository, including all the history and branches.

In very simple words, git **branches are individual projects within a git repository.** Different branches within a repository can have completely different files and folders, or it could have everything the same except for some lines of code in a file. A branch is a mechanism to handle the changes within a single repository in order to eventually merge them with the rest of code. A branch is something that is within a repository. Conceptually, it represents a thread of development

**Branching strategies**

**Feature branching** – A feature branch model keeps all of the changes for a particular feature inside of a branch. When the feature is fully tested and validated by automated tests, the branch is then merged into master.

**Task branching** – In this model, each task is implemented on its own branch with the task key included in the branch name. It is easy to see which code implements which task, just look for the task key in the branch name.

**Release branching** – Once the develop branch has acquired enough features for a release, you can clone that branch to form a Release branch. Creating this branch starts the next release cycle, so no new features can be added after this point, only bug fixes, documentation generation, and other release-oriented tasks should go in this branch.

Once it is ready to ship, the release gets merged into master and tagged with a version number. In addition, it should be merged back into the develop branch, which may have progressed since the release was initiated.

In the end tell them that branching strategies vary from one organization to another so I know basic branching operations like delete, merge, checking out a branch, etc.

***What is the purpose/need 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.

***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.

***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”.

***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]”.

**Git branch** –merged master – shows all branches that are merged into master

**Git branch** – merged – shows all branches that are merged into the head

**Git branch** – no-merged –shows all the branches that are not merged

**Git hooks**

This directory consists of **shell scripts** that are activated if you run the corresponding Git commands. For example, git will try to execute the post-commit script after you have run a commit.

**Forking workflow**

There is a fundamental difference between the forking workflow and other popular git workflows. Rather than using a single server-side to act as the “central” codebase, it gives every developer their own server-side repository. The Forking Workflow is commonly seen in public open-source projects. A crucial advantage of the Forking Workflow is that contributions can be integrated without even needing everybody to push to a single central repository that leads to clean project history. Developers can push to their own server-side repositories, but only the project maintainer can push to the official repository.

If developers are ready to publish a local commit, then they push the commit to their own public repository and not the official one. After this, they go for a pull request with the main repository that lets the project maintainer know an update is ready to be integrated.

**Gitflow workflow**

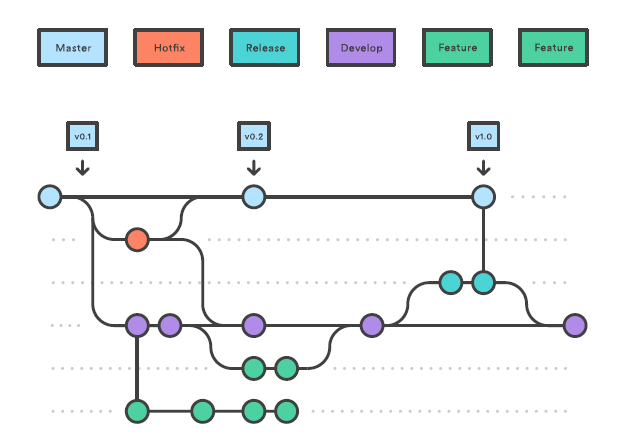
To record the history of the project, Gitflow workflow employs two parallel long-running branches – master and develop:

**Master** – this branch is always ready to be released on LIVE, with everything fully tested and approved (production-ready).

**Hotfix** – these branches are used to quickly patch production releases. These branches are a lot like release branches and feature branches except they’re based on master instead of develop.

**Develop** – this is the branch to which all feature branches are merged ad where all tests are performed. Only when everything’s been thoroughly checked and fixed it can be merged to the master.

**Feature** – each new feature should reside in its own branch, which can be pushed to the develop branch as their parent one.

–

**Commands**

**git config –global user.name”<name>”**

**git config –global user.email”<email>”**

**git fork -**

**git clone**

**git fetch**

**git merge**

**git rebase**

**git cherrypick**

**git pull**

**git push**

**git branch (--merged master, --merged, --no merged)**

**git tag**

**git revert**

**git reset(--soft, --mixed, --hard)**

**git reflog**

**git stash**

**git sqash**

**git diff**

**git status**

**git rm**

**git bisect**

**git branch -a**

**git branch –d<b.name>**

**git push origin –delete <b.name>**

**git checkout**

**git hooks**

**git remote -v**

**git remote add**