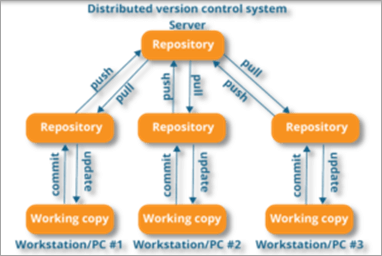
**Q #1) What is Git?**

**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.

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

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.

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

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.

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

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.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/Reliable-1.png)

**(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.

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

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:**

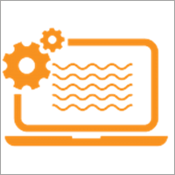
In the case of a centralized version control system, the central server must be strong enough to attend requests of the entire team. This is not a problem for smaller teams, however as the team expands, the hardware limitations of the server can be an impediment for performance.

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

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:**

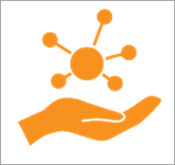
Git provides rapid branching & merging and contains particular tools for envisaging and traversing a non-linear development history. A basic notion in Git is that a change will be merged more frequently than it is written as it is sent across different reviewers.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/Supports-non-linear-Development.png)

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:**

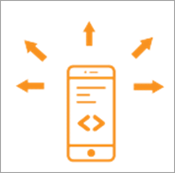
Branch management through Git is very straightforward and easy. It requires just a few jiffies to create, delete, and merge branches. Feature branches give an insulated environment to each change to your codebase.

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

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.

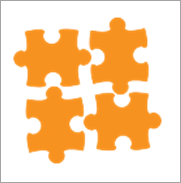
**(ix) Distributed Development:**

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.

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

**(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.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2020/01/Compatibility-along-with-present-systems-or-protocol.png)

**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.

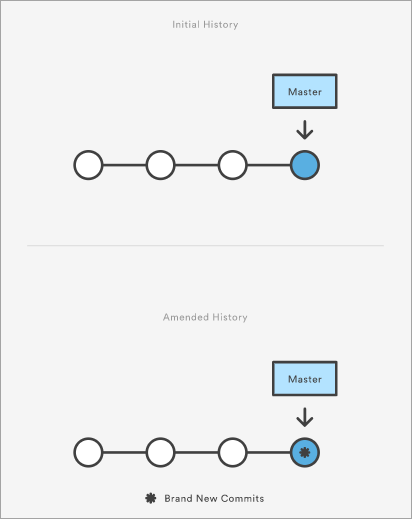
**To resolve the removed file merge conflict, you need to follow the below steps:**

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.

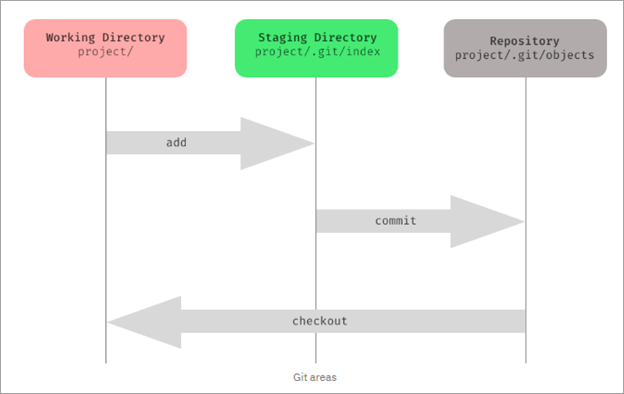
If you wish to see these changes in your target branch, then you need to perform a *git merge* after git fetch. The target branch will be updated with the latest changes only after merging it with the fetched branch.

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)

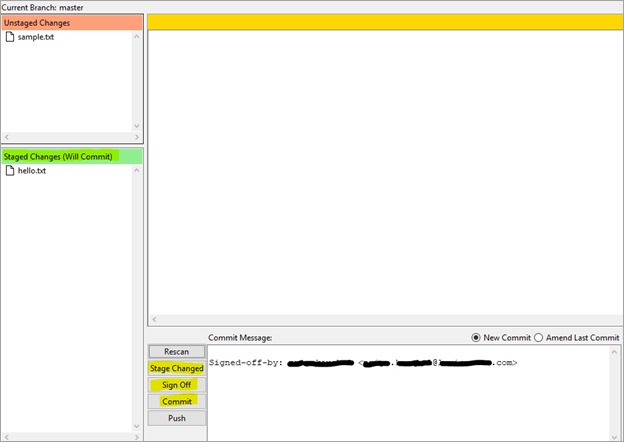
First, you make changes in your project’s working directory stored on your computer file system. All the changes remain here until you add them to an intermediate area called staging area.

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.

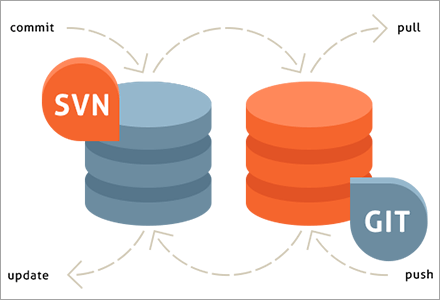
**Q #32) What is the difference between git checkout [branch name] and git checkout -b [branch name]?**

**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.

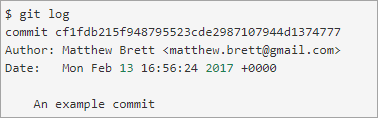
The **git status** command is used to inspect a repository. It shows the state of the working directory and staging area. It will list down the files that have been staged, which haven’t been staged and the files that are untracked.

**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.

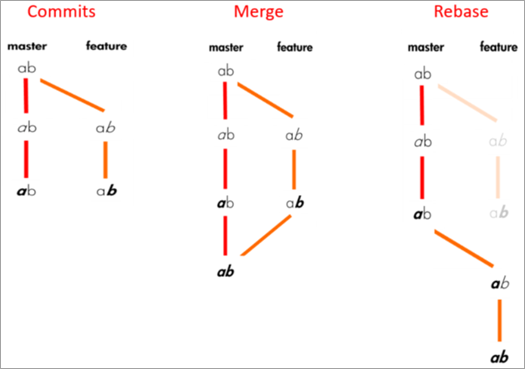
This file is present at <baseOfRepo>/.git/index. HEAD is the reference or pointer to the latest commit in the current checkout branch.

**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.

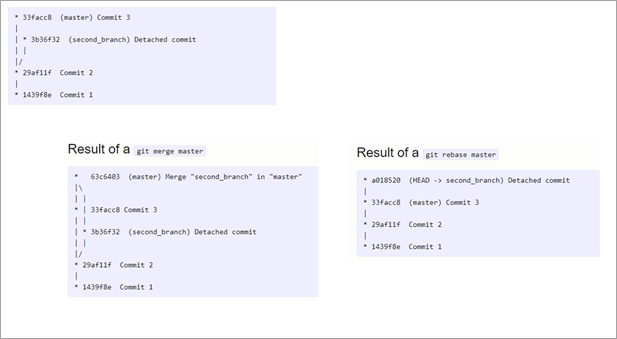
As seen in the below two images, suppose you have commits (this is before merge/rebase). After the merge, you will get the result as a combination of commits. It binds together the histories of both the branches and creates a new ‘merge commit’ in the feature branch.

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.