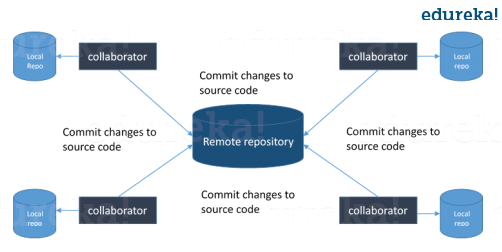
**1. What is GIT?**

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.

Its distributed architecture provides many advantages over other Version Control Systems (VCS) like SVN one major advantage is that it does not rely on a central server to store all the versions of a project’s files. Instead, every developer “clones” a copy of a repository I have shown in the diagram with “Local repository” and has the full history of the project on his hard drive so when there is a server outage all you need for recovery is one of your teammate’s local Git repository. There is a central cloud repository as well where developers can commit changes and share it with other teammates as you can see in the diagram where all collaborators are commiting changes “Remote repository”.



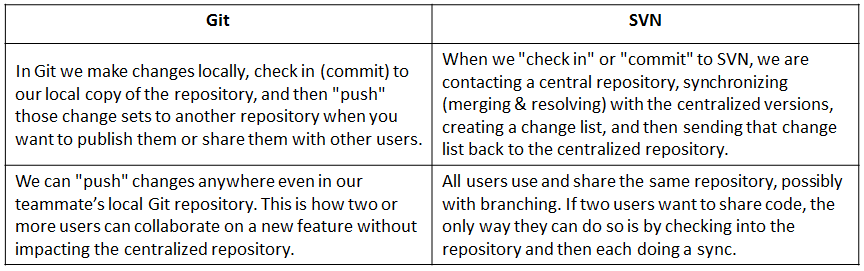
### 2. What is a repository in GIT?

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 difference between Git and SVN?**

The proper answer for this according to me will be the architectural differences between Git and SVN. So the basic difference is that Git is distributed and SVN is centralized version control system.

Then explain the same by including the below mentioned differences:



**Q4. What is ‘bare repository’ in Git?**

You are expected to tell the difference between a “working directory” and “bare repository”.

A “bare” repository in Git just contains the version control information and no working files (no tree) and it doesn’t contain the special .git sub-directory. Instead, it contains all the contents of the .git sub-directory directly in the main directory itself, where as working directory consist of:

1. A .git subdirectory with all the Git related revision history of your repo.
2. A working tree, or checked out copies of your project files.

**5. What is the difference between git pull and git fetch?**

Git pull command pulls new changes or commits from a particular branch from your central repository and updates your target branch in your local repository.

Git fetch is also used for the same purpose but it works in a slightly different way. When you perform a git fetch, it pulls all new commits from the desired branch and stores it in a new branch in your local repository. If you want to reflect these changes in your target branch, git fetch must be followed with a git merge. Your target branch will only be updated after merging the target branch and fetched branch. Just to make it easy for you, remember the equation below:

Git pull = git fetch + git merge

**6. What is 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.

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.

**7. What is Git stash drop?**

Begin this answer by saying for what purpose we use 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}**.

**8. What does commit object contains?**

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

* A set of files, representing the state of a project at a given point of time
* Reference to parent commit objects
* An SHAI name, a 40 character string that uniquely identifies the commit object.

**9. Describe branching strategies you have used?**

This question is asked to test your branching experience with Git so, tell them about how you have used branching in your previous job and what purpose does it serves, you can refer the below mention points:

* 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 develop branch, which may have progressed since the release was initiated.

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

**10. 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.

**11. What is Git rebase and how can it be used to resolve conflicts in a feature branch before merge?**

git rebase in its simplest form is a command which will merge another branch into the branch where you are currently working, and move all of the local commits that are ahead of the rebased branch to the top of the history on that branch.

From the man page: “git rebase: Forward-port local commits to the updated upstream head”.

If you’re still confused, let’s look at an example.

Say I have been doing work on a feature branch and I want to merge in the changes my teammates have made on the master branch. I’ve made 2 commits locally that aren’t shared on the origin remote, and while I’ve been working on my feature branch my co-workers have made 20 commits on origin/master that I don’t have on my branch (but I would like to).

My log looks something like this on the local branch. My two recent (unshared) commits are at the top of the history, and the rest is the history of master when I originally checked out this branch.

f48d47c Add Controller changes

fd4e046 Add DB migration

907e384 Most recent commit on master when I originally checked out feature branch

71fd630 ...

...

So how should I approach getting the changes my co-workers have made on master? I could git fetch origin and git merge origin/master, but that would cause two undesirable side effects:

1. It forces the creation of a merge commit. This is useful information if there was a conflict, but otherwise it’s just noise which pollutes the history. Not good if you update your branch frequently (which you probably should).
2. It buries my commits under an avalanche of commits from other places, when really they should be at the top of the log due to the fact that they’re “more recent to try and move upstream”. That is, it makes more sense for my buddy who is also working on this feature branch to see my commits first when he pulls in changes than it does for him to see the master changes. The things that I’ve been working on in this branch are probably more relevant to him.

So what to do? Well, as you may have predicted, git rebase comes to our rescue here.

git rebase origin/master will merge in the requested branch (origin/master in this case) and apply the commits that you have made locally to the top of the history without creating a merge commit (assuming there were no conflicts). Now our history is nice and clean, and we have avoided the two issues listed above.

**12. What is Git Tag**

Git has the ability to tag specific points in history as being important. Typically people use this functionality to mark release points (v1.0, and so on)

# **git merge**

Merging is Git's way of putting a forked history back together again. The git merge command lets you take the independent lines of development created by [git branch](https://www.atlassian.com/git/tutorials/using-branches/git-branch) and integrate them into a single branch.

Note that all of the commands presented below merge into the current branch. The current branch will be updated to reflect the merge, but the target branch will be completely unaffected. Again, this means that git merge is often used in conjunction with [git checkout](https://www.atlassian.com/git/tutorials/using-branches/git-checkout) for selecting the current branch and git branch -d for deleting the obsolete target branch.

### Usage

git merge <branch>

Merge the specified branch into the current branch. Git will determine the merge algorithm automatically (discussed below).

git merge --no-ff <branch>

Merge the specified branch into the current branch, but alwaysgenerate a merge commit (even if it was a fast-forward merge). This is useful for documenting all merges that occur in your repository.

### Discussion

Once you’ve finished developing a feature in an isolated branch, it's important to be able to get it back into the main code base. Depending on the structure of your repository, Git has several distinct algorithms to accomplish this: a fast-forward merge or a 3-way merge.

A **fast-forward merge** can occur when there is a linear path from the current branch tip to the target branch. Instead of “actually” merging the branches, all Git has to do to integrate the histories is move (i.e., “fast forward”) the current branch tip up to the target branch tip. This effectively combines the histories, since all of the commits reachable from the target branch are now available through the current one. For example, a fast forward merge of some-feature into master would look something like the following:

However, a fast-forward merge is not possible if the branches have diverged. When there is not a linear path to the target branch, Git has no choice but to combine them via a **3-way merge**. 3-way merges use a dedicated commit to tie together the two histories. The nomenclature comes from the fact that Git uses three commits to generate the merge commit: the two branch tips and their common ancestor.

While you can use either of these merge strategies, many developers like to use fast-forward merges (facilitated through [rebasing](https://www.atlassian.com/git/tutorials/rewriting-history/git-rebase)) for small features or bug fixes, while reserving 3-way merges for the integration of longer-running features. In the latter case, the resulting merge commit serves as a symbolic joining of the two branches.

#### Resolving Conflicts

If the two branches you're trying to merge both changed the same part of the same file, Git won't be able to figure out which version to use. When such a situation occurs, it stops right before the merge commit so that you can resolve the conflicts manually.

The great part of Git's merging process is that it uses the familiar edit/stage/commit workflow to resolve merge conflicts. When you encounter a merge conflict, running the [git status](https://www.atlassian.com/git/tutorials/inspecting-a-repository/git-status) command shows you which files need to be resolved. For example, if both branches modified the same section of hello.py, you would see something like the following:

# On branch master

# Unmerged paths:

# (use "git add/rm ..." as appropriate to mark resolution)

#

# both modified: hello.py

#

Then, you can go in and fix up the merge to your liking. When you're ready to finish the merge, all you have to do is run git addon the conflicted file(s) to tell Git they're resolved. Then, you run a normal git commit to generate the merge commit. It’s the exact same process as committing an ordinary snapshot, which means it’s easy for normal developers to manage their own merges.

Note that merge conflicts will only occur in the event of a 3-way merge. It’s not possible to have conflicting changes in a fast-forward merge.

### Example

#### Fast-Forward Merge

Our first example demonstrates a fast-forward merge. The code below creates a new branch, adds two commits to it, then integrates it into the main line with a fast-forward merge.

# Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Merge in the new-feature branch

git checkout master

git merge new-feature

git branch -d new-feature

This is a common workflow for short-lived topic branches that are used more as an isolated development than an organizational tool for longer-running features.

Also note that Git should not complain about the git branch -d, since new-feature is now accessible from the master branch.

#### 3-Way Merge

The next example is very similar, but requires a 3-way merge because master progresses while the feature is in-progress. This is a common scenario for large features or when several developers are working on a project simultaneously.

# Start a new feature

git checkout -b new-feature master

# Edit some files

git add <file>

git commit -m "Start a feature"

# Edit some files

git add <file>

git commit -m "Finish a feature"

# Develop the master branch

git checkout master

# Edit some files

git add <file>

git commit -m "Make some super-stable changes to master"

# Merge in the new-feature branch

git merge new-feature

git branch -d new-feature

Note that it’s impossible for Git to perform a fast-forward merge, as there is no way to move master up to new-feature without backtracking.

For most workflows, new-feature would be a much larger feature that took a long time to develop, which would be why new commits would appear on master in the meantime. If your feature branch was actually as small as the one in the above example, you would probably be better off rebasing it onto master and doing a fast-forward merge. This prevents superfluous merge commits from cluttering up the project history.

# **Merging vs. Rebasing**

[**Conceptual Overview**](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#conceptual-overview) [**The Golden Rule of Rebasing**](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#the-golden-rule-of-rebasing) [**Workflow Walkthrough**](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#workflow-walkthrough) [**Summary**](https://www.atlassian.com/git/tutorials/merging-vs-rebasing#summary)

The git rebase command has a reputation for being magical Git voodoo that beginners should stay away from, but it can actually make life much easier for a development team when used with care. In this article, we’ll compare git rebase with the related git merge command and identify all of the potential opportunities to incorporate rebasing into the typical Git workflow.

## **Conceptual Overview**

The first thing to understand about git rebase is that it solves the same problem as git merge. Both of these commands are designed to integrate changes from one branch into another branch—they just do it in very different ways.

Consider what happens when you start working on a new feature in a dedicated branch, then another team member updates the master branch with new commits. This results in a forked history, which should be familiar to anyone who has used Git as a collaboration tool.

Now, let’s say that the new commits in master are relevant to the feature that you’re working on. To incorporate the new commits into your feature branch, you have two options: merging or rebasing.

### The Merge Option

The easiest option is to merge the master branch into the feature branch using something like the following:

git checkout feature

git merge master

Or, you can condense this to a one-liner:

git merge master feature

This creates a new “merge commit” in the feature branch that ties together the histories of both branches, giving you a branch structure that looks like this:

Merging is nice because it’s a non-destructive operation. The existing branches are not changed in any way. This avoids all of the potential pitfalls of rebasing (discussed below).

On the other hand, this also means that the feature branch will have an extraneous merge commit every time you need to incorporate upstream changes. If master is very active, this can pollute your feature branch’s history quite a bit. While it’s possible to mitigate this issue with advanced git log options, it can make it hard for other developers to understand the history of the project.

### The Rebase Option

As an alternative to merging, you can rebase the feature branch onto master branch using the following commands:

git checkout feature

git rebase master

This moves the entire feature branch to begin on the tip of the master branch, effectively incorporating all of the new commits in master. But, instead of using a merge commit, rebasing re-writesthe project history by creating brand new commits for each commit in the original branch.

The major benefit of rebasing is that you get a much cleaner project history. First, it eliminates the unnecessary merge commits required by git merge. Second, as you can see in the above diagram, rebasing also results in a perfectly linear project history—you can follow the tip of feature all the way to the beginning of the project without any forks. This makes it easier to navigate your project with commands like git log, git bisect, and gitk.

<https://www.atlassian.com/git/tutorials/merging-vs-rebasing>

SVN

**1) Explain what is SVN?**

SVN or Subversion is an open source control system; it is used to trace all the changes made to your source code or files.  It is a repository used to manage files, folders, directories and the modification made to these files over a period of time.  While SVN repository provides a complete history of changes made to the files and can easily track if someone had made changes in the file.

**2) Explain what is the difference between commit and update?**

Update is used to update the local workspace with the changes made by the team to the repository, while commit is the process to implement changes from local to repository, in simple words, upload a file into repository.

**3) List out what all things should be stored in SVN repository?**

In SVN repository you can store

* Source Code
* Build scripts
* Test data used by QA
* DB schema
* Project settings (When whole team is using the same IDE)
* Project documentation (Internal and External)
* Minutes of meetings, significant e-mails and info from the web
* Expensively generated artifacts
* And other documents related to the project

**4) Explain how you can apply a patch in SVN?**  
  
To apply a patch in SVN, you are required to “Create Patch” by making changes and generating the .diff file.  Then this .diff file can be implemented to the new code base using “Apply Patch” .

**5) Mention what does the result codes G and R in svn indicates?**  
  
The result codes G and R in svn indicates  
  
 G code: Changes on the repo were automatically merged into the working copy  
 R code: This code indicates that item has been replaced in your working copy. This means the file was programmed or scheduled for deletion, and a new file with the same name was scheduled for addition in its place

**6) Mention what is the function of Revert in subversion?**  
  
“Revert” function will remove your local changes and reload the latest version from the repository.