**GIT**

**Git Interview Questions:**

1. **What is git -** Repository
2. **Diff bet git and svn**
3. **Commit command**
4. **Bare repository in git**
5. **Language used in Git – C language**
6. **How do you revert commit that has been already pushed and mase public**
7. **Difference Git Pull and Git fetch**
8. **Staging area / Index**
9. **Git stash**
10. **Git stash drop**
11. **Function of git config**
12. **How can you create a repository in Git?** **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..

1. **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

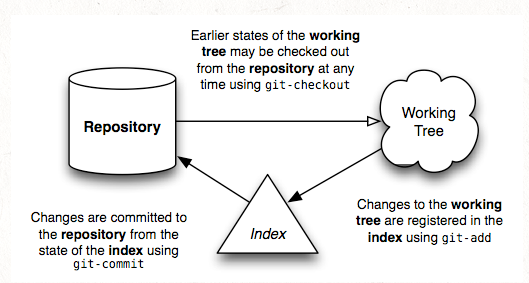
1. **What is SubGit?**

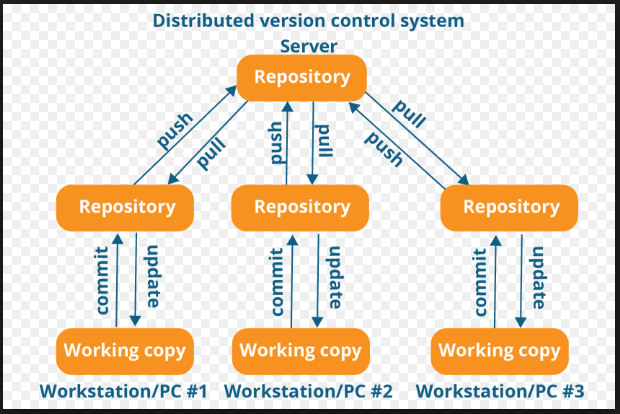
* Begin this answer by explaining what is SubGit used for.
* SubGit is a tool for SVN to Git migration. It creates a writable Git mirror of a local or remote Subversion repository and uses both Subversion and Git as long as you like.

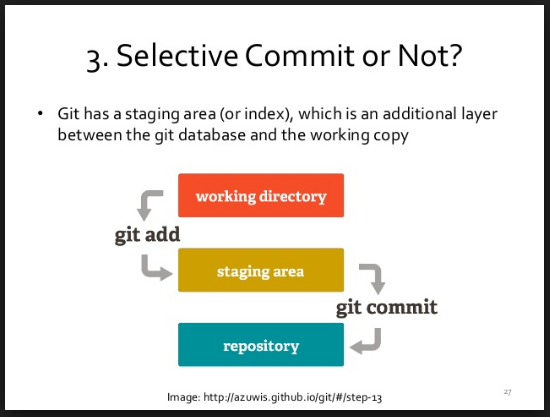
1. **What is Git rebase and how can it be used to resolve conflicts in a feature branch before merge?**

* According to me you should start by saying git rebase 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.
* Now, once you have defined Git rebase time for an example to show how it can be used to resolve conflicts in a feature branch before merge.
* If a feature branch was created from the master, and since then the master branch has received new commits, Git rebase can be used to move the feature branch to the tip of master. The command effectively will replay the changes made in the feature branch at the tip of master, allowing conflicts to be resolved in the process. When done with care, this will allow the feature branch to be merged into master with relative ease and sometimes as a simple fast-forward operation.
* You can also expect some off track questions, so the next question in this Git interview questions blog will be regarding SubGit.

**GIT Lifecycle**







**Git Bash commands:**

**Installation and initial setup:**

1. Download latest version of Git for appropriate OS .
2. Install git by fallowing **Next** and **Finish.**
3. Open cmd or Git bash (if you elected Git Bash while installation)
4. To verify git installation run command

$ git --version

Output will be -- git version 2.14.3.windows.1

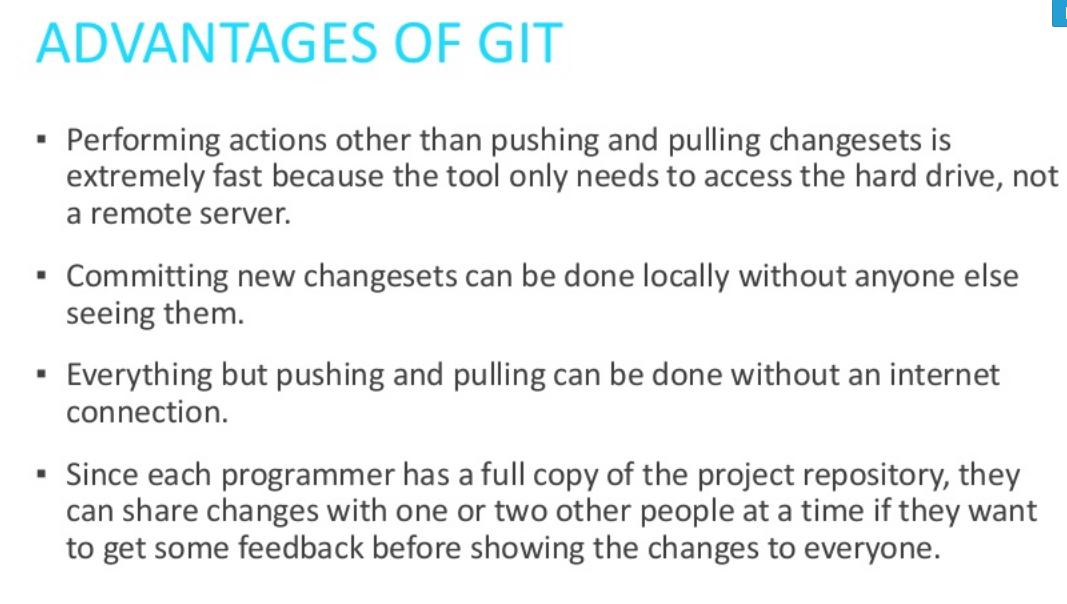
**Start git at new folder first time:**

Run the command to configure Git username and email with the help of following command

**$ git init** == to initiate git first time which create .git folder which is hidden.

**$ git config --global user.name “MayaAmol”**

**$ git config --global user.email** [**mayabcs2010@gmail.com**](mailto:mayabcs2010@gmail.com)



**Commands**

1. Create new repository

**$ git init**

1. Checkout repository

Create working copy of local repository

**$ git clone URL of repo**

e.g. **$ git clone https://github.com/MayaAmol/Devops-QA.git**

When using remote server

**$ git clone username@host:URL of repo**

1. Workflow

Local repository consist of three “trees” maintained by git

1. Working directory – holds the actual filses

**$ git add <filename>**

--- add file to the Index area

--- first step in the basic git work flow.

1. Index – which acts as staging area

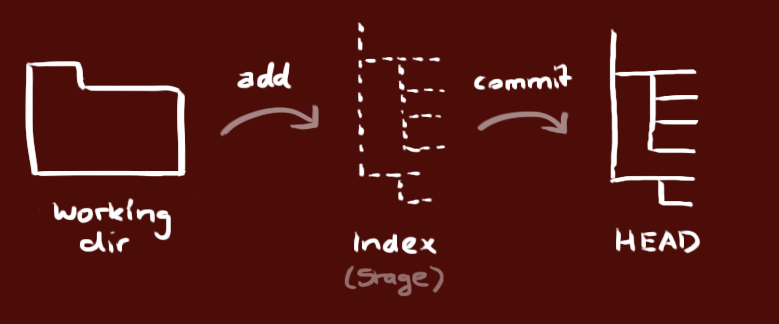
**$ git status**

--- To see the status of added and uncommitted files.

1. HEAD – points to last commit you have

**$ git commit –m “Commit Message”**

--- Now the file is committed to the HEAD, but not in your remote repository yet.



1. Push Change

Your changes now in HEAD of your local working copy. To send those changes to your remote repository execute

**Git push origin master**

Change master branch to whatever branch you want to push your changes to.

If you have not cloned an existing changes and want to connect your repository to remote server , you need to add it with

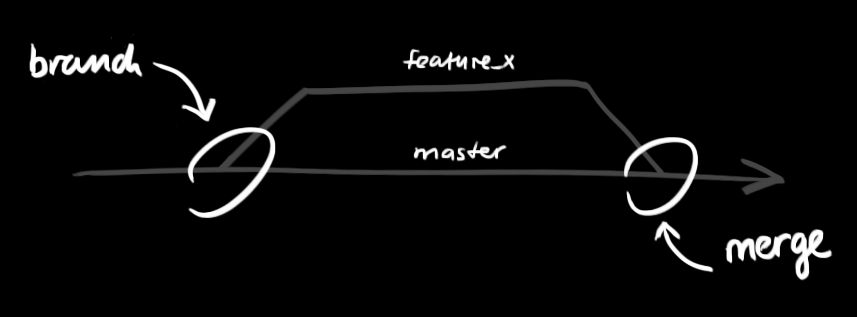
**$ git remote add origin <server>**

Now you are able to push your changes to selected remote server

1. Branching

Branches are used to develop features isolated from each other .

The master branch is the default branch when you create a repository. Use othe branches for development and merge them back to the master branch upon completion.



Create a new branch named “Branch1” and switch to it using command

**$ git checkout –b Branch1**

Switch back to master

**Git checkout master**

And delete branch again

**$ git branch –d branch1**

A branch is not available to others unless you push the branch to your remote repository.

**$ git push origin Branch1**

1. Update and merge

To update your local repository to the newest commit, run command

**$ git pull**

in your working dir to fetch and merge remote changes.

To merge another branch into your active branch (e.g. master) use

**$ git merge <branch>**

In both cases git tries to auto merge changes. Unfortunitly, this is not always possible and result in conflict. You are responsible to merge those conflict manually by editing file shown by git. After changing , you need to mark them as merged with

**$ git add <filename>**

Before merging changes, you can also preview them by using

**$ git diff <source-branch> <target-branch>**

1. Tagging

Its recommended to create tags for software release. You can create new tag named 1.0.0 by executing command

**$ git tag 1.0.0 1hkdg3ytdr**

The **1hkdg3ytdr** stands for first 10 character of commit id you want to reference with your tag. You can get the commit id by looking at.

1. Log

You can study the repository history using

**$ git log**

You can add lot of parameters to make log look like what you want

To see only commits of certain author

**$ git log --author =bob**

To see very compressed log where each commit is one line

**$ git log --pretty=oneline**

Or may be you want to see ASCII art of “TREE” of all the branches, decorate with the name of tags and branches

**$ git log --graph --oneline --decorate --all**

To see only which file have changed

**$ git log --name-status**

For more **$ git log --help**

1. Replace local changes

In case you did something wrong which for sure never happen, you can replace local changes using the command

**& git -- <filename>**

This replaces the changes in your working tree with the last content in HEAD. Changes already added to INDEX, as well as new files, will be kept.

If you instead want to drop all your local changes and commit, fetch the latest history from server and point your local master branch as like it

**$ git fetch origin**

**$ git reset --hard origin/master**

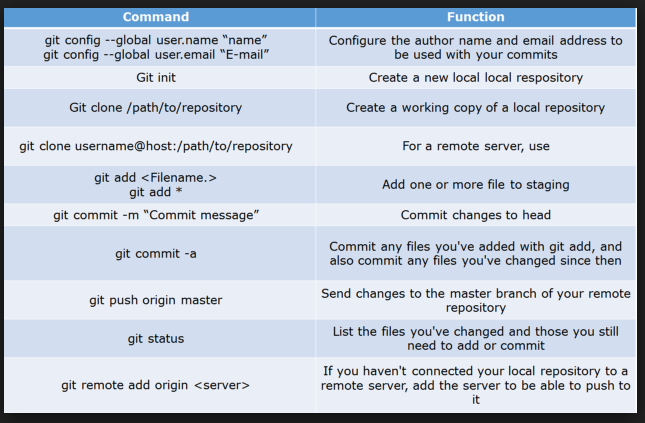
1. Useful Hints

Built-in git GUI – **gitk**

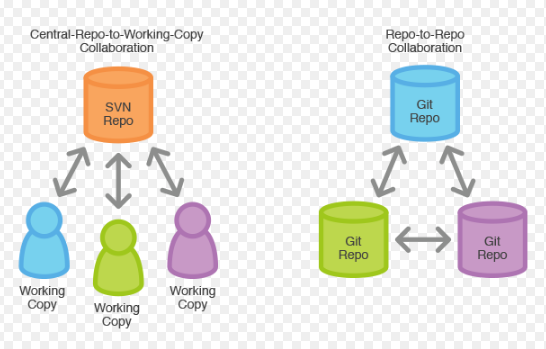
Use colorful git output – **git config color.ui true**

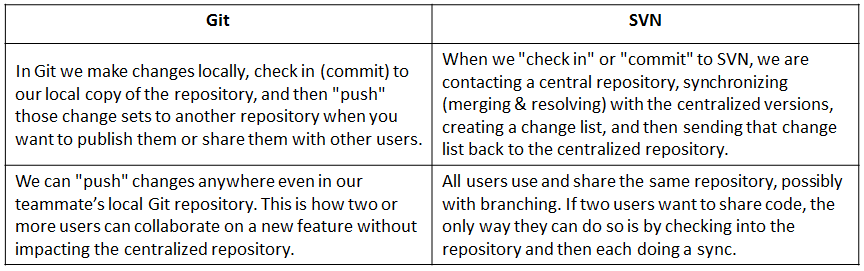
Show log on just one line per commit – **git config format.pretty oneline**

Use interactive adding – **git add -i**

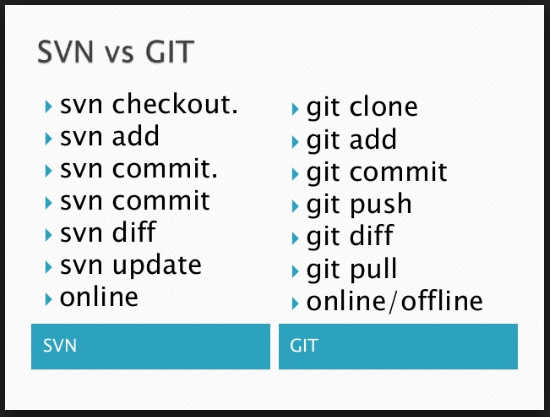


**Git and SVN difference**





|  |  |  |
| --- | --- | --- |
| **Parameters** | **SVN (Centralized)** | **Git(Distributed)** |
| **Version management** | Central | Distributed |
| **Repository** | A central repository in which working copies are generated  i.e only one repo | Locally available repository copies in which files and documents can be worked on  i.e many repo |
| **Access authorization** | Path-based | For the entire directory |
| **Speed** | Slow | fast |
| **Change history** | Only complete in the repository. Working copies only contain the newest version | Repository and working copies contain the complete history |
| **Network connection(Offline)** | Needed for access(Difficult) | Only necessary for synchronization(easy) |
| **Redundancy** | None, single point of failuare | Built in coz every one have whole copy of project |
| **Model** | Client server | Peer to peer |
| **Commits** | Centralized, public | Private, local |
| **Adv.** | * Automic commit * Fast and flexible update/commit * Ease of setup and administrate | * Fast - very fast. * Creating a new repo is very easy compared to SVN * The full repo is contained in just one .git folder - it will not add a .SVN folder in every folder of your code(not a big deal, but I like it) * Branching is easier |
| **Dis Adv.** | * Centralized copy – need to connect to central copy if you have to access branches or commit changes | * Cannot checkout a part of the repository(like just one folder or just one file) * Does not track empty folders * Bad Windows support(does not bother me much - I use Linux) * I still have not found a good GUI tool for Git yet |



Here are the advantages of each of both systems:

**Git should be used if …**

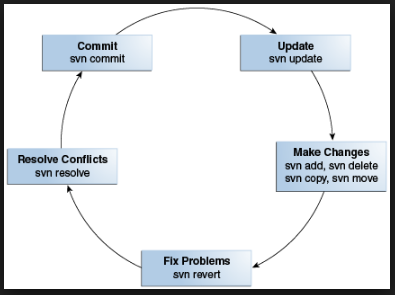
* you don’t want to be dependent on a constant network connection in order to work on all parts of your project
* you want to protect yourself from a loss or malfunction of the main repository
* you don’t need read or write permissions for special directories
* you‘re interested in quickly transferring the changes made to documents or files

**Subversion should be used if…**

* you need path-based access rights for different areas of your project
* you wish to bundle all of your work to one centrally located place
* you’re working with many large binary files
* you wish to fully register the structures of empty directories (Git disposes of these, as they contain no content)

**SVN Basics**

**LifeCycle**



**Version Control Terminologies**

Let us start by discussing some of the terms that we will be using in this tutorial.

* **Repository:** A repository is the heart of any version control system. It is the central place where developers store all their work. Repository not only stores files but also the history. Repository is accessed over a network, acting as a server and version control tool acting as a client. Clients can connect to the repository, and then they can store/retrieve their changes to/from repository. By storing changes, a client makes these changes available to other people and by retrieving changes, a client takes other people's changes as a working copy.
* **Branches:** Branch operation is used to create another line of development. It is useful when you want your development process to fork off into two different directions. For example, when you release version 5.0, you might want to create a branch so that development of 6.0 features can be kept separate from 5.0 bug-fixes.
* **Working copy:** Working copy is a snapshot of the repository. The repository is shared by all the teams, but people do not modify it directly. Instead each developer checks out the working copy. The working copy is a private workplace where developers can do their work remaining isolated from the rest of the team.
* **Commit changes:** Commit is a process of storing changes from private workplace to central server. After commit, changes are made available to all the team. Other developers can retrieve these changes by updating their working copy. Commit is an atomic operation. Either the whole commit succeeds or is rolled back. Users never see half-finished commit.

**Difference between Tags/branch/trunk**

- A **trunk** in SVN is main development area, where major development happens.

- A **branch** in SVN is sub development area where parallel development on different functionalities happens. After completion of a functionality, a branch is usually merged back into trunk.

- A **tag** in SVN is read only copy of source code from branch or tag at any point of time. tag is mostly used to create a copy of released source code for restore and backup.

**SVN Branching And Merge**

**Steps**

* Create a new branch and switch to it
* Develop in your branch, periodically merging the latest changes from trunk into your branch
* When your development is complete, merge changes from your branch back into trunk
* Delete your branch
  1. **Switch**

First off, ensure that your working copy has no local changes.  Either commit your changes or revert them.

* 1. **Create New Branch**

First, switch your local copy to trunk (if it’s not already trunk).  And again, make sure you have no local changes.

Create new branch

* 1. **Merge Trunk into Branch**

you should periodically merge the latest version of trunk into your branch to make sure you’re not straying too far away.

* 1. **Merge Branch into Trunk**
     + - When you’ve finished development in your branch, you need to merge your branch changes back into trunk, so everyone else can share in the glory of your awesome new feature
       - Make sure you only perform this merge when you are completely finished with your branch!  After merging your branch into trunk, you will no longer be able to use your branch. In fact, you should delete your branch after merging it into trunk
  2. **Delete Branch:**

 after merging a branch into trunk, the branch can no longer be used.  Subversion keeps track of merges using the svn:mergeinfo property. Because of the way it uses this property, once a branch is merged into trunk it can no longer be involved in merges properly.

**Which Branching Strategy you are using?**