**What is Version control?**

It is a system that records changes to a file or set of files over time so that you can recall specific versions later. Version control systems consist of a central shared repository where teammates can commit changes to a file or set of files. Then you can mention the uses of version control.

Version control allows you to:

* Revert files back to a previous state.
* Revert the entire project back to a previous state.
* Compare changes over time.
* See who last modified something that might be causing a problem.
* Who introduced an issue and when?

**What are the benefits of using version control?**

The following advantages of version control:

1. With Version Control System (VCS), all the team members are allowed to work freely on any file at any time. VCS will later allow you to merge all the changes into a common version.
2. All the past versions and variants are neatly packed up inside the VCS. When you need it, you can request any version at any time and you’ll have a snapshot of the complete project right at hand.
3. Every time you save a latest version of your project, your VCS requires you to provide a little description of what was changed. Additionally, you can see what exactly was changed in the file’s content. This allows you to know who has made what change in the project.
4. A distributed VCS like Git allows all the team members to have 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.

**Describe branching strategies you have used.**

* Feature branching

A feature branch model keeps all the changes for a specific 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.

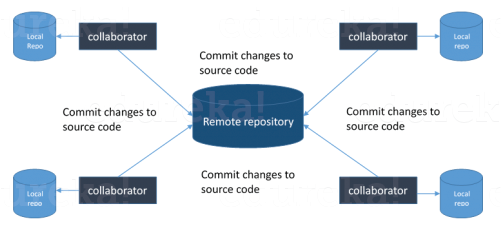
**Which VCS tool you are comfortable with?**

“I have worked on Git and one major advantage it has over other VCS tools like SVN is that it is a distributed version control system.”

Distributed VCS tools do not necessarily rely on a central server to store all the versions of a project’s files. Instead, every developer “clones” a copy of a repository and has the full history of the project on their own hard drive.

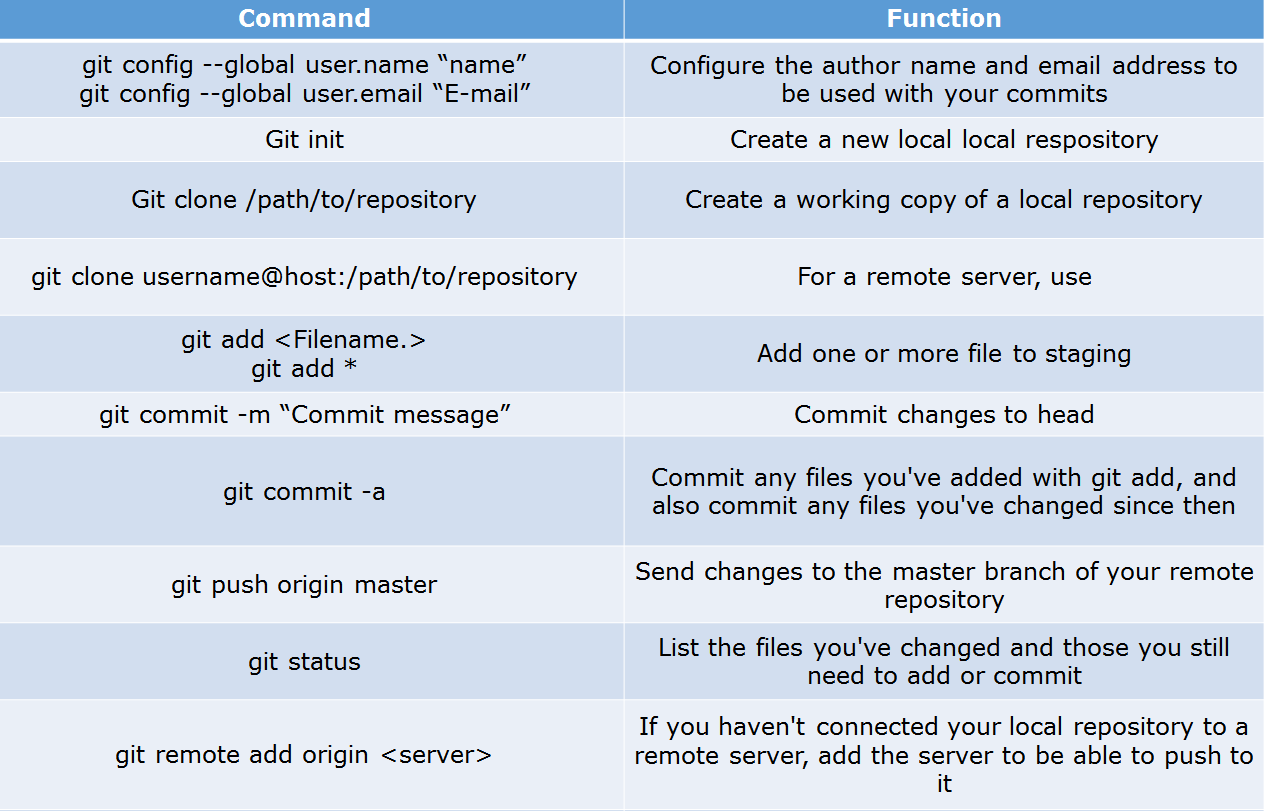
**What is Git?**

* Git is a Distributed Version Control system (DVCS). It can track changes to a file and allows you to revert back to any particular change.
* 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 below with “Local repository” and has the full history of the project on his hard drive so that 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 committing changes “Remote repository”.



**Explain some basic Git commands?**

Below are some basic Git commands:



**In Git how do you revert a commit that has already been pushed and made public?**

* Remove or fix the bad file in a new commit and push it to the remote repository. This is the most natural way to fix an error. Once you have made necessary changes to the file, commit it to the remote repository for that I will use  
  **git commit -m “commit message”**
* Create a new commit that undoes all changes that were made in the bad commit.to do this I will use a command  
  **git revert <name of bad commit>**

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

There are two options to squash 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})”**

**What does Git bisect? How can you use it to determine the source of a (regression) bug?**

Git bisect is used to find the commit that introduced a bug by using binary search. Command for Git bisect is  
**git bisect <subcommand> <options>**

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.

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

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

can be used to resolve conflicts in a feature branch before merge, if a feature branch was created from 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.

**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, A sanity or smoke testdetermines 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 pre-commit 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.

**How do you find a list of files that has changed in a particular commit?**

To get a list files that has changed in a specific commit use 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 mention 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 –name-only will only print the file names, instead of their paths.

**How do you setup a script to run every time a repository receives new commits through push?**

There are three ways to configure a script to run every time a repository receives new commits through push, one needs to define either a pre-receive, update, or a post-receive hook depending on when exactly the script needs to be triggered.

* Pre-receive hook in the destination repository is invoked when commits are pushed to it. Any script bound to this hook will be executed before any references are updated. This is a useful hook to run scripts that help enforce development policies.
* Update hook works in a similar manner to pre-receive hook, and is also triggered before any updates are actually made. However, the update hook is called once for every commit that has been pushed to the destination repository.
* Finally, post-receive hook in the repository is invoked after the updates have been accepted into the destination repository. This is an ideal place to configure simple deployment scripts, invoke some continuous integration systems, dispatch notification emails to repository maintainers, etc.

Hooks are local to every Git repository and are not versioned. Scripts can either be created within the hooks directory inside the “.git” directory, or they can be created elsewhere and links to those scripts can be placed within the directory.

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

I will suggest you include both the below mentioned commands:

git branch –merged => lists the branches that have been merged into the current branch.  
git branch –no-merged => lists the branches that have not been merged.

**What are the main advantages of Git over CVS?**

The biggest advantage is that Git is distributed while CVS is centralized. Changes in CVS are per file, while changes (commits) in Git they always refer to the whole project. Git offers much more tools than CVS.

**Potential Git Workflows**

When choosing a Git workflow, it is important to consider your team's needs. A simple workflow can maximize development speed and flexibility, while a more complex workflow can ensure greater consistency and control of work in progress. You can adapt and combine the general approaches listed below to suit your needs and the separate roles on your team. A core developer might use feature branches while a contractor works from a fork, for example.

* A **centralized workflow** provides the closest match to common SVN processes, so it's a good option to get started.
* Building on that idea, using a **feature branch workflow** lets developers keep their work in progress isolated and important shared branches protected. Feature branches also form the basis for managing changes via pull requests.
* A **Gitflow workflow** is a more formal, structured extension to feature branching, making it a great option for larger teams with well-defined release cycles.
* Finally, consider a **forking workflow** if you need maximum isolation and control over changes, or have many developers contributing to one repository.

But, if you really want to get the most out of Git as a professional team, you should consider the feature branch workflow. This is a truly distributed workflow that is highly secure, incredibly scalable, and quintessentially agile.

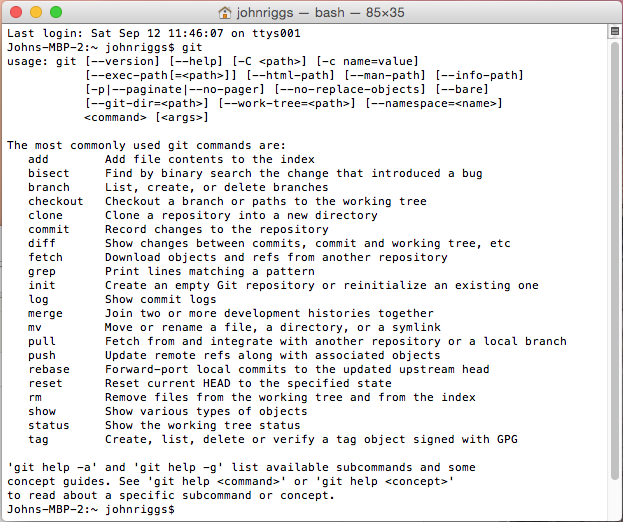
**SVN to Git Migration**

SVN to Grit migration process into 5 simple steps and they are as follows:

* Prepare the environment for the migration
* Convert SVN repository to local Git Repository
* Synchronize the local Git repository when the SVN repository changes
* Share the Git repository with your developer via Bitbucket.
* Migrate your development efforts from SVN to Git.

The prepare, convert, and synchronize steps take a SVN commit history and turn it into a Git repository.

After the synchronize phase, the migration lead should have no trouble keeping a local Git repository up-to-date with an SVN counterpart. To share the Git repository, the migration lead can share his local Git repository with other developers by pushing it to Bitbucket, a Git hosting service.



Create a Repository

From scratch -- Create a new local

repository

**$ git init [project name]**

Download from an existing repository

**$ git clone my\_url**

Make a change Synchronize

Observe your Repository

List new or modified files not yet

committed

**$ git status**

Show the changes to files not yet staged

**$ git diff**

Show the changes to staged files

**$ git diff --cached**

Show all staged and unstaged

file changes

**$ git diff HEAD**

Show the changes between two

commit ids

**$ git diff commit1 commit2**

List the change dates and authors

for a file

**$ git blame [file]**

Show the file changes for a commit

id and/or file

**$ git show [commit]:[file]**

Show full change history

**$ git log**

Show change history for file/directory

including diffs

**$ git log -p [file/directory]**

Working with Branches

List all local branches

**$ git branch**

List all branches, local and remote

**$ git branch -av**

Switch to a branch, my\_branch,

and update working directory

**$ git checkout my\_branch**

Create a new branch called new\_branch

**$ git branch new\_branch**

Delete the branch called my\_branch

**$ git branch -d my\_branch**

Merge branch\_a into branch\_b

**$ git checkout branch\_b**

**$ git merge branch\_a**

Tag the current commit

**$ git tag my\_tag**

Stages the file, ready for commit

**$ git add [file]**

Stage all changed files, ready for commit

**$ git add .**

Commit all staged files to versioned history

**$ git commit -m “commit message”**

Commit all your tracked files to

versioned history

**$ git commit -am “commit message”**

Unstages file, keeping the file changes

**$ git reset [file]**

Revert everything to the last commit

**$ git reset –hard**

Get the latest changes from origin

(no merge)

**$ git fetch**

Fetch the latest changes from origin

and merge

**$ git pull**

Fetch the latest changes from origin

and rebase

**$ git pull --rebase**

Push local changes to the origin

**$ git push**