GIT INTERVIEW QUESTIONS & ANSWERS PART 1

FAQ

1) What is the command to write a commit message in Git?

```
git commit -a
```

-a on the command line instructs git to commit the new content of all tracked files that have been modified. You can use:

```
git add <file>
```

or

```
git add -A
```

before git commit -a if new files need to be committed for the first time.

2) What is difference between Git vs SVN?

The main point in Git vs SVN debate boils down to this: Git is a distributed version control system (DVCS), whereas SVN is a centralized version control system.

3) 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.

FAQ

4) What is the difference between "git pull" and "git fetch"?

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

- 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.

5) What's the difference between a "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).

6) What is Git fork? What is difference between fork, branch and clone?

- A fork is a remote, server-side copy of a repository, distinct from the original. A fork isn't a Git concept really, it's more a political/social idea.

- A clone is not a fork; a clone is a local copy of some remote repository. When you clone, you are actually copying the entire source repository, including all the history and branches.

- 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.

FAQ

7) How does the Centralized Workflow work?

The Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. The default development branch is called master and all changes are committed into this branch.

Developers start by cloning the central repository. In their own local copies of the project, they edit files and commit changes. These new commits are stored locally.

To publish changes to the official project, developers push their local master branch to the central repository. Before the developer can publish their feature, they need to fetch the updated central commits and rebase their changes on top of them.

Compared to other workflows, the Centralized Workflow has no defined pull request or forking patterns.

8) Tell me the difference between HEAD, working tree and index, in Git?

- The working tree/working directory/workspace is the directory tree of (source) files that you see and edit.

- The index/staging area is a single, large, binary file in /.git/index, which lists all files in the current branch, their sha1 checksums, time stamps and the file name - it is not another directory with a copy of files in it.

- HEAD is a reference to the last commit in the currently checked-out branch.

9) Could you explain the Gitflow workflow?

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

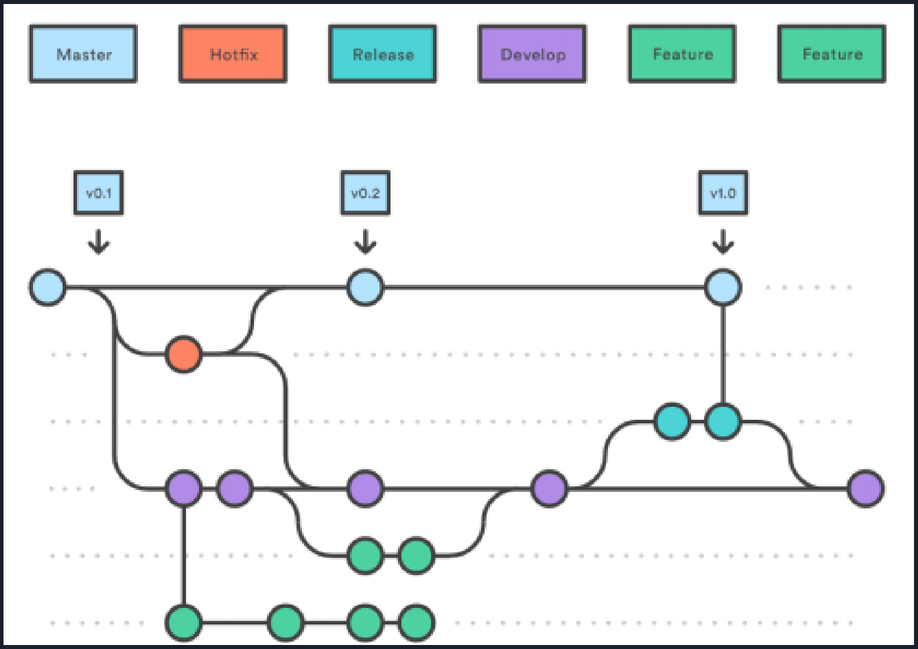
- Master - is always ready to be released on LIVE, with everything fully tested and approved (productionready).

- Hotfix - Maintenance or "hotfix" branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on master instead of develop .

FAQ

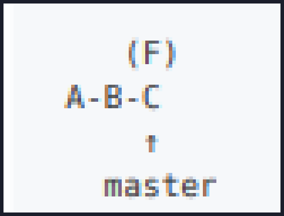
- Develop – is the branch to which all feature branches are merged and where all tests are performed. Only when everything's been thoroughly checked and fixed it can be merged to the master.

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



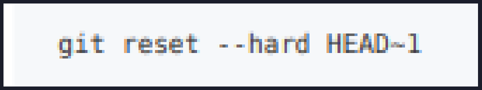
10) How to revert previous commit in git?

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



```
    (F)
A-B-C
    ↑
  master
```

1. 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.

2. 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.

3. 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.

11) Explain the advantages of Forking Workflow

The Forking Workflow is fundamentally different than other popular Git workflows. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer their own server-side repository. The Forking Workflow is most often seen in public open source projects.

The *main advantage* of the Forking Workflow is that contributions can be integrated without the need for everybody to push to a single central repository that leads to a clean project history. Developers push to their
own server-side repositories, and only the project maintainer can push to the official repository.

When developers are ready to publish a local commit, they push the commit to their own public repository—not the official one. Then, they file a pull request with the main repository, which lets the project maintainer know that an update is ready to be integrated.

FAQ

12) What is "git cherry-pick"?

The command git cherry-pick is typically used to introduce particular commits from one branch within a repository onto a different branch. A 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>
```

13) What is a "bare git" repository?

- It is a repository that is created without a working tree.
- To initializes the bare repository.

```
git init --bare .
```

- It is typically used as a remote repository that is sharing a repository among several different people.
- It only contains the repository data(refs, commits) and nothing else.

14) When should I use "git stash"?

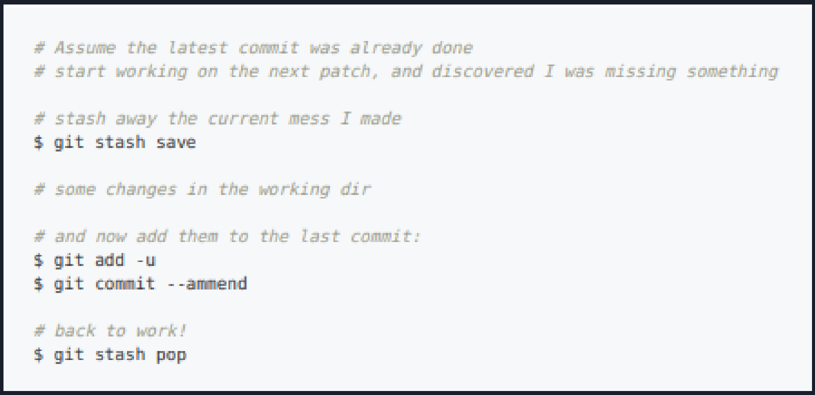
The git stash command takes your uncommitted changes (both staged and unstaged), saves them away for later use, and then reverts them from your working copy.

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
```

FAQ

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:



```
# Assume the latest commit was already done
# start working on the next patch, and discovered I was missing something

# stash away the current mess I made
$ git stash save

# some changes in the working dir

# and now add them to the last commit:
$ git add -u
$ git commit --ammend

# back to work!
$ git stash pop
```

15) What is the difference between "git stash pop" and "git stashapply"?

git stash pop throws away the (topmost, by default) stash after applying it, whereas git stash apply leaves it in the stash list for possible later reuse (or you can then git stash drop it).
This happens unless there are conflicts after git stash pop (say your stashed changes conflict with other changes that you've made since you first created the stash), in this case, it will not remove the stash, behaving
exactly like git stash apply .
Another way to look at it: git stash pop is git stash apply & git stash drop .

16) What is the "HEAD" in Git?

HEAD is a ref (reference) to the currently checked out commit.
In normal states, it's actually a symbolic ref to the branch you have checked out - if you look at the contents of .git/HEAD you'll see something like "ref: refs/heads/master". The branch itself is a reference to the commit at the tip of the branch. Therefore, in the normal state, HEAD effectively refers to the commit at the tip of the current branch.

It's also possible to have a detached HEAD . This happens when you check out something besides a (local) branch, like a remote branch, a specific commit, or a tag. The most common place to see this is during an interactive rebase, when you choose to edit a commit. In detached HEAD state, your HEAD is a direct reference to a commit - the contents of .git/HEAD will be a SHA1 hash.

Generally speaking, HEAD is just a convenient name to mean "what you have checked out" and you don't really have to worry much about it. Just be aware of what you have checked out, and remember that you probably don't want to commit if you're not on a branch (detached HEAD state) unless you know what you're doing (e.g.
are in an interactive rebase).

FAQ

17) How do you make an existing repository bare?

Run the command below:

```
git clone --mirror <repo_source_path>
```

The *--mirror* flag maps all *refs* (including remote-tracking branches, notes etc.) and sets up a refspec configuration such that all these refs are overwritten by a git remote update in the target repository

18) Can you explain what "git reset" does in plain english?

In general, git reset function is to take the current branch and reset it to point somewhere else, and possibly bring the index and work tree along.

```
- A - B - C (HEAD, master)
# after git reset B (--mixed by default)
- A - B (HEAD, master)      # - C is still here (working tree didn't change state), but there's no branch
pointing to it anymore
```

Remeber that in git you have:

- The HEAD pointer, which tells you what commit you're working on.
- The working tree, which represents the state of the files on your system.
- The staging area (also called the index), which "stages" changes so that they can later be committed together.

So consider:

- git reset --soft moves HEAD but doesn't touch the staging area or the working tree.
- git reset --mixed moves HEAD and updates the staging area, but not the working tree.
- git reset --merge moves HEAD, resets the staging area, and tries to move all the changes in your working tree into the new working tree.
- git reset --hard moves HEAD and adjusts your staging area and working tree to the new HEAD, throwing away everything

FAQ

Use cases:

Use --soft when you want to move to another commit and patch things up without "losing your place". It's
pretty rare that you need this.
Use --mixed (which is the default) when you want to see what things look like at another commit, but you don't want to lose any changes you already have.
Use --merge when you want to move to a new spot but incorporate the changes you already have into that the working tree.
Use --hard to wipe everything out and start a fresh slate at the new commit.

19) When would you use "git clone" over "git clone --mirror" ?

Suppose origin has a few branches (master (HEAD) , next , pu , and maint), some tags (v1 , v2 , v3), some remote branches (devA/master , devB/master), and some other refs (refs/foo/bar , refs/foo/baz , which might be notes, stashes, other devs' namespaces, who knows).

git clone (non-bare): You will get all of the tags copied, a local branch master (HEAD) tracking a remote branch origin/master , and remote branches origin/next , origin/pu , and origin/maint . The tracking branches are set up so that if you do something like git fetch origin , they'll be fetched as you expect. Any remote branches (in the cloned remote) and other refs are completely ignored.

git clone --mirror : Every last one of those refs will be copied as-is. You'll get all the tags, local branches master (HEAD) , next , pu , and maint , remote branches devA/master and devB/master , other refs refs/foo/bar and refs/foo/baz . Everything is exactly as it was in the cloned remote. Remote tracking is set up so that if you run git remote update all refs will be overwritten from origin, as if you'd just deleted the mirror and recloned it. As the docs originally said, it's a mirror. It's supposed to be a functionally identical copy, interchangeable with the original.

To summarize,

- git clone is used when we need to point to an existing repo and make a clone or copy of that repo at in a new directory.
- git clone --mirror is used to clone all the extended refs of the remote repository, and maintain remote branch tracking configuration.

FAQ

20) When would you use "git clone" over "git clone --bare" ?

Suppose origin has a few branches *(master (HEAD) , next , pu , and maint)*, some tags *(v1 , v2 , v3),* some remote branches *(devA/master , devB/master)*, and some other refs (*refs/foo/bar , refs/foo/baz , which might be notes, stashes, other devs'* namespaces, who knows).

- git clone (non-bare): You will get all of the tags copied, a local branch master (HEAD) tracking a remote branch origin/master , and remote branches origin/next , origin/pu , and origin/maint . The tracking branches are set up so that if you do something like git fetch origin , they'll be fetched as you expect. Any remote branches (in the cloned remote) and other refs are completely ignored.

- git clone --bare : You will get all of the tags copied, local branches master (HEAD) , next , pu , and maint , no remote tracking branches. That is, all branches are copied as is, and it's set up completely independent, with no expectation of fetching again. Any remote branches (in the cloned remote) and other refs are completely ignored.

To summarize:

- git clone is used when we need to point to an existing repo and make a clone or copy of that repo at in a new directory.

- git clone --bare is used to make a copy of the remote repository with an *omitted working directory*.