GIT INTERVIEW QUESTIONS & ANSWERS FOR DEVOPS

SCENARIO: PUSHING CODE TO GIT

You have a folder which has a calculator.sh file which you want to push to Git.

Two ways to do this:

1. Through UI on GitHub (web interface)
2. Via CLI (Command Line Interface)

We will learn the CLI approach.

Q1. HOW TO CREATE AND INITIALIZE A GIT REPOSITORY?

Command:

git init

Explanation:

- git init: Initializes a new Git repository in the current directory
- Creates a hidden .git folder
- .git folder contains all the essential information which Git uses to keep track of the changes made in the repository

What happens:

Before git init:

my-project/

calculator.sh

After git init:

my-project/

.git/ (hidden folder)

calculator.sh

The .git folder contains:

- config: Configuration settings
- objects/: All commits, files, and content
- refs/: References to branches and tags
- HEAD: Pointer to current branch
- hooks/: Scripts for automation

Example:

cd my-project

git init

Output:

Initialized empty Git repository in /path/to/my-project/.git/

Q2. HOW TO CHECK STATUS OF FILES?

Command:

git status

Explanation: Used to check the status of tracked and untracked files.

What it shows:

- Untracked files (shown in RED): Files Git is not watching
- Tracked files (shown in GREEN): Files Git is watching
- Modified files: Files that have been changed
- Staged files: Files ready to be committed

Example:

git status

Output:

On branch main

Untracked files:

(use "git add <file>..." to include in what will be committed)

calculator.sh

nothing added to commit but untracked files present (use "git add" to track)

After adding file:

On branch main

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: calculator.sh

Q3. HOW TO TRACK FILES?

Command:

git add .

OR

git add calculator.sh

Explanation: Used to track the files or folders present inside the repository.

Options:

- git add filename: Add specific file
- git add .: Add all files in current directory
- git add -A: Add all files in entire repository

What it does: Moves files from "Untracked" to "Staged" area, ready for commit.

Example:

git add calculator.sh

OR add all files:

git add .

Verify:

git status

Output:

Changes to be committed:

(use "git restore --staged <file>..." to unstage)

new file: calculator.sh

Q4. HOW TO CHECK CHANGES IN CODE?

Command:

git diff

Explanation: Used to check changes in the code between previous and present modified file.

Shows:

- Lines added (shown with + sign)
- Lines removed (shown with - sign)
- Which file was modified
- Exact changes made

Example:

Original calculator.sh:

x=a+b

Modified calculator.sh:

x=a+b

y=a-b

Run:

git diff

Output:

diff --git a/calculator.sh b/calculator.sh

index abc123..def456 100644

--- a/calculator.sh

+++ b/calculator.sh

@@ -1 +1,2 @@

x=a+b

+y=a-b

Other useful diff commands:

git diff filename: Check changes in specific file

git diff --staged: Check changes in staged files

git diff branch1..branch2: Compare two branches

git diff commit1 commit2: Compare two commits

Q5. HOW TO COMMIT CHANGES?

Command:

git commit -m "my first commit"

Explanation:

- git commit: Save the current state as a version
- -m: Message flag
- "my first commit": Commit message describing what you did

Why commit messages are important:

- Describe what changed
- Help team understand your work
- Make it easy to find specific changes later

Good commit messages:

git commit -m "Add addition and subtraction functions"

git commit -m "Fix bug in payment gateway"

git commit -m "Update user authentication logic"

Bad commit messages:

git commit -m "changes"

git commit -m "update"

git commit -m "fix"

Example:

git commit -m "Add calculator with basic operations"

Output:

[main abc1234] Add calculator with basic operations

1 file changed, 2 insertions(+)

create mode 100644 calculator.sh

Q6. HOW TO CHECK COMMIT HISTORY?

Command:

git log

Explanation: Used to check all the commit history.

Shows:

- Commit ID (unique identifier)
- Author name and email
- Date and time of commit
- Commit message

Example:

git log

Output:

commit def567890abcdef (HEAD -> main)

Author: Sushmita Hubli <sushmita@example.com>

Date: Wed Nov 6 15:30:45 2024 +0530

Add subtraction function

commit abc123456789abc

Author: Sushmita Hubli <sushmita@example.com>

Date: Wed Nov 6 15:25:30 2024 +0530

Add calculator with basic operations

Other useful log commands:

git log --oneline: Compact view (one line per commit)

git log -n 5: Show last 5 commits

git log --author="Sushmita": Show commits by specific author

git log --since="2024-01-01": Show commits after specific date

git log filename: Show commits for specific file

Example with --oneline:

git log --oneline

Output:

def5678 Add subtraction function

abc1234 Add calculator with basic operations

Q7. HOW TO PUSH CODE TO REMOTE REPOSITORY?

Command:

git push

Explanation: Put my entire code to repository on GitHub/GitLab or Bitbucket, etc.

What it does: Uploads your local commits to the remote repository (GitHub, GitLab, Bitbucket).

Full command:

git push origin main

Breakdown:

- git push: Upload command
- origin: Name of remote repository (default name)
- main: Branch name

Example:

git push origin main

Output:

Enumerating objects: 5, done.

Counting objects: 100% (5/5), done.

Writing objects: 100% (3/3), 280 bytes | 280.00 KiB/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/username/calculator.git

abc1234..def5678 main -> main

Q8. WHAT IS THE GIT WORKFLOW USED IN YOUR ORGANIZATION?

Answer:

git add . && git commit -m "message" && git push

Explanation: These are the commands we use day in and day out.

Breakdown:

1. git add .: Stage all changes
2. &&: AND operator (run next command if previous succeeds)
3. git commit -m "message": Commit with message
4. &&: Run next command
5. git push: Upload to remote repository

Alternative (separate commands):

git add .

git commit -m "Add new feature"

git push origin main

Complete daily workflow:

Step 1: Check what changed

git status

Step 2: See exact changes

git diff

Step 3: Stage changes

git add .

Step 4: Commit changes

git commit -m "Descriptive message"

Step 5: Push to remote

git push origin main

Q9. HOW TO CLONE A REPOSITORY?

Command:

git clone url

Explanation: Download the code from remote repository (GitHub) to your local machine.

Steps:

Step 1: Go to GitHub repository

- Click on "Code" button (green button)
- Copy the HTTPS or SSH URL

Step 2: Clone using command

git clone https://github.com/username/calculator.git

What happens:

- Creates a new folder with repository name
- Downloads all files
- Downloads all commit history
- Sets up remote connection automatically

Example:

git clone https://github.com/sushmita/calculator.git

Output:

Cloning into 'calculator'...

remote: Enumerating objects: 15, done.

remote: Counting objects: 100% (15/15), done.

remote: Compressing objects: 100% (10/10), done.

remote: Total 15 (delta 2), reused 15 (delta 2)

Receiving objects: 100% (15/15), done.

Resolving deltas: 100% (2/2), done.

Navigate to cloned folder:

cd calculator

Q10. HOW TO CHECK REMOTE REPOSITORY?

Command:

git remote -v

Explanation: Shows the remote repository URL that your local repository is connected to.

What -v means:

- -v: Verbose (detailed output)
- Shows both fetch and push URLs

Example:

git remote -v

Output:

origin https://github.com/username/calculator.git (fetch)

origin https://github.com/username/calculator.git (push)

Explanation of output:

- origin: Name of remote (default name)
- fetch: URL to download code from
- push: URL to upload code to

Q11. HOW TO ADD REMOTE REPOSITORY?

Command:

git remote add origin url

Explanation: Connect your local repository to a remote repository.

When to use:

- You created local repo with git init
- You want to connect it to GitHub

Steps:

Step 1: Create repository on GitHub

- Go to GitHub
- Click "New repository"
- Create empty repository

Step 2: Add remote in local

git remote add origin https://github.com/username/calculator.git

Step 3: Verify

git remote -v

Step 4: Push code

git push -u origin main

Breakdown:

- git remote add: Add remote repository
- origin: Name for remote (you can choose any name)
- url: GitHub repository URL

Example:

git remote add origin https://github.com/sushmita/calculator.git

git branch -M main

git push -u origin main

Q12. HOW TO DOWNLOAD CODE USING SSH OPTION?

Difference between HTTPS and SSH:

HTTPS:

- Uses username and password/token
- Requires authentication every time you push
- URL format: https://github.com/username/repo.git

SSH:

- Uses SSH keys (no password needed)
- Authentication happens automatically
- URL format: git@github.com:username/repo.git

PROCESS TO USE SSH:

Step 1: Generate SSH Key

On your local machine:

ssh-keygen -t rsa -b 4096 -C "your-email@example.com"

Press Enter for all prompts (use defaults)

This creates:

- Private key: ~/.ssh/id_rsa
- Public key: ~/.ssh/id_rsa.pub

Step 2: Copy Public Key

cat ~/.ssh/id_rsa.pub

Output (example):

ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAACAQC7... your-email@example.com

Copy this entire output

Step 3: Add SSH Key to GitHub

On GitHub:

1. Click your profile picture (top right)

2. Click "Settings"
3. Click "SSH and GPG keys" (left sidebar)
4. Click "New SSH key"
5. Title: Give it a name (e.g., "My Laptop")
6. Key: Paste the public key
7. Click "Add SSH key"

Step 4: Test SSH Connection

ssh -T git@github.com

Output:

Hi username! You've successfully authenticated, but GitHub does not provide shell access.

This means SSH is working!

Step 5: Clone Using SSH

Copy SSH URL from GitHub:

- Click "Code" button
- Select "SSH" tab
- Copy URL (format: git@github.com:username/repo.git)

Clone:

git clone git@github.com:username/calculator.git

No password needed!

Step 6: Push Using SSH (No Password)

git add .

git commit -m "Update code"

git push

No password prompt! Authentication happens automatically using SSH key.

Benefits of SSH:

- No password needed for push/pull
- More secure than HTTPS
- Faster authentication
- One-time setup
- Recommended for organizations

Q13. DIFFERENCE BETWEEN GIT CLONE AND GIT FORK?

Git Clone:

Definition: Download the code present on GitHub to your local machine.

What it does:

- Creates a copy of repository on your computer
- Downloads all files and commit history
- Sets up connection to original repository
- You can work locally

Command:

git clone https://github.com/originaluser/repo.git

Use case:

- You want to work on the project locally
- You have write access to the repository
- Working on your own or team's repository

Example: You work at a company, your team has a repository, you clone it to work on it.

Git Fork:

Definition: Creating a complete copy of the repository on GitHub itself (not on your local machine).

What it does:

- Creates a copy under your GitHub account
- Original: github.com/originaluser/repo
- Your fork: github.com/yourname/repo
- Both exist on GitHub
- You own your forked copy

How to fork:

1. Go to repository on GitHub
2. Click "Fork" button (top right)
3. Choose your account
4. Fork created!

Use case:

- Contributing to open source projects
- You don't have write access to original repository
- Want to experiment without affecting original
- Want your own copy of the project

Example: You want to contribute to Kubernetes (you don't have direct access):

1. Fork kubernetes/kubernetes to yourname/kubernetes
2. Clone YOUR fork to local machine
3. Make changes
4. Push to YOUR fork
5. Create Pull Request to original repository

DETAILED COMPARISON:

Aspect: Location

- Clone: GitHub to Local machine
- Fork: GitHub to GitHub (your account)

Aspect: Ownership

- Clone: You're working with original repo

- Fork: You own the forked copy

Aspect: Write Access

- Clone: Need permission to push to original
- Fork: Full access to your fork

Aspect: Purpose

- Clone: Work locally on a project
- Fork: Create your own version of someone else's project

Aspect: Contribution

- Clone: Direct push if you have access
- Fork: Push to your fork, then create Pull Request

Aspect: Relationship

- Clone: Direct connection to original
- Fork: Independent copy with link to original

TYPICAL WORKFLOW FOR OPEN SOURCE CONTRIBUTION:

Step 1: Fork the repository On GitHub, click Fork button

Original: github.com/kubernetes/kubernetes

Your fork: github.com/yourname/kubernetes

Step 2: Clone YOUR fork

git clone https://github.com/yourname/kubernetes.git

cd kubernetes

Step 3: Make changes locally

git checkout -b feature/my-improvement

(make changes)

git add .

git commit -m "Add my improvement"

Step 4: Push to YOUR fork

git push origin feature/my-improvement

Step 5: Create Pull Request

- Go to your fork on GitHub
- Click "New Pull Request"
- Request to merge YOUR changes into ORIGINAL repository
- Original maintainers review and approve

SUMMARY:

Clone:

- Downloads code to your computer
- Command: git clone url
- You work locally
- Push back to same repository

Fork:

- Creates copy on GitHub under your account
- Done via GitHub UI (Fork button)
- You own the forked copy
- Contribute back via Pull Request

Together: Fork (on GitHub) then Clone (to local) is the standard workflow for contributing to projects you don't own.

Q14. HOW TO CREATE AND SWITCH TO NEW BRANCH?

Command:

git checkout -b feature/division

Explanation:

- git checkout: Switch branch command
- -b: Create new branch
- feature/division: Name of new branch

Breakdown: This command does TWO things:

1. Creates a new branch called feature/division
2. Switches to that branch immediately

Alternative (two separate commands):

git branch feature/division (create branch)

git checkout feature/division (switch to branch)

Example:

git checkout -b feature/division

Output:

Switched to a new branch 'feature/division'

Verify:

git branch

Output:

main

* feature/division

The asterisk shows you're currently on feature/division branch.

Q15. HOW TO LIST ALL BRANCHES?

Command:

git branch

Explanation: Shows all local branches in your repository.

Example:

git branch

Output:

* feature/division

feature/login

feature/payment

main

The asterisk (*) shows the current branch.

Other useful branch commands:

git branch -r: Show remote branches

git branch -a: Show all branches (local and remote)

git branch -d branch-name: Delete local branch (safe)

git branch -D branch-name: Force delete local branch

Example with remote branches:

git branch -a

Output:

* feature/division

  main

  remotes/origin/feature/login

  remotes/origin/main

Q16. HOW TO SEE COMMITS ON SPECIFIC BRANCH?

Command:

git log feature/division

Explanation: Shows commit history for feature/division branch only.

Example:

git log feature/division

Output:

commit xyz789 (feature/division)

Author: Sushmita

Date: Thu Nov 7 10:00:00 2024

    Add division function

commit abc123

Author: Sushmita

Date: Wed Nov 6 15:00:00 2024

Initial commit

Compare branches:

git log main..feature/division

Shows commits in feature/division that are NOT in main.

Q17. WHAT IS GIT CHERRY-PICK?

Command:

git cherry-pick commit-id

Explanation: Pick a specific commit from one branch and apply it to current branch.

WHEN TO USE CHERRY-PICK:

Scenario:

- You made 10 commits on feature/division branch
- You only want commit #5 in main branch
- You don't want other commits yet

Solution: Cherry-pick commit #5

HOW TO USE CHERRY-PICK:

Step 1: Find commit ID you want

git log feature/division --oneline

Output:

abc1234 Add division function

def5678 Add multiplication function

ghi9012 Fix bug in addition

jkl3456 Update comments

Step 2: Switch to branch where you want this commit

git checkout main

Step 3: Cherry-pick the specific commit

git cherry-pick abc1234

Result: Only the "Add division function" commit is now in main branch!

LIMITATIONS OF CHERRY-PICK:

Cherry-pick is easy when there are 1 or 2 commits.

Problem: If there are many commits in feature branch, it is practically impossible to always copy-paste the commit ID and do cherry-pick.

Example:

Feature branch has 50 commits

You want all 50 commits in main

Cherry-pick would require:

git cherry-pick commit1

git cherry-pick commit2

git cherry-pick commit3

...

git cherry-pick commit50

This is tedious and error-prone!

Solution: Use git merge or git rebase instead.

Q18. DIFFERENCE BETWEEN GIT MERGE AND GIT REBASE?

Let's understand with a practical example.

SETUP: CREATING TWO BRANCHES

Step 1: Create merge example branch

git checkout -b mergeexample

Step 2: Modify calculator.sh

vi calculator.sh

Add:

z=a*b

Step 3: Commit

git add .

git commit -m "Add multiplication in mergeexample"

Step 4: Switch back to main

git checkout main

Step 5: Create rebase example branch

git checkout -b rebaseexample

Step 6: Modify calculator.sh

vi calculator.sh

Add:

w=a/b

Step 7: Commit

git add .

git commit -m "Add division in rebaseexample"

CURRENT SITUATION:

main branch:

calculator.sh:

x=a+b

y=a-b

mergeexample branch:

calculator.sh:

x=a+b

y=a-b

z=a*b

rebaseexample branch:

calculator.sh:

x=a+b

y=a-b

w=a/b

Goal: Get both changes into main branch

OPTION 1: GIT MERGE

Step 1: Switch to main

git checkout main

Step 2: Merge mergeexample branch

git merge mergeexample

What happens: A window will open (text editor) where we need to mention why this merge.

Default message:

Merge branch 'mergeexample' into main

Save and close the editor (in vi: Esc, :wq!)

Step 3: Check result

git log

Output:

commit merge123 (HEAD -> main)

Merge: abc1234 def5678

Author: Sushmita

Date: Thu Nov 7 12:00:00 2024

Merge branch 'mergeexample' into main

commit def5678 (mergeexample)

Author: Sushmita

Date: Thu Nov 7 11:00:00 2024

Add multiplication in mergeexample

commit abc1234

Author: Sushmita

Date: Thu Nov 7 10:00:00 2024

Previous commit

Notice: There's a MERGE COMMIT (merge123)

Step 4: Check file

cat calculator.sh

Output:

x=a+b

y=a-b

z=a*b

Changes from mergeexample are now in main!

OPTION 2: GIT REBASE

Step 1: Still on main branch

git checkout main

Step 2: Rebase rebaseexample branch

git rebase rebaseexample

What might happen:

CONFLICT (content): Merge conflict in calculator.sh

error: could not apply abc1234... previous commit

Resolve all conflicts manually

You may get merge conflicts!

HANDLING MERGE CONFLICTS:

Step 1: Open the file

vi calculator.sh

You'll see:

x=a+b

y=a-b

<<<<<<< HEAD

w=a/b

=======

z=a*b

>>>>>>> abc1234

Explanation:

- <<<<<<< HEAD: Current branch changes (rebaseexample)
- =======: Separator
- abc1234: Changes being applied

Step 2: Resolve conflict

Discuss with team: What should the final code be?

Option A: Keep both

x=a+b

y=a-b

z=a*b

w=a/b

Option B: Keep only multiplication

x=a+b

y=a-b

z=a*b

Option C: Keep only division

x=a+b

y=a-b

w=a/b

Let's keep both. Remove conflict markers:

x=a+b

y=a-b

z=a*b

w=a/b

Save the file

Step 3: Continue rebase

git add .

git rebase --continue

If there's a commit message prompt, save and close

Step 4: Check result

git log

Output:

commit new789 (HEAD -> main)

Author: Sushmita

Date: Thu Nov 7 12:00:00 2024

Add division in rebaseexample

commit def5678

Author: Sushmita

Date: Thu Nov 7 11:00:00 2024

Add multiplication in mergeexample

commit abc1234

Author: Sushmita

Date: Thu Nov 7 10:00:00 2024

Previous commit

Notice: NO merge commit! Linear history!

DIFFERENCE BETWEEN MERGE AND REBASE:

Git Merge:

How it works:

- Creates a new "merge commit"
- Preserves complete history
- Shows when branches were merged
- Non-linear history (has branch points)

Commit history looks like:

main: A -- B -- C ------- M (merge commit)

\ /

mergeexample: D -- E

Advantages:

- Preserves exact history
- Safe (doesn't rewrite history)
- Shows when features were integrated
- Good for team collaboration

Disadvantages:

- Many merge commits can clutter history
- Non-linear history (harder to read)
- "Messy" git log with many branches

When to use:

- Public branches (main, develop)
- When history needs to be preserved
- Team collaboration
- When you want to see when features were merged

Git Rebase:

How it works:

- Re-applies commits on top of another branch
- Rewrites commit history
- Creates linear history (straight line)
- No merge commit

Commit history looks like:

Before rebase:

main: A -- B -- C

\

feature: D -- E

After rebase:

main: A -- B -- C -- D' -- E'

Note: D and E become D' and E' (new commit IDs)

Advantages:

- Clean, linear history
- Easy to read git log
- No merge commits
- Professional-looking history

Disadvantages:

- Rewrites history (changes commit IDs)
- Can be dangerous on shared branches
- Conflicts harder to resolve
- Can confuse team members

When to use:

- Local branches before pushing
- Feature branches before merging to main
- Cleaning up your commit history
- When you want linear history

GOLDEN RULE OF REBASE:

Never rebase public/shared branches!

Why?

- Changes commit IDs
- Confuses other developers
- Can cause problems for team

Safe:

git checkout feature/my-feature

git rebase main

(Feature branch is yours, safe to rebase)

Dangerous:

git checkout main

git rebase feature/my-feature

(Main is shared, DON'T rebase!)

DETAILED COMPARISON:

Aspect: History

- Merge: Preserves original history
- Rebase: Rewrites history

Aspect: Commit Graph

- Merge: Non-linear (shows branches)
- Rebase: Linear (straight line)

Aspect: Merge Commits

- Merge: Creates merge commits
- Rebase: No merge commits

Aspect: Readability

- Merge: Can be messy with many branches
- Rebase: Clean and easy to read

Aspect: Safety

- Merge: Safe for shared branches
- Rebase: Dangerous for shared branches

Aspect: Use Case

- Merge: Public branches, team work
- Rebase: Local branches, personal cleanup

Aspect: Conflicts

- Merge: Resolve once
- Rebase: May need to resolve multiple times

Aspect: Commit IDs

- Merge: Preserves original IDs
- Rebase: Changes commit IDs

WHEN TO USE WHICH:

Use Merge when:

- Working on shared/public branches
- Want to preserve history
- Multiple people working on same branch
- Want to see when features were integrated
- Safety is priority

Use Rebase when:

- Working on personal feature branch
- Want clean history before merging
- Updating feature branch with latest main
- Want linear commit history
- Before creating Pull Request

RECOMMENDED WORKFLOW:

Step 1: Work on feature branch

git checkout -b feature/new-feature

(make commits)

Step 2: Before merging, rebase on latest main

git checkout feature/new-feature

git rebase main

(resolve conflicts if any)

Step 3: Merge to main (use merge, not rebase!)

git checkout main

git merge feature/new-feature

Result:

- Feature branch has linear history (rebased)
- Main branch has merge commit (safe)
- Best of both worlds!

VISUAL COMPARISON:

Merge Example:

git log --graph --oneline

* merge123 Merge branch 'mergeexample'

|\

| * def5678 Add multiplication

|/

* abc1234 Initial commit

Shows branching!

Rebase Example:

git log --graph --oneline

* new789 Add division

* def5678 Add multiplication

* abc1234 Initial commit

Straight line!