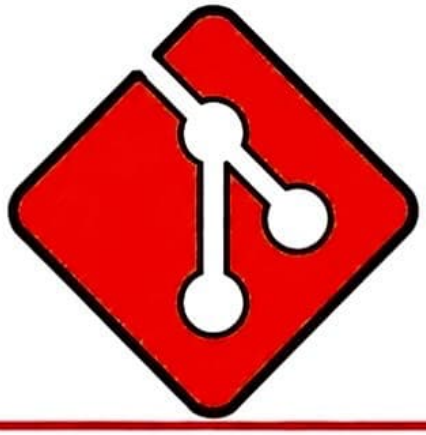
GIT

COMPLETE NOTES

(Git)



BEGINNER TO ADVANCED

SWIPE TO NEXT →

Introduction to Git

1. What is Git?

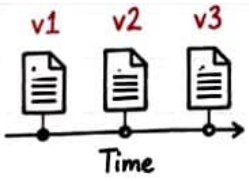
Git is a Distributed Version Control System (DVCS) used to track changes in source code during software development.

Why Git?

- ✓ Track every change
- ✓ Work with team easily
- ✓ Go back to any version
- ✓ Create branches & features
- ✓ Safe, fast & reliable

2. What is Version Control?

Version Control is a system that records changes to files over time so you can recall specific versions later.



3. Git vs GitHub

Git	GitHub
• Tool (Version Control System)	• Platform (Cloud Service)
• Works on your local machine	• Hosts Git repositories
• Command line based	• Web based interface
• Manages code history	• Collaboration & sharing

VS

4. Why Git is Important?

- ★ Every developer & tester should know Git.
- ★ Industry standard skill.
- ★ Essential for collaboration.
- ★ Helps in backup, tracking, and managing code efficiently.

5. Key Benefits of Git

Collaboration
Multiple developers can work together without overwriting each other's code.

Safety
Full history of changes is stored. You can recover anytime.

Speed
Git is lightweight and performs operations very fast.

Branching
Create branches to work on new features without affecting main code.

Backup
Your code is always safe in remote repositories.

Open Source
Free, open-source & used by millions worldwide.

Remember → Git helps you track changes, collaborate, and manage code like a pro developer!

Git Architecture

Git works like a 4-layer system where your changes move step by step from your computer to the remote server.

1 Working Directory

Where you work on files.

(Untracked/Modified)

git add →

2 Staging Area

Changes added and ready to be committed.

(Tracked)

git commit →

3 Local Repository

Changes are committed and saved locally.

(.git directory)

git push →

4 Remote Repository

Remote server like GitHub, GitLab, Bitbucket etc.

(Shared with team)

★ Git Lifecycle ★

Modified

You change a file.

→

Staged

You add changes to staging area.

→

Committed

You commit changes to local repository.

→

Pushed

You push changes to remote repository.

Easy Analogy

- Working Directory → Your Desk
- Staging Area → Your Clipboard
- Local Repository → Your Locker
- Remote Repository → Your Cloud Locker (Team Storage)

Important Commands in Flow

1. `git status` → Check current state
2. `git add <file>` → Move to staging area
3. `git commit -m "message"` → Save locally
4. `git push origin <branch>` → Push to remote

Tip → Always think: Modify → Add → Commit → Push
This is the heart of Git!

3

Installation & Configuration

1. Install Git

Download and install Git from the official website.

Official Site: https://git-scm.com/downloads

Windows macOS Linux

Follow the installation wizard for your operating system.

2. Check Git Installation

Verify Git is installed successfully.

```
$ git --version
git version 2.44.0.windows.1
```

Tip:
If command is not found, restart your terminal.

3. Configure Username & Email

Set your global username and email.

```
$ git config --global user.name "Your Name"
  # sets the username
$ git config --global user.email "youremail@example.com"
  # sets the email
```

These details will be attached to all your commits.

4. View Current Configuration

Check your current Git configuration.

```
$ git config --list
# shows all global configuration
# settings
```

Remember
Use `--global` to apply settings for all repositories on your system.
Use `--local` for a specific repository.

5. Default Branch

Git's default branch name is changing from 'master' to 'main'.

```
$ git config --global init.defaultBranch main
# sets the default branch name
# new repositories will start with 'main'
```

main

6. Check Configuration Values

Check Username

```
$ git config user.name
Your Name
```

Check Email

```
$ git config user.email
youremail@example.com
```

Check Default Branch

```
$ git config init.defaultBranch
main
```

Check All Settings

```
$ git config --list
# shows all configuration
# values
```

Best Practice: Always configure Git properly before starting any project. It saves time and avoids commit mistakes.

@qa_insights

Essential Git Commands

4

1. git init

Initializes a new Git repository in your project.

```
$ git init
```

```
Initialized empty Git repository in my-project/.git/
```

2. git clone

Clones an existing remote repository to your local machine.

```
$ git clone <url>
```

```
Cloning into 'repo'...
remote: Enumerating objects: 120, done.
Receiving objects: 100%
```

3. git status

Shows the current status of your working directory and staging area.

```
$ git status
```

```
On branch main
Changes not staged
  modified:   app.js
Untracked files:
  test.js
```

4. git add

Adds changes from working directory to the staging area.

```
$ git add <file>
$ git add .
```

```
(no output)
Moves changes to staging area.
```

5. git commit

Commits the staged changes to the local repository.

```
$ git commit -m "Your message"
```

```
[main a1b2c3d] Your message
1 file changed, 5 insertions(+)
create mode 100644 app.js
```

6. git log

Shows the commit history.

```
$ git log --oneline
```

```
a1b2c3d (HEAD -> main)
  Your message
d4e5f6g Second commit
h7i8j9k Initial commit
```

7. git diff

Shows differences between chagines. Useful to review what you modified.

1) Working Directory vs Staging

```
$ git diff
```

```
- old line
+ new line
```

2) Staged vs Last Commit

```
$ git diff --staged
```

```
- old line
+ new line
```

3) Between Commits/Branches

```
$ git diff <a> <b>
```

```
- old line
+ new line
```

Remember

- add → moves changes to staging.
- commit → saves changes permanently locally.
- push → sends changes to remote.

Tip

- Check status often using git status.
- Commit small, meaningful changes.
- Write clear commit messages.

Complete Git Workflow

Here is the complete end-to-end Git workflow you follow in any real project.

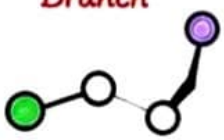
Follow this flow every time!

1. Clone

Copy a remote repository to your local machine.

```
$ git clone <url>
```

2. Branch



Create a new branch for your feature or fix.

```
$ git checkout -b feature/login
```

3. Code

Make changes in your files.

```
// Edit, Add or Update your code
```

4. Add

Add your changes from working directory to staging area.

```
$ git add .
```

5. Commit

Commit your staged changes to local repository.

```
$ git commit -m "Your message"
```

6. Pull

Pull latest changes from remote repository.

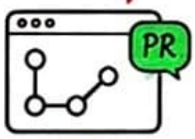
```
$ git pull origin main
```

7. Push

Push your committed changes to remote.

```
$ git push origin feature/login
```

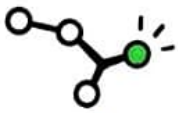
8. Pull Request



Create a Pull Request to review your changes.

```
// Open PR on GitHub, GitLab, Bitbucket etc.
```

9. Merge



After review & approval, merge your PR.

```
// Changes are merged into main branch.
```

Key Points

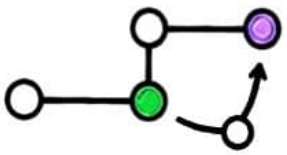
- Always work on a separate branch.
- Pull latest changes before pushing.
- Write meaningful commit messages.
- Review before creating Pull Request.

Tip

- Small commits, clear messages.
- Keep your branch up to date.
- Never push directly to main branch.

Branching

Branches help you work on new features, fixes or experiments without affecting the main code.



1. Create Branch

```
$ git branch <branch-name>
# creates a new branch
```

Example:

```
$ git branch feature/login
```

2. Switch Branch

```
$ git switch <branch-name>
# switch to a branch
```

Example:

```
$ git switch feature/login
```

3. Create & Switch

```
$ git switch -c <branch-name>
# create and switch
```

Example:

```
$ git switch -c feature/login
```

4. Rename Branch

-m → move

```
$ git branch -m <new-name>
# rename current branch
```

Example:

```
$ git branch -m feature/auth
```

5. Delete Branch

```
$ git branch -d <branch-name>
# delete merged branch
```

Example:

```
$ git branch -d feature/login
```

```
$ git branch -D <branch-name>
# force delete (not merged)
```

Example:

```
$ git branch -D temp-branch
```

Tip

- Use -d to delete safely only if merged.
- Use -D to force delete unmerged branches.

6. Branch Naming Conventions

Type	Prefix	Purpose	Examples
Feature	feature/	New feature development	feature/login, feature/payment
Bug Fix	bugfix/	Bug fixes	bugfix/header-issue, bugfix/typo
Hotfix	hotfix/	Urgent fixes in production	hotfix/critical-login, hotfix/payment
Release	release/	Release preparation	release/v1.0.0, release/v2.1.0
Chore	chore/	Maintenance / dependencies / tasks	chore/update-deps, chore/docs

Best Practice

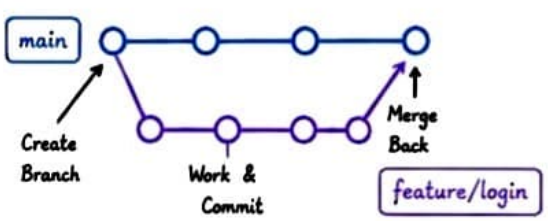
- Use lowercase letters.
- Use hyphen (-) not underscore (_).
- Keep it short, clear and meaningful.
- Follow team standards.

7. View Branches

```
$ git branch
# list local branches
```

```
$ git branch -r
# list remote branches
```

8. Branch Workflow Example



Remember

Branches are lightweight and extremely powerful. They make collaboration smooth and safe.

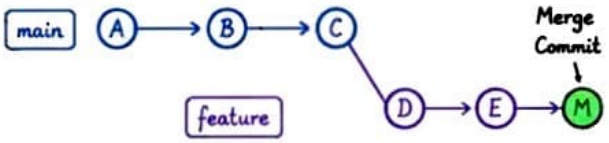
7

Merge vs Rebase

Both Merge and Rebase are used to integrate changes, but the history they create is different.

1. MERGE

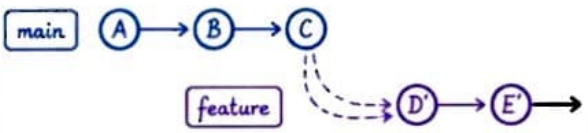
Combines histories by creating a new merge commit.




```
$ git merge feature
# merges feature into main
```

2. REBASE

Replays your commits on top of the latest commit.




```
$ git rebase main
# move feature commits to latest main
```

3. Key Differences

Aspect	Merge	Rebase
History	Keeps complete history with merge commit.	Creates linear, clean history.
Commits	Original commits are preserved.	Creates new commits (different IDs).
Clarity	History can be messy with many merges.	History looks clean and easy to read.
Collaboration	Safe for shared branches.	Avoid rebase on shared branches.
Conflicts	Conflicts resolved once during merge.	Conflicts may appear multiple times.
Use Case	Feature integration, long running branches.	Updating feature branches, before PR.

4. Advantages

Merge

- Safe and easy for teams.
- Preserves the complete project history.
- Good for long feature branches.
- Ideal for release & integration.

Rebase

- Keeps history clean and linear.
- Easier to read and understand.
- Removes unnecessary merge commits.
- Great for feature branches.

Tip

Think of Merge as "preserving history" and Rebase as "rewriting history"

5. When to Use?

Use MERGE when:

- Working on team/shared branches
- Branch is public
- You want full history
- Long running features

Use REBASE when:

- Working on your own feature branch
- Branch is not shared
- Before creating Pull Request
- You want clean history

6. Company Practices

- Never rebase on main, develop or release branches.
- Always rebase your feature branch with latest main before PR.
- Use merge commit (or Squash) while merging PR.
- Follow team's Git workflow policy.

Remember

Rebase rewrites history. Use it carefully.
Merge preserves history. Use it safely.

A, B, C = main commits
D, E = feature commits (before rebase)
D', E' = feature commits (after rebase)
M = merge commit

Remote Repository

Remote repositories allow you to share your code, collaborate with others and keep your project safe in the cloud.

1. Push

Sends your local commits to the remote repository.

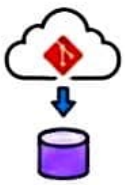
```
$ git push <remote> <branch>
# push local commits to remote
```

2. Pull

Fetches and merges changes from remote repository to local.

```
$ git pull <remote> <branch>
# fetch + merge
```

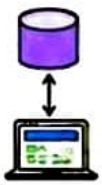
3. Fetch



Downloads changes from remote but does NOT merge automatically.

```
$ git fetch <remote>
# update remote tracking branches
```

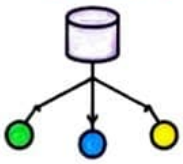
4. Origin



Default name of the remote repository you cloned from.

```
$ git remote -v
# view remote repositories
```

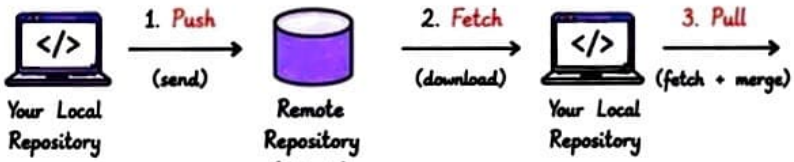
5. Upstream Branches



Remote branches that your local branches track and sync with.

```
$ git branch -vv
# show tracking information
```

How They Work Together



Check Remotes

```
$ git remote -v
origin  https://github.com/user/repo.git (fetch)
origin  https://github.com/user/repo.git (push)
```

'origin' is the default remote name.

Add / Remove / Rename Remote

Action	Command	Description
Add Remote	`$ git remote add <name> <url>`	Add a new remote repository
Remove Remote	`$ git remote remove <name>`	Remove an existing remote
Rename Remote	`$ git remote rename <old> <new>`	Rename a remote

Remote Tracking Branches (Upstream)

When you set an upstream, your local branch knows which remote branch to track.

Common Workflow Example

Tips

- Always pull before you push.
- Use meaningful branch names.
- Set upstream when creating a new branch.
- Keep your local repo in sync with remote.

Remember

- Push → upload your work to remote.
- Pull → get latest changes and merge.
- Fetch → download without merging.
- Origin → default remote name.

Undo & Recovery

Git gives you powerful tools to undo mistakes and recover your project safely.

1. git restore

Discards changes in working directory or unstage a file.

```
$ git restore <file>
$ git restore --staged <file>
# discard changes in working directory
# unstage a file
```

When to use?

When you want to discard local changes that are not committed.

2. git reset

Moves HEAD to a specific commit and updates **staging** & working directory based on mode.

```
$ git reset --soft <commit>   # keep changes staged
$ git reset --mixed <commit>  # keep changes unstaged
$ git reset --hard <commit>   # discard all changes
```

When to use?

When you need to undo commits. Use with caution!

3. git revert

Creates a new commit that undoes the changes of a previous commit.

```
$ git revert <commit>
# safe undo (does not change history)
```

When to use?

When you want to undo commits on shared/public branches.

4. git checkout

Switch branches or restore files from a specific commit.

```
$ git checkout <branch>
$ git checkout <commit> -- <file>
# switch branch
# restore file from commit
```

When to use?

When switching branches or restoring a file from a specific commit.

5. git reflog

Shows a log of all HEAD movements. Helps to recover lost commits.

```
$ git reflog      # show reflog
```

Example Output:

```
a1b2c3d HEAD@{0}: reset: moving to a1b2c3d
d4e5f6g HEAD@{1}: commit: add login test
h7i8j9k HEAD@{2}: commit: initial commit
```

When to use?

When you lost commits due to reset or want to go back to old states.

Quick Comparison

Command	What it does	Affects History?
restore	Discard changes	No
reset	Move HEAD	Yes (hard) No (soft/mixed)
revert	Undo by new commit	No
checkout	Switch / Restore	No
reflog	Show history of moves	No

When to Use Each?

Discard Changes
- Use git restore
- When you made changes but didn't commit.

→ **Undo Local Commits**
- Use git reset
- When commits are local and not pushed.

→ **Undo Shared Commits**
- Use git revert
- When commits are already pushed.

→ **Switch / Restore Files**
- Use git checkout
- When switching branches or restoring specific files.

→ **Recover Lost Commits**
- Use git reflog
- When commits are lost due to reset or other actions.

Best Practices

- Avoid using git reset --hard on shared branches.
- Always double-check before discarding changes.
- Use generically: prefer revert over reset on public branches.
- reflog is your best friend in recovery!

Tip

- Before any major undo, create a backup branch.

```
$ git branch backup-<date>
```

- Better safe than sorry!

Stash, Tags & .gitignore

These Git features help you save your work temporarily, mark important points in history and ignore unnecessary files.

1. Stash (Save Work)

Stash lets you save your uncommitted changes and clean your working directory.


```
$ git stash              # Save current changes
$ git stash list         # List all stashes
$ git stash pop          # Apply latest stash and remove it
$ git stash apply        # Apply latest stash (keep it)
$ git stash drop         # Delete a stash
```

Tip

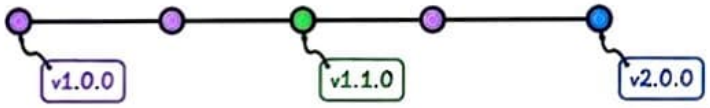
- Use stash before switching branches.
- Use stash pop to apply and remove at the same time.

```
$ git stash

Saved working directory and index state WIP
on feature/login: abc1234 Fix login UI
```

2. Tags (Release Points)

Tags are used to mark specific points in history, usually for releases.




```
$ git tag                          # List all tags
$ git tag <tagname>                # Create a lightweight tag
$ git tag -a <tagname> -m "message"  # Create annotated tag
$ git push origin <tagname>        # Push a tag
$ git push origin --tags           # Push all tags
```

```
$ git tag v1.0.0
$ git tag -a v1.1.0 -m "Release v1.1.0"
$ git tag
  v1.0.0
  v1.1.0
$ git push origin v1.1.0
```

Remember

Use annotated tags for important releases. They store message, date, author and more.

3. .gitignore (Ignore Files)

.gitignore tells Git which files or folders to ignore. Ignored files won't be tracked.


```
$ git status                  # Ignored files won't appear
$ git add .                   # Only non-ignored files are added
$ git check-ignore -v <file>  # Check why a file is ignored
```

Example .gitignore

```
# Logs
*.log
# Node modules
node_modules/
# Environment files
.env
# OS generated files
.DS_Store
# Build output
dist/
build/
# IDE files
.vscode/
*.iml
```

Tip

- Keep .gitignore in the root of your repository.
- Commit .gitignore so everyone follows the same rules.
- Add common ignores for logs, builds, dependencies, secrets, IDE files, OS files.

Quick Summary

Stash
- Save uncommitted work.
- Use before switching branches.

Tags
- Mark important points.
- Used for releases.

.gitignore
- Ignore unnecessary files.
- Keep repository clean.

GitHub Collaboration

GitHub makes collaboration easy through Forks, Pull Requests and Code Reviews.

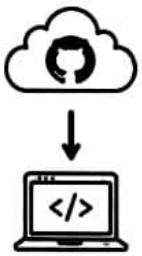
1. Fork

Create your own copy of someone else's repository.

- Click Fork on GitHub

2. Clone

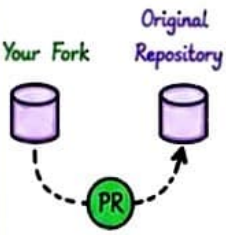
Clone your forked repository to your local machine.



```
$ git clone <your-fork-url>
```

3. Pull Request (PR)

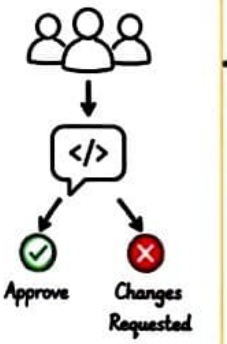
Propose your changes from your fork to the original repository.



- Click "New Pull Request" on GitHub
- Compare & describe your changes

4. Code Review

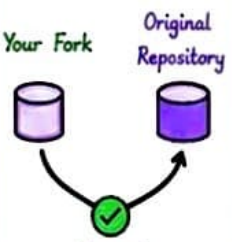
Team reviews your PR, suggests changes and approves it.



- Review code line by line
- Comment, Suggest, Approve

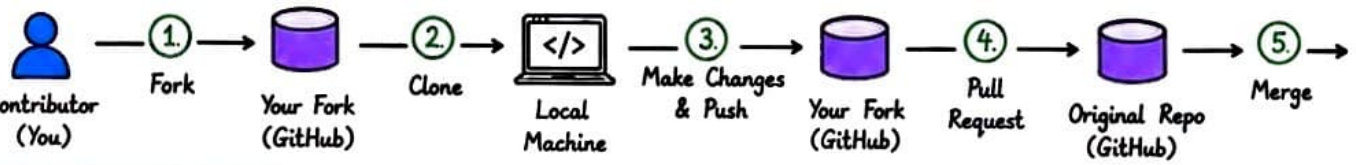
5. Merge

Once approved, your changes are merged to the main repository.



- Click "Merge Pull Request"
- Changes go into main branch

Collaboration Flow



Key Points

- Fork → Your personal copy on GitHub
- Clone → Download fork to your machine
- PR → Propose changes to original repo
- Review → Discuss and improve together
- Merge → Final changes in main repo

Best Practices

- Always keep your fork up to date with the original.
- Create meaningful branch names.
- Keep PRs small and focused.
- Write clear PR descriptions.
- Be responsive during code reviews.

Tips

- Sync your fork: git fetch upstream && git merge upstream/main
- Squash commits before PR for clean history.
- Always test your changes before opening PR.

Remember

- Collaborate → Communicate → Review → Improve
- Great code is a team effort!

@qa_insights

12

Git for QA Engineers

Git is a powerful tool for QA Engineers to manage automation code, collaborate with teams and maintain stable, versioned and reliable test frameworks.

1. Selenium Workflow

Write Tests (Selenium) → Commit & Push to Repo → Run in CI (Jenkins/GitHub Actions)

```
$ git checkout -b feature/login-test
  # create feature branch
$ git add .              # stage changes
$ git commit -m "Add login test"
  # commit
$ git push origin feature/login-test
  # push
```

2. Playwright Workflow

Write Tests (Playwright) → Commit & Push to Repo → Run in CI / Cloud (Parallel)

```
$ git checkout -b feature/signup-pw
  # create feature branch
$ git add .              # stage changes
$ git commit -m "Add signup test"
  # commit
$ git push origin feature/signup-pw
  # push
```

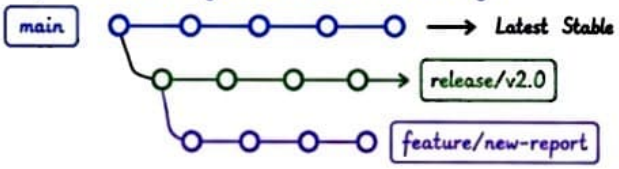
3. API Automation Workflow

Write API Tests → Commit & Push to Repo → Validate in CI & Reports

```
$ git checkout -b feature/api-auth
  # create feature branch
$ git add .              # stage changes
$ git commit -m "Add auth API test"
  # commit
$ git push origin feature/api-auth
  # push
```

4. Framework Versioning

Maintain different versions of your automation frameworks using Git branches and tags.




```
$ git checkout -b release/v2.0          # create release branch
$ git tag -a v2.0.0 -m "Release v2.0.0" # create release tag
$ git push origin release/v2.0 --tags   # push branch and tag
```

Best Practices

- Maintain clean main branch.
- Create release branches for stable versions.
- Use tags for major releases.
- Never commit directly to main.
- Keep test data, configs and docs in version control.

Example Tags

- v1.0.0 – Initial Framework
- v1.1.0 – Added API Module
- v2.0.0 – Added Playwright Support

5. Typical QA Automation Project Structure

```
qa-automation-framework/
├── tests/
│   └── pages/
├── api/
├── utils/
├── configs/
├── reports/
├── .github/
├── .gitignore
└── README.md
```

	Purpose
tests/	All test scripts
pages/	Page Object / Components
api/	API test classes / payloads
utils/	Reusable methods & helpers
configs/	Environment configs
reports/	Generated test reports
.github/	CI/CD workflows & actions
.gitignore	Ignore unwanted files
README.md	Project documentation

Tip

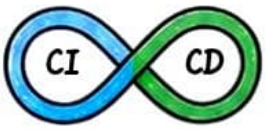
- Commit small, meaningful changes.
- Write clear commit messages.
- Use branches for new features or bug fixes.
- Review before merging.
- Automate with CI for reliable builds.

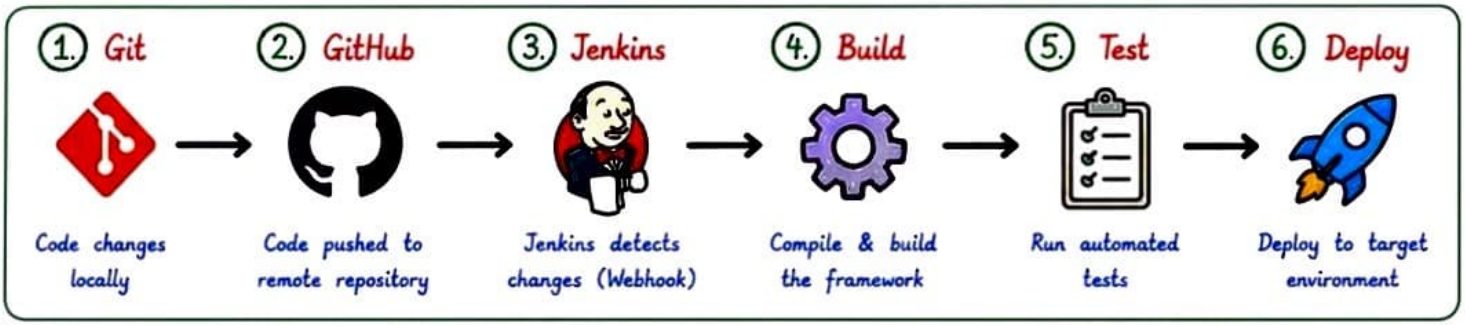
Remember

- Git helps QA teams collaborate better.
- Version control ensures safety and stability.
- Good Git habits improve code quality and productivity.
- Your tests are your product – treat them well!

Git with CI/CD

Git works hand-in-hand with CI/CD tools to automate build, test and deployment of our automation frameworks.

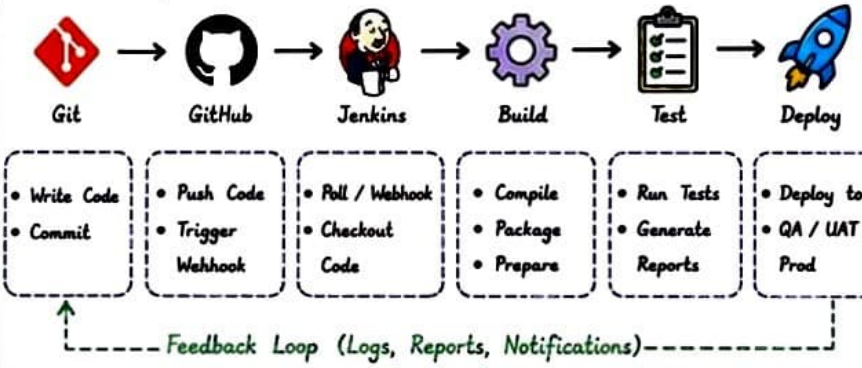




How It Works

1. Developer commits code in local Git.
2. Code is pushed to GitHub.
3. Jenkins (CI server) gets triggered.
4. Jenkins pulls code and builds the project.
5. Automated tests (Selenium / Playwright / API) are executed.
6. If tests pass, the build is deployed to the target environment.

CI/CD Pipeline Flow



Example Jenkins Pipeline (Declarative)

```
pipeline {                                  // Simple pipeline
  agent any                                 // for building,
  stages {                                  // testing and
    stage('Checkout') {                     // deploying
      steps {
        checkout scm    // Pull code from GitHub
    }
    stage('Build') {
      steps { sh 'mvn clean install' }
    }
    stage('Test') {
      steps { sh 'mvn test' }
    }
    stage('Deploy') {
      steps { sh 'echo "Deploying to QA"' }
    }
  }
}
```

Benefits

- Automates build, test and deployment.
- Ensures code quality with every commit.
- Fast feedback to developers.
- Reduces manual effort and human errors.
- Enables continuous delivery.

Tips

- Keep pipeline fast and reliable.
- Fail early, fail fast.
- Keep test data and configs in version control.
- Use branches and tags for stable releases.
- Always review logs and reports.

Popular Tools

Git → Version Control
GitHub → Code Hosting
Jenkins → CI/CD Automation
Maven → Build Tool
Selenium / Playwright / REST Assured → Test Tools

Remember

Good Git practices + Strong CI/CD pipeline
= Reliable Automation Delivery!

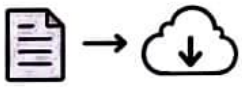
@qa_insights

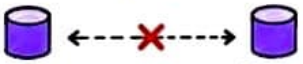
14

Common Git Errors

Mistakes happen! Here are the most common Git errors, why they occur and how to fix them quickly.

#	Error & Message	Why It Occurs	Quick Fix

1.	error: Your local changes to the following files would be overwritten by merge
	`git pull`
	`error: Your local changes would be overwritten by merge`
		• You have uncommitted changes and trying to pull new changes.
		 
			• Save or commit your changes before pulling.
			`$ git add .`
			`$ git commit -m "WIP: saving changes"`
			`$ git pull`

2.	error: failed to push some refs to 'origin'
	`$ git push origin main`
	`error: failed to push some refs to 'origin'`
		• Remote branch has new commits that you don't have locally.
		 
		You Remote
			• Pull latest changes, then push again.
			`$ git pull origin main --rebase`
			`$ git push origin main`

3.	fatal: refusing to merge unrelated histories
	`$ git pull origin main`
	`fatal: refusing to merge unrelated histories`
		You are merging two repositories that have no common history.
		 
		Repo A Repo B
			• Allow unrelated histories (use carefully).
			`$ git pull origin main --allow-unrelated-histories`

4.	error: Please commit your changes or stash them before you merge.
	`$ git merge feature/login`
	`error: Please commit your changes or stash them before you merge.`
		• You have uncommitted changes in your working directory.
		 
		Uncommitted Changes
			• Commit or stash your changes.
			`$ git add .`
			`$ git commit -m "Save changes"`
			`# or stash`
			`$ git stash`
			`$ git merge feature/login`

5.	fatal: not a git repository (or any of the parent directories)
	`$ git status`
	`fatal: not a git repository (or any of the parent directories)`
		You are not in a Git repository.
		 
		No .git folder
			• Navigate to the correct repo or initialize Git.
			`$ cd /path/to/your/repo`
			`$ git init # if new repo`

6.	Merge conflict
	`CONFLICT (content): Merge conflict in utils/login.ts`
		Same lines changed in both branches.
		 
			• Resolve conflicts manually, then commit.
			`# After resolving conflicts in files`
			`$ git add .`
			`$ git commit -m "Resolve merge conflicts"`

General Tips

- Always commit meaningful changes.
- Pull latest changes before starting work.
- Use branches for new features/bug fixes.
- Read error messages carefully – they guide you!

Quick Rescue Commands

```
$ git status              # Check current status
$ git stash               # Save changes temporarily
$ git stash pop           # Reapply stashed changes
$ git reset --hard HEAD   # Discard local changes
$ git log --oneline --graph # View clean history
```

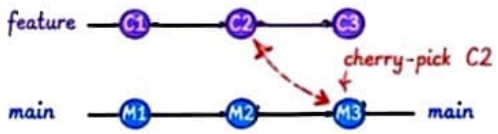
Remember

Good Git habits save time, prevent errors and keep your team happy! ☺

Advanced Git

Advanced Git commands help you manipulate history, clean up commits and recover from mistakes like a pro!

1. Cherry-pick – Apply a specific commit from one branch to another.



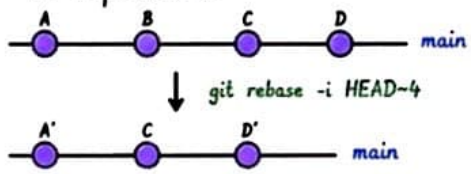
Example

```
$ git cherry-pick <commit-hash>
# applies the commit to current branch
```

Use Case

- Pick a bug fix from another branch without merging the entire branch.

2. Interactive Rebase – Edit, reorder, squash or drop commits.



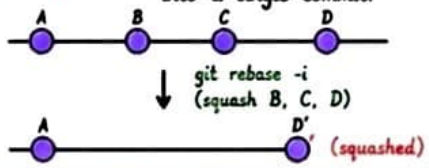
Example (rebase -i)

```
pick A      Add login
squash B    Fix typo
drop C      WIP commit
reword D    Update README
# Save & exit to apply
```

Use Cases

- Clean up messy commit history
- Squash WIP commits
- Reorder commits before merge

3. Squash – Combine multiple commits into a single commit.



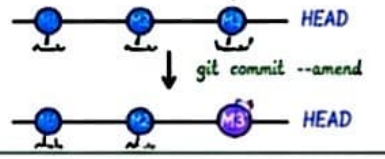
Example

```
pick A      Add base structure
squash B    Add utils
squash C    Add helpers
squash D    Add tests
# Results in 2 commits: A and D'
```

Tip

Keep history clean and easy to understand before pushing to shared repo.

4. Amend – Modify the last commit (message or content).



Examples

```
$ git commit --amend                  # modify last commit
$ git commit --amend --no-edit # only update
                                      # staged changes
```

Use Case

- Fix the last commit message
- Add missed files to last commit

5. Reflog Recovery – Recover lost commits, branches or changes.

What is Reflog?

Reflog is a log of all HEAD movements in your local repo. It helps to recover lost commits after reset, rebase, or checkout.

How it works

Reset / Rebase / Checkout (mistake)

↓

Use reflog to find the previous commit

Reflog List (example)

```
a1b2c3d  HEAD@{0}: checkout: moving
d4e5f6g  HEAD@{1}: rebase finished
h7i8j9k  HEAD@{2}: commit: Added tests
l0m1n2o  HEAD@{3}: commit: Login feature
p3q4r5s  HEAD@{4}: checkout: moving
```

Example Recovery

```
$ git reflog                          # view reflog
$ git checkout h7i8j9k                # go to lost commit
$ git branch recover h7i8j9k  # create branch
$ git checkout recover                # switch to it
```

Best Practices

- ✓ Never rewrite public history.
- ✓ Use interactive rebase on local branches only.
- ✓ Always backup important work.
- ✓ Use reflog when you mess up!

Danger Zone

- Avoid: git push --force (on shared branches)
- Be careful with: rebase, reset, cherry-pick

Think twice, commit once! ☺

Remember

Advanced Git = Power

Use it wisely and keep your repository clean & safe!

@qa_insights

Git Interview Questions

16

Top Beginner and Advanced Git interview questions with concise and easy to remember answers.

BEGINNER QUESTIONS

1. What is Git?
   → Git is a distributed version control system used to track changes in source code.

2. What is a Repository?
   → A repository (repo) is a storage location where your project and its entire history are stored.

3. What is the difference between Git and GitHub?
   → Git is a tool (VCS). GitHub is a cloud platform to host Git repositories and collaborate.

4. What is the difference between git add and git commit?
   → git add **stages changes**. git commit saves those changes to the repository.

5. What does git status do?
   → It shows the current status of changes in the working directory and staging area.

6. What is a branch?
   → A branch is an independent line of development. It allows you to work on features without affecting main code.

7. How do you merge a branch?
   → Checkout to target branch and run:

```
$ git checkout main
$ git merge feature-branch
```

8. What is the purpose of git pull?
   → It fetches changes from remote repository and merges them into current branch.

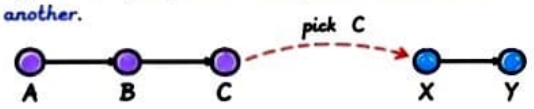
ADVANCED QUESTIONS

1. What is the difference between git fetch and git pull?
   → fetch downloads changes from remote but does not merge. pull = fetch + merge.

2. What is rebase and why use it?
   → Rebase reapplies your commits on top of another base commit. It keeps history clean and linear.

3. What is the difference between merge and rebase?
   → Merge creates a new commit (preserves history). Rebase rewrites history (linear, cleaner).

4. What is cherry-pick?
   → It applies a specific commit from one branch to another.



5. What is stash?
   → Stash temporarily saves your uncommitted changes so you can switch branches or work on something else.

6. How do you undo the last commit?
   → **If not pushed:** git reset --soft HEAD~1
   If pushed: git revert HEAD

7. What is git reflog?
   → Reflog is a log of all actions (commits, resets, checkouts). Useful to recover lost commits.

8. How do you recover a deleted branch?
   → Use reflog to find last commit and run:

```
$ git reflog
$ git checkout -b branch-name <commit-hash>
```

QUICK COMMANDS

Task	Command
Init repo	git init
Clone repo	git clone <url>
Add files	git add .
Commit	git commit -m "msg"
Push	git push origin <branch>
Pull	git pull origin <branch>
Branch	git branch <name>
Checkout	git checkout <branch>
Status	git status
Log	git log --oneline --graph

BEST PRACTICES

- Write clear and meaningful commit messages.
- Commit small, atomic changes.
- Pull latest changes before starting work.
- Use feature branches for new work.
- Review code before merging.
- Keep main branch always stable.

TIPS

- Use .gitignore to ignore unnecessary files.
- Use stash instead of committing WIP.
- Prefer rebase for personal branches.
- Always backup important work.
- Use meaningful branch names.

REMEMBER

Good Git skills show your professionalism and make team collaboration smooth!

Clean history, clear commits, happy team!

Practice daily, master Git, boost your career!

@qa_insights

Git Best Practices

17

Good Git habits = Clean history, happy team, and successful projects!

1 Commit Standards

- Write clear, concise and meaningful commit messages.
- Use present tense: "Add login test" not "Added..."
- One logical change per commit.
- Reference issues / ticket IDs when possible.
- Avoid vague commits like "fix", "update latest".

Conventional Commit Example

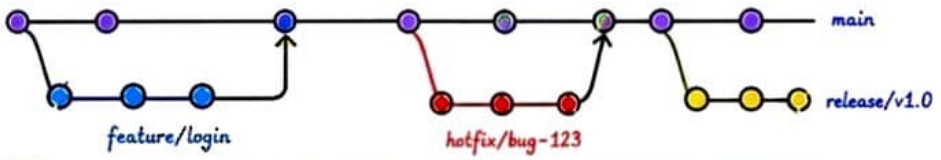
```
feat(login): add remember me checkbox
fix(api): handle 500 error gracefully
test(playwright): add test for reset password
docs(readme): update setup instructions
```

Tip:
Good commit messages make code review and debugging much easier!

2 Branch Strategy

- Use a consistent branching model.
- Keep main branch always stable.
- Create short-lived feature branches.
- Delete branches after merge.
- Use descriptive branch names.

Recommended Flow (Git Flow Simplified)



Best Practice:
Pull latest changes before starting work and before creating Pull Request.

3 Security Best Practices

- Never commit sensitive data (passwords, keys, tokens).
- Use .gitignore to exclude secrets and large files.
- Use environment variables for credentials.
- Enable 2FA on Git hosting platforms.
- Review access permissions regularly.

Tip:
Security in Git is everyone's responsibility!

4 Team Collaboration

- Pull latest changes before you start working.
- Communicate in Pull Requests (PR) clearly.
- Review code thoughtfully and provide constructive feedback.
- Keep PRs small and focused.
- Resolve conflicts locally and test before merging.
- Tag releases and maintain changelogs.

Do

- Commit small, logical changes
- Write meaningful commit messages
- Use feature branches
- Review code before merging
- Keep the repository clean

Don't

- Don't commit directly to main
- Don't push broken code
- Don't ignore .gitignore
- Don't commit secrets
- Don't leave stale branches

Quick Reference

Check status	git status
Add changes	git add .
Commit	git commit -m "message"
Push	git push origin <branch>
Pull	git pull origin <branch>
Create branch	git checkout -b <branch>
Switch branch	git checkout <branch>
Merge branch	git merge <branch>

Remember

Consistent practices build better projects and stronger teams!

Golden Rule

Commit often, commit smart, collaborate better, ship faster!

Final Tip

A clean Git history is a gift to your future self and your teammates.

@qa_insights

Ultimate Git Cheat Sheet

18

All essential Git commands, workflows, lifecycle, recovery techniques & best practices on one page!

1. ESSENTIAL COMMANDS

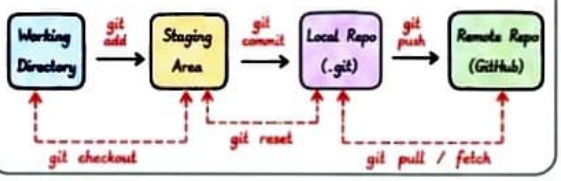
Task	Command
Initialize repo	`git init`
Clone repo	`git clone <url>`
Check status	`git status`
Add files	`git add <file>`
Add all files	`git add .`
Commit	`git commit -m "message"`
View log	`git log --oneline --graph`
View changes	`git diff`
Checkout branch	`git checkout <branch>`
Create branch	`git branch <branch>`
Switch branch	`git switch <branch>`
Merge branch	`git merge <branch>`
Pull changes	`git pull origin <branch>`
Push changes	`git push origin <branch>`
Stash changes	`git stash`
List stash	`git stash list`
Apply stash	`git stash apply`
Delete branch	`git branch -d <branch>`
Force push	`git push --force`

2. GIT WORKFLOW

1. Clone → 2. Code → 3. Add → 4. Commit → 5. Push

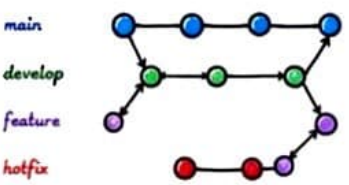
- Clone the repo
- Make changes in code
- Add changes to staging
- Commit with meaningful message
- Push to remote repository

3. GIT LIFECYCLE



4. BRANCHING STRATEGY

- `main / master`
  Stable production code
- `develop`
  Integration branch for features
- `feature/*`
  New feature development
- `hotfix/*`
  Urgent fixes for production
- `release/*`
  Release preparation



5. ADVANCED COMMANDS

- **Cherry-pick**
  `git cherry-pick <commit-hash>`
  Apply specific commit from another branch.
- **Rebase**
  `git rebase <branch>`
  Reapply commits on top of another base.
- **Interactive Rebase**
  `git rebase -i HEAD~n`
  Edit, reorder, squash commits.
- **Squash**
  `git merge --squash <branch>`
  Combine all commits into one.
- **Amend Last Commit**
  `git commit --amend`
  Modify the last commit.
- **Rename Branch**
  `git branch -m <new-name>`
- **Delete Remote Branch**
  `git push origin --delete <branch>`
- **Set Upstream**
  `git push -u origin <branch>`
  Set remote upstream for branch.

Tip: Use tab completion & `git --help` for help!

6. RECOVERY & UNDO

- **Uncommit (keep changes)**
  `git reset --soft HEAD~1`
- **Unstage files**
  `git reset <file>`
- **Discard changes in file**
  `git checkout -- <file>`
- **Reset (remove changes)**
  `git reset --hard HEAD~1`
- **Revert a commit**
  `git revert <commit-hash>`
- **View Reflog**
  `git reflog`
  Find lost commits, reset, checkout from history.

7. COMMON SCENARIOS

- **Start a new project**
  `git init`
- **Clone existing project**
  `git clone <url>`
- **Create & switch branch**
  `git switch -c feature/login`
- **Work, add & commit**
  `git add .`
  `git commit -m "Add login page"`
- **Push branch**
  `git push -u origin feature/login`
- **Update local repo**
  `git pull origin develop`
- **Merge feature to develop**
  `git checkout develop`
  `git merge feature/login`
- **Delete feature branch**
  `git branch -d feature/login`
- **Fix & push again**
  `git add .`
  `git commit -m "Fix bug"`
  `git push`

8. BEST PRACTICES

- ✓ Commit small, logical changes frequently.
- ✓ Write clear, concise commit messages.
- ✓ Use meaningful branch names.
- ✓ Pull latest changes before starting work.
- ✓ Never commit sensitive data or secrets.
- ✓ Review code before merging (PR).
- ✓ Keep `main` branch always stable.
- ✓ Use `.gitignore` to ignore unnecessary files.
- ✓ Delete merged branches regularly.
- ✓ Backup important work/stash when needed.
- ✓ Communicate clearly with your team.

9. USEFUL ALIASES (Examples)

```
alias gs='git status'          # status
alias ga='git add .'           # add all
alias gc='git commit -m'       # commit
alias gp='git push'            # push
alias gl='git log --oneline'   # log
alias co='git checkout'        # checkout
alias br='git branch'          # branch
```

Add to ~/.gitconfig

10. .GITIGNORE EXAMPLE (Node.js)

```
node_modules/
dist/
.env
*.log
coverage/
.DS_Store
Thumbs.db
*.local
```

11. COMMIT MESSAGE GUIDE

`<type>(scope): <short description>`

Types: `feat`, `fix`, `docs`, `style`, `refactor`, `test`, `chore`, `perf`

Example: `feat(login): add login API test`

12. REMEMBER

- Clean history = Happy team!
- Commit often, push safely!
- Think before you force push.
- Automate what you repeat.

Git is powerful, but with great power comes great responsibility! Use Git wisely & keep your repo clean!