yatlas

GIT & GITHUB
THE SAUCE

Git Fundamentals - The "Why" and the "What"

What is Version Control?

Analogy: Saving a file as "Project_v1.docx", "Project_v2.docx", etc., but infinitely more powerful.
The problem it solves: Tracking changes, reverting mistakes, understanding who changed what and why.

What is Git?

A Distributed Version Control System (DVCS). Explain "distributed" – everyone has a full copy of the repository's history.

Terminology:

- **Commit**: A snapshot of your project at a specific point in time. Each commit has a unique ID (hash) and records what changed, who changed it, and a message explaining why. It's like a save point in a video game that you can always return to.
- **Branch**: A lightweight, movable pointer to a specific commit. The default branch is usually main or master. Branches allow you to develop features or fix bugs in an isolated environment without affecting the main codebase. Think of it as creating an alternate timeline for your project.
- **Merge**: The act of combining the changes from one branch into another. This typically creates a special "merge commit" that has two parents, explicitly recording the point where the histories were joined.
- **Rebase**: An alternative to merging. Rebasing takes the commits from one branch and "replays" them on top of another branch. This results in a cleaner, linear project history, but it rewrites the commit history, which can be dangerous if misused on shared branches.
- **HEAD**: A special pointer that represents your current position in the repository's history. In most cases, HEAD points to the tip of the branch you are currently working on. When you check out a specific commit (instead of a branch), you enter a "detached HEAD" state.

Terminology 2:

- **Remote**: A version of your repository that is hosted on the internet or another network, such as on GitHub or GitLab. The default name for a remote is origin.
- **Clone**: The process of creating a full copy of a remote repository on your local machine. This is usually the first command you run when joining a project.
- **Push** (git push): Uploads your local commits to a remote repository, sharing your work with others.
- **Pull** (git pull): Fetches changes from a remote repository and integrates them into your current local branch. It is essentially a git fetch followed by a git merge.
- **Staging Area** (or Index): A preparatory area where you assemble the changes you want to include in your next commit. It allows you to commit related changes together, even if you've made unrelated changes in your working directory.

GitHub & Collaboration - The "Where" and "With Whom"

What is GitHub? (It's not Git!)

- A hosting service for Git repositories. It's a central place for teams to share code.
- Key concepts: Remote Repository, Forking, Cloning.

Basic Remote Operations (git CMDs on the next page)

- git clone <url>: Download a remote repo to your machine.
- git push <remote> <branch>: Upload your local commits to GitHub.
- git pull <remote> <branch>: This is the detailed explanation **[it's actually a git fetch (downloads new data) + git merge (integrates it into your current branch)]**

Essential git Commands

Before diving into advanced concepts, here's a solid foundation of the commands you'll use daily. Master this flow: Modify -> Stage -> Commit.

Command	Purpose	Example
git init	Turns the current directory into a new Git repository.	git init
git status	Shows the state of your working directory and staging area. It tells you what has changed and what is ready to be committed.	git status
git add <file>	Moves changes from your working directory to the staging area, preparing them for a commit.	git add index.html git add src/ (adds a folder) git add . (adds all changes)
git commit -m "message"	Takes a snapshot of the changes in the staging area and saves it to the repository's history.	git commit -m "Add user login form"
git log	Displays the commit history for the current branch.	git log git log --oneline (compact view) git log --all --graph (visualizes branches)
git checkout <branch-name>	Switches to a different branch.	git checkout main git checkout -b new-feature (creates and switches)
git branch	Lists all branches in your repository. The asterisk (*) shows your current branch.	git branch
git diff	Shows the exact differences (lines added/removed) in files that are not yet staged.	git diff

The Professional Daily Workflow (what is expected of you?)

Scenario: You arrive at your desk to work on a new feature.

1-Sync with the Team: Always start with a clean slate.

- git checkout main
- git pull origin main (Fetches the latest changes from the team and updates your local main branch).

2-Create a Feature Branch: Work in isolation.

- git checkout -b feature/user-authentication (The -b flag creates and switches to the branch in one step). Use descriptive names!

3- The Development Loop:

Code, Test, Stage, Commit, Repeat. Make small, atomic commits with clear messages ("Add login form HTML" is better than "fixed stuff").

4- Share Your Progress & Get Early Feedback (Advanced Workflow):

- After a few commits, you can push the branch and create a Draft Pull Request (PR) on GitHub.
- git push origin feature/user-authentication
- A Draft PR tells your team: "I'm working on this, feel free to look, but it's not ready for a full review yet." This is great for collaboration.

5- Working on a Subtask Without Blocking Yourself:

- You realize you need to fix an unrelated bug before finishing the feature.
- Commit your current work on the feature branch.
- Create a new branch from the latest main for the bug fix. Do NOT create it from your feature branch.
- git checkout main
- git pull origin main
- git checkout -b hotfix/typo-on-homepage
- This keeps changes isolated and makes merging easier.

6- Completing the Feature & Formal Review:

- Once the feature is complete, mark the Draft PR as "Ready for Review".
- Your teammates will review the code, add comments, and request changes.

7- Addressing Feedback:

- Make the requested changes on your same feature branch.
- Add new commits (e.g., "Address review comments: refactor validation logic").
- Push again: git push origin feature/user-authentication. The PR updates automatically.

8- The Merge:

- Once approved, a teammate (or you) clicks "Merge" on GitHub.
- This integrates your feature into the main branch on the remote repository.

9- Cleanup:

- git checkout main
- git pull origin main (Now your local main has the new feature!).
- Delete the feature branch locally (git branch -d feature/user-authentication) and on GitHub (there's usually a button after merging).

Advanced Concepts & Common Confusions

git pull vs git fetch

- git fetch: A "safe" command. It goes to the remote repo, downloads all new data (commits, branches), and updates your remote-tracking branches (e.g., origin/main). It does not change your working files.
- git pull: A "powerful" command. It does a git fetch and then immediately a git merge to integrate the new changes into your current branch. This can sometimes cause merge conflicts. Best practice: Often, it's safer to git fetch first, inspect what changed with git log origin/main, and then decide to merge (git merge origin/main) or rebase.

Merge vs. Rebase (click for youtube short explanation)

- Goal of Both: Integrate changes from one branch into another.
- Merge: Creates a new "merge commit" that ties the two histories together. Pros: Preserves exact history. Cons: History can get messy ("merge commits everywhere").
- Rebase: "Replays" your commits from one branch on top of another. It makes history look like a straight line. Pros: Clean, linear history. Cons: Rewrites history. **Golden Rule: Never rebase commits that have been shared with others (pushed to a public branch).If you rebase commits that others have already pulled, their local history will no longer match yours. This causes confusing conflicts and breaks collaboration because everyone's timeline becomes different.**

git reset vs git revert (click for youtube short explanation)

- git reset: Moves the branch pointer backwards. It erases history. Dangerous if you've already pushed the commits you're resetting.
- git revert: Creates a new commit that undoes the changes of a previous commit. Safe for public history because it doesn't erase anything. This is the preferred method for undoing changes on shared branches.

git checkout vs git switch / git restore

- git checkout is a confusing "Swiss Army knife" command. Newer versions of Git introduced:
- git switch <branch>: Use this specifically to switch branches. (Clearer than git checkout <branch>).
- git restore <file>: Use this to discard changes in your working directory. (Clearer than git checkout -- <file>).

Pull Requests (PRs)

When to Create a PR?

- For any non-trivial change (a new feature, a significant refactor, a bug fix).
- Even for small changes, if your team culture encourages review.

How to Write a Great PR Description:

- Title: Clear and concise. e.g., "FEAT: Add user login API endpoint".
- Template: Use a template if your team has one. Otherwise, include:

1. What & Why: What does this PR do? Why is this change necessary?
2. How: Briefly explain the implementation, if relevant.
3. Testing: How was this tested? "Tested locally by logging in with X and Y."
4. Screenshots/GIFs: If it's a UI change, a visual is worth a thousand words.
5. Checklist: e.g., [ ] Code reviewed, [ ] Tests pass, [ ] Documentation updated.

What to Do After Submitting a PR:

1. Notify your team (e.g., in Slack/Teams channel: "PR ready for review: [link]").
2. Address review comments promptly. Be polite and see it as a learning opportunity.
3. Keep the PR updated. If main changes, rebase your feature branch onto main to avoid conflicts. (git fetch origin, git rebase origin/main).
4. Once merged, celebrate and clean up (delete the branch).

Dealing with Merge Conflicts Step-by-Step

A conflict happens when Git cannot automatically merge changes because edits were made to the same part of a file in two different branches.

1- Identify the Conflict: Git will mark the conflicting sections in the file. You'll see markers like this:

```
<<<<<<< HEAD
code from your current branch
=======
code from the branch you're trying to merge
>>>>>>> feature-branch
```

2- Resolve the Conflict: Open the file in your editor. Decide what the final code should be. You must choose one set of changes, combine them, or write new code.

- Delete the conflict markers (<<<<<<<, =======, >>>>>>>).
- Keep the code that should be in the final version.

3- Complete the Merge: After resolving all conflicts in all files:

- Stage the resolved files: git add <file-name>.
- Commit the resolution: git commit -m "Merge branchX, resolve conflicts". Git will provide a default commit message.

.gitignore: Protecting Your Project

1. The .gitignore file is a plain text file where you list patterns for files and directories that Git should completely ignore. It is essential for several reasons:
2. Security: Prevents you from accidentally committing sensitive information like passwords, API keys, or SSH keys. Once a secret is committed, it becomes part of the history and is very difficult to remove entirely.
3. Cleanliness and Performance: Excludes generated files and dependencies that don't need to be version-controlled, such as:
4. node_modules/ (for Node.js projects)
5. *.log (log files)
6. build/ or dist/ (compiled code)
7. .env (environment variables)
8. How to use it: Create a file named .gitignore in the root of your repository. Each line is a pattern. You can use * as a wildcard and # for comments.

Example .gitignore for a Node.js project:

```
# Dependency directories
node_modules/
# Logs
npm-debug.log*
*.log
# Environment variables
.env
.env.local
# Build output
dist/
build/
```

Appendix: Complete Git Command Reference 1/4

Setup & Configuration

Command	Description
git config --global user.name "Your Name"	Set your commit author name
git config --global user.email "your.email@example.com"	Set your commit author email
git config --global alias.st "status"	Create shortcut: git st for git status

Repository Basics

Command	Description
git init	Initialize new Git repository in current
git clone <url>	Download repository from URL
git status	Show working directory status
git status -s	Short format status

Making Changes

Command	Description
git add <file>	Stage specific file
git add .	Stage all changes in current directory
git add -A	Stage all changes (including deletions)
git commit -m "message"	Commit staged changes with message
git commit -am "message"	Stage tracked files and commit (skip git add)
git commit --amend	Modify the previous commit

Appendix: Complete Git Command Reference 2/4

Viewing History

Command	Description
git log	Show commit history
git log --oneline	Compact one-line per commit
git log --all --graph	Visual history with branches
git log -p	Show changes in each commit
git show <commit>	Show specific commit details
git diff	Show unstaged changes
git diff --staged	Show staged changes

Branches

Command	Description
git branch	List all branches
git branch <name>	Create new branch
git branch -d <name>	Delete branch (safe)
git branch -D <name>	Force delete branch
git checkout <branch>	Switch to branch
git checkout -b <new-branch>	Create and switch to new branch
git switch <branch>	Switch to branch (newer command)
git switch -c <new-branch>	Create and switch (newer command)

Appendix: Complete Git Command Reference 3/4

Merging & Rebasing

Command	Description
git merge <branch>	Merge branch into current branch
git rebase <branch>	Rebase current branch onto target branch
git rebase -i <commit>	Interactive rebase (squash, edit commits)

Remote Repositories

Command	Description
git remote -v	List remote repositories
git remote add <name> <url>	Add new remote
git push <remote> <branch>	Push branch to remote
git push -u <remote> <branch>	Push and set upstream branch
git push --force-with-lease	Safe force push (recommended over -f)
git fetch <remote>	Download changes without merging
git pull <remote> <branch>	Fetch and merge changes
git pull --rebase	Fetch and rebase instead of merge

Appendix: Complete Git Command Reference 4/4

Undoing Changes

Command	Description
git restore <file>	Discard unstaged changes in file
git restore --staged <file>	Unstage file (keep changes)
git reset --hard HEAD	Discard all uncommitted changes
git reset --soft HEAD~1	Undo commit but keep changes staged
git reset --hard HEAD~1	Completely remove last commit
git revert <commit>	Create new commit that undoes changes

Stashing

Command	Description
git stash	Temporarily save uncommitted changes
git stash list	List all stashes
git stash pop	Apply most recent stash and remove it
git stash apply	Apply stash but keep it in list
git stash drop	Remove specific stash
Command	Description
git stash	Temporarily save uncommitted changes
git stash list	List all stashes

Good work you finished reading
“practice makes perfect”