Introduction to Git/GitHub: Version Control for Collaboration

1. What is Version Control?

Version Control is a system that records changes to a file or set of files over time so that you can recall specific versions later. It's essential for **tracking every modification** and allows you to:

- **Revert** files back to a previous state.
- **Restore** the entire project back to a previous state.
- **Compare** changes over time.
- **Detect** who last modified something that might be causing a problem.
- **Enable** multiple people to **collaborate** without overwriting each other's work.

There are **two** main types of Version Control:

- **Centralized Version Control Systems (CVCS):** Uses a single server to contain all the versioned files, and clients check out files from that central place.
  Example: Subversion (SVN).
- **Distributed Version Control Systems (DVCS):** Clients **fully mirror** the repository, including its full history. If the central server dies, any of the client repositories can be copied back up to restore the server. **Example:** Git.

2. What is Git?

Git is a free, open-source **Distributed Version Control System (DVCS)** designed to handle everything from small to very large projects with speed and efficiency. It was created in 2005 by **Linus Torvalds** after the Linux kernel community lost access to the version control system they had been using. Its architecture focused on **integrity**, **branching efficiency**, and **decentralized collaboration** quickly made it the **dominant** version control system in the software world.

Key Concepts

- **Local Focus:** Most operations in Git are performed locally on your machine, making it extremely fast.
- **Integrity:** Git is designed to ensure the integrity of your code. Everything is checksummed (verified by unique integrity hash) before it's stored, and it's retrieved by that checksum.
- Git manages your project in **three** main states:
    1. **Working Directory:** The files you are currently editing.
    2. **Staging Area (or Index):** A place where you prepare a snapshot of your changes before committing them.
    3. **Git Directory (or Repository):** Where Git permanently stores the history and metadata for your project.

3. What is GitHub?

GitHub is the world's most popular **web-based hosting service for Git repositories**. It provides a central location for teams to share, collaborate, and manage Git projects.

Key Features of GitHub

- **Remote Storage:** Provides a backup and central point of collaboration for your Git repository.
- **Collaboration Tools:** Includes features like **Pull Requests** (for reviewing and merging code), **Issue Tracking**, and project management tools.

Other Git Hosting Services (Clients)

GitHub is one of many remote Git hosting services. Other popular alternatives include:

- **GitLab:** A complete DevSecOps platform that can be self-hosted.
- **GitKraken:** A graphical Git client with a visual commit graph, conflict resolver, and productivity-focused UI.
- **Bitbucket:** Popular with teams using other Atlassian products like Jira.
- **Azure DevOps (Azure Repos):** Microsoft's platform offering Git repos integrated with a full suite of DevOps tools.

4. Setting Up a Repository (Repo)

A **repository** is the folder that contains your project files and the Git history.

Core Configuration Commands

Before using Git, you should set your identity:

Command	Description
git config --global user.name "[firstname lastname]"	Sets a name that is identifiable for credit when reviewing version history.
git config --global user.email "[valid-email]"	Sets an email address that will be associated with each history marker.

Starting a New Project

To create a new, empty Git repository in your current directory:

git init

Cloning an Existing Project

To get a copy of an existing remote repository onto your local machine:

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

5. Making and Tracking Changes

This is the standard workflow: Edit files in the **Working Directory**, move them to the **Staging Area**, and then **Commit** them to the **Git Directory** (history).

The Essential Workflow Commands

Command	Description
git status	Shows the state of the working directory and staging area. It tells you which files are untracked, modified, and staged.
git add [file]	Stages the specified file, preparing it to be included in the next commit.
git remote add [url]	Adds a remote repository URL to your local Git project. It tells Git where to push and pull changes from.
git commit -m "[message]"	Records the staged snapshot permanently into the repository history with a descriptive message.
git diff	Shows the differences between the working directory and the staging area, or between two commits/branches.
git log	Shows the commit history, including commit hashes, authors, dates, and messages.

Sending/Receiving Changes (Remote Operations)

Push: When you **upload** your local **commits** to a remote repository like GitHub. It updates the remote branch with your new changes so others can **access** them.

Pull Request (PR): A GitHub feature where you **propose changes** from one branch to be **merged** into another.

Command	Description
git push	Sends your local committed changes to the remote repository.
git pull	Fetches changes from the remote repository and automatically merges them into your current local branch.

6. Branching and Merging

Branches allow you to diverge from the main line of development and work in isolation. This is essential for developing new features, fixing bugs, and experimenting without affecting the stable code.

Command	Description
git branch	Lists all local branches.
git branch [name]	Creates a new branch.
git checkout [branch-name]	Switches your working directory to the specified branch.
git checkout -b [name]	Shortcut to **create and switch** to a new branch.
git merge [branch-name]	Integrates the specified branch's history into your current branch.

7. Handling a Merge Conflict

A **merge conflict** occurs when Git is unable to automatically integrate changes because the **same line** of the same file has been modified differently in the two branches being merged.

Conflict Resolution Steps

1. Git **notifies** you of the conflict.
2. **Check status:** Run git status to see the conflicted files.
3. **Manually edit** the conflicted file(s) to choose which changes to keep.
   Conflicts are marked by special lines:
   - <<<<<<< HEAD (Your changes)
   - ======= (Separator)
   - >>>>>>> [branch-name] (Incoming changes)
4. **Remove** the conflict markers (<<<<<<<, =======, >>>>>>>).
5. **Stage** the resolved file: *git add [conflicted-file]*
6. **Commit** the resolution: *git commit -m "Resolved merge conflict..."*

8. Undo and History Rewriting

Git offers powerful tools for modifying history, but use these commands **cautiously**, especially on shared branches.

Command	Description
git commit --amend	**Replaces the last commit** with a new commit containing the staged changes and the original commit's message (or a new one). Used to fix a small mistake or forget a file in the last commit.
git reset [mode] [commit-hash]	Moves the **HEAD pointer** to a specified commit, effectively **rewriting history** by abandoning commits. **Modes:** *--soft* (keeps changes staged), *--mixed* (default, keeps changes in working directory), *--hard* (deletes all changes).
git revert [commit-hash]	Creates a **new commit** that **undoes** the changes introduced by the specified commit. This is a **safe** way to undo shared history, as it does not rewrite history.
git rebase [branch-name]	**Moves** or **combines** a sequence of commits to a **new** base commit. Useful for keeping history clean by incorporating upstream changes without merge commits.
git checkout -- [file]	**Discards changes** in the working directory for a specific file (reverts to the last committed or staged version).

9. Conclusion

Git and GitHub are fundamental tools in modern software development for several reasons:

- **Collaboration:** Enables multiple developers to work on the same codebase simultaneously and safely.
- **Auditing:** Provides a complete, immutable record of every change made to the project.
- **Safety:** Allows developers to experiment and make mistakes, knowing they can always revert to a stable version.
- **Industry Standard:** Proficiency in Git/GitHub is a core requirement for almost every technical role.