Git SCM and Versioning

Introduction to Version Control

- - What is Version Control?
- - Types of Version Control Systems
- - Importance of Versioning in Software Development

Why Use Version Control?

1. **Tracks Changes Over Time**
   - Allows developers to see what was changed, who changed it, and when.
2. **Collaboration and Teamwork**
   - Enables multiple people to work on the same project simultaneously without conflicts.
3. **Backup and Recovery**
   - Provides a safety net by allowing rollback to previous versions in case of mistakes.
4. **Branching and Merging**
   - Developers can create separate branches to work on features or bug fixes independently and later merge them.
5. **Audit and Compliance**
   - Maintains a history of changes for accountability and regulatory compliance

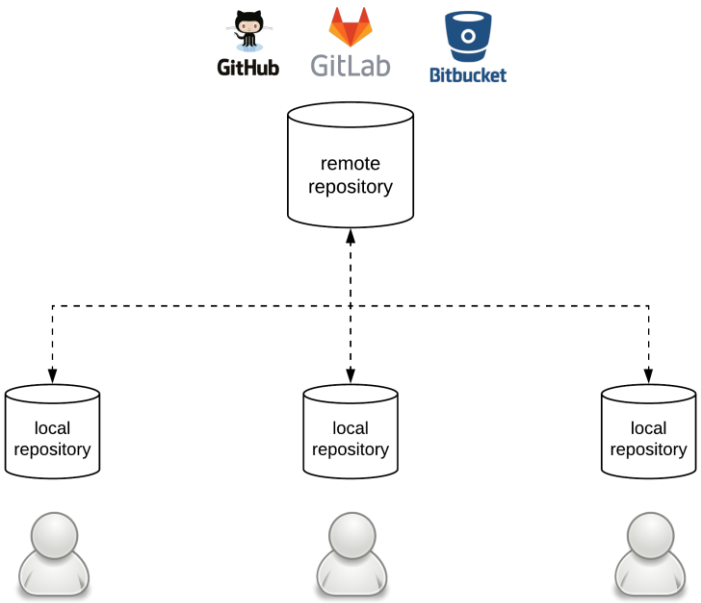
Types of Version Control Systems (VCS)

1. **Local Version Control Systems (LVCS)**
   - Tracks file versions on a single computer. Example: Saving multiple copies of a file with different names.
2. **Centralized Version Control Systems (CVCS)**
   - A central server holds all versions, and developers pull/push changes. Example: **SVN, Perforce**.
3. **Distributed Version Control Systems (DVCS)** *(Most Modern Approach)*
   - Each developer has a complete history of the repository, allowing offline work. Example: **Git, Mercurial**.

Introduction to Git

- What is Git?
- Why use Git?
- Git vs. Other Version Control Systems

GIT



- **What is Git?**
  Git is a **Distributed Version Control System (DVCS)** that allows developers to track changes in their code, collaborate efficiently, and manage software development projects with ease. It was created by **Linus Torvalds** in 2005 to manage the development of the Linux kernel.
- **Why Git?**
- Git is widely used because it is:
  - **Distributed**: Every developer has a full copy of the repository, making it independent of a central server.
  - **Fast and Efficient**: Git handles large projects with speed and efficiency.
  - **Secure**: Uses cryptographic hashing (SHA-1) to track every change.
  - **Supports Branching and Merging**: Developers can work on features separately and merge them seamlessly.

Key Features of GIT

1. **Distributed Version Control**
   - Every user has a full copy of the repository (not just a snapshot).
   - Can work offline and sync later.
2. **Branching and Merging**
   - Developers can create **branches** to work on different features.
   - Merge branches back into the main codebase when ready.
3. **History Tracking**
   - Git maintains a **commit history** of all changes.
   - Each commit has a unique **hash ID**.
4. **Collaboration & Remote Repositories**
   - Developers can **push** and **pull** code to/from remote repositories (like GitHub, GitLab, Bitbucket).
5. **Lightweight and Fast**
   - Git is optimized for speed, making it efficient for large projects

Git Basics

- Installing Git
- Basic Git Commands (init, clone, add, commit, status, log)
- Understanding Git Workflow

Create new repository – command line

- **Step 1: Initialize a New Repository**
  - mkdir my-project
  - cd my-project
  - Git init
- **Step 2 : Add a New File and Track changes**
  - echo "Hello, Git!" > README.md
  - git add README.md
- **Step 3: Commit the Changes (** This saves the changes to your local repository.)
  - git commit -m "Initial commit“
- **Step 4: Add a Remote Repository (Optional)**
  - git remote add origin <repository_url>
  - Git push –u origin main
- **Step 5: Verify**
  - Git status
  - Git log --oneline

Branching and Merging

- What are Branches?
- Creating and Managing Branches (checkout, branch, merge, rebase)
- Handling Merge Conflicts

GIT Rebase

- Git **rebase** is a way to integrate changes from one branch into another by **moving** or **replaying** commits on top of another branch, resulting in a **clean and linear history**.

Feature	git merge	git rebase
History	Creates a merge commit (fast-forward or non-fast-forward).	Moves commits to the latest base for a cleaner history.
Commit Order	Keeps original branch history intact.	Rewrites commit history by applying commits on top of the latest branch.
Usage	Good for preserving commit history.	Good for keeping history linear.

GIT Rebase example

- Check out the feature branch
  - git checkout feature-branch
- Ensure main is updated
  - git checkout main
  - git pull origin main
- Switch Back to feature-branch
  - git checkout feature-branch
  - git rebase main

Working with Remote Repositories

- - Setting up GitHub/GitLab
- - push, pull, fetch Commands
- - Cloning and Forking

Git Advanced Features

- - Stashing Changes
- - Interactive Rebase
- - Git Hooks
- - Git Bisect for Debugging

Best Practices in Git

- - Writing Good Commit Messages
- - Using .gitignore Properly
- - Keeping Repositories Clean
- - Git Workflows (Feature Branching, GitFlow)

Hands-on Demonstration

- - Live Demo of Git Commands
- - Sample GitHub Repository Walkthrough