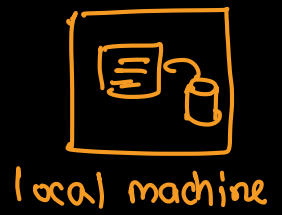
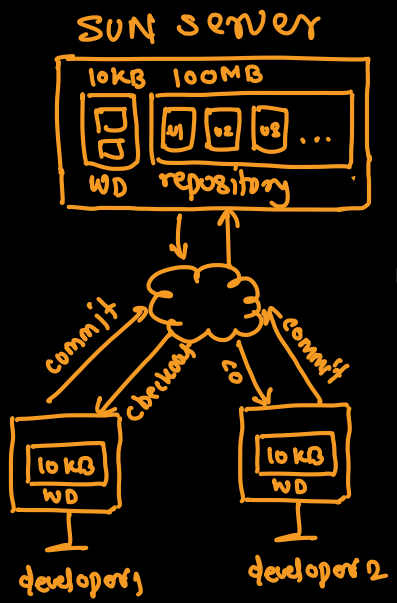
Source Code Management / Version Control System

Types of Version Control Systems

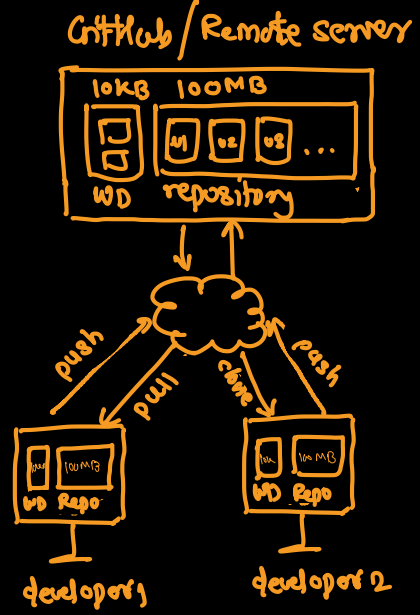
- **Local VCS** → decreprecated
  - Example: RCS (Revision Control System)
  - **Architecture:**
    - Tracks changes on your local machine only
  - **Advantages:**
    - The simplest form
  - **Disadvantages:** single point of failure
    - Limited because there's no collaboration capability



- **Centralized VCS (CVCS)**
  - Examples: **SVN (Subversion)**, **Perforce**, **CVS**
  - **Architecture:**
    - Single central server holds the repository
    - Developers check out files from this server
  - **Advantages:**
    - Simple mental model
    - Fine-grained access control
    - Easier to understand for beginners
  - **Disadvantages:**
    - Single point of failure
    - Requires network access to commit
    - Slower operations (network dependent)
    - If the server crashes without backups, you lose everything



- **Distributed VCS (DVCS)**
  - Examples: **Git**, **Mercurial**, **Bazaar**
  - **Architecture:**
    - Every developer has a complete copy of the repository including full history
  - **Advantages:**
    - Work offline completely
    - Fast operations (local)
    - No single point of failure
    - Flexible workflows
    - Easy branching and merging *
  - **Disadvantages:**
    - Steeper learning curve
    - More complex mental model
    - Large repositories can be unwieldy




Git

What is Git ?

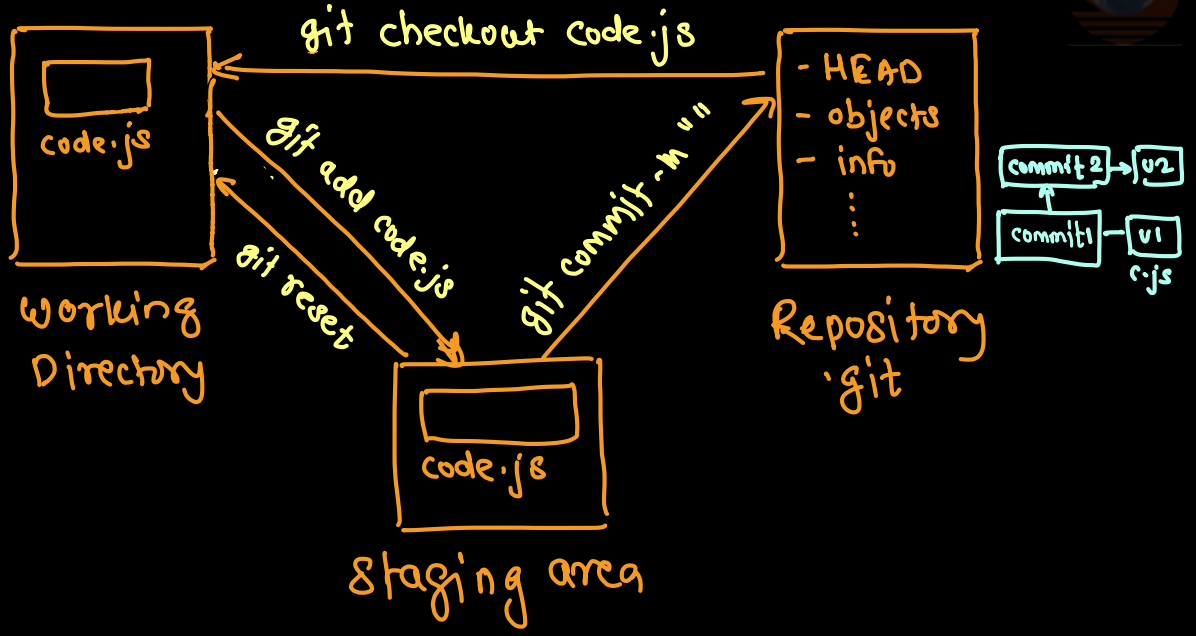
- Git is a **distributed version control system** created by Linus Torvalds in 2005 (the same person who created Linux)
- It's now the most widely used version control system in the world
- At its core, Git is a **content-addressable filesystem** with a version control interface on top
- It's essentially a database that tracks changes to your files over time, allowing you to
  - Save snapshots of your project at any point
  - Go back to previous versions
  - Create parallel versions (branches) to experiment
  - Collaborate with others without overwriting their work
- **Why Git Was Created**
  - Linus Torvalds needed a VCS for Linux kernel development after the existing system (BitKeeper) became unavailable
  - He designed Git with these goals:
    - **Speed:** Operations should be fast
    - **Distributed:** No single point of failure
    - **Support for non-linear development:** Thousands of parallel branches
    - **Handle large projects:** Like the Linux kernel efficiently

Basic Git Workflow

- Initialize or Clone
  - git init
  - git clone <url>
- Make Changes
- Check Status
  - git status
- Stage Changes
  - git add file.txt
  - git add .
- Commit Changes
  - git commit -m "Add new feature"
- View History
  - git log
  - git log --oneline

The Three-Stage Architecture

- **Working Directory**
  - Your actual project files on disk
  - Where you make all your edits
  - Contains both tracked and untracked files
- **Staging Area (Index)** → virtual area (no physical directory)
  - A **holding area** between working directory and repository
  - Stores a snapshot of what will go into the next commit
  - Lets you craft precise, logical commits
  - Lives in .git/index file
- **Repository (.git directory)**
  - The permanent history
  - Contains all committed snapshots (versions)
  - The database of your project's complete history



Branching

Branching

- Git branching is one of the most powerful features that sets Git apart from other version control systems
- A branch in Git is simply a **lightweight, movable pointer to a commit**
- It's not a copy of your code, not a separate directory - just a 41-byte file containing a SHA-1 hash pointing to a commit
- Because branches are just pointers, creating and switching between branches is nearly instantaneous, even in massive projects
- This is fundamentally different from older VCS systems where branching meant copying entire directories

How Branches Work Internally

- **The Commit Graph**
  - Git stores history as a directed acyclic graph (DAG) of commits
  - Each commit points to its parent(s):
    - A --- B --- C --- D (main)
  - Each letter represents a commit with:
    - A unique SHA-1 hash (like a3f5b2c...)
    - A pointer to its parent commit
    - A pointer to a tree (snapshot of files)
    - Metadata (author, date, message)
- The branch main is literally just a file in .git/refs/heads/main containing the contents
- The HEAD Pointer
  - HEAD is a special pointer that tells Git which branch you're currently on
  - HEAD → main → Commit C
  - When you're on the main branch, HEAD points to main, which points to the latest commit

Branching Workflow

- Creating a Branch
  - git branch feature-login
- Switching Branches
  - git checkout feature-login
  - git switch feature-login
- Create and Switch in One Command
  - git checkout -b feature-login
  - git switch -c feature-login
- Rename branch
  - git branch –M <new name>
- Merge branch
  - git checkout main
  - git merge feature-login
- Delete branch
  - git branch –d feature-login

Merge Conflicts

- Conflicts happen when
  - Both branches modified the same lines in the same file
  - One branch deleted a file the other modified
  - Both branches created files with the same name
- **On main** (file.txt)

```
function login() {
    return authenticateUser();
}
```

On feature (file.txt)

```
function login() {
    return validateAndAuthenticate();
}
```

- After git merge feature

```
function login() {
<<<<<<< HEAD
    return authenticateUser();
=======
    return validateAndAuthenticate();
>>>>>>> feature
}
```

Resolving Conflicts

- Open the conflicted file and find the markers
  - <<<<<<< HEAD: Your current branch's version
  - =======: Separator
  - >>>>>>> branch-name: The incoming branch's version
- Edit the file to keep what you want

```
function login() {
    return validateAndAuthenticate();
}
```

- Mark as resolved
  - git add file.txt
- Complete the merge
  - git commit