Introduction to Version Control & Git

What is Version Control?

Definition: Version control is a system that records changes to files over time so you can recall specific versions later. It's like a **time machine** for your code.

Why use it?

- Tracks changes over time.
- Restores older versions when something breaks.
- Allows multiple people to work on the same project.
- Helps experiment safely.

The problem without version control:

```
index.html
index_final.html
index_final_final.html
index_final_v2_really_final.html
```

This is messy, error-prone, and hard to manage.

Centralized vs Distributed Version Control

- **Centralized (CVCS)** – e.g., SVN
    - One central server stores the code.
    - If the server goes down, no one can commit changes.

- **Distributed (DVCS)** – e.g., Git
    - Every developer has a complete copy of the repository.
    - You can commit changes offline.
    - More robust against server failures.

What is Git?

- Git is a **distributed version control system** created by **Linus Torvalds** (the creator of Linux) in 2005.
- Git tracks changes in your files, especially source code.
- Works **locally first** and then syncs with remote repositories.

Git vs GitHub

- **Git** → The tool that manages your code history (installed on your computer).
- **GitHub** → A hosting service for Git repositories (like Google Drive for your Git projects). Also alternatives: GitLab, Bitbucket.

Installing Git

Windows

1. Go to https://git-scm.com/.
2. Download the installer.
3. Follow the prompts (use default settings if unsure).

macOS

```
brew install git
```

Linux (Debian/Ubuntu)

```
sudo apt update
sudo apt install git
```

Configuring Git

Run these commands in your terminal after installing Git:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global core.editor "code --wait"  # optional, sets VS Code as editor
```

To check your config:

```
git config --list
```

Git Workflow Basics

Git has three key areas:

1. **Working Directory** – Where you edit files.
2. **Staging Area** – Where you prepare files for committing.
3. **Repository** – Where committed changes are stored permanently.

Basic flow:

```
Edit files → git add → git commit
```

First Git Commands

Let's try:

```
mkdir my-first-git-project
cd my-first-git-project
git init
```

Output:

```
Initialized empty Git repository in ...
```

You've just created your first **local Git repository**.

Mini Exercise

1. Install Git on your system.
2. Configure your name and email.
3. Create a folder named `my-first-repo`.
4. Initialize it with `git init`.
5. Run `git status` and see what it says.

First Steps with Git

Creating a Repository

There are two ways to start working with Git:

1. Starting from scratch

```
mkdir my-first-repo
cd my-first-repo
git init
```

You'll see:

```
Initialized empty Git repository in /path/to/my-first-repo/.git/
```

This `.git` folder is the brain of your repository — it stores the entire history of your project.

2. Cloning an existing repository

```
git clone https://github.com/user/repo.git
```

This downloads the entire project **with history** from a remote server like GitHub.

Adding Files

Let's create a file:

```
echo "Hello Git!" > hello.txt
```

Check what Git sees:

```
git status
```

You'll see:

```
Untracked files:
  hello.txt
```

Untracked means Git sees the file but hasn't started tracking it.

Staging Changes

To start tracking the file:

```
git add hello.txt
```

To stage everything at once:

```
git add .
```

At this point, the file is **staged** — ready to be saved into history.

Committing Changes

A commit is like taking a snapshot of your project:

```
git commit -m "Add hello.txt with a greeting"
```

Commit messages should describe *why* the change was made, not just *what* changed.

Viewing History

To see the commit history:

```
git log
```

For a shorter, cleaner view:

```
git log --oneline
```

Example:

```
a1b2c3d Add hello.txt with a greeting
```

Understanding the Git Workflow

Files in Git move through three main states:

1. **Untracked** – not being tracked yet.
2. **Staged** – ready to be committed.
3. **Committed** – saved permanently in the Git history.

Basic flow:

```
Working Directory → git add → Staging Area → git commit → Repository
```

Making Further Changes

Edit the file:

```
echo "This is my first change" >> hello.txt
```

Check the status:

```
git status
```

You'll see it's "modified." Stage and commit:

```
git add hello.txt
git commit -m "Update hello.txt with a new line"
```

Quick Command Recap

```
git init                # start a new repository
git clone <url>         # copy a repository
git status              # check file states
git add <file>          # stage changes
git commit -m "msg"     # save staged changes
git log --oneline       # view history in short form
```

Practice Challenge

1. Create a folder `git-practice`.
2. Initialize it as a Git repo.
3. Add a file `notes.txt` with some text.
4. Stage and commit it with a clear message.

5. Modify `notes.txt` and commit the change.
6. Run `git log --oneline` to see your commits.

Tracking & Managing Changes

Checking Status

The most important diagnostic command in Git:

```
git status
```

This tells you:

- Which branch you're on
- What files are modified
- What files are staged
- What files are untracked

Status States

```
# Untracked (new file)
Untracked files:
  newfile.txt

# Modified (changed but not staged)
Changes not staged for commit:
  modified: existing.txt

# Staged (ready to commit)
Changes to be committed:
  new file: newfile.txt
  modified: existing.txt
```

Viewing Changes

See Unstaged Changes

```
git diff
```

Shows what you've changed but haven't staged yet.

See Staged Changes

```
git diff --staged
```

Shows what will go into your next commit.

See Changes Between Commits

```
git diff HEAD~1 HEAD
```

Compares the last commit with the current one.

Undoing Changes

Git provides several ways to undo changes depending on where they are:

1. Discard Working Directory Changes

To restore a file to its last committed state:

```
git restore file.txt
```

Or for all files:

```
git restore .
```

Warning: This permanently discards uncommitted changes!

2. Unstage Files

To remove files from staging area but keep changes:

```
git restore --staged file.txt
```

Or the older syntax:

```
git reset HEAD file.txt
```

3. Amend the Last Commit

Forgot to include a file or want to change the commit message?

```
# Stage the forgotten file
git add forgotten.txt

# Amend the previous commit
git commit --amend -m "New commit message"
```

4. Reset to a Previous Commit

Soft Reset (keeps changes in staging):

```
git reset --soft HEAD~1
```

Mixed Reset (keeps changes in working directory):

```
git reset HEAD~1
```

Hard Reset (discards all changes):

```
git reset --hard HEAD~1
```

Warning: `--hard` permanently deletes uncommitted work!

Ignoring Files

Not all files should be tracked by Git (e.g., passwords, compiled files, system files).

Creating .gitignore

Create a `.gitignore` file in your repository root:

```
touch .gitignore
```

Common .gitignore Patterns

```
# Ignore specific files
secret.txt
config.env

# Ignore file types
*.log
*.tmp
*.cache

# Ignore directories
node_modules/
build/
dist/

# Ignore files in any directory
**/*.bak
```

```
# Exception: Track this file even if ignored
!important.log
```

Global .gitignore

Set up a global ignore file for all repositories:

```
git config --global core.excludesfile ~/.gitignore_global
```

Common .gitignore Templates

For Node.js projects:

```
node_modules/
npm-debug.log
.env
dist/
*.log
```

For Python projects:

```
__pycache__/
*.py[cod]
*$py.class
venv/
.env
*.egg-info/
```

For IDE/Editor files:

```
.vscode/
.idea/
*.swp
.DS_Store
Thumbs.db
```

Removing Files from Git

Remove File from Repository and Disk

```
git rm file.txt
git commit -m "Remove file.txt"
```

Remove File from Repository but Keep on Disk

```
git rm --cached file.txt
git commit -m "Stop tracking file.txt"
```

This is useful when you accidentally committed a file that should be ignored.

Moving/Renaming Files

Git tracks file movements:

```
git mv oldname.txt newname.txt
git commit -m "Rename oldname.txt to newname.txt"
```

This is equivalent to:

```
mv oldname.txt newname.txt
git rm oldname.txt
git add newname.txt
```

Practical Examples

Example 1: Fixing a Mistake

```
# You accidentally staged a file
git add passwords.txt

# Unstage it
git restore --staged passwords.txt

# Add it to .gitignore
echo "passwords.txt" >> .gitignore

# Stage and commit .gitignore
git add .gitignore
git commit -m "Add .gitignore to exclude sensitive files"
```

Example 2: Cleaning Up Working Directory

```
# See what's changed
git status

# Review the changes
git diff

# Discard changes to a specific file
git restore style.css

# Or discard all changes
git restore .
```

Command Summary

Command	Description
`git status`	Show working tree status
`git diff`	Show unstaged changes
`git diff --staged`	Show staged changes
`git restore [file]`	Discard working directory changes
`git restore --staged [file]`	Unstage files
`git reset --soft HEAD~1`	Undo last commit, keep changes staged
`git reset HEAD~1`	Undo last commit, keep changes unstaged
`git reset --hard HEAD~1`	Undo last commit, discard changes
`git rm [file]`	Remove file from repository
`git rm --cached [file]`	Stop tracking file
`git mv [old] [new]`	Rename/move file

Exercise

1. Create a new repository with several files
2. Make changes to multiple files
3. Use `git status` and `git diff` to review changes
4. Stage only some changes
5. Create a `.gitignore` file and add patterns
6. Practice undoing changes with `git restore`
7. Try amending a commit with `git commit --amend`
8. Experiment with different reset options (be careful with `--hard`!)

Challenge: Create a file with sensitive data, commit it, then properly remove it from history and add it to `.gitignore`.

Branching & Merging

Why Branches Matter

Branches allow you to:

- Work on features without affecting the main code
- Experiment safely
- Collaborate without conflicts
- Maintain multiple versions of your project

Think of branches as parallel universes of your code.

Understanding Branches

What is a Branch?

A branch is a movable pointer to a commit. The default branch is usually called `main` (or `master` in older repositories).

View Current Branch

```
git branch
```

The asterisk (*) shows your current branch:

```
* main
  feature-login
  bugfix-header
```

View All Branches (Including Remote)

```
git branch -a
```

Creating and Switching Branches

Create a New Branch

```
git branch feature-navbar
```

Switch to a Branch

```
git checkout feature-navbar
```

Or with the newer command:

```
git switch feature-navbar
```

Create and Switch in One Command

```
git checkout -b feature-navbar
```

Or:

```
git switch -c feature-navbar
```

Working with Branches

Making Changes on a Branch

```
# Create and switch to new branch
git checkout -b feature-login

# Make changes
echo "Login form" > login.html
git add login.html
git commit -m "Add login form"

# Your changes exist only on this branch
```

Switching Between Branches

```
# Switch back to main
git checkout main

# login.html doesn't exist here!

# Switch back to feature branch
git checkout feature-login

# login.html is back!
```

Merging Branches

Fast-Forward Merge

When there are no divergent commits, Git simply moves the pointer forward:

```
# On main branch
git checkout main
```

```
# Merge feature branch
git merge feature-navbar
```

Output:

```
Fast-forward
 navbar.html | 10 ++++++++++
 1 file changed, 10 insertions(+)
```

Three-Way Merge

When branches have diverged, Git creates a merge commit:

```
git checkout main
git merge feature-login
```

Git will open an editor for the merge commit message.

Resolving Merge Conflicts

Conflicts occur when the same lines are changed in different branches.

What a Conflict Looks Like

```
git merge feature-branch
```

Output:

```
Auto-merging index.html
CONFLICT (content): Merge conflict in index.html
Automatic merge failed; fix conflicts and then commit the result.
```

Conflict Markers in File

```
<<<<<<< HEAD
<h1>Welcome to Our Site</h1>
=======
<h1>Welcome to My Website</h1>
>>>>>>> feature-branch
```

Resolving Conflicts

1. Open the conflicted file
2. Decide which changes to keep
3. Remove conflict markers
4. Stage and commit

```
# After editing the file
git add index.html
git commit -m "Resolve merge conflict in index.html"
```

Conflict Resolution Strategies

Understanding "ours" and "theirs":

During a merge conflict, Git uses specific terminology: - **"ours"** = The branch you're currently on (the branch you're merging INTO) - **"theirs"** = The branch you're merging FROM (the incoming changes)

Keep current branch changes (ours):

```
git checkout --ours index.html
```

This keeps the version from your current branch, discarding all changes from the incoming branch. Works properly when: - You're certain your current branch has the correct implementation - The incoming changes are outdated or incorrect - You want to maintain consistency with other files in your branch

Keep incoming branch changes (theirs):

```
git checkout --theirs index.html
```

This accepts all changes from the branch you're merging, discarding your current branch's version. Works properly when: - The incoming branch has the most up-to-date or correct version - Your current changes are no longer needed - You want to fully adopt the incoming implementation

Important Note: These commands work ONLY during an active merge conflict. They replace the entire file with either version, not individual conflict sections.

Use a merge tool:

```
git mergetool
```

Branch Management

Delete a Branch

After merging, you can delete the branch:

```
# Delete local branch
git branch -d feature-navbar

# Force delete (if not merged)
git branch -D feature-navbar
```

Rename a Branch

```
# Rename current branch
git branch -m new-name

# Rename a different branch
git branch -m old-name new-name
```

List Merged/Unmerged Branches

```
# Show merged branches
git branch --merged

# Show unmerged branches
git branch --no-merged
```

Branching Strategies

Feature Branch Workflow

```
# 1. Create feature branch
git checkout -b feature-shopping-cart

# 2. Work on feature
# ... make commits ...

# 3. Merge back to main
git checkout main
git merge feature-shopping-cart

# 4. Delete feature branch
git branch -d feature-shopping-cart
```

Hotfix Workflow

```
# 1. Create hotfix from main
git checkout main
git checkout -b hotfix-security

# 2. Fix the issue
# ... make changes ...
git commit -m "Fix security vulnerability"
```

```
# 3. Merge to main
git checkout main
git merge hotfix-security

# 4. Also merge to develop if exists
git checkout develop
git merge hotfix-security
```

Visualizing Branches

See Branch Graph

```
git log --graph --oneline --all
```

Output:

```
* 3a4f5d6 (HEAD -> main) Merge feature-login
|\
| * 8b9c0d1 (feature-login) Add login form
* | 7e2f3a5 Update homepage
|/
* 1d2e3f4 Initial commit
```

See Branch Divergence

```
git log main..feature-branch
```

Shows commits in feature-branch that aren't in main.

Best Practices

1. **Keep branches focused** - One feature per branch

2. **Use descriptive names** - `feature-user-auth` not `new-stuff`
3. **Delete merged branches** - Keep repository clean
4. **Merge regularly** - Don't let branches diverge too much
5. **Test before merging** - Ensure branch works correctly

Common Branch Naming Conventions

- `feature/` - New features (feature/user-login)
- `bugfix/` - Bug fixes (bugfix/header-alignment)
- `hotfix/` - Urgent production fixes (hotfix/security-patch)
- `release/` - Release preparation (release/v2.0)
- `chore/` - Maintenance tasks (chore/update-dependencies)

Command Summary

Command	Description
`git branch`	List branches
`git branch [name]`	Create branch
`git checkout [branch]`	Switch branch
`git checkout -b [branch]`	Create and switch
`git switch [branch]`	Switch branch (newer)
`git switch -c [branch]`	Create and switch (newer)
`git merge [branch]`	Merge branch into current
`git branch -d [branch]`	Delete branch
`git branch -m [new-name]`	Rename branch
`git log --graph --oneline --all`	Visualize branches

Practice Exercise

1. Create a new repository
2. Create a file `main.txt` with “Main branch content”
3. Create a branch called `feature-a`
4. Add a file `feature-a.txt` and commit
5. Switch back to main
6. Create another branch `feature-b` from main
7. Add a file `feature-b.txt` and commit
8. Merge `feature-a` into main
9. Merge `feature-b` into main
10. Create a conflict intentionally and resolve it

Advanced: Try rebasing instead of merging to maintain a linear history.

Essential Remote Repository Commands

Connecting to Remotes

View Remotes

```
git remote -v
```

Add Remote

```
git remote add origin https://github.com/username/repository.git
```

Change Remote URL

```
git remote set-url origin https://github.com/username/new-repo.git
```

Core Operations

Clone Repository

```
git clone https://github.com/username/repository.git
```

Push Changes

```
# First push (set upstream)
git push -u origin main
```

```
# Regular push
git push

# Push specific branch
git push origin branch-name
```

Pull Changes

```
# Pull (fetch + merge)
git pull

# Pull specific branch
git pull origin branch-name
```

Fetch Changes

```
# Fetch without merging
git fetch

# Fetch all remotes
git fetch --all
```

Branch Management

List Remote Branches

```
git branch -r
```

Delete Remote Branch

```
git push origin --delete branch-name
```

Git Stash - Essential Commands

What is Git Stash?

Git stash temporarily saves your uncommitted changes so you can work on something else, then come back and re-apply them later.

Core Stash Commands

Save Changes to Stash

```
# Stash all changes
git stash

# Stash with a message
git stash save "work in progress on feature X"

# Include untracked files
git stash -u
```

View Stashes

```
# List all stashes
git stash list
```

Output example:

```
stash@{0}: On main: work in progress on feature X
stash@{1}: WIP on develop: 5002d47 fix conflict
```

Apply Stash

```
# Apply most recent stash
git stash apply

# Apply specific stash
git stash apply stash@{2}

# Apply and remove from stash list
git stash pop
```

Remove Stashes

```
# Remove most recent stash
git stash drop

# Remove specific stash
git stash drop stash@{1}

# Clear all stashes
git stash clear
```

Useful Stash Operations

View Stash Contents

```
# Show files in latest stash
git stash show

# Show detailed diff
git stash show -p

# Show specific stash diff
git stash show -p stash@{1}
```

Create Branch from Stash

```
# Create new branch and apply stash
git stash branch new-feature-branch
```

Stash Specific Files

```
# Interactive stash
git stash -p
```

Common Use Cases

Switch Branches Quickly

```
# Working on feature, need to fix bug on main
git stash
git checkout main
# Fix bug...
git checkout feature-branch
git stash pop
```

Pull Without Committing

```
git stash
git pull
git stash pop
```

Commands Summary

Command	Description
`git stash`	Save changes to stash
`git stash list`	List all stashes
`git stash apply`	Apply stash without removing
`git stash pop`	Apply and remove stash
`git stash drop`	Delete a stash
`git stash show`	View stash contents
`git stash clear`	Remove all stashes
`git stash branch [name]`	Create branch from stash