Git

Session 1

- Developing an application in a team
- Issues developers face when working in a team
- Introduction to code versioning system
- History of code versioning system
    - Different tools available for versioning
    - Software development workflow
- Introduction to git
- Introduction to git repository and git structure
- Adding code to git
- Creating and merging different git branches

Developing an application in a team

Why to use Git?

To follow a clear **workflow** which supports collaboration, minimizes conflicts and maintains code quality.

Steps to Develop in a team:

Step 1 - Initial Setup

- Create a Remote Repository (e.g., GitHub, GitLab, Bitbucket)

  One team member creates the repo and shares access with the team

- Clone the Repository

  Each team member runs: **git clone https://github.com/your-org/your-repo.git**

Step 2 - Use a Branching Strategy

Common models include:main or master

- **feature/xyz:** for each feature or task
- **bugfix/xyz, hotfix/xyz, release/xyz**: for other purposes
- **Create a new branch for each task:** git checkout -b feature/login-page

Steps to Develop in a team:

Step 3 - Development Workflow

1 - Pull latest changes before you start:

```
git checkout main
git pull origin main
```

2 - Create and switch to a feature branch:

```
git checkout -b feature/user-auth
```

3 - Make changes, stage, and commit:

```
git add
git commit -m "Add user authentication functionality"
```

4 - Push your feature branch:

```
git push origin feature/user-auth
```

Step 4 - Pull Requests (PR) / Merge Requests (MR)

1 - Create a PR/MR from your feature/xyz branch to main or develop.

2 - Have at least one team member **review** it.

3 - Resolve any **merge conflicts** if needed.

4 - Once approved, **merge** it.

Step 5 - Keeping Your Branch Updated

To avoid conflicts, keep your branch updated:

```
git fetch origin
git rebase origin/main  # or git merge origin/main
```

Step 6 - Team Collaboration Best Practices

- Use clear commit messages and Push often, but only working code
- Use .gitignore to avoid pushing unnecessary files
- Document setup steps in a README.md
- Agree on a code style guide or use a formatter/linter

Example Daily Workflow Summary

```
# 1. Sync main
    git checkout main
    git pull origin main

# 2. Create a new feature branch
    git checkout -b feature/new-dashboard

# 3. Work and commit
    git add .
    git commit -m "Add new dashboard UI"

# 4. Push feature branch
    git push origin feature/new-dashboard

# 5. Open a PR and assign a reviewer
```

Issues developers face when working in a team

- Merge Conflicts - Two or more developers change the same line(s) in a file or edit nearby lines.
- Not Pulling Before Pushing - Developers make local changes and push without syncing with the remote first.
- Working Directly on the main or master Branch - Developers forget or ignore branching strategies.
- Improper or Inconsistent Commit Messages - Lack of a team-wide convention.
- Lack of Code Review or Peer Review
- Confusion Between fetch, pull, and merge

Introduction to code versioning system

- A **code versioning system**, also known as a **version control system (VCS)**, is a tool that helps developers track and manage changes to source code over time

Types of Version Control Systems:

Type	Description	Example Tools
Local VCS	Stores versions on a local system. Simple but limited.	RCS, SCCS
Centralized VCS (CVCS)	Single central server stores all files and history.	CVS, Subversion (SVN)
Distributed VCS (DVCS)	Every developer has a full copy of the repo and history.	Git

Common Tools Used in Versioning:

Tool	Type	Description
Git	DVCS	Most popular VCS, supports local/remote workflows
GitHub	Hosting	Cloud-based Git repository hosting
GitLab	Hosting	Git repo manager with CI/CD
Subversion (SVN)	CVCS	Centralized version control tool

HISTORY OF CODE VERSIONING SYSTEM

Pre-Version Control Era

Before 1970's
Developers manually maintained by renaming files

First Generation - Local version Control Systems

1970s - 1980s
SCCS (Source Code Control System) , RCS (Revision Control System)

Second Generation - Centralized Version Control Systems

1990s - 2000s
CVS (Concurrent Versions System), Subversion (SVN)

Third Generation - Distributed Version Control Systems (DVCS) (2005-2010)

2005 - 2010
Git , Mercurial

Modern Era

2010 - Present
Git hosting and Collaboration
GitHub - 2008
Bitnucket - 2008
GitLab - 2011
Azure DevOps - 2013

Software development workflow

- A software development workflow in a code versioning system is like a well-choreographed dance—it ensures that developers can collaborate, experiment, and deliver features without stepping on each other's toes

Step 1 **Repository Setup**

```
git clone https://github.com/org/project.git
```

Step 2 **Branching Strategy**

Branch Type	Purpose
main or master	Stable, production-ready code
develop	Integration of all features (optional)
feature/xyz	For new features
bugfix/xyz	For bug fixes
hotfix/xyz	For critical production fixes
release/xyz	For preparing a new release

Software development workflow

Step 3 Development Steps

- **Create a Feature Branch**

```
git checkout -b feature/login-system
```

- **Make Changes & Test Locally**
- **Stage & Commit Changes**

```
git add
git commit -m "Add login system UI and validation"
```

- **Pull Latest from Main to Stay Updated**

```
git pull origin main --rebase
```

Step 4 Push to Remote Repository

```
git push origin feature/login-system
```

Step 5 Create a Pull Request (PR)

Open a PR/Merge Request to merge feature/xyz into main or develop.

Step 6 Code Review & Merge

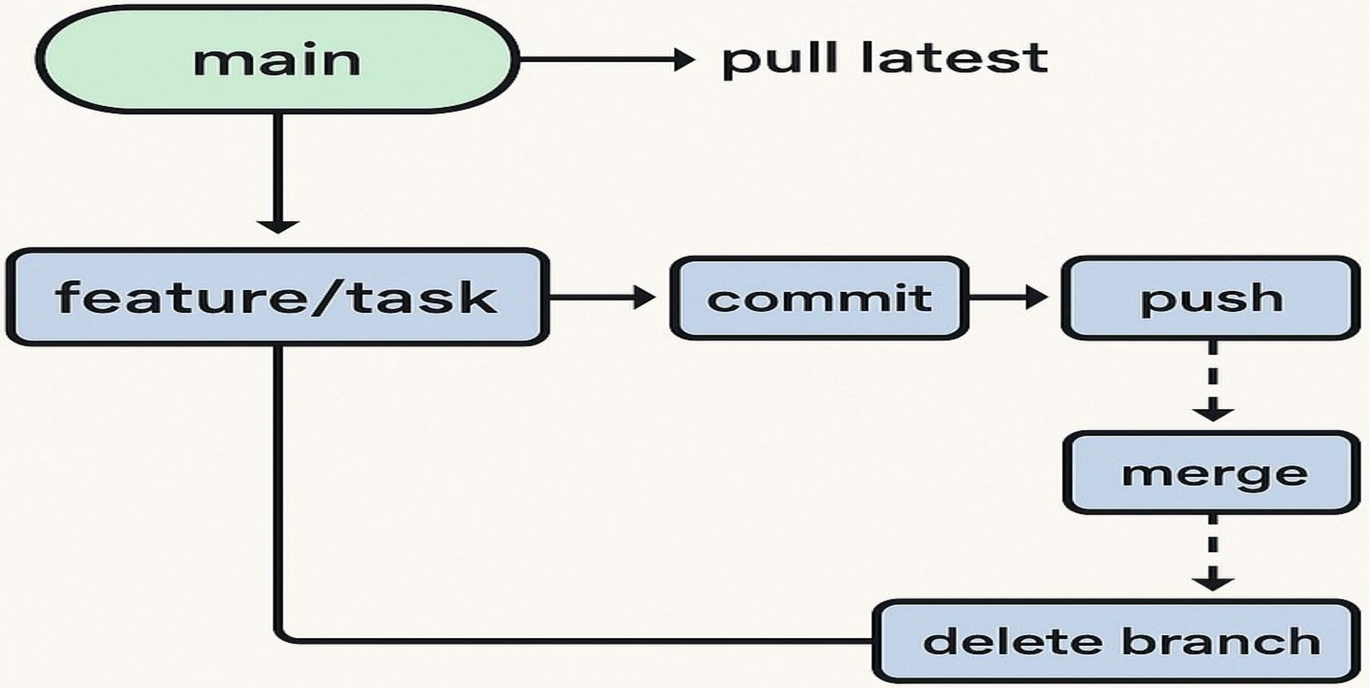
- Add reviewers from the team
- Discuss, suggest, and approve changes.

Step 7

Continuous Integration/Deployment (CI/CD)

Tools like **GitHub Actions**, **GitLab CI**, or **Jenkins** run automated tests and deploys.

Development Workflow (Git)



Introduction to git

- Git is a source code version control system(VCS).
- Helps to manage code changes in a better manner
- Best suited for source code of computer programming languages: C/C++, Java, Python etc
- Since it is a distributed version control system, It allows multiple developers to work on a project simultaneously without overwriting each other’s work.
- Created by **Linus Torvalds** in 2005 to manage the development of the Linux kernel.
- Available for Linux, Mac and Windows
- Web Interfaces available in form of github.com, gitlab.com, bitbucket.com etc
- Big community for support

Introduction to git repository and git structure

git repository

- A **Git repository** (or **repo**) is a virtual storage space where Git tracks all your project's files, changes, and version history.

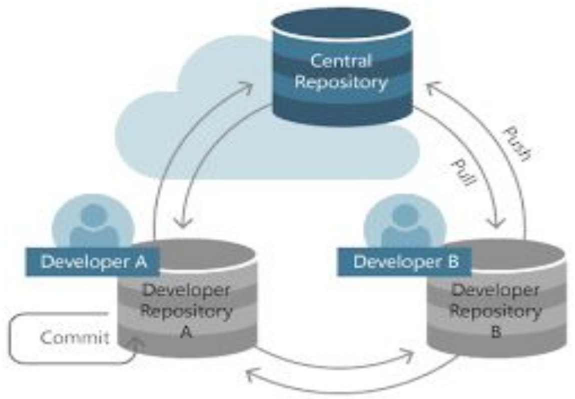
There are two types:

- **Local Repository**: Stored on your computer.
- **Remote Repository**: Hosted on platforms like **GitHub**, **GitLab**, or **Bitbucket** for team collaboration.
- To create a new Git repository: **git init**
- To copy (clone) a remote repository: **git clone https://github.com/username/repo-name.git**

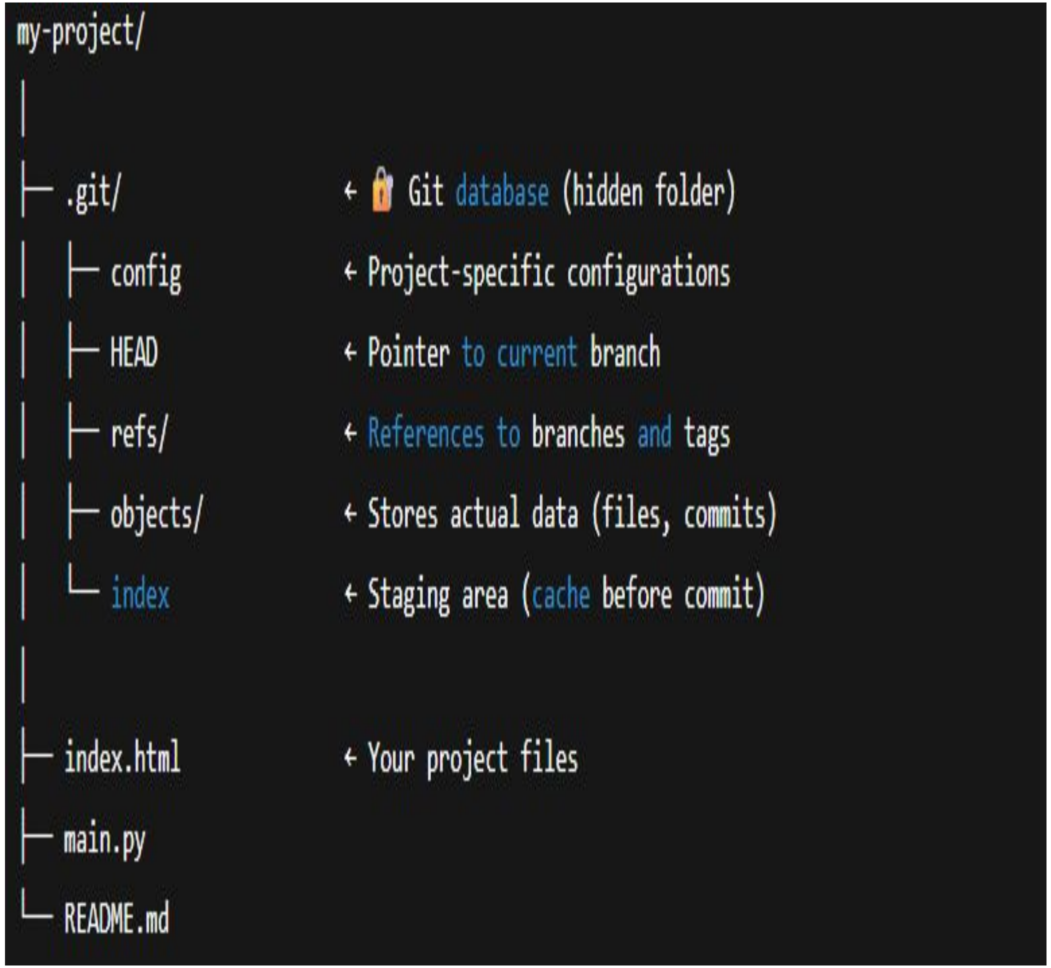
Structure of a Git Repository

- When you initialize a Git repository using git init, it creates a hidden directory called **.git/.**
- This is where Git stores all the information it needs to manage the project.

git repository

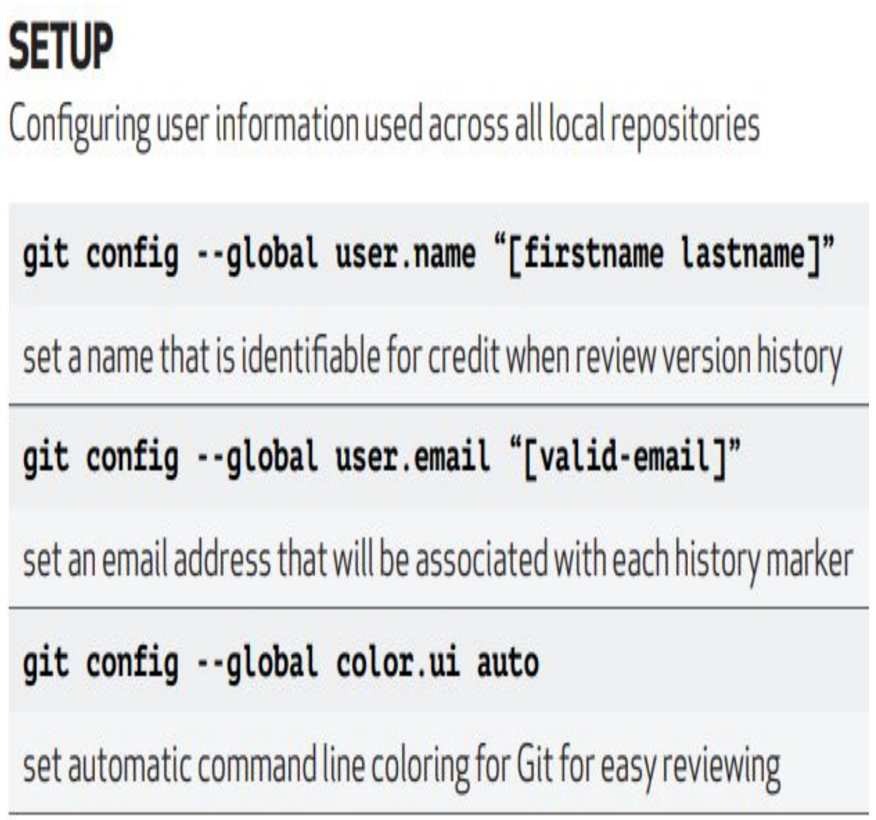


git structure



```
my-project/
│
├─ .git/                ← 🔐 Git database (hidden folder)
│   ├─ config           ← Project-specific configurations
│   ├─ HEAD             ← Pointer to current branch
│   ├─ refs/            ← References to branches and tags
│   ├─ objects/         ← Stores actual data (files, commits)
│   └─ index            ← Staging area (cache before commit)
│
├─ index.html           ← Your project files
├─ main.py
└─ README.md
```

Git Commands



SETUP

Configuring user information used across all local repositories

```
git config --global user.name “[firstname lastname]”
```

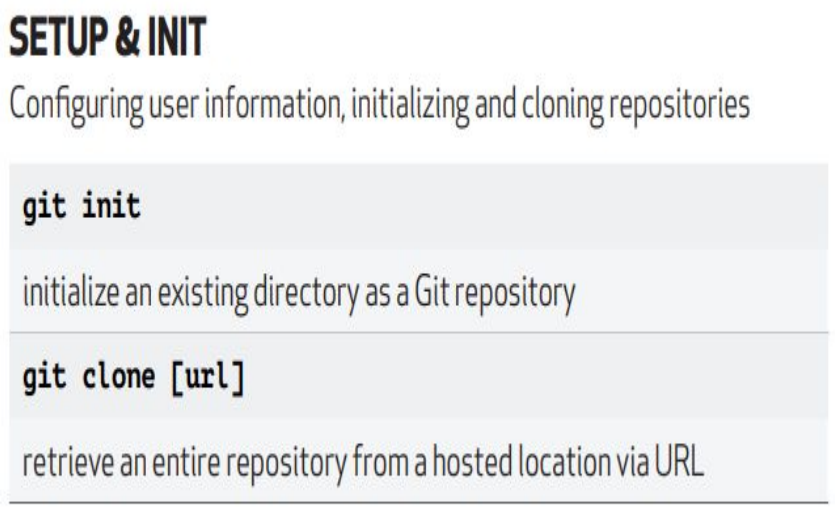
set a name that is identifiable for credit when review version history

```
git config --global user.email “[valid-email]”
```

set an email address that will be associated with each history marker

```
git config --global color.ui auto
```

set automatic command line coloring for Git for easy reviewing



SETUP & INIT

Configuring user information, initializing and cloning repositories

```
git init
```

initialize an existing directory as a Git repository

```
git clone [url]
```

retrieve an entire repository from a hosted location via URL

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository
`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

```
git branch
```

list your branches. a * will appear next to the currently active branch

```
git branch [branch-name]
```

create a new branch at the current commit

```
git checkout
```

switch to another branch and check it out into your working directory

```
git merge [branch]
```

merge the specified branch's history into the current one

```
git log
```

show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`

show the commit history for the currently active branch

`git log branchB..branchA`

show the commits on branchA that are not on branchB

`git log --follow [file]`

show the commits that changed file, even across renames

`git diff branchB...branchA`

show the diff of what is in branchA that is not in branchB

`git show [SHA]`

show any object in Git in human-readable format

TRACKING PATH CHANGES

Versioning file removes and path changes

```
git rm [file]
```

delete the file from project and stage the removal for commit

```
git mv [existing-path] [new-path]
```

change an existing file path and stage the move

```
git log --stat -M
```

show all commit logs with indication of any paths that moved

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

```
git config --global core.excludesfile [file]
```

system wide ignore pattern for all local repositories

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

```
git remote add [alias] [url]
```

add a git URL as an alias

```
git fetch [alias]
```

fetch down all the branches from that Git remote

```
git merge [alias]/[branch]
```

merge a remote branch into your current branch to bring it up to date

```
git push [alias] [branch]
```

Transmit local branch commits to the remote repository branch

```
git pull
```

fetch and merge any commits from the tracking remote branch

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`
apply any commits of current branch ahead of specified one
`git reset --hard [commit]`
clear staging area, rewrite working tree from specified commit

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`
Save modified and staged changes
`git stash list`
list stack-order of stashed file changes
`git stash pop`
write working from top of stash stack
`git stash drop`
discard the changes from top of stash stack

Assignment

- Create a local git repository
- Commit the initial code
- Update the code
- Use git commands to
    - o Get the updated files
    - o List the changes
    - o Create branch
    - o Merge branch