Version Control Tutorial with Git

Contents

1 What is Version Control?

Version control systems (VCS) track changes to files over time so you can:

- Revert back to previous versions
- Collaborate with others without overwriting each other's work
- Keep history of what was changed, when, and by whom

The most popular VCS is **Git** (created by Linus Torvalds).

2 Installing Git

Windows

- Download: https://git-scm.com/download/win
- Install with default settings

Mac

```
brew install git
```

Linux

```
sudo apt install git          # Debian/Ubuntu
sudo yum install git          # Fedora/CentOS
```

Verify installation

```
git --version
```

Configure identity

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
```

3 Git Basics – Local Workflow

Create a repository

```
mkdir myproject
cd myproject
git init
```

Add files

Create `index.html`:

```
<!DOCTYPE html>
<html><body>Hello World</body></html>
```

Check status:

```
git status
```

Stage and commit

```
git add index.html
git commit -m "Add initial HTML page"
```

Edit & track changes

Make edits (e.g., add `style.css`). Check and commit:

```
git status
git diff
git add style.css
git commit -m "Add basic CSS"
```

View history

```
git log
```

4 Understanding Branching in Git

What is Branching?

In Git, a **branch** is simply a lightweight, movable pointer to a commit. By default, Git creates a branch named `main` (or sometimes `master`) that points to the latest commit in your project.

Creating a new branch allows you to diverge from the main code line and work on new features, bug fixes, or experiments without affecting the stable code in `main`.

Why Branching Matters

Branching makes it easy to:

- Develop new features in isolation
- Fix bugs without disturbing other work
- Experiment and discard changes easily if needed
- Enable multiple team members to work in parallel

This helps avoid conflicts and keeps the `main` branch production-ready.

Common Branching Strategies

Different teams adopt different branching strategies depending on project size and workflow:

Feature Branch Workflow:
Create a separate branch for each new feature. Merge back into `main` after review.

Git Flow:
Uses two main branches, `main` and `develop`, along with supporting branches for features, releases, and hotfixes.

Trunk-Based Development:
Developers work in small, short-lived branches and merge back frequently (often daily) into the `main` branch.

Branch Naming Conventions

Consistent branch names make collaboration clearer:

- `feature/login-page` for new features
- `bugfix/typo-header` for bug fixes
- `hotfix/payment-crash` for urgent fixes

Merging vs Rebasing

- **Merging:** Combines the changes of two branches, preserving the history of both. Often used in team workflows.
- **Rebasing:** Moves or "replays" your commits on top of another branch, creating a linear history. Useful for cleaning up local commits before merging.

Best Practices for Branching

- Keep branches small and focused on a single task
- Regularly pull updates from `main` to keep your branch up to date
- Delete branches after merging to keep the repository clean
- Use pull requests (or merge requests) for code reviews

Visualization

Git's branching is lightweight because each branch is just a reference to a commit. This makes it easy to create, switch, and merge branches without duplicating files.

Create and switch branch

```
git branch feature-login
git checkout feature-login
```

Or shortcut:

```
git checkout -b feature-login
```

Commit changes on branch

```
git add .
git commit -m "Start login feature"
```

Merge to main branch

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

If there are conflicts, fix them, then:

```
git add .
git commit -m "Fix merge conflicts"
```

Delete branch

```
git branch -d feature-login
```

5 Working with Remote Repositories

Create repository on GitHub

Go to GitHub → New repository → name: `myproject`.

Link local repo

```
git remote add origin https://github.com/yourname/myproject.git
```

Push code

```
git push -u origin main
```

Next pushes:

```
git push
```

Pull updates

```
git pull
```

Clone a repository

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

6 Best Practices & Tips

- Write meaningful commit messages
- Commit small logical units
- Use `.gitignore` for unnecessary files (e.g., `node_modules`, `.env`)
- Sync often (`git pull`) when collaborating
- Use branches for features and experiments
- Protect `main` branch (e.g., via pull requests)

7 GUI Tools & Advanced Tips

- GUI tools: GitKraken, SourceTree, GitHub Desktop
- VS Code built-in Git integration
- `git stash` to temporarily save changes
- `git rebase` to clean up history
- `git tag` to mark versions (e.g., `v1.0`)

8 Cheat Sheet

Action	**Command**
Init repo	`git init`
Stage	`git add filename` or `git add .`
Commit	`git commit -m "message"`
View history	`git log`
Create branch	`git branch branchname`
Switch branch	`git checkout branchname`
Merge branch	`git merge branchname`
Add remote	`git remote add origin URL`
Push	`git push -u origin branchname`
Pull	`git pull`
Clone	`git clone URL`

9 Learn More

- https://git-scm.com/book/en/v2Pro Git book (free)
- https://docs.github.com/GitHub Docs
- https://learngitbranching.js.org/Interactive learning: Learn Git Branching