Devops Shack

Mastering Git

A Comprehensive Guide

By DevOps Shack

[Click here for DevSecOps & Cloud DevOps Course]

DevOps Shack

Mastering Git: A Comprehensive Guide

Table of Contents

- Working with Remote Repositories (git remote, git fetch, git pull, git push)
- Forking and Cloning Repositories
- Working with Multiple Remotes
- GitHub, GitLab, and Bitbucket Basics
- Pull Requests and Code Reviews
- Configuring SSH Authentication

5. Undoing Changes and Debugging

- Undoing Local Changes (git checkout, git restore, git reset)
- Undoing Commits (git revert, git reset --soft/hard/mixed)
- Stashing Changes (git stash, git stash pop, git stash apply)
- Finding Issues in History (git blame, git bisect, git reflog)

6. Git Advanced Features

- Interactive Rebase (git rebase -i)
- Squashing Commits (git rebase -i, git merge --squash)
- Git Hooks (Pre-commit, Post-commit, Pre-push Hooks)
- Submodules (git submodule)
- Managing Large Repositories (git LFS)

7. Git Workflows and Best Practices

- Git Workflows (Feature Branch, Git Flow, Trunk-based Development)
- Writing Good Commit Messages
- Handling Conflicts Efficiently
- Keeping a Clean Commit History
- Best Practices for Collaboration

8. Git Internals and Performance Optimization

- How Git Works Internally (Objects, Trees, Blobs, Hashing)
- Understanding .git Directory Structure

- Optimizing Repositories (git gc, git prune)
- Handling Large Repositories Efficiently

9. GitHub, GitLab, and CI/CD Integration

- GitHub Actions for Automation
- GitLab CI/CD Pipelines
- Using Git in DevOps (Automated Deployments)
- Code Review and Collaboration

10. Troubleshooting and Debugging Git Issues

- Resolving Merge Conflicts
- Fixing Detached HEAD Issues
- Debugging Network Issues with Git Remotes
- Resolving Common Git Errors

11. Conclusion and Further Learning

- Additional Resources (Books, Courses, Documentation)
- Common Git Mistakes to Avoid
- Next Steps in Mastering Git

1. Introduction to Git

Git is a powerful, open-source distributed version control system designed to handle everything from small to very large projects with speed and efficiency. It was created by **Linus Torvalds** in 2005 for Linux kernel development and has since become the most widely used version control system in the world.

What is Git?

Git is a **Version Control System (VCS)** that helps developers track changes in their code, collaborate with others, and maintain a history of their project. Some key characteristics of Git include:

- **Distributed**: Every developer has a full copy of the repository, making it independent of a central server.
- **Fast and Efficient**: Git is designed to be fast, even for large repositories.
- **Reliable**: It ensures data integrity using cryptographic hashing (SHA-1).
- **Supports Branching & Merging**: Git allows developers to work on different features without affecting the main codebase.

Why Use Git?

Git provides many benefits for individuals and teams working on software development projects:

- **Collaboration**: Multiple developers can work on the same project simultaneously.
- **History Tracking**: Git records every change, making it easy to revert to a previous state.
- **Branching & Merging**: Allows developers to experiment with new features safely.
- **Code Integrity**: Git uses checksums to detect corruption and changes.
- **Widely Adopted**: Supported by platforms like GitHub, GitLab, and Bitbucket.

Version Control Systems: Centralized vs. Distributed

There are two main types of Version Control Systems (VCS):

1. Centralized Version Control Systems (CVCS)

- Uses a single central server to store all files and version history.
- Developers check out files, make changes, and commit them back to the central server.
- Examples: SVN, Perforce
- **Drawbacks**:
    - Single point of failure (if the server is down, no one can work).
    - Slower operations due to network dependency.

2. Distributed Version Control Systems (DVCS)

- Each user has a complete copy of the entire repository, including history.
- Users can commit changes locally before pushing them to a remote server.
- Examples: Git, Mercurial
- **Advantages**:
    - Faster operations (committing and branching are local).
    - Work offline and sync later.
    - No single point of failure.

Git falls under **Distributed Version Control Systems (DVCS)**, making it highly flexible and reliable.

Installing Git

Git can be installed on various operating systems:

Windows

1. Download the installer from git-scm.com.
2. Run the installer and follow the setup wizard.
3. Choose **Git Bash** as the command-line tool (recommended).
4. Set **Use Git from the Windows Command Prompt** option.
5. Verify installation by

running: git --version

macOS

1. Install Git using

Homebrew: brew install git

2. Alternatively, install Xcode Command Line

Tools: xcode-select --install

3. Verify installation:

git --version

Linux (Ubuntu/Debian)

1. Install Git via package

manager: sudo apt update

sudo apt install git

2. Verify installation:

git --version

Linux (Fedora)

sudo dnf install git

Linux (Arch-

based) sudo

pacman -S git

Setting Up Git

After installing Git, configure it with your identity and preferred settings.

Configuring User Identity

Set your name and email, which will be associated with every commit: git

config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

Setting Default Branch Name

By default, Git used to name the first branch **"master"**, but now it is commonly set to **"main"**.

To set it explicitly:

```
git config --global init.defaultBranch main
```

Verifying Configuration

Check your global Git settings:

```
git config --list
```

Now that Git is installed and set up, you are ready to start using it for version control.

2. Understanding Git Basics

Now that Git is installed and configured, let's dive into its fundamental concepts and commands.

Git Terminology

Before using Git, it's essential to understand its key terms:

Term	Description
Repository (repo)	A directory that contains all the files, history, and metadata of a project.
Commit	A snapshot of changes saved to the repository.
Branch	A parallel version of the repository that allows independent development.
Merge	Combining changes from different branches.
Remote	A reference to a repository stored on a server (e.g., GitHub, GitLab).
Clone	Creating a local copy of a remote repository.
Pull	Fetching and integrating changes from a remote repository.
Push	Sending local commits to a remote repository.
Staging Area	A place where changes are prepared before committing.
HEAD	A pointer to the latest commit in the current branch.

Initializing a Git Repository (git init)

To start using Git in a new project:

1. Navigate to the project

folder: cd /path/to/your/project

2. Initialize Git:

```
git init
```

This creates a hidden .git folder that stores version control data.

Cloning an Existing Repository (git clone)

If you want to work on an existing Git project:

git clone <repository-url> Example:

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

This downloads the project and its history to your local machine.

Tracking Files (git add, .gitignore)

Adding Files to Staging Area

To track a new or modified file:

git add <file-name> Example:

```
git add index.html
```

To add all modified files:

```
git add .
```

Ignoring Files with .gitignore

Some files (e.g., logs, temporary files, secrets) should not be tracked. Add them to a .gitignore file:

Example .gitignore file:

```
node_modules/
.env
*.log
```

Committing Changes (git commit)

Once files are staged, save them in Git with a commit message:

git commit -m "Add homepage layout"

This creates a commit with a unique identifier (hash) and saves a snapshot of the changes.

Viewing Commit History (git log, git show)

To see the commit

history: git log

For a single commit's details:

git show <commit-hash>

Example:

git show 3a5d9b2

3. Branching and Merging

Branching is one of Git's most powerful features, allowing multiple developers to work on different features simultaneously without interfering with the main codebase.

Understanding Branches in Git

A branch in Git is essentially a pointer to a commit, allowing you to work on different versions of a project in parallel.

Why Use Branches?

- Isolate new features from the main codebase.
- Prevent incomplete features from breaking production code.
- Enable collaboration by allowing multiple developers to work independently.

Creating and Switching Branches (git branch, git checkout, git switch)

Viewing Existing Branches

To list all branches in a repository:

```
git branch
```

Creating a New Branch

To create a new branch:

```
git branch feature-branch
```

This only creates the branch; it does not switch to it.

Switching to a Branch

To move to a different

branch: `git checkout feature-`

`branch`

Alternatively, in newer Git versions, use:

```
git switch feature-branch
```

Creating and Switching in One Command

git checkout -b feature-

branch or

git switch -c feature-branch

Merging Branches (git merge)

Once development in a branch is complete, merge it into the main branch.

1. Switch to the main

branch: git checkout main

2. Merge the feature

branch: git merge feature-

branch

Fast-Forward vs. Three-Way Merge

- **Fast-Forward Merge**: If no new commits were made on main, Git moves the branch pointer forward.
- **Three-Way Merge**: If both branches have new commits, Git creates a new commit combining changes.

Handling Merge Conflicts

If Git cannot automatically merge changes, a **merge conflict** occurs.

Resolving Merge Conflicts

1. Identify conflicts

using: git status

2. Open conflicting files and manually edit sections marked by <<<<<<< and >>>>>>>.
3. After resolving, stage the

file: git add <file>

4. Complete the merge:

```
git commit -m "Resolve merge conflict"
```

Rebase vs. Merge (git rebase vs. git merge)

Merging

- Preserves commit history.
- Creates a new merge

commit. `git merge feature-branch`

Rebasing

- Moves commits from one branch to another, keeping history

linear. `git rebase main`

When to Use Which?

- **Use merge** for collaborative projects to keep history.
- **Use rebase** for a clean, linear commit history.

Cherry-Picking Commits (git cherry-pick)

To apply a specific commit from one branch to another: `git cherry-pick <commit-hash>`

Example:

```
git cherry-pick 3a5d9b2
```

4. Remote Repositories and Collaboration

Git is designed for collaboration. Remote repositories allow multiple developers to work together by sharing and syncing code.

Working with Remote Repositories (git remote, git fetch, git pull, git push)

Adding a Remote Repository

To connect a local project to a remote repository (e.g., GitHub, GitLab, Bitbucket):

git remote add origin <repository-url>

Example:

git remote add origin https://github.com/user/repo.git

origin is the default name for the remote repository.

Viewing Remote Repositories

git remote -v

Fetching Changes from a Remote Repository (git fetch)

To check for updates without merging them:

git fetch origin

This downloads new commits but does not apply them to your branch.

Pulling Changes (git pull)

To fetch and merge changes from the remote repository:

git pull origin main

Equivalent to:

git fetch origin

git merge origin/main

Pushing Changes (git push)

To upload local commits to a remote repository:

git push origin main

If pushing for the first time:

git push -u origin main

-u sets origin main as the default upstream branch.

Forking and Cloning Repositories

Forking

Forking is creating a copy of someone else's repository under your GitHub account.

Cloning a Repository (git clone)

To copy an existing remote repository:

git clone <repository-url>

Example:

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

Working with Multiple Remotes

You can connect a project to multiple remote repositories.

Adding Another Remote

git remote add upstream <repository-url>

Useful when contributing to open-source projects where origin is your fork, and upstream is the original repository.

Fetching from Upstream

git fetch upstream

git merge upstream/main

GitHub, GitLab, and Bitbucket Basics

These platforms provide hosting for Git repositories and additional collaboration features.

- **GitHub**: Most popular, offers pull requests, issues, and CI/CD through GitHub Actions.
- **GitLab**: Includes built-in CI/CD and DevOps tools.
- **Bitbucket**: Supports Mercurial and Git, commonly used with Atlassian tools.

Pull Requests and Code Reviews

A **pull request (PR)** is a request to merge changes into a remote repository.

Creating a Pull Request on GitHub

1. Push your branch to

GitHub: git push origin feature-

branch

2. Go to GitHub and open a pull request.
3. Request a review and discuss changes.
4. Once approved, merge the pull request.

Configuring SSH Authentication

Instead of using passwords, SSH authentication allows secure access.

Generating an SSH Key

ssh-keygen -t rsa -b 4096 -C "your.email@example.com"

Add the key to GitHub/GitLab/Bitbucket under **SSH Keys** in settings.

Using SSH to Clone

git clone git@github.com:user/repo.git

5. Undoing Changes and Debugging

Git provides powerful tools to undo changes, fix mistakes, and debug issues efficiently.

Undoing Local Changes (git checkout, git restore, git reset)

Discarding Unstaged Changes (git restore)

If you made changes but haven't staged them yet, you can discard them:

git restore <file>

To discard all unstaged changes:

git restore .

Reverting Staged Changes (git reset)

If you have already staged changes using git add, you can unstage them: git reset <file>

To unstage everything:

git reset

This does not delete the changes, just removes them from staging.

Undoing Commits (git revert, git reset --soft/hard/mixed)

Reverting a Commit (git revert)

If you need to undo a commit but keep history intact, use git revert:

git revert <commit-hash>

This creates a new commit that undoes the changes from the specified commit.

Resetting a Commit (git reset)

- **Soft Reset (--soft)**: Moves HEAD to a previous commit but keeps changes staged.

git reset --soft HEAD~1

- **Mixed Reset (--mixed)**: Moves HEAD and unstages changes but the files.

git reset --mixed HEAD~1

- **Hard Reset (--hard)**: Moves HEAD and deletes changes

permanently. git reset --hard HEAD~1

Stashing Changes (git stash)

If you need to temporarily save changes without committing:

git stash

To apply the last stashed changes:

git stash pop

To apply a specific stash:

git stash apply stash@{1}

To see all stashes:

git stash list

Finding Issues in History (git blame, git bisect, git reflog)

Finding Who Made a Change (git blame)

To see who modified each line of a file:

git blame <file>

Finding a Bug Using git bisect

Git bisect helps find which commit introduced a bug:

git bisect start

git bisect bad # Mark current commit as bad

git bisect good <commit-hash> # Mark a known good commit

Git will now guide you through testing commits.

Viewing Reference History (git reflog)

To see recent changes to

git reflog

This helps recover lost commits.

6. Git Advanced Features

Now that we've covered the basics, let's explore some advanced Git features that can improve your workflow.

Interactive Rebase (git rebase -i)

Rebasing allows you to modify commit history by rewriting, reordering, or squashing commits.

Starting an Interactive Rebase

```
git rebase -i HEAD~3
```

This will open an interactive list of the last 3 commits, allowing you to:

- **pick** → Keep commit as is.
- **reword** → Change commit message.
- **edit** → Modify the commit.
- **squash** → Merge commits into one.
- **drop** → Remove a

commit. Example:

```
pick a1b2c3 Add login feature
squash d4e5f6 Fix login bug
reword g7h8i9 Improve login message
```

Squashing Commits (git rebase -i, git merge --squash)

Squashing reduces multiple commits into a single commit, creating a cleaner history.

Squashing with Rebase

1. Start interactive rebase:

```
git rebase -i HEAD~3
```

2. Change pick to squash for commits you want to merge.

3. Save and edit the commit message.

Squashing with Merge

```
git merge --squash feature-branch
git commit -m "Squashed commits from feature-branch"
```

Git Hooks (Pre-commit, Post-commit, Pre-push Hooks)

Git hooks are scripts that execute before or after Git events like commits or pushes.

Common Git Hooks

- **pre-commit** → Runs before git commit (e.g., linting, formatting).
- **post-commit** → Runs after git commit (e.g., notifications).
- **pre-push** → Runs before git push (e.g., running tests).

Setting Up a Git Hook

1. Navigate to .git/hooks/ directory.
2. Create or edit a hook

script: `nano .git/hooks/pre-commit`

3. Add a script, e.g., prevent committing debug

code:

```
#!/bin/sh
if grep -q "console.log" *.js; then
echo "Remove console.log before committing!"
  exit 1
fi
```

4. Make it executable:

```
chmod +x .git/hooks/pre-commit
```

Submodules (git submodule)

Git submodules allow including one repository inside another, useful managing dependencies.

Adding a Submodule

git submodule add <repo-url> <path>

Example:

git submodule add https://github.com/user/library.git libs/library

Initializing and Updating Submodules

git submodule update --init --recursive

Managing Large Repositories (git LFS)

Git LFS (Large File Storage) helps manage large files efficiently.

Installing Git LFS

git lfs install

Tracking Large Files

git lfs track "*.psd"

Commit the tracking info:

git add .gitattributes

git commit -m "Track PSD files with LFS"

7. Git Workflows and Best Practices

Choosing the right Git workflow can greatly improve collaboration and project management. This section covers common workflows and best practices for using Git effectively.

Git Workflows

Different teams use different workflows based on their needs. Here are the most common ones:

Feature Branch Workflow

Each new feature or fix is developed in a separate branch before merging into the main branch.

1. Create a feature

branch: git checkout -b

feature-branch

2. Work on the feature and commit changes.
3. Merge the feature branch into the main

branch: git checkout main

git merge feature-branch

Git Flow Workflow

A structured workflow with specific branches for development, releases, and fixes.

- **main** → Stable production branch.
- **develop** → Active development branch.
- **Feature branches** → For new features.
- **Release branches** → For preparing a new release.
- **Hotfix branches** → For urgent fixes to production.

Using Git Flow

First, install Git Flow:

```
git flow init
```

Start a new feature:

```
git flow feature start new-feature
```

Complete and merge the feature:

```
git flow feature finish new-feature
```

Trunk-Based Development

Developers work on short-lived branches and merge changes frequently into main.

Forking Workflow

Used in open-source projects where contributors fork a repository and submit pull requests.

Writing Good Commit Messages

A good commit message makes the history easy to understand.

Best Practices

1. **Use imperative mood**
   - ☑ "Fix login bug"
   - ❌ "Fixed login bug"
2. **Keep it concise**

```
git commit -m "Add validation to user input"
```

3. **Use multi-line commits for more details**

```
git commit -m "Improve password hashing" -m "Uses bcrypt instead of SHA-256 for stronger security"
```

Handling Conflicts Efficiently

1. Check for conflicts:

git status

2. Edit the conflicted files (look for <<<<<<< HEAD).
3. Mark the conflict as

resolved: git add <file>

4. Complete the merge:

git commit -m "Resolve merge conflict"

Keeping a Clean Commit History

- **Use rebase to keep history**

clean: git rebase -i HEAD~3

- **Squash unnecessary commits** before

merging: git merge --squash feature-branch

- **Delete merged**

branches: git branch -d

feature-branch

Best Practices for Collaboration

- **Pull before pushing** to avoid

conflicts: git pull origin main

- **Use meaningful branch names** (e.g., feature/login-page instead of dev123).
- **Review pull requests before merging** to maintain code quality.

8. Git Internals and Performance Optimization

Understanding how Git works internally can help you troubleshoot issues and optimize performance for large repositories.

How Git Works Internally (Objects, Trees, Blobs, Hashing)

Git stores data as snapshots, not diffs. The main components are:

1. Git Objects

Git uses four main object types stored in the .git/objects directory:

- **Blobs**: Store file contents.
- **Trees**: Store directory structures and file references.
- **Commits**: Store metadata, author info, and parent commit reference.
- **Tags**: Reference specific commits.

Each object is identified by a **SHA-1 hash** (e.g., a1b2c3d4e5...).

2. Viewing Git Objects

- Show the internal structure of a

commit: git cat-file -p HEAD

- Show a tree

structure: git ls-tree HEAD

- Show a blob (file

contents): git cat-file -p <blob-

hash>

Understanding .git Directory Structure

Every Git repository contains a hidden .git/ directory that stores all version control data.

Key Directories and Files

- .git/objects/ → Stores all commits, trees, and blobs.

- .git/refs/ → Stores references to branches and tags.

- .git/HEAD → Points to the current branch.
- .git/config → Stores repository settings.

Viewing Configuration

git config --list

Optimizing Repositories (git gc, git prune)

1. Garbage Collection (git gc)

Cleans up unnecessary files and optimizes storage. git gc --aggressive

2. Removing Unreachable Objects (git prune)

Removes old objects no longer referenced by any commit. git prune

3. Cleaning Up Local Repository

git fsck

git reflog expire --all --expire=now

git repack -a -d

Handling Large Repositories Efficiently

1. Using Git LFS (Large File Storage)

Tracks large files outside of the main repository. git lfs track "*.zip"

git add .gitattributes

git commit -m "Track ZIP files with LFS"

2. Shallow Cloning

Speeds up cloning by fetching only the latest commits. git clone --depth=1 <repository-url>

3. Sparse Checkout

Check out only specific directories from a large repository.

git sparse-checkout init

git sparse-checkout set src/

9. GitHub, GitLab, and CI/CD Integration

Integrating Git with platforms like GitHub and GitLab allows for better collaboration, automation, and deployment using CI/CD pipelines.

GitHub, GitLab, and Bitbucket Basics

These platforms provide hosting for Git repositories with additional collaboration tools like pull requests, issues, and CI/CD.

Cloning a Repository from GitHub/GitLab

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

Forking a Repository (GitHub)

1. Click the **Fork** button on GitHub.
2. Clone your fork:

```
git clone https://github.com/your-username/repo.git
```

3. Add the original repo as a remote:

```
git remote add upstream https://github.com/original-user/repo.git
```

4. Fetch and merge updates from the original

repo: `git fetch upstream`

```
git merge upstream/main
```

Pull Requests and Code Reviews

A **pull request (PR)** allows contributors to propose changes before merging.

Creating a Pull Request

1. Push your branch to

GitHub/GitLab: `git push origin feature-branch`

2. Go to GitHub/GitLab and create a PR.
3. Request a review, address comments, and merge when approved.

Reviewing a Pull Request

- Add comments inline.
- Approve or request changes.
- Squash and merge when ready.

Configuring SSH Authentication

Using SSH keys allows secure authentication without entering passwords.

Generating an SSH Key

```
ssh-keygen -t rsa -b 4096 -C "your-email@example.com"
```

Copy the public key:

```
cat ~/.ssh/id_rsa.pub
```

Add it to **GitHub/GitLab → Settings → SSH Keys**.

Testing SSH Connection

```
ssh -T git@github.com
```

GitHub Actions for Automation

GitHub Actions automates workflows like running tests and deploying code.

Creating a Workflow File

1. Create .github/workflows/main.yml in your repo.
2. Add the following CI pipeline:name:

```
CI on: [push, pull_request]
jobs:
  build:
    runs-on: ubuntu-
    latest steps:
     - uses: actions/checkout@v3
     - name: Install dependencies
```

```
      run: npm install
    - name: Run tests
      run: npm test
```

3. Push the file to trigger the workflow.

GitLab CI/CD Pipelines

GitLab provides built-in CI/CD pipelines via .gitlab-ci.yml.

Example GitLab CI/CD Pipeline

Create .gitlab-ci.yml in your repo:

```
stages:
  - test
  - deploy

test:
 stage: test
  script:
    - npm install
    - npm test

deploy:
 stage: deploy
  script:
    - echo "Deploying..."
  only:
    - main
```

Push the file to trigger the pipeline.

Using Git in DevOps (Automated Deployments)

Connecting Git to a Server for Deployment

1. Set up SSH access to your server.
2. Use Git hooks to trigger

deployments: git pull origin main && npm

run deploy

10. Troubleshooting and Debugging Git Issues

Even experienced Git users encounter issues. This section covers common Git problems and how to fix them.

Resolving Merge Conflicts

Merge conflicts occur when two branches modify the same line in a file.

Steps to Resolve a Merge Conflict

1. Identify conflicts

using: git status

2. Open the conflicted file; it will contain markers like this:

<<<<<<< HEAD

Your changes

=======

Incoming changes

>>>>>>> branch-name

3. Edit the file to keep the correct changes.
4. Mark the conflict as

resolved: git add <file>

5. Complete the merge:

git commit -m "Resolve merge conflict"

Aborting a Merge if Needed

git merge --abort

Fixing Detached HEAD Issues

A detached HEAD occurs when you check out a commit instead of a branch.

Reattaching HEAD to a Branch

git checkout main

If You Want to Keep Changes

Create a new branch:

git checkout -b new-branch

Debugging Network Issues with Git Remotes

Checking Remote Repositories

git remote -v

Fixing Authentication Issues

If authentication fails, verify your credentials:

git credential reject https://github.com

Then, re-authenticate and try again.

Fixing SSH Connection Issues

Test the SSH connection:

ssh -T git@github.com

If it fails, check SSH key permissions:

chmod 600 ~/.ssh/id_rsa

Resolving Common Git Errors

1. Accidentally Committed to the Wrong Branch

Move the commit to the correct branch:

git checkout correct-branch

git cherry-pick <commit-

hash> git checkout wrong-

branch

git reset --hard HEAD~1

2. Undoing the Last Commit

If the commit hasn't been pushed:

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

If the commit has been pushed:

```
git revert HEAD
```

3. Restoring a Deleted Branch

If the branch was deleted but still exists in reflog:

```
git reflog
```

```
git checkout -b recovered-branch <commit-hash>
```

4. Recovering Lost Commits

Check the reflog for lost commits:

```
git reflog
```

```
git checkout <commit-hash>
```

11. Conclusion and Further Learning

Congratulations on making it through this Git guide! By now, you should have a solid understanding of Git fundamentals, workflows, and advanced features.

Additional Resources

To deepen your knowledge, explore these resources:

Official Documentation

- Git Documentation
- GitHub Docs
- GitLab Docs

Interactive Learning

- Learn Git Branching
- GitHub Learning Lab

Common Git Mistakes to Avoid

1. **Committing secrets or sensitive data**
    - Use .gitignore to prevent accidental commits.
2. **Not pulling before pushing**

git pull origin main

Always pull the latest changes before pushing.

3. **Using force push (git push --force) recklessly**
    - This can overwrite work from others.
    - Use git push --force-with-lease instead.
4. **Not writing meaningful commit messages**
    - Good commit messages help maintain a readable history.

Next Steps in Mastering Git

- **Contribute to Open Source**
  - Find a project on GitHub and submit a pull request.
- **Automate with Git Hooks**
  - Use pre-commit hooks for linting and testing.
- **Explore Advanced Git Tools**
  - Try Git worktrees, bisect, and submodules.

Final Thoughts

Git is an essential tool for developers, enabling efficient collaboration and version control. By practicing regularly and applying best practices, you'll become a Git expert in no time! 🚀