Complete Git Workflow Guide

A comprehensive, structured guide to Git fundamentals and workflows

Federica Gazzelloni

2026-01-29

Table of contents

1 1. Understanding Terminal and Shell

1.1 What's the Difference?

Terminal = The window/container
Shell = The interpreter running inside that window

The shell provides a command-line interface (CLI) for Unix-like operating systems to interact with your machine.

1.2 Shell Types by Platform

Platform	Default Shell	Notes
macOS	Zsh	Default since Catalina (10.15)
RStudio	Bash	Older shell, still common
Windows	PowerShell/CMD	Git typically used via WSL (Windows Subsystem for Linux)

1.3 Check Your Current Shell

```
echo $SHELL
```

Expected output: - `/bin/zsh` (macOS) - `/bin/bash` (RStudio, Linux)

---

2 2. Git Fundamentals

2.1 Core Concept

Git works completely locally. - No GitHub required - No registration needed - No internet connection needed

A Git repository = A folder containing a hidden `.git` directory

---

3 3. Initial Setup (One-Time Configuration)

3.1 Set Your Identity

Every commit is labeled with author information:

```
# Set your name
git config --global user.name "Your Name"

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

# Verify configuration
git config --list
```

3.2 Optional: Set Default Editor

```
# Use nano (beginner-friendly)
git config --global core.editor nano

# Or use VS Code
git config --global core.editor "code --wait"

# Or use vim (default, more complex)
git config --global core.editor vim
```

---

4 4. Creating Your First Repository

4.1 Step-by-Step Workflow

```
# 1. Create a project folder
mkdir backup
cd backup

# 2. Check current directory
pwd

# 3. List contents (including hidden files)
ls -a

# 4. Check Git status (will show error - not a repo yet)
git status

# 5. Initialize Git repository
git init

# 6. Verify .git was created
ls -a
# Output: .  ..  .git

# 7. Check status again
git status
# Output: On branch main, No commits yet
```

---

5 5. Working with Files

5.1 Creating and Adding Files

```
# Method 1: Create empty file (bash/zsh)
touch test.txt

# Method 2: Create file with content
echo "Hello Git!" > test.txt

# Method 3: Use text editor
nano test.txt
# Type content
# Save: Ctrl + O, then Enter
# Exit: Ctrl + X

# Alternative: vim (more complex)
vim test.txt
# Press 'i' to insert text
# Type content
# Press ESC, then type :wq to save and quit
```

5.2 Checking Status

```
# See what files changed
git status

# View detailed differences
git diff

# Diff for specific file
```

```
git diff test.txt

# Word-by-word diff (easier to read)
git diff --color-words test.txt
```

---

6 6. Staging and Committing

6.1 The Three States

```
Working Directory → Staging Area → Repository
      ↓                 ↓              ↓
   (modified)        (staged)      (committed)
```

6.2 Commands

```
# Stage specific file
git add test.txt

# Stage all changes
git add .

# Commit staged changes
git commit -m "Add test file"

# Stage and commit in one step (tracked files only)
git commit -a -m "Update test notes"

# Amend last commit (change message or add files)
git commit --amend -m "Corrected commit message"
```

---

7 7. Viewing History

7.1 Log Commands

```
# Full commit history
git log

# Compact one-line format
git log --oneline

# Visual branch graph
git log --graph --oneline --all

# Show changes in specific commit
git show <sha>

# Show last 5 commits
git log --oneline -5

# Show commits by specific author
git log --author="Your Name"
```

Example output:

```
a1b2c3d (HEAD -> main) Add new feature
e4f5g6h Update documentation
i7j8k9l Initial commit
```

---

8 8. Undoing Changes

8.1 Decision Tree

```
Have you committed yet?

  NO → Use git restore

  YES → Is it pushed to remote?

      NO → Use git reset

      YES → Use git revert
```

8.2 Commands

8.2.1 Discard Unstaged Changes

```
# Restore single file to last commit
git restore test.txt

# Restore all files
git restore .
```

8.2.2 Unstage Files (Keep Changes)

```
# Unstage specific file
git restore --staged test.txt

# Unstage all
git restore --staged .
```

8.2.3 Undo Commits

```
# Method 1: Soft reset (keep changes staged)
git reset --soft HEAD~1

# Method 2: Mixed reset (keep changes unstaged)
git reset HEAD~1

# Method 3: Hard reset ( discard all changes)
git reset --hard HEAD~1

# Method 4: Revert (safe for shared repos)
git revert <sha>
```

8.2.4 Recover File from Specific Commit

```
# View history to find commit
git log --oneline

# Restore file from that commit
git checkout <sha> -- test.txt
```

---

9 9. Quick Reference Table

Scenario	Command	Result
Unstage file	`git reset <file>`	Removes from staging area
Discard changes	`git restore <file>`	Reverts to last commit
Undo last commit	`git reset --soft HEAD~1`	Keeps changes staged
Safe undo (public)	`git revert <sha>`	Creates new commit
Edit commit history	`git rebase -i HEAD~3`	Rewrites last 3 commits

---

10 10. Advanced Operations

10.1 Stashing Changes

```
# Temporarily save uncommitted changes
git stash

# View stashed changes
git stash list

# Apply most recent stash
git stash apply

# Apply and remove stash
git stash pop

# Clear all stashes
git stash clear
```

10.2 File Operations

```
# Remove file from Git and filesystem
git rm <file>

# Remove from Git, keep in filesystem
git rm --cached <file>

# Rename/move file
git mv <old-name> <new-name>
```

10.3 Interactive Rebase

```
# Edit last 3 commits
git rebase -i HEAD~3

# Opens editor with options:
# pick   = use commit
# reword = change commit message
# edit   = modify commit
# squash = combine with previous commit
# drop   = remove commit
```

Editor workflow: 1. Press `i` to start editing (vim) 2. Make changes 3. Press `ESC` 4. Type `:wq` to save and quit

---

11 11. Branches

11.1 Why Use Branches?

- Develop features independently
- Experiment without affecting main code
- Collaborate with team members
- Maintain multiple versions

11.2 Branch Commands

```
# List all branches
git branch

# Create new branch
git branch feature-navbar

# Switch to branch
git checkout feature-navbar
# Or (newer syntax):
git switch feature-navbar

# Create and switch in one command
git checkout -b feature-navbar

# Merge branch into current branch
git merge feature-navbar

# Delete branch
git branch -d feature-navbar

# Force delete (unmerged changes)
```

```
git branch -D feature-navbar

# View all branches (including remote)
git branch -a
```

---

12 12. Complete Workflow Example

12.1 Feature Development Workflow

```
# 1. Initialize repository
git init my-project
cd my-project

# 2. Create and edit files
echo "Hello World" > index.html

# 3. Check status
git status

# 4. Stage and commit
git add index.html
git commit -m "Initial commit"

# 5. Create feature branch
git checkout -b feature-navbar

# 6. Make changes
echo "<nav>Navigation</nav>" >> index.html

# 7. Stage and commit changes
git add index.html
git commit -m "Add navigation bar"

# 8. Switch back to main branch
git checkout main

# 9. Merge the feature branch
git merge feature-navbar
```

```
# 10. View history
git log --oneline --graph --all

# 11. Delete feature branch (cleanup)
git branch -d feature-navbar

# 12. Verify
git branch
```

---

13 13. Repository Management

13.1 Remove Git Repository

```
# Remove Git (keep files)
rm -rf .git

# Remove everything ( dangerous!)
rm -rf *
```

13.2 Check Repository Structure

In Terminal:

```
# Visualize project tree
tree -a -L 2
```

In RStudio (R code):

```
# View directory tree
fs::dir_tree(all = TRUE, recurse = 1)
```

---

14 14. Finding Git Repositories

14.1 Search Your System

```
# Find all Git repositories on your machine
find ~ -type d -name ".git" -prune 2>/dev/null | sed 's|/.git||'
```

What it does: - Searches from home directory (`~`) - Finds directories named `.git` - Removes `.git` from output to show parent folder - Hides permission errors

---

15 15. Working with GitHub (Remote Repository)

15.1 Adding a Remote Repository

```
# Add GitHub as remote
git remote add origin https://github.com/username/repo-name.git

# Verify remote connection
git remote -v

# Rename branch to main (GitHub standard)
git branch -M main

# Push to GitHub (first time)
git push -u origin main

# Future pushes (after first time)
git push
```

15.2 Cloning a Repository

```
# Clone existing repository
git clone <repository-url>

# Clone with custom folder name
git clone <repository-url> custom-folder-name

# Navigate into cloned repo
cd custom-folder-name
```

```
# Check remote configuration
git remote -v
```

15.3 Syncing with Remote

```
# Push changes to remote
git push <remote> <branch>
# Example: git push origin main

# Pull changes from remote
git pull <remote> <branch>
# Example: git pull origin main

# Fetch changes without merging
git fetch origin
```

---

16 16. Local Backup Strategy

16.1 Create Bare Repository Backup

```
# Navigate to your working project
cd ~/Projects/important-app

# Create bare backup repository
git init --bare ~/Backups/important-app.git

# Add backup as remote
git remote add backup ~/Backups/important-app.git

# Push to backup
git push backup main

# Push all branches and tags
git push backup --all
git push backup --tags
```

16.2 Why Use Bare Repositories for Backups?

- No working directory (saves space)
- Only Git database (pure backup)
- Can push/pull like any remote
- Local = fast and secure

---

17 17. Essential Git Operations (Local)

17.1 Fast Local Commands

```
# Read local database (instant)
git log

# Compare local versions (instant)
git diff

# Create branch pointer (instant)
git branch new-feature

# Save to local repo (instant)
git commit -m "Changes"
```

Why so fast? - Everything is local (no network) - No server communication needed - Full history on your machine

---

18 18. Common Git Workflows

18.1 Daily Development Workflow

```
# 1. Start your day - update from remote
git pull origin main

# 2. Create feature branch
git checkout -b feature-login

# 3. Make changes
# ... edit files ...

# 4. Check what changed
git status
git diff

# 5. Stage changes
git add .

# 6. Commit with descriptive message
git commit -m "Add login form validation"

# 7. Push feature branch
git push origin feature-login

# 8. After code review/approval, merge to main
git checkout main
git merge feature-login

# 9. Push to remote
git push origin main
```

```
# 10. Clean up
git branch -d feature-login
```

18.2 Hotfix Workflow

```
# 1. Critical bug found in production
git checkout main

# 2. Create hotfix branch
git checkout -b hotfix-critical-bug

# 3. Fix the bug
# ... edit files ...

# 4. Commit fix
git add .
git commit -m "Fix critical authentication bug"

# 5. Merge back to main
git checkout main
git merge hotfix-critical-bug

# 6. Deploy/push
git push origin main

# 7. Clean up
git branch -d hotfix-critical-bug
```

---

19 19. Best Practices

19.1 Commit Messages

Good commit messages:

```
git commit -m "Add user authentication"
git commit -m "Fix database connection timeout"
git commit -m "Update documentation for API endpoints"
```

Bad commit messages:

```
git commit -m "fix"
git commit -m "changes"
git commit -m "asdf"
git commit -m "stuff"
```

19.2 When to Commit

Commit when: - Feature is complete and working - Bug is fixed - Logical unit of work is done - Tests pass - Code is reviewed

Don't commit: - Broken code - Half-finished features - Debugging code (console.log, print statements) - Sensitive data (passwords, API keys)

---

20 20. Troubleshooting

20.1 Common Issues

Issue: Commit rejected

```
# Error: Updates were rejected
# Solution: Pull first, then push
git pull origin main
git push origin main
```

Issue: Merge conflict

```
# Error: Automatic merge failed
# Solution: Resolve manually
# 1. Open conflicted files
# 2. Edit to resolve conflicts
# 3. Stage resolved files
git add <resolved-file>
git commit -m "Resolve merge conflict"
```

Issue: Accidentally committed wrong files

```
# Solution: Undo last commit
git reset --soft HEAD~1
# Files are unstaged, make corrections
git add <correct-files>
git commit -m "Correct commit"
```

---

21 21. Summary Cheat Sheet

21.1 Essential Commands

```
# Setup
git init                          # Initialize repository
git config --global user.name     # Set name
git config --global user.email    # Set email

# Basic workflow
git status                        # Check status
git add <file>                    # Stage file
git add .                         # Stage all
git commit -m "message"           # Commit
git log --oneline                 # View history

# Undoing
git restore <file>                # Discard changes
git restore --staged <file>       # Unstage
git reset --soft HEAD~1           # Undo commit
git revert <sha>                  # Safe undo

# Branches
git branch                        # List branches
git checkout -b <name>            # Create and switch
git merge <branch>                # Merge branch
git branch -d <branch>            # Delete branch

# Remote
git remote add origin <url>       # Add remote
git push -u origin main           # First push
git push                          # Subsequent pushes
git pull                          # Pull changes
```

```
git clone <url>                    # Clone repository
```

---

22 22. Next Steps

22.1 Continue Learning

1. **Practice locally** without GitHub
2. **Create test repositories** to experiment
3. **Use branches** for different features
4. **Read commit messages** in open-source projects
5. **Explore `.git` directory** to understand internals
6. **Try interactive rebase** to clean up history
7. **Set up local backups** with bare repositories

Master Git locally first, then add GitHub later!