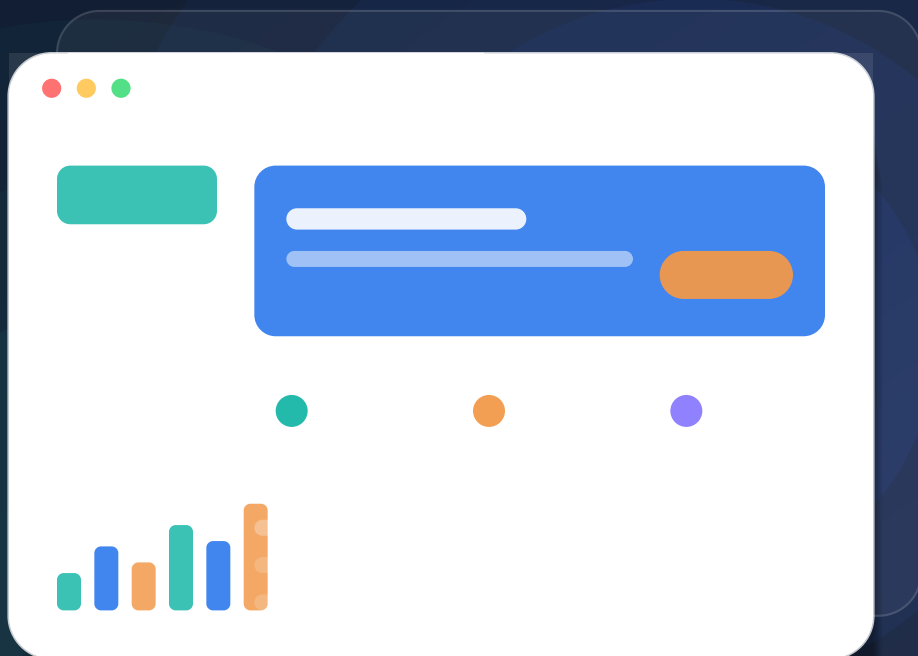


GIT AND GITHUB FOR BEGINNERS

VERSION CONTROL MADE SIMPLE



**From First Commit to
Your First Pull Request**

COMMIT • BRANCH • PUSH • COLLABORATE

Git and GitHub for Beginners

Version Control from First Commit to Pull Request

This guide is provided for general information. Git commands are stable, but GitHub menus and plan features change over time, so check GitHub documentation for current details and practise on a throwaway repository first.

Contents

- Introduction 4
- 1. What Is Version Control? 6
- 2. Installing and Setting Up Git 9
- 3. Your First Repository 12
- 4. Good Commits and Ignoring Files 16
- 5. Looking Back: History and Time Travel 19
- 6. Branches 22
- 7. Meet GitHub 25
- 8. Connecting Your Computer to GitHub 28
- 9. Collaborating with Pull Requests 32
- 10. Merge Conflicts Without Fear 35
- 11. Undoing Mistakes 38
- 12. Going Further with GitHub 42
- Appendix A: Command Cheat Sheet 46
- Appendix B: Glossary 49

Introduction

Have you ever saved a file as *report_final.docx*, then *report_final_v2.docx*, then *report_final_v2_REALLY_final.docx*? Version control exists to end that mess. It records every change you make to a project, lets you travel back to any earlier moment, and lets many people work on the same files without overwriting each other.

Git is the version control tool that almost everyone uses. **GitHub** is a website that stores Git projects online and adds tools for collaboration: sharing code, reviewing changes, tracking tasks and publishing websites. This book teaches both from scratch, with no previous experience needed.

You will learn the everyday commands, understand what they actually do, collaborate with other people through pull requests, and recover calmly when something goes wrong. The examples use code, but Git works on any text files, such as documents, notes, configuration or web pages.

What you will need

- A computer running **Windows, macOS or Linux**, and a terminal (Git Bash on Windows, Terminal on macOS and Linux).
- A **free GitHub account** from github.com (needed from Chapter 7).
- A **text editor**, such as Visual Studio Code, or any editor you like.

How to use this book

- **Complete beginner?** Read Chapters 1 to 6 in order, typing every command.
- **Want to share your work?** Go to Chapters 7 and 8.
- **Working in a team?** Chapters 9 and 10 cover branches, pull requests and conflicts.

- **Made a mistake?** Chapter 11 is the rescue chapter.
- **Need a quick lookup?** Appendix A is a command cheat sheet.

TIP

Git rewards practice more than reading. Make a throwaway folder and try every command as you go. Nothing you do in a practice project can harm anything else.

TIP

Commands in this book were written for the current Git 2.x series. Menus, screens and options on GitHub change from time to time, so if a button has moved, look for the same words nearby or check GitHub's documentation.

What Is Version Control?

Before learning commands, it helps to understand the idea. Version control is a safety net and a time machine for your work.

The problem it solves

- **Undo:** you broke something and cannot remember what you changed.
- **History:** you need to know who changed a line, when, and why.
- **Experiments:** you want to try an idea without risking the working version.
- **Teamwork:** two people edit the same file at the same time.
- **Backup:** your laptop dies and the project is gone.

Snapshots, not copies

Git records **snapshots** of your project, called **commits**. Each commit captures what every file looked like at one moment, together with a message saying what changed and why. Commits are linked in a chain, so you can walk backwards through the whole history.

```
A ---- B ---- C ---- D      (each letter is a commit)
first  add   fix   new
files  login bug   page
```

Git is distributed

Git keeps the **entire history on your own computer**. You can commit, look at old versions and create branches with no internet connection. When you are ready to share, you **push** your commits to a remote copy, usually on GitHub, and others can **pull** them down.

Git versus GitHub

	Git	GitHub
What it is	A free program on your computer	A website and service
Purpose	Tracks changes to files	Hosts Git projects and helps people collaborate
Needs internet?	No	Yes
Made by	Open-source community (started by Linus Torvalds)	A company, now owned by Microsoft
Alternatives	Few in practice	GitLab, Bitbucket, Codeberg and others

Everything you learn about Git works the same with GitLab or any other host. Only the website differs.

The three places your files live

Understanding this one picture explains most of Git:



- The **working directory** is your normal project folder, where you edit files.
- The **staging area** is a waiting room where you choose which changes go into the next commit.
- The **repository** is the hidden history of all commits, stored in a folder called `.git`.

You edit, then **add** the changes you want to the staging area, then **commit** them to history. The staging area looks like an extra step at first, but it lets you make tidy commits that each do one thing.

TIP

A **repository** (or **repo**) is simply a project folder that Git is tracking. Delete its `.git` folder and it becomes an ordinary folder again.

CHAPTER 2

Installing and Setting Up Git

Setting up takes about ten minutes, and you only do it once per computer.

Install Git

System	How to install
Windows	Download Git for Windows from git-scm.com and run the installer. The defaults are fine. It includes Git Bash, a terminal to use with this book
macOS	Open Terminal and type <code>git --version</code> . If missing, macOS offers to install developer tools. Or use Homebrew: <code>brew install git</code>
Debian / Ubuntu	<code>sudo apt install git</code>
Fedora / RHEL family	<code>sudo dnf install git</code>

Check that it worked:

```
$ git --version
git version 2.43.0
```

Your version number will differ. Any recent 2.x version is fine.

Tell Git who you are

Every commit records its author. Set your name and email once:

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

Use the same email address as your GitHub account (or the private no-reply address GitHub offers in its email settings) so that GitHub links your commits to you.

Choose a default branch name and editor

```
$ git config --global init.defaultBranch main
$ git config --global core.editor "code --wait"
```

The first line makes new repositories start on a branch called **main**, which is the common convention. The second sets Visual Studio Code as the editor Git opens for commit messages. Skip it to use your system's default (often nano or vim).

TIP

Stuck in **vim** after Git opens it? Press **Esc**, then type **:q!** and Enter to quit without saving. To save and quit, type **:wq**. Or avoid the problem by setting a friendlier editor as above.

Check your settings

```
$ git config --list
$ git config user.name
```

Getting help

```
$ git help commit      # full manual page
$ git commit -h        # short summary of options
$ git status           # often explains your next step
```

Git's own messages are unusually helpful. Read them before searching online, because they frequently contain the exact command you need.

Basic terminal skills you will use

Command	Meaning
<code>pwd</code>	Show the folder you are in
<code>ls</code>	List files (use <code>ls -a</code> to show hidden ones)
<code>cd folder</code>	Move into a folder
<code>cd ..</code>	Move up one folder
<code>mkdir name</code>	Create a folder
<code>cat file</code>	Show a file's contents
<code>echo "text" > file</code>	Write text into a file

CHAPTER 3

Your First Repository

Now for the real thing. In this chapter you will create a project, record your first commits and look at the history.

Create a project and start tracking it

```
$ mkdir my-first-repo
$ cd my-first-repo
$ git init
Initialized empty Git repository in .../.git/
```

git init creates the hidden `.git` folder, and the project is now a repository. Nothing is recorded yet.

Check the status

```
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use
"git add" to track)
```

git status is the command you will type most often. It tells you where you are and what has changed. Use it constantly.

Make a file and stage it

```
$ echo "# My First Repo" > README.md
$ git status
Untracked files:
  README.md

$ git add README.md
$ git status
Changes to be committed:
  new file:   README.md
```

The file started as **untracked**, meaning Git knew it existed but was not recording it. **git add** moved it into the staging area.

Make your first commit

```
$ git commit -m "Add README"
[main (root-commit) 3f2a9c1] Add README
1 file changed, 1 insertion(+)
```

The **-m** flag supplies the message. The mix of letters and numbers is the commit's unique ID (a hash). You will rarely need to type it in full. The first seven or so characters are enough.

Change a file and commit again

```
$ echo "A practice project." >> README.md
$ git status
    modified:   README.md

$ git diff
+A practice project.

$ git add README.md
$ git commit -m "Describe the project"
```

git diff shows exactly what you changed since the last commit, with lines added in green (+) and removed in red (-). Looking at it before committing is a great habit.

View the history

```
$ git log
$ git log --oneline
9b1d4e7 Describe the project
3f2a9c1 Add README

$ git log --oneline --graph --all    # with branches
```

The everyday loop

1. **Edit** files.
2. **Check** with `git status` and `git diff`.
3. **Stage** with `git add`.
4. **Commit** with `git commit -m "message"`.
5. **Repeat**, in small steps.

Command	What it does
<code>git init</code>	Start tracking the current folder
<code>git status</code>	Show what has changed and what is staged
<code>git add file</code>	Stage one file
<code>git add .</code>	Stage everything that changed here
<code>git commit -m "msg"</code>	Save staged changes as a snapshot
<code>git diff</code>	Show unstaged changes
<code>git diff --staged</code>	Show staged changes
<code>git log --oneline</code>	Compact history

TIP

Commit **small and often**. A commit that does one thing is easy to understand, review and undo. A commit that changes everything is none of those.

Good Commits and Ignoring Files

Anyone can make commits. Good commits make a project's history readable, and that pays off every time you have to find something later.

Writing good commit messages

A commit message is a note to your future self and your teammates. A widely used shape is a short summary line, then (if needed) a blank line and more detail.

```
Fix login error for users with long emails

The form rejected addresses over 30 characters
because of an old length check. Raised the limit
to 254, the standard maximum.
```

- Keep the first line **short**, ideally under about 50 characters.
- Use the **imperative mood**, as if giving an order: *Add*, *Fix*, *Remove*, not *Added* or *Fixes*.
- Say **what** and **why**, not how. The code already shows how.
- One logical change per commit.

Weak message	Better message
fixed stuff	Fix crash when the cart is empty
update	Update shipping rates for 2026
asdf	Add contact form validation
Final version!!	Prepare release 1.0

Stage parts of a file

If you changed two unrelated things in one file, you can commit them separately:

```
$ git add -p file.txt
```

Git shows each changed chunk and asks **y** (stage it), **n** (skip it) or **s** (split it smaller). This is how people keep commits tidy.

Fix the last commit

Forgot a file, or made a typo in the message? You can amend the most recent commit, as long as you have not shared it yet:

```
$ git add forgotten-file.txt
$ git commit --amend -m "Better message"
```

Telling Git what to ignore

Some files should never be committed: passwords, build output, temporary files and folders full of downloaded libraries. List them in a file called `.gitignore` at the top of your project:

```
# .gitignore
.env
*.log
node_modules/
__pycache__/.
.DS_Store
build/
secrets.txt
```

Pattern	Meaning
*.log	Any file ending in .log
folder/	A whole folder

Pattern	Meaning
/notes.txt	Only this file at the top level
!keep.log	Exception: do track this file
# comment	A comment line

Commit the `.gitignore` itself so the whole team shares the same rules. GitHub offers ready-made templates for most languages when you create a repository, and github.com/github/gitignore collects many more.

WARNING

Never commit passwords, API keys or private keys. Once a secret is in the history, deleting the file later does not remove it from the history. Keep secrets in an ignored file such as `.env` or in a secrets manager. If you ever commit one by accident, treat it as stolen and **change it immediately**.

Files that are already tracked

Adding a pattern to `.gitignore` does not affect files Git already tracks. Stop tracking one without deleting it:

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

Removing, moving and renaming

```
$ git rm old.txt           # delete and stage
$ git mv old.txt new.txt   # rename and stage
```

CHAPTER 5

Looking Back: History and Time Travel

One of Git's superpowers is that nothing is lost. This chapter shows how to explore the past and compare it with the present.

Reading the log

```
$ git log                # full detail
$ git log --oneline      # one line each
$ git log -5            # last five commits
$ git log --author="Alice" # by author
$ git log --since="2 weeks ago" # by date
$ git log -- path/to/file.txt # one file's history
$ git log -p            # with the changes
```

Inspect a single commit

```
$ git show 9b1d4e7      # what that commit changed
$ git show HEAD         # the latest commit
$ git show HEAD~2       # two commits before
```

HEAD is Git's name for where you are now, usually the latest commit on your current branch. **HEAD~1** means one commit before it.

Compare versions

```
$ git diff              # working dir vs staged
$ git diff --staged     # staged vs last commit
$ git diff 3f2a9c1 9b1d4e7 # two commits
$ git diff main feature  # two branches
```

Who changed this line?

```
$ git blame file.txt
```

git blame shows who last changed each line and in which commit. Despite the name, it is meant for finding the *reason* for a line, not for pointing fingers. Look up the commit message to learn why.

Search the history

```
$ git log -S"password" --oneline    # text added/removed
$ git log --grep="login"           # search messages
$ git grep "TODO"                  # search files
```

Look at an old version safely

You can visit an old commit to see how the project looked, then come back:

```
$ git switch --detach 3f2a9c1    # look around
$ git switch main                # come back
```

While detached, you are viewing history, not on any branch. Do not make new work there unless you create a branch first (Chapter 9).

Restore one file from the past

```
$ git restore --source=3f2a9c1 README.md
```

This brings that file's old contents into your working directory. Review it, then commit if you want to keep it.

Tags: bookmarks for important commits

```
$ git tag v1.0                # lightweight tag
$ git tag -a v1.0 -m "First release" # annotated tag
$ git tag                    # list tags
$ git push origin v1.0       # share a tag
```

Tags give a readable name to a commit, usually for release versions such as **v1.0** or **v2.3.1**.

TIP

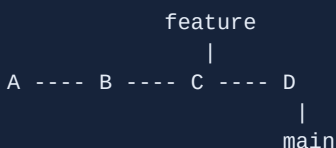
A nicer log: `git log --oneline --graph --decorate --all`. Create a shortcut with `git config --global alias.lg "log --oneline --graph --decorate --all"` and then type `git lg`.

Branches

Branches are what make Git feel magical. A **branch** is an independent line of work, so you can try new ideas, fix bugs or build features without touching the stable version, and combine the work later.

What a branch is

A branch is just a movable label pointing at a commit. Creating one is instant and costs almost nothing.



Everyday branch commands

```
$ git branch                # list branches
$ git branch feature-login   # create one
$ git switch feature-login   # move to it
$ git switch -c feature-login # create and move
$ git branch -d feature-login # delete (if merged)
```

Older tutorials use `git checkout` for switching branches. It still works, but `git switch` is clearer.

A complete example

```
$ git switch -c add-footer
$ echo "<footer>Contact us</footer>" > footer.html
$ git add footer.html
$ git commit -m "Add footer"

$ git switch main
$ ls                                     # footer.html is not here!
```

Switching back to **main** makes the footer file disappear, because it exists only on the other branch. Switch again, and it returns. Git rewrites your working folder to match whichever branch you are on.

Merging branches together

When the feature is ready, bring it into **main**:

```
$ git switch main
$ git merge add-footer
Updating 9b1d4e7..c4e8a02
Fast-forward
 footer.html | 1 +

$ git branch -d add-footer
```

There are two kinds of merge:

Type	When it happens	Result
Fast-forward	main has not moved since you branched	Git just moves the label forward. No new commit
Merge commit	Both branches have new commits	Git creates a new commit that joins the two histories

Before merge:

```
      C---D feature
     /
A-B---E main
```

After merge:

```
      C---D
     /   \
A-B---E---M main
```

Good habits with branches

- Keep **main** working and releasable at all times.
- Make a **new branch for each task**, such as a feature or a fix.
- Give branches **descriptive names**: `fix-login-bug`, `add-search`.
- Keep branches **short-lived**. Long-running branches become hard to merge.
- Delete branches after merging. The commits remain in history.

Saving unfinished work: stash

Need to switch branches but are not ready to commit? Shelve your changes temporarily:

```
$ git stash                # put changes aside
$ git switch other-branch
# ... do something else ...
$ git switch original-branch
$ git stash pop            # bring them back
$ git stash list           # see what is stashed
```

TIP

If you are unsure where you are, run `git status` or `git branch`. The current branch is marked with a star.

Meet GitHub

Git alone is a personal tool. GitHub is where your work meets the world: a backup, a portfolio and a meeting place for teams and open-source communities.

What GitHub gives you

- **Remote storage** for your repositories, an off-site backup.
- **Collaboration:** pull requests, code review and discussion.
- **Issues:** track bugs, requests and tasks.
- **GitHub Pages:** publish a website straight from a repository.
- **Actions:** automatically run tests and tasks when code changes.
- **Projects:** simple boards for planning work.
- **Discover and reuse** millions of open-source projects.

Create your account

1. Go to **github.com** and choose **Sign up**.
2. Pick a **username** carefully. It appears in your repository addresses and is public.
3. Verify your email address.
4. Turn on **two-factor authentication** in your account's security settings, and save your recovery codes somewhere safe.

Create a repository on GitHub

1. Click the **+** icon at the top and choose **New repository**.
2. Enter a **name** (no spaces; use hyphens) and an optional description.

3. Choose **Public** (anyone can see it) or **Private** (only you and people you invite).
4. For an existing local project, leave the options to add a README, .gitignore and licence **unticked**, to avoid conflicts.
5. Click **Create repository**.

GitHub then shows the commands for connecting a local project. They look like the ones in the next chapter.

A tour of a repository page

Tab or area	What it is for
Code	The files, branches and the README shown beneath
Issues	Bugs, ideas and tasks
Pull requests	Proposed changes awaiting review
Actions	Automated workflows and their results
Insights	Activity, contributors and statistics
Settings	Access, branches, Pages and security (owners only)

The README

The file `README.md` is shown on the front page of your repository. It is written in **Markdown**, a simple way to format text. A good README says what the project is, how to use it and how to contribute.

```
# Project Name

A short description of what this does.

## Install

    npm install

## Usage

Run `npm start` and open the browser.

## Licence

MIT
```

Choosing a licence

If you make a public project and want others to be allowed to reuse it, add a **licence**. Without one, the default is that nobody else may legally use your code. Popular choices include **MIT** (very permissive) and **Apache 2.0** or **GPL** (with different conditions). GitHub's site **choosealicense.com** explains the differences in plain language. For anything important, check the licence terms or ask a qualified person.

WARNING

Everything in a public repository is visible to the whole world, including its full history. Before making a project public, check that no passwords, keys, private data or embarrassing messages are anywhere in the history.

Connecting Your Computer to GitHub

Your local repository and your GitHub repository are separate copies. A **remote** is the link between them. This chapter covers the connection, the sign-in methods, and the three commands you will use all the time: push, pull and clone.

Sign-in methods

GitHub no longer accepts your account password for Git commands over HTTPS. You have three good options:

Method	How it works	Good for
GitHub CLI (gh)	Run gh auth login, follow the browser prompts	The easiest start
SSH key	A key pair on your computer; upload the public half to GitHub	Long-term use, no repeated prompts
HTTPS with a token	A personal access token or the Git Credential Manager sign-in	Where SSH is blocked

Option 1: the GitHub CLI

```
$ gh auth login
# choose GitHub.com, HTTPS or SSH, and log in via browser
$ gh auth status
```

The **gh** tool is a free download from cli.github.com. It also lets you create repositories and pull requests from the terminal.

Option 2: an SSH key

```
$ ssh-keygen -t ed25519 -C "you@example.com"
$ cat ~/.ssh/id_ed25519.pub      # copy the output
```

On GitHub, open **Settings**, then **SSH and GPG keys**, then **New SSH key**, and paste the public key (the file ending in **.pub**). Then test the connection:

```
$ ssh -T git@github.com
Hi username! You've successfully authenticated...
```

WARNING

Never share your **private** key (the file without **.pub**). Only the public key goes to GitHub. Protect the private key with a passphrase.

Connect a local project to GitHub

Suppose you made `my-first-repo` in Chapter 3, and have created an empty repository on GitHub with the same name:

```
$ git remote add origin \
    git@github.com:USERNAME/my-first-repo.git
$ git branch -M main
$ git push -u origin main
```

If you use HTTPS, the address looks like `https://github.com/USERNAME/my-first-repo.git` instead. The name **origin** is the conventional nickname for your main remote. The **-u** flag remembers the link, so later you can type just `git push`.

The three big commands

Command	What it does
<code>git push</code>	Send your new commits to the remote
<code>git pull</code>	Fetch new commits from the remote and merge them in
<code>git clone URL</code>	Download a whole repository for the first time
<code>git fetch</code>	Download new commits without merging them
<code>git remote -v</code>	Show your configured remotes

Clone an existing project

```
$ git clone https://github.com/OWNER/project.git
$ cd project
$ git log --oneline
```

Cloning copies the files, the complete history and the remote link, all in one step.

Fetch versus pull

git pull is really two steps combined: **fetch** (download) plus **merge**. If you want to look before combining, run them separately:

```
$ git fetch origin
$ git log main..origin/main --oneline    # what is new?
$ git merge origin/main
```

A daily rhythm

1. **git pull** at the start of the day, to get others' changes.
2. Create a **branch**, work, and commit in small steps.
3. **git push** your branch to GitHub.
4. Open a **pull request** (next chapter).

TIP

If `git push` is rejected with a message about the remote having work you do not have, run `git pull` first, resolve anything it asks for, then push again.

Collaborating with Pull Requests

A **pull request** (PR) is a proposal to merge your branch into another one, usually **main**. It is the heart of teamwork on GitHub, because it gives everyone a place to discuss and review changes before they become part of the project.

The workflow, step by step

1. **Update** your local main: `git switch main` then `git pull`.
2. **Create a branch**: `git switch -c fix-typo`.
3. **Make changes and commit** them with clear messages.
4. **Push the branch**: `git push -u origin fix-typo`.
5. On GitHub, click **Compare and pull request** (a banner appears after a push).
6. **Write a title and description** explaining what and why.
7. **Request a review** from a teammate.
8. **Respond to feedback** by pushing more commits to the same branch. The PR updates itself.
9. When approved, **merge** it. Then delete the branch.
10. Back on your computer: `git switch main`, `git pull`, `git branch -d fix-typo`.

Writing a good pull request

- A **clear title**: *Fix crash when the cart is empty*.
- A **description** of the problem, your solution, and how to test it.
- **Small and focused**: one purpose per PR. Large PRs are slow to review and easy to get wrong.

- Link to the related issue, for example by writing **Closes #42**, which closes the issue when the PR merges.
- Add **screenshots** for visual changes.

Reviewing someone else's pull request

- Open the **Files changed** tab and read the differences.
- Click the **+** beside a line to leave a **comment**.
- Choose **Approve**, **Comment** or **Request changes** when you finish the review.
- Be **kind and specific**. Criticise the code, never the person, and explain your reasons.
- Say what is **good** too. Encouragement matters.

Ways to merge a pull request

Option	What happens
Create a merge commit	Keeps every commit, adds a commit joining the branches
Squash and merge	Combines all the branch's commits into one tidy commit
Rebase and merge	Replays the commits on top of main for a straight history

Teams choose a favourite. **Squash and merge** is popular because it keeps the main history short and readable.

Forks: contributing to projects you do not own

You usually cannot push to someone else's repository. Instead, you make your own copy, a **fork**, on GitHub, work there, and propose your changes back.

1. Click **Fork** on the project's page.

2. **Clone your fork:** `git clone https://github.com/YOU/project.git`.
3. Add the original as **upstream**: `git remote add upstream https://github.com/OWNER/project.git`.
4. Create a branch, make changes, commit, push to your fork.
5. Open a **pull request** from your fork to the original.
6. Keep your fork current: `git fetch upstream` then `git merge upstream/main`.

Protecting main

Teams often turn on **branch protection** in the repository settings, so nobody can push directly to main. Changes must arrive through a reviewed pull request, and can be required to pass automatic checks first. It is a great habit even for small teams.

Issues

Issues are GitHub's built-in to-do list and bug tracker. Each has a title, a description, labels, assignees and a discussion thread. A good bug report says what you did, what you expected, what happened instead, and your system details.

TIP

Your very first contribution can be tiny: fix a typo in documentation. Look for issues labelled **good first issue**, which maintainers use to welcome newcomers.

Merge Conflicts Without Fear

Sooner or later you will see a message that begins *CONFLICT*, and it feels alarming. It is not. A merge conflict is simply Git saying: *two people changed the same lines, and I need a human to decide*.

Why conflicts happen

Git merges most changes automatically. It only asks for help when both branches changed the **same part of the same file** in different ways, or one deleted a file the other edited.

What a conflict looks like

```
$ git merge feature
Auto-merging menu.txt
CONFLICT (content): Merge conflict in menu.txt
Automatic merge failed; fix conflicts and then
commit the result.
```

Open the file. Git marks both versions:

```
Opening hours
<<<<<< HEAD
Monday to Friday, 8am to 5pm
=====
Monday to Saturday, 9am to 4pm
>>>>>> feature
```

- Between <<<<<< HEAD and ===== is **your current branch's** version.
- Between ===== and >>>>>> feature is the **incoming** version.

Resolving it, step by step

1. Run `git status` to see which files are in conflict.
2. **Open each file** and decide what the final text should be. Keep one side, the other, or combine them.
3. **Delete the marker lines** (`<<<<<<`, `=====`, `>>>>>>`) completely.
4. **Test** that the result still works.
5. Stage the file: `git add menu.txt`.
6. Finish: `git commit` (Git prefills a merge message).

```
Opening hours
Monday to Saturday, 8am to 4pm
```

The final text above combines the ideas of both sides. Only the people involved can decide that, which is why Git asks.

Changed your mind? Abort

```
$ git merge --abort
```

This returns everything to how it was before you started the merge. It is a safe way out.

Tools that help

- **Visual Studio Code** shows conflicts with clickable *Accept Current*, *Accept Incoming* and *Accept Both* buttons.
- **GitHub's web editor** can resolve simple conflicts inside a pull request.
- **git mergetool** opens a configured visual comparison tool.

Avoiding conflicts

- **Pull often** so your branch stays close to main.
- Keep branches and pull requests **small and short-lived**.
- **Talk** to teammates about who is editing which files.
- Avoid reformatting whole files, which touches every line.
- Split large files into smaller ones.

A word on rebase

Rebase is an alternative to merging. It replays your commits on top of the latest main, giving a straight history:

```
$ git switch feature  
$ git rebase main
```

WARNING

Rebase **rewrites history**. Only rebase branches that you alone are using, and never rebase commits others have already pulled. When unsure, use merge. It is the safer choice for beginners, and perfectly professional.

TIP

Conflicts are normal in teams. Take a breath, read the markers, decide, and move on. After you have done it three times, it is routine.

Undoing Mistakes

Everyone makes mistakes in Git. The good news is that Git is built to make most of them recoverable, if you pick the right tool. The key is knowing **where** the mistake is: unsaved edits, staged changes, a local commit, or something already pushed.

Choose the right undo

Situation	What to do
Edited a file and want the old version back	<code>git restore file</code>
Staged a file by mistake	<code>git restore --staged file</code>
Typo in the last commit message	<code>git commit --amend</code>
Forgot a file in the last commit	<code>git add file</code> , then <code>git commit --amend</code>
Want to undo a commit that is already shared	<code>git revert COMMIT</code>
Want to discard local commits that are not shared	<code>git reset</code> (see below)
Lost commits and need them back	<code>git reflog</code>

Throw away uncommitted changes

```
$ git restore notes.txt      # one file
$ git restore .              # everything
```

WARNING

This cannot be undone. Uncommitted edits that you discard with `restore` are gone for good, because Git never saved them. Check with `git diff` first.

Unstage a file

```
$ git restore --staged notes.txt
```

The file keeps your edits. It just leaves the staging area.

Undo a shared commit safely: revert

```
$ git revert 9b1d4e7
```

git revert creates a **new commit** that does the opposite of an earlier one. History stays intact, which makes it the right choice for anything you have already pushed.

Move back locally: reset

git reset moves the current branch to an earlier commit. It has three common modes:

Mode	Effect on commits	Effect on your files
--soft	Undone	Changes stay staged
--mixed (default)	Undone	Changes stay in files, unstaged
--hard	Undone	Changes are deleted

```
$ git reset --soft HEAD~1 # undo last commit, keep work
$ git reset HEAD~1        # undo it, keep edits
$ git reset --hard HEAD~1  # undo it and delete edits
```

WARNING

Use **--hard** with great care, and never use `reset` on commits that have been pushed and shared. It rewrites history and causes trouble for everyone else. Use `git revert` instead.

The safety net: reflog

Git quietly records every place your branch tip has been, for a while after (typically some weeks). If you think you have lost commits, look there:

```
$ git reflog
9b1d4e7 HEAD@{0}: reset: moving to HEAD~1
3f2a9c1 HEAD@{1}: commit: Add search page

$ git reset --hard 3f2a9c1    # return to that point
```

The reflog has rescued countless people who thought their work was lost. It only exists on your own computer, though, so it is no substitute for pushing.

Committed a secret?

1. **Change the secret immediately** (new password or key). Assume it is stolen.
2. Removing it from history needs special tools such as **git filter-repo**, and a force-push, and everyone must re-clone. Do this only after rotating the secret.
3. Add the file to `.gitignore` so it never happens again.

Force pushing

After rewriting history, a normal push is refused. The safer forced version is:

```
$ git push --force-with-lease
```

WARNING

Never force-push to **main** or any branch other people use.

Force-pushing branches that only you use (such as your own PR branch) is common and fine.

A general rule

Before anything risky, make a safety branch: `git branch backup`. It costs nothing, and if things go badly you can return to it.

TIP

When panicking, stop typing and run `git status`, `git log --oneline` and `git reflog`. Understand where you are before changing anything else.

Going Further with GitHub

With the basics in place, GitHub has several powerful features worth knowing. This chapter introduces them briefly, so that you know what exists and where to look.

Publish a website with GitHub Pages

GitHub Pages hosts a static website (HTML, CSS and JavaScript) directly from a repository, for free for public repositories on most plans. Availability for private repositories depends on your plan.

1. Put your site files, with an `index.html`, in a repository.
2. Open **Settings**, then **Pages**.
3. Choose the branch (and folder) to publish, and save.
4. After a minute or two, GitHub shows your site's address, in the form `https://USERNAME.github.io/REPO/`.

Every time you push changes, the site updates. You can also attach your own domain name in the same settings page.

Automate things with GitHub Actions

Actions run tasks automatically whenever something happens in your repository, such as testing code on every push. A workflow is a small YAML file saved in `.github/workflows/`:

```
name: Tests
on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - run: echo "Run your tests here"
```

The version after the @ sign changes over time. Check the action's own page for the current recommended version. GitHub gives a free allowance of minutes, with limits that depend on your plan and on whether the repository is public.

Organise work with Projects and milestones

- **Projects** are boards and tables where issues move from *To do* through *Done*.
- **Milestones** group issues toward a target such as a release.
- **Labels** categorise issues: *bug*, *enhancement*, *documentation*.

Keep your code safe

- **Two-factor authentication** on your account. Always.
- **Dependabot** alerts and updates warn about vulnerable libraries.
- **Secret scanning** detects keys accidentally committed (features vary by plan).
- Choose **private** repositories for anything confidential.
- Review who has **access** regularly, and use the least permission needed.

Working with teams and organisations

An **organisation** lets a group own repositories together, with roles for members, teams and permissions. Collaborators on personal repositories can be invited under **Settings, then Collaborators**.

Role	Typical abilities
Read	View and clone; open issues
Write	Push branches and merge pull requests
Admin	Change settings and manage access

Useful GitHub shortcuts

- Press **.** (full stop) on a repository page to open an in-browser editor, or press **?** to see keyboard shortcuts.
- Press **t** to search for a file by name.
- Click a line number to link to it. Add a commit hash to a comment, and it becomes a link.
- Mention people with **@username** and issues with **#number**.

Other tools worth exploring

- **GitHub Desktop** is a friendly graphical app for people who prefer clicking.
- **VS Code** has Git built in, with a visual changes panel.
- **GitHub CLI (gh)** handles issues and pull requests from the terminal.
- **GitHub Codespaces** provides a development environment in the browser (usage limits and costs depend on your plan).
- **GitLab** and **Bitbucket** offer alternatives with the same Git underneath.

Where to go next

- Practise by contributing to a small open-source project's documentation.
- Read the free **Pro Git** book at git-scm.com/book, which goes deeper on everything here.
- Learn **interactive rebase** to tidy history, and **git bisect** to hunt down which commit introduced a bug.
- Explore **Git hooks** and continuous integration to automate quality checks.
- Use Git for **everything**: notes, writing, configuration files. The more you use it, the more natural it becomes.

TIP

The best way to learn Git is to use it daily on something real. Put your own project in a repository today, and commit as you work.

Appendix A: Command Cheat Sheet

Setup and starting

Command	What it does
<code>git config --global user.name "N"</code>	Set your name
<code>git config --global user.email "E"</code>	Set your email
<code>git init</code>	Make this folder a repository
<code>git clone URL</code>	Copy a remote repository

Everyday work

Command	What it does
<code>git status</code>	What has changed?
<code>git add file</code>	Stage a file
<code>git add -p</code>	Stage chosen chunks
<code>git commit -m "msg"</code>	Commit staged changes
<code>git commit --amend</code>	Fix the last commit
<code>git diff</code>	Unstaged changes
<code>git diff --staged</code>	Staged changes
<code>git log --oneline</code>	Short history
<code>git show COMMIT</code>	One commit in detail

Branches

Command	What it does
<code>git branch</code>	List branches
<code>git switch -c name</code>	Create and switch
<code>git switch name</code>	Switch to a branch
<code>git merge name</code>	Merge a branch into the current one
<code>git branch -d name</code>	Delete a merged branch
<code>git stash / git stash pop</code>	Shelve and restore changes

Remotes

Command	What it does
<code>git remote add origin URL</code>	Link a remote
<code>git remote -v</code>	Show remotes
<code>git push -u origin main</code>	First push, remember link
<code>git push</code>	Send commits
<code>git pull</code>	Fetch and merge
<code>git fetch</code>	Fetch only

Undoing

Command	What it does
<code>git restore file</code>	Discard edits (cannot undo)
<code>git restore --staged file</code>	Unstage
<code>git revert COMMIT</code>	Undo a commit with a new commit
<code>git reset --soft HEAD~1</code>	Undo last commit, keep staged
<code>git reflog</code>	Find lost commits

Before you push: a checklist

1. **git status** shows only what you expect.
2. **git diff --staged** contains no passwords, keys or junk.
3. The commit **message** explains the change.
4. You are on the **right branch**.
5. You have **pulled** the latest changes.
6. Your **tests** pass, if you have any.

Appendix B: Glossary

Branch. An independent line of development; a movable pointer to a commit.

Clone. A complete local copy of a remote repository.

Commit. A saved snapshot of the project, with a message and an author.

Conflict. When Git cannot combine changes automatically and needs a human decision.

Fetch. Download new commits from a remote without merging them.

Fork. Your own copy of someone else's repository on GitHub.

git. The version control program that tracks changes on your computer.

GitHub. A website that hosts Git repositories and adds collaboration tools.

HEAD. Your current position: usually the latest commit on the current branch.

Hash. The long unique ID of a commit, often shortened to seven characters.

Issue. A tracked bug, task or idea on GitHub.

Main. The conventional name of a project's primary branch.

Merge. Combine the changes from one branch into another.

Origin. The usual nickname for the main remote repository.

Pull. Fetch new commits from a remote and merge them in.

Pull request. A proposal to merge a branch, with review and discussion.

Push. Send your commits to a remote repository.

Rebase. Replay commits on top of another branch to make a straight history.

Remote. A copy of the repository hosted elsewhere, such as on GitHub.

Repository. A project folder with its complete tracked history.

Revert. Create a new commit that undoes an earlier one.

Staging area. Where changes wait to be included in the next commit.

Stash. A temporary shelf for unfinished changes.

Tag. A named bookmark on a commit, often used for releases.

Working directory. The files in your project folder that you edit.