codeop

Software success

Learning the essentials of version control and Git to excel when working in remote teams as a new developer.

A bit about me

01 Graduated CodeOp in May 2020

Completed the full-stack development bootcamp and was the first cohort to go fully remote.

02 Landed my first developer role in April 2021

After freelancing for a year, I interned at a Dutch design agency and quit 5 months later. Sometimes it's just not a match.

03 Found my happy place

Joined a 2-person startup and stayed for a year and a half. I now work as a software engineer at the FAO of the United Nations.

Prerequisites

01 Code Editor

You should have a code editor installed such as Visual Studio Code, Atom, or Cursor to write and edit code

02 Terminal or Command Line Basics

A basic understanding of using the terminal or command line interface is essential.

03 GitHub Account

You should have a GitHub account set up. This will allow you to create join projects, and practice using Git features like branching and pull requests after the workshop.

What is Git anyway?

01 Version Control

Git tracks changes in code, allowing multiple developers to collaborate without conflicts.

02 Snapshots

It saves the entire project state at specific points (commits), enabling easy reversion and history tracking

03 Branching/Merging

Git allows developers to create branches for new features, which can be merged back into the main codebase once complete.

How is Git different to GitHub?

01 Tool vs. Platform

Git is a local version control system; GitHub is a web-based platform for hosting Git repositories.

02 Local vs. Remote

Git works on your machine; GitHub stores repositories online for collaboration.

03 Collaboration Features

Git handles version control; GitHub adds social features like pull requests and project management tools.

The standard flow: Cloning the repo

git clone <repo-url>

- Creates a new directory:
  - It automatically makes a folder with the repository's name.
- Initializes Git:
  - It sets up the .git directory, so there's no need to run git init.
- Downloads the repo:
  - It copies all files, branches, and commits to your machine.

git pull origin main

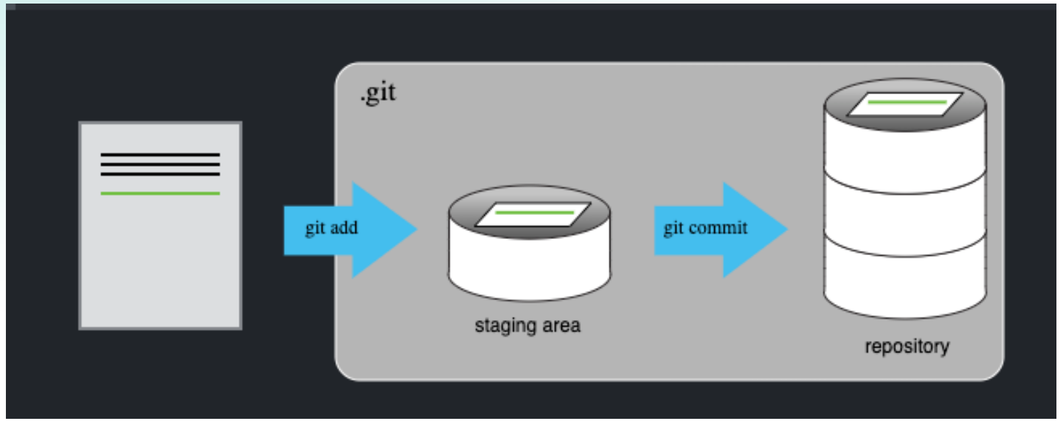
- This command syncs your local main branch with the remote version to avoid conflicts later.

The standard flow: Create a feature branch

git checkout -b feature/your-feature-name

- Working on a feature branch keeps your work separate from main until it’s ready to be merged, making collaboration cleaner and more organised.

The standard flow: Make changes and commit



git add <file name>

If you think of Git as taking snapshots of changes over the life of a project, git add specifies **what** will go in a snapshot (putting things in the staging area)

git commit –m “commit message”

git commit then **actually** takes the snapshot, and makes a permanent record of it (as a commit)

What makes a good commit message?

A good commit message is clear, concise, and informative.

Start with a short summary in the imperative mood (e.g., "Add user login functionality" instead of "Added" or "Adding").

Focus on the “Why” and the “What”

Avoid describing how the change was made in detail, as this is often evident in the code itself.

The good, the bad and the ugly

Fix broken image links on homepage

Add feature for task

Change header styling

Add email validation to registration form

The verdicts are in

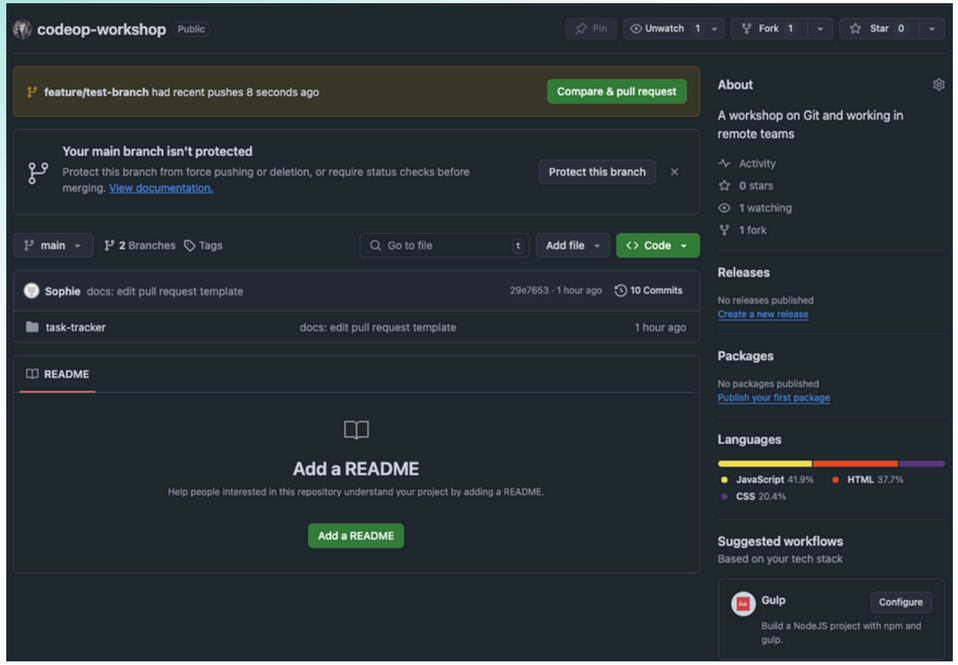
Fix broken image links on homepage
Clearly states what was fixed (image links) and where (homepage), making the purpose and location of the change easy to understand.

Add feature for task
This message is unclear about which feature was added and how it relates to the task. A clearer version might be: Add filtering option for task list.

Change header styling
Too vague. A better version could be: Increase header font size for better readability.

Add email validation to registration form
Specifies the action taken (adding validation) and its purpose (ensuring correct email format), giving clear context for the change.

The standard flow: Push the feature branch to remote



git push origin feature/your-feature-name

Using git push alone works if the branch is already linked to a remote (like origin). However, specifying origin explicitly (git push origin feature/your-feature-name) ensures:

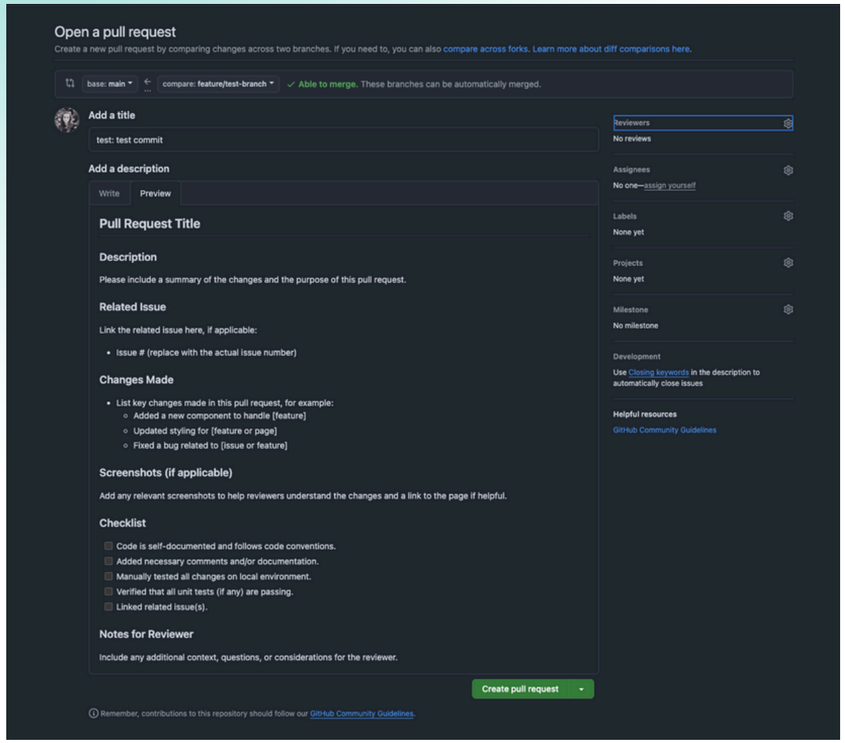
Clear Targeting

It directly pushes to the specified remote (e.g., origin), which is useful if multiple remotes exist.

Less Error-Prone

Explicitly naming the remote avoids pushing to the wrong repository, especially in multi-remote setups.

The standard flow: Create a pull request (PR)



Create a Pull Request (PR)

Go to the repository on GitHub, find your feature branch, and create a pull request. A PR lets teammates review your code before it's merged into main.

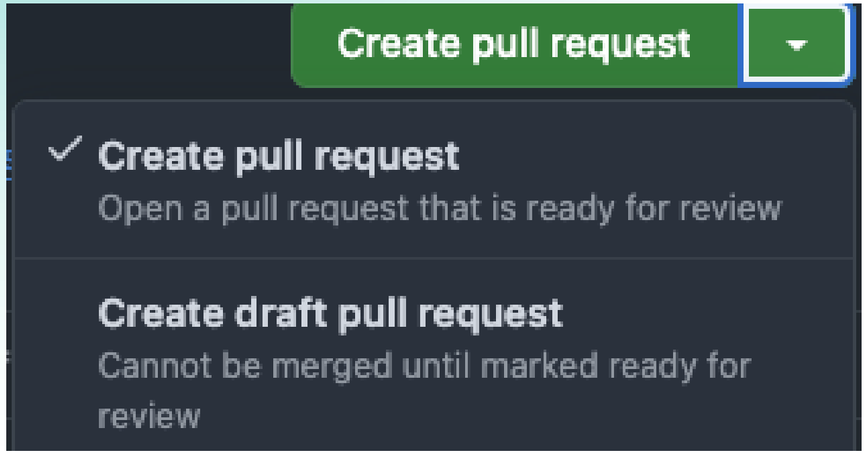
Add a reviewer

Choose a team member as a reviewer and assign the PR to them in GitHub.

Assign the PR

The assignee is responsible for the PR or issue. They own the task and are expected to complete it or address feedback. Think of the assignee as the primary person working on or managing the change.

The standard flow: Submitting a pull request



Keep it Focused

Address a single feature or bug fix to keep the PR manageable.

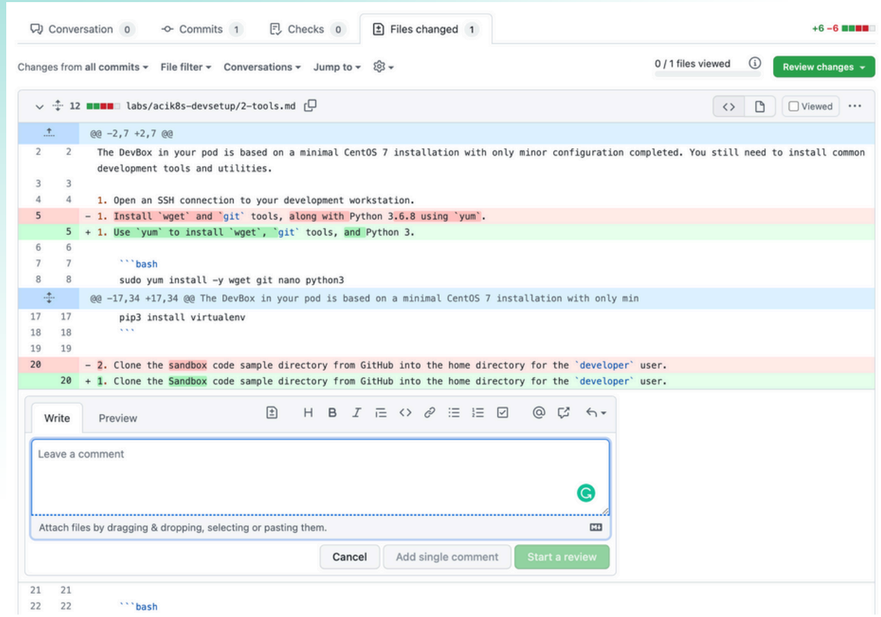
Write Clear Descriptions

Explain what changes were made and why, including any related issues from the GitHub Issues board.

Review Your Code

Check your changes for clarity and correctness before submitting to ensure everything looks good.

The standard flow: Reviewing code



Understand the Changes

Read the code and the PR description to grasp what was changed and why.

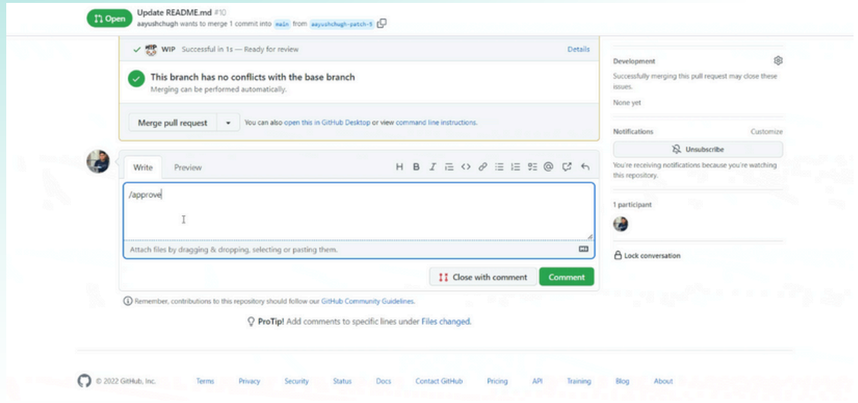
Check Clarity and Quality

Ensure the code is clear, follows best practices, and is easy to read.

Provide Constructive Feedback

Offer specific, actionable suggestions while highlighting strengths to encourage growth.

The standard flow: Approve and merge



Review and Approve

Go through the code changes, ensure everything meets quality standards, and approve the PR if it looks good.

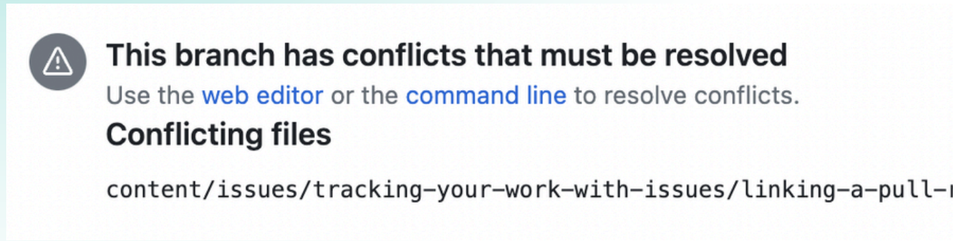
Merge the PR

Click the merge button (often labeled "Merge pull request") to integrate the changes into the main branch.

Delete the Branch

After merging, delete the feature branch to keep the repository tidy and reduce clutter.

The standard flow: Merge conflicts



Stay Calm and Analyse

Take a deep breath and carefully read the conflict markers in the files to understand what changes are conflicting.

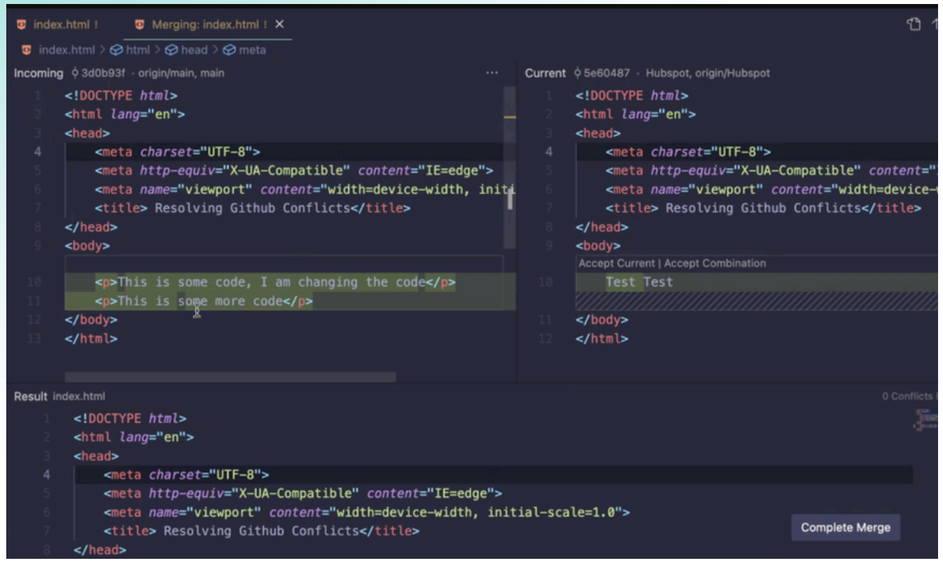
Choose the Right Changes

Decide whether to keep your changes, the incoming changes, or a combination of both. Make the necessary edits to resolve the conflict.

Test After Merging

After resolving conflicts, test your code to ensure everything works correctly before committing the changes.

The standard flow: Merge conflicts



git pull origin main

Make sure your local branch is up to date with the main branch by running this command

Identify Conflicts

When you pull, Git will indicate which files have conflicts. Open those files in your code editor. Look for conflict markers (<<<<<<<, =======, >>>>>>>) that show the conflicting changes.

Resolve the Conflicts

Review the conflicting sections and decide which changes to keep (your changes, the incoming changes, or a combination).

The standard flow: Merge conflicts

git add <file-name>

After resolving all conflicts, stage the updated files

git commit -m "Resolved merge conflicts in [file names]“

Commit your resolved changes with a clear message

git push origin your-branch-name

Finally, push your branch to the remote repository

Git Cheatsheet

git init

Initialize a new Git repository

git status

Check the status of your repo

git log

View commit history

git checkout <branch-name>

Switch to an existing branch

git branch

List all branches

Git Cheatsheet

git remote -v

View remote repositories

git diff

View files changes

git reset HEAD~1

Undo last commit (but keep changes)

git stash

Stash your changes (save changes temporarily)

git stash apply

Apply the latest stashed changes

Key takeaways

1. Nothing that is committed to version control is ever lost, unless you work really, really hard at losing it.
2. Version control matters, even if you aren't collaborating with other people.
3. Conflicts are inevitable, it's just about knowing how to manage them.

Let’s practice

CodeOp Task Tracker

- Github repository here
- Github issues board here

Step 1: Clone the repo
Step 2: Pull the latest changes from main
Step 3: Assign yourself an issue
Step 4: Create a feature branch
Step 5: Code
Step 6: Commit your changes
Step 7: Push to the remote repo
Step 8: Submit a pull request
Step 9: Assign a teammate as a reviewer
Step 10: Approve and merge :)

Questions

Email

sophie.ogden@fao.org

LinkedIn

Find me on LinkedIn here

Resources

Up for Grabs
First Timers Only
Code Triage
Conventional Commits