Git Learning Journal Noah Staley

0. Initializing a Repository

**Command:** git init

**What it does:** Creates a new, empty Git repository in the current folder (makes a hidden .git folder).

**Example:**git init

Initialized empty Git repository in C:/Users/gladi/Desktop/module 2/.git/

* **Mistakes I made:**
  + Tried adding files before running git init → Git said it wasn’t a repo yet.
* **Reminder:** Always run git init once in a new project before using other Git commands.

1. Checking Status

**Command:** git status

* **What it does:** Shows what branch you’re on, what files are staged, modified, untracked, or deleted.

**Example:** git status

On branch main

nothing to commit, working tree clean

* **Mistakes I made:**
  + Typed status without git → bash: status: command not found.
  + Typed git stauts (spelling error).
* **Reminder:** Always type git status with correct spelling. This should be your “home base.”

2. Listing Files

**Command:** ls

* **What it does:** Shows the files in the current folder.

**Example:** ls

DONE.txt 'IN PROGRESS.txt' PRIVATE.txt TODO.txt

* **Mistakes I made:**
  + Tried cls (Windows habit), but Git Bash uses clear to clear the screen.
* **Reminder:** Use clear instead of cls.

3. Adding Files

**Commands:** git add <file>

git add .

git add -u .

* **What it does:** Stages changes so they’re ready to commit.

**Examples:** git add TODO.txt

git add .

* **Mistakes I made:**
  + Tried git add -m "deleted" → wrong (you can’t add with a message).
  + Tried git add .gotognore (spelling mistake) instead of .gitignore.
* **Reminder:** Use git add <file> for specific files, git add . for everything. Commit messages only go with git commit -m.

4. Committing Changes

**Command:** git commit -m "message"

* **What it does:** Saves a snapshot of staged changes into history.

**Examples:** git commit -m "I added text"

* **Mistakes I made:**
  + Forgot -m → got stuck in commit editor.
  + Used vague commit messages like "idk".
* **Reminder:** Always use clear messages like "Add TODO.txt" or "Ignore PRIVATE.txt".

5. Removing Files from Git (but keeping them locally)

**Command:** git rm --cached <file>

* **What it does:** Stops Git from tracking a file but doesn’t delete it locally.

**Example:** git rm --cached PRIVATE.txt

* **Mistakes I made:**
  + Typed git rm -- cached (extra space).
  + Forgot that .gitignore needs the filename inside to keep ignoring it.
* **Reminder:** No space between -- and cached.

6. Resetting and Restoring

**Commands:** git reset

git restore <file>

git reset --soft HEAD~1

* **What they do:**
  + reset unstages or rewinds commits.
  + restore undoes changes in working directory.

**Examples:** git reset HEAD TODO.txt

git restore TODO.txt

* **Mistakes I made:**
  + Typed git rest instead of restore.
* **Reminder:** Use restore to undo file edits, reset to unstage or undo commits.

7. Looking at History

**Commands:** git log

git show <commit-hash>

* **What it does:** Lets me see what I committed and when.

**Examples:** git log

git show 397c7873486abe727cffcd72585a6b534af8b98b

* **Mistakes I made:**
  + Tried git show Noah Staley (author name doesn’t work). Must use commit hash.
* **Reminder:** Copy the commit hash from git log.

8. Checking Out Old Commits

**Command:** git checkout <commit-hash>

* **What it does:** Lets me “time travel” to an old commit.

**Examples:** git checkout 397c7873486abe727cffcd72585a6b534af8b98b

* **Mistakes I made:**
  + Tried git checkout master but my branch is main.

**Reminder:** To return to latest work, always run:  
  
 git checkout main

9. Connecting to GitHub

A) Linking a local project to a new GitHub repo

1. On GitHub → make a new repo (don’t add README if I already have files).

In Bash:  
  
 git remote add origin https://github.com/USERNAME/REPO-NAME.git

git branch -M main

git push -u origin main

Next pushes can just be:  
  
 git push

B) Cloning an existing GitHub repo

1. Copy the repo’s HTTPS link from GitHub.

In Bash:  
  
 git clone https://github.com/USERNAME/REPO-NAME.git

cd REPO-NAME

1. Run git status to confirm you’re inside the repo.

Mistakes I might make:

* Typing git remote add orgin (spelling mistake → should be origin).
* Forgetting cd into the folder after cloning.
* Mixing up SSH vs HTTPS links.

Reminder:

* Use git remote -v to check what GitHub repo my local project is linked to.
* origin = nickname for the GitHub repo.
* main = my branch name (not master).

## **Key Concepts I Want to Retain**

* **Git** is a **distributed version control system** that runs locally on a machine. It tracks changes to files, allows branching, and maintains a history of project development.
* **GitHub** is a **remote hosting service** that stores Git repositories online, enabling collaboration, sharing, and backup.
* Git can exist **without GitHub**, but GitHub extends Git by providing features like issue tracking, pull requests, code reviews, and centralized collaboration.
* Interaction between Git and GitHub:  
  + git init → starts a local repo.
  + git remote add origin [URL] → links the local repo to GitHub.
  + git push → sends commits to GitHub.
  + git pull → brings updates from GitHub to local repo.

## **Creating and Using a Repository in GitHub**

1. **Create repository on GitHub**:  
   * Choose a name, description, public/private option.
   * Initialize with a README if desired.

**Connect local Git to GitHub repo**:  
  
 git init

git remote add origin https://github.com/[username]/[repo].git

**Stage and commit files locally**:  
  
 git add .

git commit -m "Initial commit"

**Push files to GitHub**:  
  
 git push -u origin main

1. **Ongoing workflow**:  
   * Make edits → git add → git commit → git push.
   * Sync changes with team using git pull.

## **Difficulties Encountered**

* Remembering the difference between **Git (tool)** and **GitHub (platform)** was initially confusing.
* First time linking local repo to GitHub required learning how **remotes** work.
* Conflicts can occur when multiple people push to the same branch — requires using git pull and sometimes resolving merge conflicts.

## **Notes for Future Reference**

* Always check git status to see which files are staged, unstaged, or untracked.
* Use .gitignore to prevent unnecessary files (e.g., large media files, system files) from being tracked.
* Best practice: make frequent commits with clear messages to maintain a clean project history.
* For collaboration, branching is safer than working directly on main.

# **Module 4**

### **Topic: Creating, Managing, and Pushing Repositories with Git and GitHub**

## **Key Concepts and Actions**

* **Creating and Editing Files**
  + Initially tried git add README.md but Git reported the file did not exist.
  + Used echo "# SDEV-143-module-4" >> README.md to create the README file, then successfully added and committed it.
* **Staging and Committing**
  + Added both README.md and index.html.
  + Committed changes with the message *“My HTML FIX fix: #1”*.
* **Setting Branches and Remote**
  + Renamed default branch to main with git branch -M main.

Added the remote repository URL, but initially got an error (remote origin already exists) and had to reset it using:  
  
 git remote set-url origin https://github.com/Gladiator87s/SDEV-143-module-4.git

* **Push Errors and Fix**
  + First push attempt failed with a **non-fast-forward error** because the remote repo already had commits.
  + Tried git pull origin main --rebase but encountered *“couldn’t find remote ref main”* (GitHub repo didn’t have a main branch yet).

Finally succeeded by pushing the local branch to GitHub:  
  
 git push -u origin main

* **Branch Management**
  + Experimented with creating and deleting branches:  
    - git branch Noahsbranch
    - git branch develop
    - git branch improve-readme-description
  + Learned that you cannot delete a branch you are currently on (e.g., Noahsbranch).
  + Deleted other test branches (DeleteMe).
* **Switching and Merging**
  + Checked out Noahsbranch and later switched to develop.
  + Created a feature branch improve-readme-description, edited README, committed, then merged into develop with a **fast-forward merge**.
* **Pushing Branches to GitHub**
  + Successfully pushed multiple branches to GitHub:  
    - git push origin main
    - git push origin Noahsbranch
    - git push origin develop
  + GitHub responded with pull request links for collaborative merging.

## **Difficulties Encountered**

* Confusion between **local and remote branches**. At first, pushing failed because the remote repo had different history than the local repo.
* The error *“couldn’t find remote ref main”* was caused by the remote not having a main branch yet.
* Accidentally attempted to delete a branch currently in use, which Git disallowed.
* The warning *“LF will be replaced by CRLF”* showed up — related to line ending differences between Windows (CRLF) and Linux (LF).

## **Lessons Learned / Future Reference**

* Always check git status and git branch -a to confirm current branch and what will be pushed.

To fix a mismatched remote, use:  
  
 git remote set-url origin [new-URL]

* When encountering push rejections, **pull first** (with rebase if needed) before retrying a push.
* Branching workflow:  
  1. Create a new branch for a feature.
  2. Make commits there.
  3. Merge into develop or main only when ready.
* Deleting branches requires switching to another branch first.
* GitHub automatically provides pull request links after pushing new branches — useful for collaboration.

# **GitHub Issues and Project Management**

### **Topic: Creating and Managing Issues in GitHub**

## **Key Concepts and Actions**

### **1. Creating an Issue**

* Learned how to open a new issue in GitHub.
* Issues act like **tickets** that track bugs, features, or tasks.
* Includes: **title**, **description**, and optional metadata like labels, assignees, and milestones.

### **2. Interacting with an Issue**

* Added **comments** to issues to communicate progress and clarifications.
* Recognized issues as **collaborative discussion threads** where multiple contributors can provide updates.

### **3. Labels**

* Used labels such as *bug*, *enhancement*, or *documentation*.
* Labels are a way to **categorize and prioritize** issues for better project organization.

### **4. Assignees**

* Assigned issues to myself or teammates.
* Helps establish **accountability** and clarity on who is responsible for resolving a task.

### **5. Linking Issues with Commits**

Connected commits to specific issues using references like:  
  
 git commit -m "Fix login bug, see #5"

* GitHub automatically linked the commit to the issue, providing **traceability** between code changes and project tasks.

### **6. Working on the Commit**

* Made changes in a local branch, committed them, and pushed to GitHub.
* Practiced good commit messages to clearly explain what issue was being addressed.

### **7. Referencing an Issue**

* Learned to reference issues in commit messages and pull requests with **#issue-number**.
* Example: *“Added error handling to form submission (ref #7)”*.
* References appear in the issue timeline for context.

### **8. Closing an Issue Using Keywords**

* Used GitHub’s **special keywords** in commit messages or pull requests to automatically close issues.

Example:  
  
 git commit -m "Fix login redirect error. Closes #10"

* After merging, GitHub automatically closes the referenced issue.

### **Journal Entry – Git Practice** **Topic:** Creating a pull request, handling merge conflicts, and branch management in Git

#### **Actions Taken:**

1. **Force Pushed Branches:**
   * Pushed changes to both main and improve-app-style using git push origin <branch> -f.
   * This updated remote branches with local changes.
2. **Branch Switching and Pulling:**
   * Used git switch main and git pull origin main to sync the local main branch with the remote.
   * Switched between branches (main, develop, improve-app-style) to merge changes and keep them updated.
3. **Merging Branches:**
   * Merged main into improve-app-style and develop.
   * Initially mistyped git megre instead of git merge, corrected it.
   * Encountered a merge conflict when merging main into improve-app-style.
4. **Resolving Merge Conflict:**
   * Conflict occurred because working.txt was **deleted in main** but **modified in improve-app-style**.
   * Resolved by re-adding working.txt and writing "hello" into it.
   * Committed the resolution with message: "fixed wotrking text merge fail".
5. **File Deletion Attempt & Correction:**
   * Attempted invalid commands (git -d working.txt, git -rm working.txt).
   * Correct command: git rm working.txt.
   * Successfully committed with: "removed working from master".
6. **Editing Files:**
   * Accidentally tried invalid Git commands (git echo, git append).
   * Correct approach: echo "hello" > working.txt.
   * Added the file, committed with: "added text".
7. **Final Steps:**
   * After resolving conflicts and syncing, pushed branch improve-app-style successfully to GitHub.
   * Confirmed clean working directory with git status.

#### **Lessons Learned:**

* **Force push (-f)** overwrites remote history—should be used carefully.
* Always specify origin main when pulling/pushing to avoid Git misinterpretation.
* Merge conflicts can arise when one branch deletes a file while another modifies it. These must be resolved manually.
* Be precise with Git syntax (git rm, git add ., git commit -m "message").
* Commands like echo are **shell commands**, not Git commands.

#### **Reflection:**

Today’s session was valuable for practicing **pull requests, branch merging, and conflict resolution**. I made mistakes with syntax but learned how to properly delete files, resolve merge conflicts, and push updated branches. This exercise gave me hands-on experience with real-world Git workflows and how to troubleshoot errors effectively.

# **Journal Entry – Resolving Merge Conflicts in Git**

### **Topic: Practical Workflow for Merge Conflicts**

## **Key Concepts and Actions**

### **1. Creating and Working in a Feature Branch**

Created a branch for adding a new habit:  
  
 git checkout -b add-gym

Added a new file and committed changes:  
  
 echo "- Gym" > habits.txt

git add habits.txt

git commit -m "Change to gym habit"

### **2. Making Conflicting Changes on Main Branch**

Switched back to main:  
  
 git checkout main

Edited habits.txt with different content:  
  
 echo "- Sleep more" > habits.txt

git add habits.txt

git commit -m "updated sleep"

### **3. Merging and Resolving Conflicts**

Attempted merge of add-gym into main:  
  
 git merge add-gym

→ Encountered a conflict:  
  
 CONFLICT (add/add): Merge conflict in habits.txt

Resolved conflict manually in habits.txt, then staged and committed:  
  
 git add habits.txt

git commit -m "updated sleep merge conflicts"

git commit -m "updated gym merge conflicts"

git commit -m "updated gym and sleep merge conflicts"

### **4. Pushing Changes and Syncing Branches**

Pushed resolved changes to GitHub:  
  
 git push

git push origin develop

Attempted to check out master, but branch did not exist:  
  
 git checkout master

* → Correct branch was main.

### **5. Merging Development Branches**

Merged develop into main:  
  
 git checkout main

git merge develop

* → Merge completed using the **ORT strategy**.

Attempted to abort merge when no merge was in progress:  
  
 git merge --abort

* → Received fatal: There is no merge to abort (MERGE\_HEAD missing).

## **Difficulties Encountered**

* Merge conflict occurred when changes were made to the same file (habits.txt) on different branches.
* Confusion between branch names (master vs main).
* Learned the importance of committing and merging carefully to avoid losing work.

## **Lessons Learned / Future Reference**

* **Branching** is a safe way to work on features without affecting the main codebase.
* **Merge conflicts** happen when two branches make changes to the same lines in a file; they must be resolved manually.

Always check branch names with:  
  
 git branch

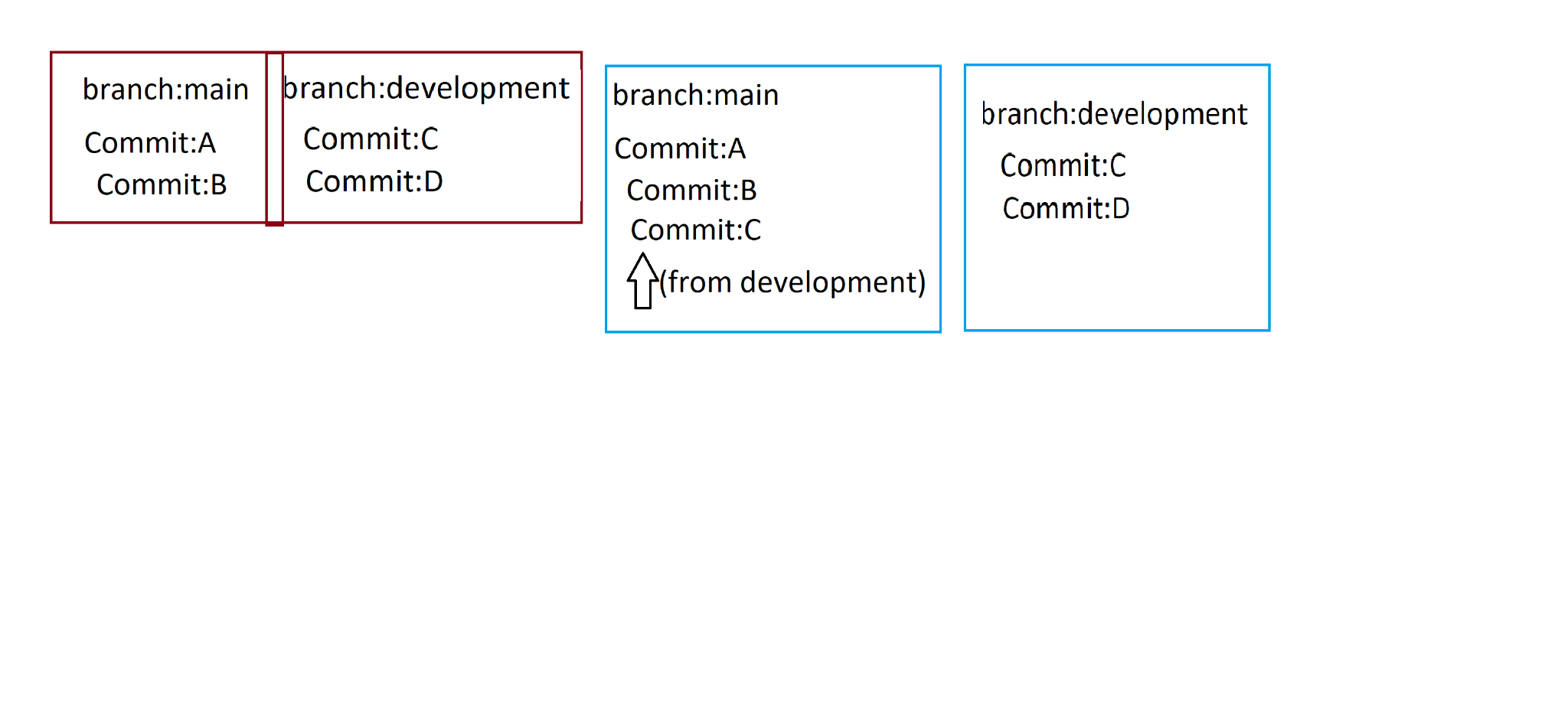
Resolve conflicts by editing the file, then staging the changes:  
  
 git add [file]

git commit -m "resolved conflict"

* Use git status often to track what files are staged or in conflict.
* Pushing changes ensures that your work is backed up remotely.

If you accidentally try to check out a branch that doesn’t exist, verify branch names with:  
  
 git branch -a

* The --abort option only works during an active merge conflict.



Pros:You can use it to test parts of the commits from other branches or even whole features. Like for example if I was developing a whole new section of the website but I don't understand why there is a bug I could pull it to another branch and try to isolate the file to help with bug testing.

Cons: you will now have multiple of the same commit and if you get confused on what you are editing and what is being edited it will be a nightmare to try to fix.

git clean -n or --dry-run Preview the files and directories that would be removed.

git clean -f Remove untracked files in the working directory.

git clean -fd Remove untracked files and directories.

git clean -i Interactively choose files to remove.

git clean -X Remove ignored files only (from .gitignore).

git clean -x Remove all untracked files, including ignored ones.

Journal Entry – Managing Branch and Stash Operations in Git

**Used git stash to Save Work in Progress** git stash push

Saved the current working directory and index state as a stash entry labeled:  
  
 WIP on seprate-code-and-style: 0c194f7 read me

**Verified and Restored Stash** git stash list

git stash show

git stash pop

Successfully reapplied the stashed changes and confirmed there were no remaining stash entries.

**Created a New Commit** git commit -m "bad commit to read me"

Logged the new commit using:  
  
 git log --oneline

(Initially mistyped as --online.)

**Hard Reset** git reset --hard origin/seprate-code-and-style

Learned the correct sequence of staging, committing, and restoring files.

Practiced using git stash to temporarily save work and restore it later.

Identified the importance of correct branch spelling when using remote reset commands.

Reinforced understanding of the difference between:

* git restore --staged <file> (unstages changes)
* git checkout -- <file> (reverts working directory changes)
* git stash (temporarily saves all changes)

### **GUI Exercises**

#### **1. Reverting**

In Git GUI, I practiced reverting a previous commit to undo unwanted changes while maintaining commit history.  
 **Insight:** The revert creates a *new* commit that undoes previous modifications — the original commit remains visible in the log, preserving history integrity.

#### **2. Stashing**

In Git GUI, I tested saving temporary changes without committing using **Stash Changes**.  
 This feature is useful when switching branches without losing current progress. The stashed work can later be reapplied or discarded.

#### **3. Resetting**

Using the GUI, I explored **Reset Commit** to move the branch pointer to a previous commit.  
 This reinforced the three reset modes:

* **Soft:** Keeps changes staged.
* **Mixed:** Keeps changes but unstages them.
* **Hard:** Discards all changes.

#### **4. Creating an Issue**

Through the GitHub interface, I created a new **issue** to track a bug related to the README.md edits.  
 **Purpose:** Demonstrated how issues serve as formal tasks or problem reports to be addressed in later commits or pull requests.

#### **5. Making Another Commit**

Made an additional commit through Git GUI to document the resolution for the issue created earlier.  
 This was a straightforward process using **Stage Changed Files → Commit → Push**.

#### **6. Creating a Pull Request**

Finally, I created a **pull request** from the seprate-code-and-style branch to merge into the main branch.  
 This allowed for a review and approval workflow, ensuring code changes are checked before integration.