The Ultimate Git & GitHub Survival Guide for Your Final Year Project

Welcome, Future Developers!

You're about to embark on your Final Year Project with three teammates. You'll be writing code, lots of it, all working on the same application. Right now, you might be thinking: "How will we share our code? What if we accidentally delete each other's work? What if everything breaks?"

Here's the truth: **Without version control, team coding is a nightmare.** Imagine four people trying to edit the same Word document by emailing it back and forth, creating files like `project_final_FINAL_v3_akash_edits_REAL_FINAL.docx`. Now imagine that with code, where one wrong change can break everything.

Git and GitHub exist to solve exactly this problem. Think of them as: - **Git:** A time machine for your project that saves every version and lets you work in parallel universes (branches) - **GitHub:** A cloud headquarters where all four of you sync your work safely

By the end of this guide, you'll go from zero to confidently collaborating like professional developers. You'll learn exactly what you need—no more, no less—to build your FYP without losing sleep over lost code or merge disasters.

Let's turn this potential chaos into a smooth, professional workflow. One step at a time.

Module 1: The Solo Foundation (Your Local Time Machine)

The Problem: "How do I save different versions of my work without creating a million folders?"

Right now, if you're working on code, you might save versions like this:

```
my_project/
  my_project_v1/
  my_project_v2_working/
  my_project_backup/
  my_project_ACTUALLY_WORKING/
```

This is messy, confusing, and takes up tons of space. Worse: you can't see *what* changed between versions or easily go back to a specific moment in time.

The Git Mental Model: Your Project's Snapshot Album

Git is a time machine and snapshot camera for your project folder.

Imagine you're working on a photo album. Every time you finish arranging a page, you take a photograph of the entire album (a "snapshot"). These photos are stored in a special box (the `.git` folder). You can: - Look back at any previous photo (version) - Compare two photos to see what changed - Jump back to an old photo if you mess up the current one

In Git terms: - **Repository (repo):** Your project folder with the magic `.git` folder inside - **Commit:** One snapshot/photograph with a description - **Working Directory:** The current state of your files (what you see in your folder) - **Staging Area:** The "preview" before taking the snapshot—you choose which changes to include

Setting Up Your Time Machine

Step 1: Install Git **For Windows:** 1. Download Git from: https://git-scm.com/download/win 2. Run the installer 3. Use all default options (just keep clicking "Next") 4. When asked about text editor, choose "Nano" or "Visual Studio Code" if you have it

For Mac: 1. Open Terminal (search "Terminal" in Spotlight) 2. Type: `git --version` and press Enter 3. If not installed, it will prompt you to install Xcode Command Line Tools—click "Install"

For Linux (Ubuntu/Debian):

```
sudo apt update
sudo apt install git
```

Verify Installation: Open your terminal/command prompt and type:

```
git --version
```

You should see something like `git version 2.43.0`

Step 2: Introduce Yourself to Git Git needs to know who you are to label your snapshots. Run these commands (replace with YOUR information):

```
git config --global user.name "Your Full Name"
git config --global user.email "your.email@example.com"
```

What each part means: - `git config` — the command to set configuration settings - `--global` — applies this setting to all your projects (not just one) - `user.name` / `user.email` — the settings you're changing

Check it worked:

```
git config --global --list
```

You should see your name and email printed.

Step 3: Create Your First Repository Let's create a practice project:

```
# 1. Create a new folder for your project
mkdir my-first-repo
# mkdir = "make directory" (make a new folder)

# 2. Enter that folder
cd my-first-repo
# cd = "change directory" (move into the folder)

# 3. Initialize Git (turn this folder into a repository)
git init
# Creates the hidden .git folder that tracks everything
```

Success looks like: You see the message `Initialized empty Git repository in /path/to/my-first-repo/.git/`

Step 4: Create Your First Snapshot (Commit) Let's create a file and take our first snapshot:

```
# 1. Create a simple text file
echo "# My First Project" > README.md
# This creates a file called README.md with one line of text

# 2. Check the status of your repository
git status
```

What you'll see:

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        README.md

nothing added to commit but untracked files present (use "git add" to track)
```

Translation: Git sees a new file (`README.md`) but you haven't told it to include this file in the next snapshot yet.

```
# 3. Add the file to the staging area (prepare it for the snapshot)
git add README.md
# You're saying: "Include this file in my next snapshot"

# 4. Check status again
git status
```

Now you'll see:

```
Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   README.md
```

Translation: The file is "staged" and ready to be committed.

```
# 5. Take the snapshot (commit)
git commit -m "Add README file"
# -m means "message" - every commit needs a description
```

Success looks like:

```
[main (root-commit) a1b2c3d] Add README file
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

That random code `a1b2c3d` is your commit's unique ID (like a photograph number).

Step 5: View Your History

```
git log
# Shows the history of all snapshots
```

You'll see:

```
commit a1b2c3d4e5f6g7h8i9j0k1l2m3n4o5p6q7r8s9t0
Author: Your Name <your.email@example.com>
Date:   Fri Jan 30 10:30:00 2026 +0800

    Add README file
```

To exit this view, press `q` (for "quit").

For a simpler, one-line-per-commit view:

```
git log --oneline
```

The Basic Git Workflow (Repeat This Constantly)

This is the cycle you'll repeat hundreds of times in your FYP:

```
1. Make changes to your files
   ↓
2. git status (see what changed)
   ↓
3. git add <filename> (stage the changes)
   ↓
4. git commit -m "description" (take the snapshot)
```

Let's practice:

```bash
# Edit the README file (add a new line)
echo "This is my FYP project" >> README.md

# Check what changed
git status
# Shows: modified: README.md

# See the exact changes
git diff
# Shows line-by-line what was added/removed (press 'q' to exit)

# Stage the change
git add README.md

# Commit it
git commit -m "Add project description to README"

# View history
git log --oneline
```

Writing Good Commit Messages

Bad Examples: - "updated stuff" - "fixes" - "asdjklasdh" - "commit"

Good Examples: - "Add user login form" - "Fix bug in product search function" - "Update database schema for orders table" - "Remove unused CSS from homepage"

The Rule: A good commit message completes the sentence: "This commit will _____"

Mini Practice Task #1: Your Personal Time Machine

Task: Each team member should do this individually on their own computer.

1. Create a folder called `fyp-practice`
2. Initialize it as a Git repository
3. Create a file named `teaminfo.txt`
4. Write your name and student ID in it
5. Stage and commit the file with the message "Add team member info"
6. Edit the file to add your email
7. Commit again with message "Add email address"
8. View your commit history

Expected Output of `git log --oneline`:

```
b2c3d4e Add email address
a1b2c3d Add team member info
```

Expected Project State: - You have a folder with a `.git` subfolder (hidden) - You have `teaminfo.txt` with your name, ID, and email - You have 2 commits in your history

---

Module 2: The Shared Cloud (Your Team's Central Hub)

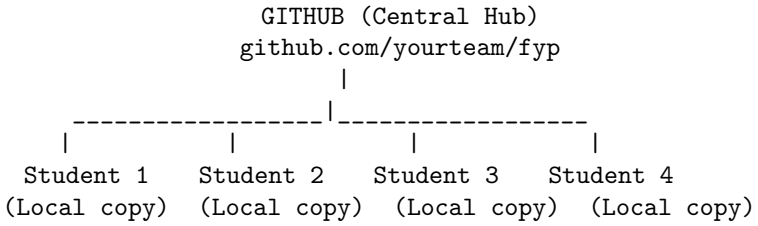
The Problem: "How do we all work on the same project without using USB drives?"

Right now, your project only exists on YOUR computer. How do your three teammates get the code? Email? Google Drive? USB drive? What happens when all four of you make changes? How do you combine them?

This is why GitHub exists.

The GitHub Mental Model: Your Project's Cloud Headquarters

Think of your team's workflow like this:



- **GitHub:** The "source of truth"—the main, official version of your project that lives online
- **Remote:** The name for the GitHub version (specifically, it's usually called `origin`)
- **Clone:** Making a copy of the GitHub project onto your computer
- **Push:** Sending your local commits up to GitHub
- **Pull:** Downloading new commits from GitHub to your computer

Key Concept: Everyone has their own FULL copy of the project, complete with all history. You work on your copy, then sync changes through GitHub.

Setting Up Your Team's GitHub Hub

Step 1: Create a GitHub Account (All 4 Students)

1. Go to: https://github.com
2. Click "Sign up"
3. Use your student email if possible (GitHub offers free perks for students)
4. Choose a professional username (this will be visible to employers!)
    - Good: `akash-kumar`, `sarah-chen`, `davidlee-dev`
    - Bad: `cooldude123`, `ihatecode`, `xxxgamerxxx`

Step 2: Create Your Team's Repository (ONE PERSON ONLY) Decide who will be the "Repository Owner" (usually the team leader). Only this person does these steps:

1. Log into GitHub
2. Click the + icon in the top-right corner
3. Select "New repository"
4. Fill in the details:
   - **Repository name:** `fyp-team-project` (or your actual project name, no spaces!)
   - **Description:** "Final Year Project - [Your App Name]"
   - **Public or Private:** Choose **Private** (keeps your FYP code confidential)
   - **DO NOT check "Add a README file"** (we'll add it ourselves)
   - Click "Create repository"

Success looks like: You see a page with setup instructions and a URL like:

```
https://github.com/your-username/fyp-team-project.git
```

Step 3: Connect Your Local Project to GitHub (Repository Owner) Remember the project you created in Module 1? Let's put it on GitHub.

```
# Make sure you're inside your project folder
cd fyp-practice

# Add the GitHub repository as a "remote" (a link to the cloud version)
git remote add origin https://github.com/YOUR-USERNAME/fyp-team-project.git
# "origin" is just a nickname for this URL

# Verify it worked
git remote -v
# You should see the URL listed twice (for fetch and push)

# Send your commits to GitHub
git push -u origin main
# "push" = upload your commits
# "-u" = set "origin main" as the default destination for future pushes
# "origin" = the nickname for GitHub
# "main" = the branch name (we'll explain branches in Module 3)
```

If you see an error about "main" vs "master":

```
git branch -M main
# This renames your branch to "main"
# Then try the push command again
```

You might be asked to authenticate: - GitHub now requires a Personal Access Token (not your password) - Follow this guide: https://docs.github.com/en/authentication/keeping-

your-account-and-data-secure/creating-a-personal-access-token - When prompted, paste your token (it won't show characters—that's normal)

Success looks like:

```
Enumerating objects: 6, done.
Counting objects: 100% (6/6), done.
Writing objects: 100% (6/6), 450 bytes | 450.00 KiB/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/YOUR-USERNAME/fyp-team-project.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Step 4: Add Your Teammates as Collaborators (Repository Owner)

1. Go to your repository on GitHub
2. Click "Settings" (top menu)
3. Click "Collaborators" (left sidebar)
4. Click "Add people"
5. Enter each teammate's GitHub username or email
6. They'll receive an invitation email—they must accept it

Each teammate should check their email and accept the invitation.

Step 5: Clone the Repository (The Other 3 Students) Now the other three students need to download the project to their computers.

IMPORTANT: Do NOT create a new folder and run `git init`. Instead:

```
# 1. Navigate to where you want to store the project
cd ~/Documents
# or wherever you keep your coding projects

# 2. Clone (download) the repository from GitHub
git clone https://github.com/REPOSITORY-OWNER-USERNAME/fyp-team-project.git
# Use the actual URL from the GitHub repository

# 3. Enter the new folder that was created
cd fyp-team-project

# 4. Check the remote is set up
git remote -v
```

Success looks like: - A new folder `fyp-team-project` appears in your directory - When you run `git remote -v`, you see the GitHub URL - When you run `git log --oneline`, you see the commits the owner pushed

All four students now have identical copies of the project!

The Team Sync Workflow

This is the daily cycle your team will follow:

Morning (Before Starting Work):

```
# Download any new changes from GitHub
git pull origin main
```

This ensures you're working with the latest version of the code.

Throughout the Day (After Each Feature/Fix):

```
# 1. Check what you changed
git status

# 2. Stage your changes
git add .
# The "." means "add all changed files"
# Or add specific files: git add filename.js

# 3. Commit your changes
git commit -m "Clear description of what you did"

# 4. Upload to GitHub
git push origin main
```

Before Ending Your Session:

```
# Make sure you've pushed everything
git status
# Should say "nothing to commit, working tree clean"
```

Mini Practice Task #2: The Team Assembly

Task: Execute these steps in order as a team.

Step 1 (Repository Owner - let's say Student 1): 1. Create a new private GitHub repository called `fyp-practice-collab` 2. Add all 3 teammates as collaborators 3. Create a local folder with the same name 4. Initialize Git: `git init` 5. Create `README.md` with: "# FYP Practice - Team [Your Team Name]" 6. Commit and push to GitHub

Step 2 (Students 2, 3, and 4): 1. Accept the email invitation 2. Clone the repository to your computers 3. Verify you can see the README file

Step 3 (Each student, one at a time): 1. Create a file named `your-name.txt` (e.g., `akash.txt`) 2. Write inside: "Hello from [Your Name]! My role: [Frontend/Backend/Database/etc.]" 3. Stage, commit, and push:

`bash git add your-name.txt git commit -m "Add introduction from [Your Name]" git push origin main` 4. **IMPORTANT:** Before the next person starts, they must run: `bash git pull origin main`

Step 4 (Everyone): After all 4 files are pushed, everyone runs:

```
git pull origin main
git log --oneline
```

Expected Output: - The repository has 5 commits (1 initial + 4 introductions) - Each student has all 4 `.txt` files on their computer - `git log --oneline` shows all 5 commits

Expected Project State:

```
fyp-practice-collab/
  README.md
  student1.txt
  student2.txt
  student3.txt
  student4.txt
```

Module 3: The Parallel Universe (Working Without Stepping on Toes)

The Problem: "How do we work on different features at the same time without breaking everything?"

Imagine this scenario: - Student 1 is building the login page - Student 2 is designing the database - Student 3 is creating the homepage - Student 4 is writing tests

If everyone commits to the same `main` branch, the project is constantly in a half-working state. Student 1 pushes broken code at 2pm that stops Student 3 from working. Student 2's database changes conflict with Student 4's tests. Chaos.

You need a way to work in isolation, then merge your completed work.

The Branch Mental Model: Parallel Timelines

Think of your repository as a tree:

```
                main (the trunk - stable, working code)
                  |

       |          |          |
  feature-A  feature-B  feature-C  (branches - experimental)
```

1 2 3

Branches are parallel universes for your code: - **`main` branch:** The stable, working version of your project. This should always work—never broken code here. - **Feature branches:** Temporary branches where you build one specific feature. If you mess up, it doesn't affect `main`. - **Merging:** When your feature is done and tested, you combine it back into `main`.

The Rule: Never commit directly to `main`. Always work on a branch.

Working with Branches

Step 1: Understanding Your Current Branch

```
# See all branches and which one you're on
git branch
# The branch with a * is your current branch

# See which branch you're on (alternative)
git status
# The first line says "On branch main" or whatever branch you're on
```

By default, you start on the `main` branch.

Step 2: Create a New Branch Branch Naming Convention: - Good: `feature-login-page`, `fix-database-bug`, `add-payment-gateway` - Bad: `mycode`, `test`, `branch1`, `akash`

```
# Create a new branch and switch to it
git checkout -b feature-login-page
# "checkout" = switch to
# "-b" = create a new branch
# "feature-login-page" = the name of your new branch

# Verify you switched
git branch
# You should see * next to feature-login-page
```

What just happened: You created a parallel timeline. Any commits you make now will ONLY exist on `feature-login-page`, not on `main`.

Step 3: Work on Your Branch

```
# Create a new file for your feature
echo "Login form HTML goes here" > login.html

# Stage and commit (exactly like before!)
git add login.html
git commit -m "Add login form structure"
```

```
# Make more changes
echo "CSS styling for login form" > login.css
git add login.css
git commit -m "Add login form styling"

# Check your history
git log --oneline
```

Step 4: Push Your Branch to GitHub

```
# Push your branch to GitHub
git push origin feature-login-page
# This creates the branch on GitHub too
```

Success looks like:

```
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/your-username/fyp-team-project.git
 * [new branch]      feature-login-page -> feature-login-page
```

If you go to GitHub now, you'll see your branch in the "branches" dropdown.

Step 5: Switch Between Branches

```
# Switch back to main
git checkout main
# Notice: login.html and login.css disappear! (They only exist on the other branch)

# Check your files
ls
# Your login files aren't here

# Switch back to your feature branch
git checkout feature-login-page
# The files reappear!

# Check your files again
ls
# Now you see login.html and login.css
```

This is the magic of branches: Each branch is a separate workspace.

The Branch Workflow for Your Team

For each new feature/task:

```
# 1. Make sure you're on main and it's up to date
git checkout main
```

```
git pull origin main

# 2. Create a new branch for your task
git checkout -b feature-your-feature-name
# Always create new branches FROM main

# 3. Do your work (edit files)

# 4. Commit your changes (can do this multiple times)
git add .
git commit -m "Describe what you did"

# 5. Push your branch to GitHub
git push origin feature-your-feature-name

# 6. (We'll learn how to merge in Module 4)
```

Keeping Your Branch Updated

If you're working on a long feature and your teammates have merged their branches into `main`, you need to update your branch:

```
# You're on your feature branch
git checkout feature-your-feature-name

# Get the latest main branch
git checkout main
git pull origin main

# Go back to your feature branch
git checkout feature-your-feature-name

# Merge the latest main INTO your feature branch
git merge main
# This brings in your teammates' changes
```

If there are no conflicts (files you both changed), it will merge automatically. We'll handle conflicts in Module 5.

Deleting Branches (After They're Merged)

Once your feature is merged into `main`, delete the branch to keep things clean:

```
# Delete local branch
git branch -d feature-login-page
# "-d" means delete
```

```
# Delete the branch on GitHub
git push origin --delete feature-login-page
```

Mini Practice Task #3: Branch Chaos (Controlled)

Task: Each student creates their own feature branch.

Step 1 (Everyone): 1. Make sure you're in the `fyp-practice-collab` folder 2. Get the latest code: `bash git checkout main git pull origin main`

Step 2 (Each student individually): 1. Create a branch named `feature-yourname-section` (e.g., `feature-akash-header`) 2. Create a file named `section-yourname.md` (e.g., `section-akash.md`) 3. Write inside: `# My Section: [Header/Footer/Sidebar/etc.] This section will contain: [brief description] Status: In Progress` 4. Commit your changes 5. Push your branch to GitHub

Commands:

```
git checkout -b feature-yourname-section
# Create your .md file with content
git add section-yourname.md
git commit -m "Add initial structure for [your section]"
git push origin feature-yourname-section
```

Step 3 (Everyone): 1. Go to GitHub and click the "branches" dropdown 2. Verify you see all 4 feature branches

Step 4 (Everyone): Switch between branches to see the magic:

```
# Switch to main
git checkout main
ls  # Your section-yourname.md is NOT here

# Switch to your branch
git checkout feature-yourname-section
ls  # Now it IS here

# Switch to a teammate's branch (replace with actual branch name)
git checkout feature-akash-header
ls  # See Akash's file, not yours!
```

Expected State: - GitHub shows 5 branches: `main` + 4 feature branches - Each feature branch has 1 unique file - `main` branch still only has the original files - Everyone can switch between branches and see different files

Expected Project State on GitHub:

```
Branches:
- main (no changes yet)
```

```
- feature-student1-section (has section-student1.md)
- feature-student2-section (has section-student2.md)
- feature-student3-section (has section-student3.md)
- feature-student4-section (has section-student4.md)
```

Module 4: The Collaboration Merge (Bringing It All Together)

The Problem: "How do we safely combine everyone's work into the main project?"

You've all worked on separate branches. Now you need to combine them. But you can't just throw all the code together—what if there are conflicts? What if someone's code breaks the project? What if you merge too early?

This is why we use Pull Requests and controlled merging.

The Pull Request Mental Model: Code Review Before Merging

Think of a Pull Request (PR) like submitting an assignment:

```
Your Branch (feature-login)
        |
        | "Hey team, I finished the login feature.
        | Please review my code before it goes into main!"
        ↓
    PULL REQUEST (GitHub)
        ↓
    Team Reviews:
      Student 2: "Looks good!"
      Student 3: "Change the button color, otherwise approved"
        ↓
    You fix the feedback
        ↓
    Someone clicks "Merge"
        ↓
    Your code is now in main!
```

A Pull Request (PR) is: - A formal request to merge your branch into another branch (usually `main`) - A place for teammates to review your code - A final check before code becomes "official"

Benefits: - Catch bugs before they reach `main` - Learn from each other's code - Ensure someone else understands your work - Keep `main` stable and working

Creating a Pull Request

Step 1: Push Your Completed Branch to GitHub

```
# Make sure all your work is committed
git status
# Should say "nothing to commit, working tree clean"

# Push your branch
git push origin feature-login-page
```

Step 2: Create the PR on GitHub

1. Go to your repository on GitHub
2. You'll see a yellow banner saying "Your recently pushed branches: feature-login-page" with a button "Compare & pull request"—click it
   - (If you don't see this: Click "Pull requests" tab → "New pull request" → select your branch)
3. Fill in the PR details:
   - **Title:** "Add login page feature" (clear and descriptive)
   - **Description:** Write what you did:

     ```
     ## What I Did
     - Created login.html with form structure
     - Added CSS styling in login.css
     - Tested the form validation

     ## How to Test
     1. Open login.html in browser
     2. Try submitting empty fields (should show error)
     3. Try valid credentials

     ## Screenshots
     (If applicable, add screenshots)
     ```
4. On the right side:
   - **Reviewers:** Select your teammates
   - **Assignees:** Select yourself
5. Click "Create pull request"

Success looks like: You see a PR page with your title, description, and the list of commits and file changes.

Step 3: Team Reviews the PR For Reviewers (your teammates):

1. Go to the "Pull requests" tab on GitHub
2. Click on the PR to review
3. Click on "Files changed" tab to see the code
4. Review the changes:
   - Does the code do what it claims?

- Are there any obvious bugs?
- Is the code readable?
- Does it follow your team's style?

To leave feedback: - Hover over a line of code → Click the `+` icon → Write a comment → "Add single comment" - If requesting changes: "Request changes" before submitting review - If approving: "Approve" before submitting review

Example Comments: - "This looks good!" - "Can you add a comment explaining this function?" - "Should we use `let` instead of `var` here?" - "This might cause an error if the user doesn't enter an email"

Step 4: Respond to Feedback If reviewers requested changes:

```
# Make the changes in your local branch
git checkout feature-login-page

# Edit the files as requested

# Commit the changes
git add .
git commit -m "Address code review feedback"

# Push to the same branch
git push origin feature-login-page
```

The PR automatically updates with your new commits!

Step 5: Merge the PR Once approved (at least 1-2 teammates have reviewed):

1. Go to the PR page on GitHub
2. Click the green "Merge pull request" button
3. Click "Confirm merge"
4. **IMPORTANT:** Click "Delete branch" (the branch on GitHub, not local)

Success looks like: The PR shows "Merged" in purple, and the branch is deleted on GitHub.

Step 6: Update Everyone's Local Main Everyone on the team (including the person who merged) must now run:

```
# Switch to main
git checkout main

# Get the newly merged code
git pull origin main
```

```
# Delete your local feature branch (if you're done with it)
git branch -d feature-login-page
```

Now everyone has the login feature on their main branch!

Direct Merge (Fast Method - Use Carefully)

For very small changes or if you're working alone on a part, you can merge without a PR:

```
# Make sure your feature branch is committed and pushed
git checkout feature-small-fix

# Switch to main
git checkout main

# Merge your feature branch into main
git merge feature-small-fix

# Push the updated main to GitHub
git push origin main
```

When to use direct merge: - Fixing a typo - Updating documentation - Very small, non-critical changes

When to use PR: - Any new feature (ALWAYS) - Bug fixes that change logic - Changes to critical files (database, authentication, etc.)

Your Team's PR Workflow

Establish these rules:

1. **Never push directly to `main`**
   - All work happens on feature branches
2. **Every feature needs a PR**
   - Even if you think it's perfect
3. **Every PR needs at least 1 approval**
   - Ideally 2 people review
4. **Merge frequency:**
   - Merge small PRs frequently (daily)
   - Better to merge 5 small features than 1 giant feature
5. **Pull main often:**
   - Start every work session with `git pull origin main`
   - Update your feature branch if you're working on it for multiple days

Mini Practice Task #4: The Great Merge

Task: Merge all 4 feature branches from Task #3 into `main` using Pull Requests.

Step 1 (Student 1): 1. Make sure your branch is pushed: `bash git checkout feature-yourname-section git push origin feature-yourname-section` 2. Go to GitHub and create a Pull Request for your branch 3. Title: "Add [your section] structure" 4. Description: Brief explanation of what's in your file 5. Assign Students 2 and 3 as reviewers

Step 2 (Students 2 and 3): 1. Go to the PR, click "Files changed" 2. Review the file 3. Leave at least one comment (even if just "Looks good! ") 4. Click "Review changes" → "Approve" → "Submit review"

Step 3 (Student 1): 1. Once you have 2 approvals, click "Merge pull request" 2. Click "Delete branch" on GitHub

Step 4 (Everyone):

```
git checkout main
git pull origin main
```

Verify Student 1's file is now in `main`.

Step 5-7 (Repeat for Students 2, 3, and 4): Each student creates a PR, gets reviews, merges, and everyone pulls.

Expected Final State: - `main` branch has all 4 `section-*.md` files - All feature branches are deleted on GitHub - Everyone's local `main` branch is identical - GitHub shows 8-9 closed Pull Requests (4 merge PRs + maybe some from setup)

Expected Project State:

```
fyp-practice-collab/ (main branch)
  README.md
  student1.txt
  student2.txt
  student3.txt
  student4.txt
  section-student1.md  ← NEW
  section-student2.md  ← NEW
  section-student3.md  ← NEW
  section-student4.md  ← NEW
```

Module 5: The "Oh No!" Handbook (Fixing Common Mistakes)

The Reality: "We're going to mess up. Let's learn how to fix it."

You will make mistakes. Everyone does. Git is incredibly powerful, which means there are many ways to get into confusing situations. This module covers the most common "OH NO" moments and how to escape them.

Golden Rule: Git almost never deletes data permanently. There's usually a way to undo.

Scenario 1: "I committed to the wrong branch!"

Problem: You made commits on `main` instead of a feature branch.

```
# You're on main (oops)
git branch
# * main

# You made changes and committed
git commit -m "Add new feature"
```

Fix:

```
# 1. Create a new branch from your current position
git branch feature-correct-branch
# This creates a branch with your commits

# 2. Switch to the new branch
git checkout feature-correct-branch

# 3. Go back to main
git checkout main

# 4. Reset main to match GitHub (removes your local commits)
git reset --hard origin/main
# --hard means "throw away my local changes"
# BE CAREFUL: This deletes uncommitted work!

# 5. Go back to your feature branch and continue working
git checkout feature-correct-branch
```

Verify:

```
git checkout main
git log --oneline  # Should NOT show your commit

git checkout feature-correct-branch
git log --oneline  # SHOULD show your commit
```

Scenario 2: "My commit message is terrible!"

Problem: You wrote `git commit -m "asdfjkl"` and need to fix it.

Fix (if you haven't pushed yet):

```
# Change the message of your most recent commit
git commit --amend -m "Add user authentication feature"
```

If you already pushed: Don't amend! Instead, push a new commit:

```
# Just make a new commit with clarification
git commit --allow-empty -m "Previous commit added user authentication"
git push origin feature-your-branch
```

Scenario 3: "I want to undo my last commit!"

Problem: You committed something you shouldn't have.

Fix (if you HAVEN'T pushed):

```
# Undo the commit but keep your changes
git reset --soft HEAD~1
# HEAD~1 means "the commit before the current one"
# --soft means "keep my file changes, just undo the commit"

# Your changes are now unstaged
git status  # Shows your files as modified

# You can now edit and commit again
```

Fix (if you HAVE pushed):

Don't rewrite history! Instead, create a new commit that undoes the changes:

```
# Revert the last commit (creates a new commit that undoes it)
git revert HEAD
# This opens an editor for the revert message, save and close

# Push the revert
git push origin feature-your-branch
```

Scenario 4: "I have merge conflicts!"

Problem: You tried to merge, and Git says:

```
CONFLICT (content): Merge conflict in homepage.html
Automatic merge failed; fix conflicts and then commit the result.
```

What happened: You and a teammate both changed the same lines in the same file. Git doesn't know which version to keep.

Fix:

```
# 1. See which files have conflicts
git status
# Files under "both modified" have conflicts

# 2. Open the conflicted file
# You'll see something like this:

<<<<<<< HEAD
<h1>Welcome to Our App</h1>
=======
<h1>Welcome to the Best App Ever</h1>
>>>>>>> feature-other-branch
```

How to read this: - `<<<<<<< HEAD`: Your version (the current branch) - `=======`: Separator - `>>>>>>> feature-other-branch`: Their version (the branch you're merging)

3. Manually edit the file to what it SHOULD be:

Remove the markers and choose what to keep:

```
<!-- Option 1: Keep yours -->
<h1>Welcome to Our App</h1>

<!-- Option 2: Keep theirs -->
<h1>Welcome to the Best App Ever</h1>

<!-- Option 3: Combine both -->
<h1>Welcome to Our App - The Best Ever</h1>
```

4. Save the file, then:

```
# Mark the conflict as resolved
git add homepage.html

# Complete the merge
git commit -m "Merge feature-other-branch and resolve conflicts"

# Push the merge
git push origin main
```

Pro Tip: Communicate with your teammate! Ask them why they made their change before deciding what to keep.

Scenario 5: "I accidentally deleted a file!"

Problem: You deleted a file and committed it, but now you need it back.

Fix:

```
# Find the commit where the file still existed
git log --oneline --all -- path/to/deleted-file.js

# Restore the file from a specific commit
git checkout <commit-hash> -- path/to/deleted-file.js
# Replace <commit-hash> with the actual hash from the log

# The file is back! Commit it
git add path/to/deleted-file.js
git commit -m "Restore accidentally deleted file"
```

Scenario 6: "Help! I don't know what I did!"

Problem: Your repository is in a weird state and you just want to start fresh.

Nuclear Option (be careful!):

```
# See the current state
git status

# If you have uncommitted changes you want to keep:
git stash
# This saves your changes temporarily

# Reset your branch to match GitHub exactly
git fetch origin
git reset --hard origin/main
# This throws away ALL local changes and matches GitHub

# If you stashed changes and want them back:
git stash pop
```

Scenario 7: "I pushed to the wrong branch!"

Problem: You pushed your feature branch commits to `main` by accident.

Fix (DANGEROUS - only if no one else has pulled):

```
# Find the commit hash BEFORE your mistake
git log --oneline

# Reset main to that commit
git reset --hard <good-commit-hash>
```

```
# Force push (overwrites GitHub)
git push --force origin main

# NEVER use --force if teammates have pulled!
```

Safer Fix (if others have pulled):

```
# Revert each bad commit
git revert <commit-hash1>
git revert <commit-hash2>

# Push the reverts
git push origin main
```

Scenario 8: "My pull doesn't work!"

Problem: You run `git pull` and see:

```
error: Your local changes to the following files would be overwritten by merge:
    homepage.html
Please commit your changes or stash them before you merge.
```

Fix:

```
# Option 1: Commit your changes first
git add .
git commit -m "WIP: Work in progress"
git pull origin main

# Option 2: Stash your changes
git stash  # Temporarily saves your changes
git pull origin main
git stash pop  # Brings your changes back
```

Emergency Commands Reference

```
# See what's going on
git status
git log --oneline --graph --all

# Undo uncommitted changes to a file
git checkout -- filename.js

# Undo ALL uncommitted changes (DESTRUCTIVE!)
git reset --hard
```

```
# Undo last commit but keep changes
git reset --soft HEAD~1

# Undo last commit and delete changes (VERY DESTRUCTIVE!)
git reset --hard HEAD~1

# Save current work temporarily
git stash
git stash list  # See stashed work
git stash pop   # Restore stashed work

# See who changed what line in a file
git blame filename.js

# Find a lost commit (Git's recycling bin)
git reflog
```

Mini Practice Task #5: Controlled Chaos

Task: Practice handling conflicts safely.

Step 1 (Students 1 and 2):

Both students do this at the SAME TIME:

```
# Both: Make sure main is up to date
git checkout main
git pull origin main

# Both: Create a branch
git checkout -b conflict-practice-yourname

# Both: Edit README.md
# Change the SAME LINE to DIFFERENT things:
# Student 1: "# FYP Practice - Amazing Team"
# Student 2: "# FYP Practice - Awesome Collaboration"

# Both: Commit
git add README.md
git commit -m "Update project title"
```

Step 2 (Student 1 first):

```
# Student 1: Push and merge first
git push origin conflict-practice-student1

# Student 1: Create PR and merge it on GitHub
```

```
# Everyone: Pull the changes
git checkout main
git pull origin main
```

Step 3 (Student 2):

```
# Student 2: Try to merge
git checkout main
git pull origin main  # Gets Student 1's change

git checkout conflict-practice-student2
git merge main  # CONFLICT!

# You'll see:
# CONFLICT (content): Merge conflict in README.md
```

Step 4 (Student 2):

1. Open README.md
2. You'll see conflict markers
3. Decide which title to keep (or combine them)
4. Remove the markers
5. Save the file

```
# Resolve the conflict
git add README.md
git commit -m "Merge main and resolve title conflict"

# Push and create PR
git push origin conflict-practice-student2
```

Step 5 (Everyone): Review Student 2's PR, approve, merge, and pull.

Success: You've successfully resolved a real merge conflict!

Final Module: Your FYP Day-to-Day Workflow

The Daily Ritual (Every Team Member)

Every morning before coding:

```
cd fyp-team-project
git checkout main
git pull origin main
git checkout -b feature-todays-task
```

Throughout the day (after completing a small piece):

```
git status
git add .
git commit -m "Clear description"
```

End of day (or when feature is done):

```
git push origin feature-todays-task
# Go to GitHub, create PR, get reviews, merge
```

After your PR is merged:

```
git checkout main
git pull origin main
git branch -d feature-todays-task
```

Project Organization Best Practices

File Structure Example:

```
fyp-team-project/
  README.md                 # Project overview
  docs/                     # Documentation
      setup.md
      api-reference.md
  src/                      # Source code
      frontend/
      backend/
      database/
  tests/                    # Test files
  .gitignore               # Files to not track
```

Create a `.gitignore` file to exclude files you don't want in Git:

```
# In your project root
cat > .gitignore << EOL
# Dependencies
node_modules/
venv/

# Environment variables
.env

# IDE settings
.vscode/
.idea/

# OS files
.DS_Store
```

```
Thumbs.db

# Build outputs
dist/
build/
*.pyc
EOL

git add .gitignore
git commit -m "Add gitignore file"
git push origin main
```

Communication Protocol

1. Before Starting a Task: - Announce in your team chat: "I'm working on [feature X]" - Create a branch: `feature-[feature-name]`

2. While Working: - Commit often (every 30-60 minutes of work) - Push to your branch at least once per day

3. When Done: - Create a PR with clear description - Tag reviewers - Wait for approval (don't merge your own PR)

4. During Code Review: - Be kind: "Consider changing X" not "This is wrong" - Explain WHY you suggest changes - Approve quickly (don't leave teammates waiting)

5. After Merging: - Announce in chat: "[Feature X] is merged into main" - Everyone pulls the latest main

---

Commit Message Style Guide

Format:

```
<type>: <short summary>

<detailed explanation (optional)>
```

Types: - `feat`: New feature - `fix`: Bug fix - `docs`: Documentation only - `style`: Formatting, missing semicolons - `refactor`: Code restructuring - `test`: Adding tests - `chore`: Maintenance tasks

Examples:

```
git commit -m "feat: Add user registration form"

git commit -m "fix: Resolve database connection timeout"
```

```
git commit -m "docs: Update API endpoints in README"

git commit -m "refactor: Simplify login validation logic"
```

Weekly Maintenance (Assign to One Person)

Every Friday afternoon:

1. **Review open PRs:** Close stale PRs, remind teammates
2. **Clean up branches:** Delete merged branches

   ```
   # Delete all merged branches
   git branch --merged main | grep -v "main" | xargs git branch -d
   ```

3. **Update documentation:** Keep README and docs current
4. **Run tests:** Make sure main branch works

Emergency: "Main is Broken!"

If someone merges bad code and `main` doesn't work:

Step 1: Identify the bad commit

```
git checkout main
git log --oneline
# Find the commit that broke things
```

Step 2: Revert it

```
git revert <bad-commit-hash>
git push origin main
```

Step 3: Notify the team Post in chat: "Main was broken by [commit X]. Reverted. Please pull."

Everyone pulls:

```
git checkout main
git pull origin main
```

Conclusion: You're Ready!

You've learned:

Module 1: Git basics—commits, staging, local version control
Module 2: GitHub setup—remotes, cloning, pushing, pulling
Module 3: Branching—parallel work without conflicts
Module 4: Pull Requests—code review and safe merging
Module 5: Troubleshooting—fixing common mistakes

Your Cheat Sheet (Print This!)

```
# Daily Start
git checkout main
git pull origin main
git checkout -b feature-my-task

# While Working
git status
git add .
git commit -m "description"

# When Done
git push origin feature-my-task
# Create PR on GitHub

# After PR Merged
git checkout main
git pull origin main
git branch -d feature-my-task

# Emergency
git status                    # What's happening?
git stash                     # Save work temporarily
git reset --hard origin/main  # Nuclear reset
```

Before Your First Real Coding Session

1. **Set up your repository:**
   - One person creates the GitHub repo
   - Add all teammates as collaborators
   - Everyone clones it
2. **Create your project structure:**
   - Add folders for frontend, backend, docs
   - Create README.md with project description
   - Add .gitignore
3. **Test the workflow:**
   - Each person creates a test branch
   - Makes a small change
   - Creates a PR

- Everyone reviews and merges

4. **Establish team rules:**
   - Never commit to main directly
   - Every PR needs 1-2 approvals
   - Use clear branch and commit names
   - Communicate in your team chat

Resources for Further Learning

- **Official Git Documentation:** https://git-scm.com/doc
- **GitHub Guides:** https://guides.github.com
- **Interactive Git Tutorial:** https://learngitbranching.js.org
- **Git Cheat Sheet:** https://education.github.com/git-cheat-sheet-education.pdf

Final Advice

1. **Start simple:** Don't try to learn advanced Git. Master the basics first.
2. **Commit often:** Small commits are easier to understand and revert.
3. **Communicate constantly:** Tell your team what you're working on.
4. **Don't panic:** Git is forgiving. Almost everything can be undone.
5. **Ask for help:** If you're stuck for more than 30 minutes, ask a teammate or search online.

Remember: Every professional developer uses Git. You're learning an essential skill that will serve you throughout your career. The confusion you feel now is temporary—with practice, these commands will become second nature.

Now go build something amazing!

One Last Thing: Bookmark this guide. You'll refer back to it constantly during your FYP. That's not a sign of weakness—it's smart. Even experienced developers look up Git commands regularly.

Good luck with your Final Year Project!