**Laboratory 3: Project Setup and Version Control**

**Part 1: Setting Up Git and GitHub**

**A. Install Git**

1. **Download Git**: Visit [Git’s official website](https://git-scm.com/downloads) and download the Git installer for your operating system.
2. **Install Git**: Run the downloaded installer and follow the installation prompts. Make sure to accept the default options unless you have specific preferences.

**B. Create GitHub Accounts**

1. **Sign Up**: Go to [GitHub’s website](https://github.com/) and sign up for a new account.
2. **Verify Email**: Check your email inbox for a verification email from GitHub and follow the instructions to verify your account.

**C. Create and Set Up a Repository on GitHub**

1. **Create a New Repository**: One team member should log into GitHub, click on the “+” icon in the upper-right corner, and select “New repository”.
2. **Repository Details**: Fill in the repository name, add a description (optional), and choose if the repository should be public or private. No need to initialize the repository with a README file at the moment.
3. **Add Collaborators**: In the repository, go to “Settings” > “Manage access” > “Invite collaborators”. Enter the GitHub usernames of your team members and send the invitations.

**D. Initialize and Push Local Project**

1. **Create Local Project Folder**:
2. mkdir project-name
3. cd project-name
4. **Initialize Git Repository**:
5. git init
6. **Add README File** (optional):
7. echo "# Project Title" > README.md
8. git add README.md
9. git commit -m "Initial commit with README"
10. **Link Local Repository to GitHub**:
11. git remote add origin https://github.com/username/repository-name.git
12. **Push to GitHub**:
13. git push -u origin main

**Understanding git remote add origin https://github.com/username/repository-name.git**

* **git remote add origin**: This command is used to add a new remote to your local Git repository.
  + **git remote**: This part of the command deals with the remote repositories. Remote repositories are versions of your project that are hosted on the internet or network somewhere.
  + **add**: This tells Git you want to add a new remote.
  + **origin**: This is the conventional name for the primary remote repository. You can name your remote anything, but origin is the standard and most commonly used name in Git.
  + **https://github.com/username/repository-name.git**: This is the URL of the remote repository on GitHub. Replace username with your GitHub username and repository-name with the name of your repository. This URL is where Git will push to and pull from.
* **Purpose**: The purpose of this command is to link your local repository with the remote repository on GitHub. After executing this command, origin will be a reference to the GitHub repository, and you can push and pull changes to and from the remote repository.

**Understanding git push -u origin main**

* **git push**: This command is used to upload the content of your local repository to a remote repository.
  + **-u** or **--set-upstream**: This flag sets the upstream for your branch. Upstream refers to the default remote branch that Git will push to or pull from. When you use -u, Git not only pushes your current branch to the remote branch but also sets this remote branch as the upstream for your local branch.
  + **origin**: As mentioned earlier, this is the name of your remote repository (in this case, on GitHub).
  + **main**: This is the name of the branch you are pushing. In this context, main is the default branch name for most repositories.
* **Purpose**: This command pushes your main branch to the origin (GitHub in this case) and sets it as the default branch to track against the remote main branch.
* **Future Pushes**: After running git push -u origin main, your local main branch will “remember” the remote branch to push to. So, in future, you only need to use git push when pushing changes from your local main branch. You don’t have to specify -u origin main again for this branch.

**E. Clone the Repository for Other Team Members**

1. **Clone the Repository**:  
   Each team member (other than the one who created the repo) should open their terminal or Git Bash and run:
2. git clone https://github.com/username/repository-name.git
3. **Navigate to Project Directory**:
4. cd repository-name

**Part 2: Practice making changes**

Below is a set of tasks. Each task is designed to be completed by a different team member. Before you begin your task, read the General Procedure for Each Task at the bottom of step 2.

**Task 1: Create a “meeting-minutes” Folder and Add a Document**

* **Student Assigned**: Student 1
* **Task**: In your working copy, create a new folder named meeting-minutes. Inside this folder, create a document named 2024-26-01-notes.docs and add some text to it.

**Task 2: Update the README File**

* **Student Assigned**: Student 2
* **Task**: Edit the README.md file in the root of the repository to include a paragraph describing the game the team is working on.

**Task 3: Create Essential Project Directories**

* **Student Assigned**: Student 3
* **Task**: Create the following directories: src, test, docs, assets, data.
  + **src**: This directory will contain the source code of the project.
  + **test**: For storing test scripts and test-related resources.
  + **docs**: To keep project documentation.
  + **assets**: For non-code resources like images or sound files.
  + **data**: To store data files used or generated by the project.

**Task 4: Add a Python Script**

* **Student Assigned**: Student 4
* **Task**: In the src directory, add some basic Python code, for example, a simple “Hello World” program.

**Task 5: Modify the Python Script (Potential for Merge Conflict)**

* **Student Assigned**: Students 5 and 6 (Simultaneously)
* **Task**: Both students modify the same Python file created by Student 4. Students should make different changes to the same part of the file.
* **Purpose**: This task is designed to potentially create a merge conflict, which is a common scenario in team-based development. Resolving merge conflicts is a crucial skill in using Git.

**General Procedure for Each Task:**

1. **Pull the Latest Changes**: Start by pulling the latest changes from the repository to ensure you’re working on the most recent version.
2. git pull origin main
3. **Make the Changes**: Perform the tasks as described above.
4. **Stage the Changes**: Add the changes to the staging area.
5. git add .
6. **Commit the Changes**: Commit the changes with an appropriate commit message.
7. git commit -m "Descriptive message about the change"
8. **Push the Changes**: Push the commit to the GitHub repository.
9. git push origin main
10. **Handling Merge Conflicts**: If a merge conflict arises (likely in Task 5), Git will notify the student. They will then need to manually resolve the conflict in the file, stage the resolved file, and complete the push. Read below for more details.

**When a Merge Conflict Occurs**

**Git Notifies of a Conflict**: When a team member tries to pull or push changes where the same lines of code have been altered by different team members, Git will stop and mark the file as having a conflict.

**Resolving the Merge Conflict**

a. **Identify Conflicted Files**: First, identify which files have conflicts. Git will mark these files in the output of the git status command.

b. **Edit the Files**: Open the conflicted files in a text editor. Git will have marked the areas with conflicts with special markers:

* <<<<<<< HEAD: Everything after this marker and before the ======= marker represents your changes (the changes in the current branch).
* =======: Separates your changes from the changes made in the other branch.
* >>>>>>> [other branch name]: Everything after this marker represents the changes from the other branch that are conflicting with your changes.

c. **Resolve the Conflicts**: Decide what the final content should be. You might choose to keep your changes, the other person’s changes, or combine both. Remove the conflict markers (<<<<<<<, =======, >>>>>>>) and edit the content to how you want it to appear in the final merge.

d. **Save the File**: After resolving the conflicts in the file, save it.

**Completing the Merge**

a. **Stage the Resolved File**:

git add [file]

Replace [file] with the name of the file you resolved.

b. **Commit the Resolution**:

git commit -m "Resolved merge conflict in [file]"

This commit will complete the merge process. You do not need to specify the file name in the commit message, but it can be helpful for documentation purposes.

c. **Push the Changes**:

git push origin main

**Final Steps**

* **Check Status**: Use git status to ensure all conflicts are resolved and all changes are committed.
* **Syncing with Remote**: After resolving conflicts and pushing, it’s a good practice for other team members to pull the latest changes to ensure everyone is working with the updated codebase.

**Part 3: Working with Branches and Pull Requests**

**Introduction to Branches in Git**

**What are Branches?**

* **Branches in Git** represent an independent line of development in your project. You can think of them as a way to request a brand new working directory, staging area, and project history.

**Why Use Branches?**

* **Isolation**: Branches allow you to work on different features, fixes, or experiments in isolation, without affecting the main codebase (usually the main or master branch).
* **Parallel Development**: Multiple team members can work simultaneously on different aspects of a project without interfering with each other. Each member works on their own branch.
* **Safety**: The main codebase remains safe from untested or unstable changes. Features are only merged into the main branch once they are ready and reviewed.

**Common Branching Strategies**

* **Feature Branches**: For adding new features or components to your project.
* **Bugfix/Branches**: Specifically for fixing bugs in your project.
* **Release Branches**: Used for preparing a project for a new release.

**Lifecycle of a Branch**

1. **Creation**: You create a branch when you want to add a new feature or fix a bug.
2. **Development**: You make your changes in this branch, committing them as you go.
3. **Pull Request**: Once the work is complete, you might make a pull request to merge your changes into the main branch.
4. **Merge**: After review and approval, your branch is merged into the main branch.
5. **Deletion**: Optionally, you can delete the branch after its merge, especially if it was for a specific task.

**Creating and Checking Out a Branch**

* To create and switch to a new branch:
* git checkout -b new-branch-name
* To only create a new branch without switching to it:
* git branch new-branch-name
* To switch between branches:
* git checkout branch-name

Certainly! When working with multiple branches in Git, it’s important for team members to understand how to switch between branches and what that means for their working copy. Here’s an explanation that you can include in your instructions:

**Moving Between Branches in Git**

**Understanding Working Copy**

* **Working Copy**: Your working copy is the local directory where you have your project files. It changes based on the branch you’re currently checked out in. There’s only one working copy, but it reflects the state of the branch you’re working on.

**Switching Branches**

1. **Check Current Branch**: First, it’s good practice to check which branch you’re currently on:
2. git branch

This command lists all branches in your local repository and marks the current branch with an asterisk (\*).

1. **Switch Branches**: To switch from one branch to another, use:
2. git checkout branch-name

Replace branch-name with the name of the branch you want to switch to.

1. **What Happens to the Working Copy**:
   * When you switch branches, the files in your working directory change to reflect the state of the new branch.
   * Any committed changes specific to the branch you’re moving from are saved in that branch.
   * Uncommitted changes will be carried over to the new branch unless they conflict with that branch’s files. If there’s a conflict, Git won’t let you switch branches until you either commit or stash your changes.

**Example Scenario:**

* **Developer on feature-branch-1**: Imagine a developer is working on a feature in feature-branch-1. They have made some changes and committed them to this branch.
* **Switching to main**: Now, if the developer wants to switch to the main branch, they use:
* git checkout main
* **Changes in Working Copy**: After switching, their working copy will represent the state of the main branch. The changes they made in feature-branch-1 won’t be visible until they switch back to that branch.

**Handling Uncommitted Changes:**

* If the developer has uncommitted changes in feature-branch-1 and tries to switch to main, they have two options:
  1. **Commit the Changes**: Commit them to feature-branch-1 before switching.
  2. **Stash the Changes**: Use git stash to temporarily store the changes, switch branches, and then retrieve the changes later with git stash pop after switching back.

**Best Practices:**

* **Frequent Commits**: It’s good practice to commit changes frequently. This way, you can switch between branches without carrying over a lot of uncommitted changes.
* **Clean Working Directory**: Try to keep your working directory clean (without uncommitted changes) before switching branches to avoid confusion.

**Your task: Feature Development Using Branches and Pull Requests**

Branching and pull requests are fundamental to managing features, fixes, and updates in a structured and reviewable manner.

**Objective**

Each team member will create a new feature or make a significant update to the project in a separate branch. They will then submit their changes as a pull request on GitHub for review by other team members.

**Step-by-Step Guide**

1. **Create a New Branch**: Each team member should create a new branch for their task. Branches should be named meaningfully, often based on the feature or fix they’re working on.
2. git checkout -b feature-branch-name

Replace feature-branch-name with a relevant name for the branch.

1. **Develop the Feature**: Make changes or add new features in the branch. This could involve adding new files, editing existing ones, etc. For this test scenario, you can ddd files (to be deleted later) and add some text into them.
2. **Commit Changes in the Branch**: As you work, you should regularly commit your changes changes.
3. git add .
4. git commit -m "Describe the changes made"

After you commit, you can switch to the main branch to see how that changes your working copy, then switch back to your feature branch.

1. **Push the Branch to GitHub**: Once the feature is complete, push the branch to the remote repository.
2. git push -u origin feature-branch-name
3. **Create a Pull Request (PR)**:
   * Go to the GitHub repository page.
   * GitHub often shows a quick option to create a pull request for new branches right on the main page of the repository. If not, you can go to the “Pull requests” tab and click “New pull request”.
   * Select the feature-branch-name as the source and main (or the relevant branch) as the destination for the merge.
   * Fill in the details of the pull request: title, a brief description of the changes, and any other relevant information.
   * Create the pull request.
4. **Review and Merge the Pull Request**:
   * Other team members review the code, discuss it, and suggest changes if necessary.
   * Once the pull request is approved by the team or the designated reviewers, it can be merged into the main branch.
   * Ideally, the team member who created the pull request shouldn’t be the one to merge it. This encourages peer review and collaboration.
5. **Delete the Branch Post-Merge** (optional but recommended for cleanliness):
   * After the pull request is merged, the feature branch can be deleted to keep the repository tidy.
   * GitHub provides an option to delete the branch immediately after merging the PR.
6. **Pull the Latest Changes**:
   * Once the feature branch is merged into main, all team members should update their local main branch.
7. git checkout main
8. git pull origin main

**Why this is important in a collaborative project**

* **Branching**: learn to work on features in isolation, reducing the risk of disrupting the main codebase.
* **Pull Requests**: They facilitate code reviews, discussions, and provide a history of changes.
* **Collaboration and Communication**: This workflow encourages team communication and collaboration through code review.