Week 3

Modified State: This is when you make changes to files within your repository. Git recognizes that the files have changed, but it doesn't track these changes yet.

Staged State: To start tracking changes, you need to move the modified files to the staged area. Staging allows you to prepare specific changes for commit, and it acts as a sort of pre-commit area. You can choose which changes to include in the next commit.

Committed State: Committing is like saving a snapshot of the current state of your project. When you commit changes, Git records them permanently in the repository's history. Each commit has a unique identifier (hash) and includes a message describing the changes.

Remote Repository: In addition to your local repository, Git also allows you to work with remote repositories, typically hosted on platforms like GitHub. You can push your local commits to the remote repository, and others can fetch, check out, or merge those changes.

Checking Repository Status: You run git status to check the status of your Git repository. It shows that you're on the main branch, up to date with the remote repository, and your working tree is clean.

Adding a New File: You create a new file named test.txt using the touch command and check the status again. Git recognizes it as an untracked file.

Staging the File: You use the git add test.txt command to stage the file, indicating that you want to include it in the next commit.

Checking Status After Staging: After staging, you run git status again, and it shows the file as staged, ready to be committed.

Made by: Omar Madany

Week 3

Unstaging a File: You demonstrate how to unstage a file using git restore --stage test.txt, which puts the file back in an untracked state.

Staging Again: You add the file to the staging area again with git add test.txt.

Committing Changes: You commit the staged changes using git commit -m "adding a new file for testing". This creates a commit with the specified message.

Checking Status After Committing: Finally, you use git status once more, and it reports that there's nothing to commit, and the working tree is clean.

Creating a New Branch: To create a new branch, you use the git checkout -b command followed by the branch name you want to create. In your example, you create a branch called "feature/lesson.

Adding and Committing Changes: You create a new file called "test2.txt" using the touch test2.txt command.

To add this new file to the staging area, you use the git add command: git add test2.txt.

After staging the changes, you commit them using the git commit command

with a commit message: git commit -m "Adding a new file for testing."

The commit captures the changes you've made in your branch.

Pushing Changes to the Remote Repository: To push the changes to the remote repository (GitHub), you use the git push command with the -u flag and specify the remote repository and branch name. In your case: git push -u origin feature/lesson.

This command sends your branch and commits to the remote repository on GitHub.

Made by: Omar Madany

Week 3

Creating a Pull Request (PR): You switch to the GitHub web interface and see that GitHub recognizes the new branch.

You click on the "Compare and pull request" button to create a pull request.

The pull request allows you to compare the changes in your branch (feature/lesson) with the main branch.

You provide details about the changes in the pull request, including the commit message.

Review and Merge: Team members can review your pull request and provide feedback or approve it.

Once approved, you can merge the changes into the main branch through the GitHub interface.

Cleaning Up: After merging, you have the option to delete the feature branch if you no longer need it.

Updating the Local Main Branch: Back in your local environment, you can switch to the main branch using git checkout main.

You run git pull to retrieve the latest changes, including the merged changes from the feature branch.

Now, your local main branch is up to date with the changes made in the feature branch.

Local: Local refers to your local machine, whether it's a laptop, desktop, or any device where you have a Git repository.

Your local repository is accessible only to you, and it's where you make changes, commits, and work on your code.

You can work offline in your local repository and commit changes when you're ready.

Changes made in the local repository are not automatically visible to others; you need to push them to a remote repository for others to see.

Made by: Omar Madany

Week 3

Remote: Remote refers to any other Git repository that is not on your local machine.

Remote repositories can be hosted on platforms like GitHub or other servers.

When you want to collaborate with others or back up your code, you interact with remote repositories.

You can clone a remote repository to create a copy on your local machine, allowing you to work on the code locally.

After making changes locally, you can push those changes to the remote repository so that others can access them.

To see the changes made by others in the remote repository, you can pull those changes into your local repository.

Workflow Example: You demonstrated the process of creating a new local repository using git init.

You showed how to set up a connection between a local repository and a remote repository on GitHub using git remote add origin <URL>.

You explained that pulling changes from the remote repository using git pull updates your local repository with the latest changes from the remote.

You mentioned that branches play a role in tracking specific lines of development and that you can use git checkout <branch> to set up a local branch that tracks a remote branch.

---

git push: git push is used to upload your local changes to a remote repository, typically hosted on a platform like GitHub.

Before pushing, it's essential to check which branch you're currently on using git status or git branch.

You specify the remote repository (often referred to as "origin") and the branch to which you want to push your changes. For example, git push origin main.

When pushing, you may be prompted for your login information, especially if you are using HTTPS to connect to the remote repository.

After a successful push, your local changes are now available in the remote repository, making them accessible to other team members.

Made by: Omar Madany

Week 3

git pull: git pull is used to retrieve changes from a remote repository and apply them to your local repository.

It's a crucial step to perform before making any new changes locally to ensure you're working with the latest code.

Running git pull fetches changes from the remote repository and merges them into your local branch.

If there are no conflicts between your local and remote changes, Git will perform an automatic merge.

After a successful pull, your local repository is up to date with the latest changes from the remote.

Workflow Example: You demonstrated how to use git push to upload your local changes to the remote repository.

You emphasized the importance of performing a git pull before making new changes to ensure you have the most recent code.

You showed an example of making changes directly on the GitHub website, committing them to the main branch, and then using git pull to update your local repository with those changes.

HEAD Pointer Basics: The HEAD pointer is essentially a reference to the latest commit in the currently checked-out branch. It tells Git where your working directory is in the repository's history.

Viewing the HEAD Reference: You can directly view the contents of the HEAD reference file by running the following command:

// cat .git/HEAD

This will show you the path to the branch or commit that HEAD is currently pointing to.

Switching Branches: When you switch to a different branch using git checkout, the HEAD pointer is updated to point to the new branch. For example:

//git checkout feature/testing

This command moves the HEAD pointer to the feature/testing branch.

Week 3

Commit Changes: As you make changes and commit them to the repository, the HEAD pointer moves forward to the latest commit in the currently checked-out branch. The HEAD always points to the tip of the branch.

Tracking Commits: The HEAD pointer is updated automatically by Git as you create new commits. It always reflects the latest commit you are working on.

Understanding Commit IDs: Each commit in Git is identified by a unique commit ID (usually a long hash). When you make a new commit, the HEAD pointer is updated to point to the new commit's ID.

Resetting HEAD: You can use commands like git reset to move the HEAD pointer to a different commit. This is useful for undoing changes or moving to a previous state of the repository.

Detached HEAD: In some cases, you might find yourself in a "detached HEAD" state. This means HEAD is pointing directly to a commit (not a branch). Be cautious in this state, as changes won't be associated with a branch.

Moving Between Branches: You can freely move between branches using git checkout. This changes which branch HEAD points to, effectively switching your working context.

Compare Working Directory with the Last Commit: Use **git diff** without any additional arguments to compare the changes in your working directory with the last commit.

Compare Staging Area with the Last Commit: Use **git diff --staged** to compare the changes in the staging area (changes added with git add) with the last commit.

Compare a File with the Last Commit: Use **git diff <file>** to compare a specific file in your working directory with the last commit.

Compare Between Two Commits: Use **git diff <commit1> <commit2>** to compare changes between two specific commits. Replace <commit1> and <commit2> with commit hashes or references.

Made by: Omar Madany

Week 3

Compare Between Branches: Use **git diff <branch1>..<branch2>** to compare changes between two branches. Replace <branch1> and <branch2> with branch names or references.

Compare with the HEAD (Last Commit): Use **git diff HEAD** to compare the changes in your working directory with the last commit (HEAD).

Compare with External Files: You can also use git diff to compare a file in your repository with an external file not in the repository.

Basic Usage: To use **git blame**, open your terminal and navigate to the Git repository where the file is located.

Run the following command to see the history of changes for a specific file (replace <filename> with the actual file name)**: git blame <filename>**

Understanding the Output: The git blame output includes several columns for each line of the file:

Commit ID: This is a reference to the specific commit where the change occurred.

Author: The name of the developer who made the change.

Timestamp: The date and time when the change was committed.

Line Number: Indicates the line number in the file where the change occurred.

Code: The actual code or content that was added or modified.

Navigating through the Output: Use the arrow keys or scroll through the output to view all changes.

To exit the git blame view, press the Q key.

Limiting the Output: You can limit the output to a specific range of lines using the -L option. For example, to see changes from line 5 to 15:

// git blame -L 5,15 <filename>

Made by: Omar Madany

Week 3

Customizing Output Format: You can customize the format of the output using the -L option. For example, to show the full commit hash, use -L %H. You can also customize the date format and include email addresses. Refer to the documentation for more formatting options.

Viewing Detailed Changes: If you want to see the detailed changes made in a specific commit, you can combine git blame with git log. First, run git blame to find the commit ID of interest. Then, use **git log -p <commit-id>** to view the detailed changes for that commit.

Using git blame with Different Files and Branches: You can use git blame with different files and branches by specifying the file path and, if needed, the branch name or commit hash.

Creating a Fork: Forking starts with an existing Git repository hosted on a platform like GitHub. To create a fork, you go to the repository you want to fork and click the "Fork" button. This action creates a copy of the original repository in your GitHub account.

Working on Your Fork: Once you have forked a repository, you can clone it to your local machine using Git. This creates a local copy of your fork that you can work on.

You can make changes, add new features, fix bugs, or do whatever you want in your forked repository without affecting the original project.

Pushing Changes: After making changes in your local fork, you commit those changes to your fork's Git history using git commit and git push commands. This updates your fork on the Git hosting platform.

Creating Pull Requests (PRs): If you want to contribute your changes back to the original repository, you typically create a Pull Request (PR). A PR is a request to merge your changes into the original project.

Made by: Omar Madany

Week 3

The maintainers of the original repository can review your changes, provide feedback, and decide whether to accept or reject the PR.

Keeping Your Fork Updated: While you're working in your fork, the original repository might receive new updates, bug fixes, or improvements. It's a good practice to keep your fork up-to-date with the latest changes from the original repository.

You can do this by adding the original repository as a remote (git remote add upstream <original-repo-url>) and regularly fetching and merging or rebasing from it.

Collaboration and Open Source: Forking is often used in open-source projects where multiple developers want to contribute to a single project. Each developer creates their fork, works on their changes, and then submits PRs to the main project.

It allows for a distributed and collaborative development workflow, enabling developers from around the world to contribute to a project.
