**Current Directory and Staging Area in GIT**

In Git, the current directory refers to the directory that you are currently working in. It is the directory that contains the files and directories that you want to track with Git.

The staging area, also known as the index, is a place where you can prepare changes for a commit. When you make changes to files in your working directory, those changes are not automatically added to the next commit. Instead, you must explicitly add them to the staging area using the `git add` command. Once the changes are in the staging area, you can review them and make any necessary modifications before committing them to the repository using `git commit`.

Think of the staging area as a place where you can review and organize your changes before committing them to your repository. It allows you to selectively choose which changes you want to include in your next commit, and gives you an opportunity to review your changes before making them permanent.

**How to create a initial commit in GIT ?**

To create an initial commit in Git, you can follow these steps:

1. First, initialize a new Git repository in your project directory by running the command `**git init**`.

2. Add the files you want to track to the staging area using the command `**git add <filename>**` or `**git add .**` to add all files.

3. Commit the changes using the command **`git commit -m** "Initial commit"`.

Lab exercise for creating an initial commit in Git:

|  |
| --- |
| 1. Create a new directory for your project and navigate to it using the command line.  ```  mkdir myproject  cd myproject  ```  2. Initialize a new Git repository in the directory using the command `git init`.  ```  git init  ```  3. Create a new file in the directory and add some content to it.  ```  touch README.md  echo "Hello, world!" > README.md  ```  4. Add the file to the staging area using the command `git add`.  ```  git add README.md  ```  5. Commit the changes with a message using the command `git commit`.  ```  git commit -m "Initial commit"  ```  6. Verify that the commit was created by running the command `git log`.  ```  git log |

GIT Logs:

To read the logs in Git, you can use the `git log` command. This command displays a list of all the commits in your repository, along with some information about each commit, such as the commit hash, author, date, and commit message.

Here's an example lab exercise for reading the logs in Git:

1. Create a new directory for your project and navigate to it using the command line.

```

mkdir myproject

cd myproject

```

2. Initialize a new Git repository in the directory using the command `git init`.

```

git init

```

3. Create a new file in the directory and add some content to it.

```

touch README.md

echo "Hello, world!" > README.md

```

4. Add the file to the staging area using the command `git add`.

```

git add README.md

```

5. Commit the changes with a message using the command `git commit`.

```

git commit -m "Initial commit"

```

6. Make some changes to the file and commit them.

```

echo "This is a new line." >> README.md

git add README.md

git commit -m "Added a new line to README.md"

```

7. View the logs using the `git log` command.

```

git log

```

**Branches in GIT**

In Git, a branch is a lightweight movable pointer to a commit. It is essentially a way to create a separate line of development that is independent from the main line of development. Each branch represents a different version of the codebase, and changes made in one branch do not affect the code in other branches until they are merged together.

The main branch in Git is usually called "master", but you can create as many branches as you need for your project. Branches are useful for working on new features, fixing bugs, or experimenting with new ideas without affecting the main codebase. Once you are satisfied with the changes in a branch, you can merge it back into the main branch to incorporate the changes into the main codebase.

Using branches in Git allows multiple developers to work on different parts of a project simultaneously without interfering with each other's work. It also provides a way to experiment with new features or ideas without affecting the stability of the main codebase.

**Use Cases of GIT Branches:**

There are many use cases for Git branches, including:

1. **Feature development**: You can create a new branch for each new feature you are working on, allowing you to develop and test the feature without affecting the main codebase.

2. **Bug fixing**: If you discover a bug in the code, you can create a new branch to fix the bug without affecting the main codebase.

3. **Experimentation**: You can create a branch to experiment with new ideas or technologies without affecting the stability of the main codebase.

4. **Code reviews**: You can create a branch to review changes made by another developer before merging them into the main codebase.

5. **Release management**: You can create a branch for each release of your software, allowing you to maintain different versions of the codebase with different features and bug fixes.

6. **Hotfixes**: If you need to make an urgent fix to the code, you can create a branch to fix the issue and merge it into the main codebase as soon as possible.

**How to create a branch :**

To create a new branch in Git, you can use the `git branch` command. Here are the steps to create a new branch:

1. Make sure you are in the main branch by running `git branch` command. The current branch will be highlighted with an asterisk.

```

git branch

\* master

```

2. Create a new branch by running the `git branch <branch-name>` command, where `<branch-name>` is the name of the new branch you want to create.

```

git branch feature-branch

```

3. Switch to the new branch by running the `git checkout <branch-name>` command.

```

git checkout feature-branch

```

Alternatively, you can combine steps 2 and 3 into a single command by running `git checkout -b <branch-name>`, which creates a new branch and switches to it in one step.

```

git checkout -b feature-branch

```

That's it! You have created a new branch in Git and switched to it. Now you can make changes to the code in this branch without affecting the main codebase.

Lab 02:

Lab exercise for creating a branch in Git:

|  |
| --- |
| 1. Create a new directory for your project and navigate to it using the command line.  ```  mkdir myproject  cd myproject  ```  2. Initialize a new Git repository in the directory using the command `git init`.  ```  git init  ```  3. Create a new file in the directory and add some content to it.  ```  touch README.md  echo "Hello, world!" > README.md  ```  4. Add the file to the staging area using the command `git add`.  ```  git add README.md  ```  5. Commit the changes with a message using the command `git commit`.  ```  git commit -m "Initial commit"  ```  6. Create a new branch using the `git branch` command.  ```  git branch feature-branch  ```  7. Switch to the new branch using the `git checkout` command.  ```  git checkout feature-branch  ```  8. Make some changes to the file and commit them.  ```  echo "This is a new feature." >> README.md  git add README.md  git commit -m "Added a new feature to README.md"  ```  9. Switch back to the main branch using the `git checkout` command.  ```  git checkout master  ```  10. Verify that the changes made in the feature branch did not affect the main branch.  ```  cat README.md  ```  You should only see the original content of the file, "Hello, world!". |

**GIT Merge:**

In Git, merging is the process of combining changes from one branch into another. When you merge a branch, Git takes the changes made in the source branch and applies them to the target branch, creating a new commit that incorporates the changes.

Merging is typically used to incorporate changes made in a feature branch back into the main branch, or to incorporate bug fixes made in a bugfix branch back into the main branch. It allows multiple developers to work on different parts of a project simultaneously without interfering with each other's work.

When you merge a branch, Git will automatically try to merge the changes in the two branches. If there are any conflicts between the changes, Git will prompt you to resolve them manually before completing the merge.

Merging is an important part of Git workflow and allows teams to collaborate on a codebase effectively.

Lab Exercise:

|  |
| --- |
| Sure, here's an example of how to merge two branches in Git:  1. Create a new directory for your project and navigate to it using the command line.  ```  mkdir myproject  cd myproject  ```  2. Initialize a new Git repository in the directory using the command `git init`.  ```  git init  ```  3. Create a new file in the directory and add some content to it.  ```  touch README.md  echo "Hello, world!" > README.md  ```  4. Add the file to the staging area using the command `git add`.  ```  git add README.md  ```  5. Commit the changes with a message using the command `git commit`.  ```  git commit -m "Initial commit"  ```  6. Create a new branch called `feature-branch` using the `git branch` command.  ```  git branch feature-branch  ```  7. Switch to the new branch using the `git checkout` command.  ```  git checkout feature-branch  ```  8. Make some changes to the file and commit them.  ```  echo "This is a new feature." >> README.md  git add README.md  git commit -m "Added a new feature to README.md"  ```  9. Switch back to the main branch using the `git checkout` command.  ```  git checkout master  ```  10. Create a new branch called `bugfix-branch` using the `git branch` command.  ```  git branch bugfix-branch  ```  11. Switch to the new branch using the `git checkout` command.  ```  git checkout bugfix-branch  ```  12. Make some changes to the file to fix a bug and commit them.  ```  sed -i '' 's/world/universe/g' README.md  git add README.md  git commit -m "Fixed a bug in README.md"  ```  13. Switch back to the main branch using the `git checkout` command.  ```  git checkout master  ```  14. Merge the changes from `feature-branch` into `master` using the `git merge` command.  ```  git merge feature-branch  ```  15. Merge the changes from `bugfix-branch` into `master` using the `git merge` command.  ```  git merge bugfix-branch  ```  Congratulations, you have successfully merged two branches in Git! Now both the new feature and bug fix are incorporated into the main codebase. |
| Deleting Data:   1. Working Directory Files 2. Unstaged Changes 3. Staged Changes 4. Latest Commits 5. Branches   Working Directory files:  To delete a file and commit the changes in Git, you can use the following commands:    1. To delete the file: `git rm filename`    This will remove the file from your local repository.    2. To commit the changes: `git commit -m "Delete filename"`    This will commit the changes to your local repository along with a message describing the changes made.      For example, if you want to delete a file named "example.txt" and commit the changes, you can use the following commands:    ```  git rm example.txt  git commit -m "Delete example.txt"  git push  ```    Note that these commands assume you are currently in the Git repository directory and have the necessary permissions to make changes.  How to restore a file:  The `git restore` command is used to restore files in a Git repository. It can be used to restore files to a previous commit or to discard local changes made to a file. Here are some examples:    - To restore a file to its state in the last commit: `**git restore filename**`    This will replace the contents of the file with the version in the last commit.    - To restore a file to its state in a specific commit: `**git restore --source=commit\_hash filename**`    This will replace the contents of the file with the version in the specified commit.    For example, if you want to restore a file named "example.txt" to its state in the last commit, you can use the following command:    ```  git restore example.txt  ```    If you want to discard local changes made to the file and restore it to its state in the last commit, you can use the following command:    ```  git restore --staged example.txt  ```    Note that these commands assume you are currently in the Git repository directory and have the necessary permissions to make changes.  The **`git checkout <file\_name>**` command is used to discard changes made to a specific file in the working directory and replace it with the version from the most recent commit. This can be useful if you have made changes to a file and want to discard those changes and start over with the version from the last commit.    Here's an example:    Suppose you have made changes to a file named "example.txt" and want to discard those changes and restore the file to its state in the last commit. You can use the following command:    ```  git checkout example.txt  ```    This will replace the contents of "example.txt" with the version from the most recent commit, effectively discarding any local changes made to the file.    Note that if you have already staged changes to the file using `git add`, you will need to unstage those changes first using `git reset` before using `git checkout` to discard the changes in the working directory.  **GIT Restore:**  Here's an example of how to use the `git restore` command and some of its use cases:    Suppose you have made changes to a file named "example.txt" and want to restore it to its state in the last commit. You can use the following command:    ```  git restore example.txt  ```    This will replace the contents of "example.txt" with the version from the most recent commit, effectively discarding any local changes made to the file.    Use cases for `git restore` include:    1. **Discarding local changes:** If you have made changes to a file and want to discard those changes and start over with the version from the last commit, you can use `git restore` to restore the file to its previous state.    2. **Unstaging changes**: If you have staged changes to a file using `git add` and want to unstage those changes, you can use `git restore --staged` followed by the filename to unstage the changes.    3. **Reverting to a previous commit**: If you want to revert a file to its state in a previous commit, you can use `git restore --source=<commit>` followed by the filename to restore the file to its state in the specified commit.    4. **Restoring deleted files**: If you have accidentally deleted a file and want to restore it, you can use `git restore` followed by the filename to restore the file to its previous state.    Note that the `git restore` command was introduced in Git 2.23 and replaces the older `git checkout -- <file>` command for restoring files.  **Undo the staged Changes:**  This will unstage all changes that you have added to the staging area. If you want to unstage a specific file, you can specify the file name after the HEAD parameter:      ```  git reset HEAD <file>  git checkout <file>  ```    The first command `git reset HEAD <file>` will unstage the changes for the specified file. The second command `git checkout <file>` will discard the changes made to the file and restore it to the last committed state.    Note that if you have made changes to multiple files, you can use `git reset` to unstage all changes and then use `git checkout` to restore all files to the last committed state:    ```  git reset HEAD  git checkout .  ```    This will unstage all changes and restore all files to the last committed state. |
| To move the `HEAD` to a different commit in Git, you can use the `git reset` command. Here's how you can do it:    1. \*\*Find the Commit ID:\*\*  First, find the commit ID that you want to move the `HEAD` to. You can use the `git log` command to see a list of commits:    ```bash  git log  ```    Identify the commit ID you want to move to; it's a long hexadecimal string associated with each commit.    2. \*\*Use `git reset` to Move `HEAD`:\*\*  Once you have the commit ID, use the following command to move the `HEAD` to that commit:    ```bash  git reset --hard <commit\_id>  ```    Replace `<commit\_id>` with the actual commit ID you want to move to.    The `--hard` option resets both the `HEAD` and the working directory to the specified commit. Be cautious when using `--hard` as it discards changes irreversibly.    3. \*\*Optional: Force Push (Use with Caution):\*\*  If you have already pushed the branch to a remote repository, you may need to force-push the changes. However, be careful with force-pushing, as it can rewrite history and cause issues for collaborators.    ```bash  git push origin <branch\_name> --force  ```    Replace `<branch\_name>` with the name of your branch.    Remember that moving the `HEAD` and force-pushing can have implications, especially in a shared repository. Communicate with your collaborators if you're making significant changes to the commit history. Additionally, be cautious with `--force` as it can overwrite changes on the remote repository. |

**Finding the difference of files:**

|  |
| --- |
| The `git diff` command is used to show the differences between the working directory, the staging area, and the last commit. It helps you review changes before committing them. Here are some examples of using the `git diff` command:    1. \*\*Compare Working Directory with Staging Area:\*\*  This shows the changes between the files in your working directory and the files in the staging area (index).    ```bash  git diff  ```    2. \*\*Compare Staging Area with Last Commit:\*\*  This shows the changes that are staged but not yet committed.    ```bash  git diff --staged  ```    3. \*\*Compare Specific Files:\*\*  To see the changes in a specific file, provide the filename(s) after the `git diff` command.    ```bash  git diff file1.txt file2.txt  ```    4. \*\*Compare Between Commits:\*\*  To compare changes between two specific commits, you can use commit hashes or branch names.    ```bash  git diff commit\_hash1 commit\_hash2  ```    or    ```bash  git diff branch\_name1 branch\_name2  ```    5. \*\*View Changes in Color:\*\*  You can add the `--color` option for a more visually appealing output.    ```bash  git diff --color  ```    6. \*\*Unified Diff Format:\*\*  The unified diff format is a standard way of representing the differences between two sets of files.    ```bash  git diff --unified=3  ```    This example shows three lines of context around each change.    7. \*\*Output to a File:\*\*  You can redirect the output of `git diff` to a file for further analysis.    ```bash  git diff > changes.txt  ```    These are just a few examples of how you can use the `git diff` command to examine changes in your Git repository. Adjust the options based on your specific needs and preferences. |

**Adding only the updated files ; don’t add any new files in the local repo**

|  |
| --- |
| The `git add -u` command is used to stage and prepare for the next commit all the changes to tracked files in the working directory that have been modified or deleted. It does not include untracked files. Here's an example of how to use `git add -u`:    1. \*\*Modify Tracked Files:\*\*  Make changes to one or more files in your Git repository. For example, let's modify a file named `example.txt`.    # echo "Updated content" > example.txt    2. \*\*Check the Status:\*\*  Before using `git add -u`, you can use the `git status` command to see the current status of your working directory.    ```bash  git status  ```    This will show you which files have been modified or deleted.    3. \*\*Use `git add -u`:\*\*  Now, use the `git add -u` command to stage the modifications and deletions of tracked files:    ```bash  git add -u  ```    This command stages all modifications and deletions, but not additions of new files.    4. \*\*Check the Status Again:\*\*  After using `git add -u`, you can run `git status` again to see the changes that have been staged.    ```bash  git status  ```    You should see the modified files in the "Changes to be committed" section.    5. \*\*Commit the Changes:\*\*  Finally, commit the changes to make them a part of the version history:    ```bash  git commit -m "Update example.txt"  ``` |

**The git rm command is used to remove files from both your working directory and the Git repository.**

|  |
| --- |
| The `git rm` command is used to remove files from both your working directory and the Git repository. It's essentially a combination of the Unix command `rm` and `git add`. Here's the basic syntax:    ```bash  git rm [options] <file>  ```    - Options: Various options can be used with `git rm` to control its behavior.  - `<file>`: The file or files you want to remove from the repository.    ### Examples:    1. \*\*Remove a file from the working directory and staging area:\*\*    ```bash  git rm filename.txt  ```    This removes the file `filename.txt` from both your working directory and the staging area.    2. \*\*Remove multiple files:\*\*    ```bash  git rm file1.txt file2.txt  ```    You can specify multiple files to be removed in a single command.    3. \*\*Remove a file and commit the change:\*\*    ```bash  git rm --cached myfile.txt  git commit -m "Remove myfile.txt from the repository"  ```    The `--cached` option removes the file from the staging area (index) but keeps it in your working directory. After that, you commit the change.    4. \*\*Remove a directory and its contents:\*\*    ```bash  git rm -r directoryname  ```    The `-r` option is used for recursive removal, which is necessary when removing directories.    5. \*\*Force removal of a file (without checking):\*\*    ```bash  git rm -f filename.txt  ```    Use the `-f` option to force removal without checking if the file is modified.    6. \*\*Remove files matching a pattern:\*\*    ```bash  git rm \*.log  ``` |

**Git MV command**

|  |
| --- |
| The `git mv` command is used to move or rename files and directories while simultaneously staging the changes. It is essentially a shortcut that combines the actions of moving a file, removing the old file, and adding the new file in Git. Here are some examples:    ### 1. \*\*Move a file to a new directory:\*\*    ```bash  git mv file.txt new\_directory/  ```    This command moves the file `file.txt` to the directory `new\_directory/`. The change is staged, and you can then commit the move.    ### 2. \*\*Rename a file:\*\*    ```bash  git mv oldname.txt newname.txt  ```    This renames the file `oldname.txt` to `newname.txt`. The command takes care of both renaming the file and staging the change.    ### 3. \*\*Move and rename a file:\*\*    ```bash  git mv old\_directory/oldfile.txt new\_directory/newfile.txt  ```    You can move a file to a different directory and rename it simultaneously using the `git mv` command.    ### 4. \*\*Move a directory:\*\*    ```bash  git mv old\_directory/ new\_directory/  ```    This command moves the entire directory `old\_directory` into `new\_directory`. All the files and subdirectories are moved, and the changes are staged.    ### 5. \*\*Force move/rename (overwrite if necessary):\*\*    ```bash  git mv -f oldfile.txt newfile.txt  ```    The `-f` option forces the move or rename, overwriting the destination file if it already exists.    ### 6. \*\*Move and commit in one step:\*\*    ```bash  git mv file.txt new\_directory/ && git commit -m "Move file.txt to new\_directory/"  ```    If you want to move a file and commit the change in a single step, you can use the `&&` operator to combine the `git mv` and `git commit` commands.    Remember that after using `git mv`, you still need to commit the changes using `git commit` to make them permanent in the repository. The `git mv` command helps you perform file moves and renames more efficiently by handling both the move and the necessary staging in one step. |

**GIT STASH**

|  |
| --- |
| The `git stash` command is used to save changes in your working directory that are not ready to be committed yet, allowing you to switch branches or perform other operations without committing the changes. The changes are temporarily "stashed away" so that you can return to them later. This is particularly useful when you need to switch branches or perform some other operation that requires a clean working directory.    ### Basic Usage:    - \*\*Stash Changes:\*\*    ```bash  git stash  ```    This command saves your local modifications to a new stash entry and reverts the working directory to match the `HEAD` commit.    - \*\*Stash Changes with a Message:\*\*    ```bash  git stash save "Your stash message here"  ```    You can provide a descriptive message to the stash, making it easier to identify later.    ### Viewing and Applying Stashes:    - \*\*List Stashes:\*\*    ```bash  git stash list  ```    This command displays a list of all stashes you have created.    - \*\*Apply the Latest Stash:\*\*    ```bash  git stash apply  ```    This applies the changes from the latest stash to your working directory. The stash itself is not removed.    - \*\*Apply a Specific Stash:\*\*    ```bash  git stash apply stash@{2}  ```    You can apply a specific stash by providing its reference (e.g., `stash@{2}`).    ### Additional Stash Operations:    - \*\*Pop the Latest Stash:\*\*    ```bash  git stash pop  ```    This command applies the changes from the latest stash and removes that stash from the stash list.    - \*\*Pop a Specific Stash:\*\*    ```bash  git stash pop stash@{1}  ```    Similar to `git stash apply`, but it also removes the stash from the stash list.    - \*\*Drop a Stash:\*\*    ```bash  git stash drop stash@{0}  ```    This permanently deletes a stash from the stash list.    - \*\*Clear All Stashes:\*\*    ```bash  git stash clear  ```    This removes all stashes, permanently discarding them.    ### Example Workflow:    1. \*\*Stash Changes:\*\*    ```bash  git stash  ```    2. \*\*Switch Branch:\*\*    ```bash  git checkout feature-branch  ```    3. \*\*Work on the Feature Branch:\*\*    Make your changes on the new branch.    4. \*\*Switch Back to Original Branch:\*\*    ```bash  git checkout main  ```    5. \*\*Apply Stash:\*\*    ```bash  git stash apply  ```    or    ```bash  ```    This brings back your stashed changes to the original branch.    Using `git stash` is a helpful way to temporarily set aside changes and switch between branches without committing incomplete work. It's a useful tool in a variety of scenarios where you need to juggle multiple branches or temporarily set aside changes. |

|  |
| --- |
| **Deleting Branch in GIT:**    To delete a branch in Git, you can use the `git branch -d` command followed by the name of the branch you want to delete. For example, if you want to delete a branch named `my-branch`, you can run the following command:    ```  git branch -d my-branch  ```    If the branch has not been merged yet, you may need to use the `-D` option instead of `-d` to force the deletion:    ```  git branch -D my-branch  ```    This will permanently delete the branch and all of its commits, so make sure you don't need it anymore before running this command.    **Detached HEAD:**    When you checkout to a specific commit in Git, you are said to be in a "detached HEAD" state. This means that your current state is not pointing to the branch head, but rather to a specific commit.    In this state, any changes you make will not be associated with a branch, and if you make a new commit, it will not be on any branch and may be lost if you switch to another branch.    Here's an example of how you can end up in a detached HEAD state:    1. You create a new branch called "feature" from the "master" branch: `git checkout -b feature`  2. You make some changes, commit them, and push them to the remote repository: `git commit -m "some changes"; git push origin feature`  3. You checkout a specific commit in the "feature" branch: `git checkout <commit\_hash>`  4. Now you are in a detached HEAD state, and any changes you make will not be associated with the "feature" branch.    To get back to the "feature" branch, you can use the following command: `git checkout feature`. This will move your HEAD back to the "feature" branch head and any changes you make will be associated with the branch again.    **#** git branch  # Add a dummy.txt file  # git add .  # git commit -m “dmmy file added”  # git log  # git checkout      mkdir project1  cd project1/  ls  ls -lr  touch file1  66 nano.exe file1  67 git init  68 git add file1  69 git status  70 git commit -m "Added first file"  71 git status  72 git logs  73 git log  74 touch file2  75 vim file2  76 git status  77 git add .  78 git commit -m "file2"  79 git status  80 ls  81 ls -ltr  82 git branch  83 git log  84 git checkout be92f498cb4d295de003f046844de3376621092d  85 git branch  86 git status  87 git status  88 ls  89 vi file1  90 git add .  91 git stauts  92 git status  93 git commit -m "Vaithee-updated" .  94 git statys  95 git status  96 git branch  97 git switch master  98 git branch  99 git branch detached-head be92f49  100 git branch  101 git switch detached-head  102 git branch  103 git branch master  104 git checkout master  105 git checkout detached-head  106 git checkout master  107 git merge detached-head      **GITIGNORE**:    `.gitignore` is a file in Git that allows you to specify files and directories that should be ignored by Git. This means that any changes made to these files or directories will not be tracked, committed, or pushed to the remote repository.    You can use `.gitignore` to exclude files like temporary files, log files, build artifacts, and other files that are not necessary to include in the repository. This can help keep your repository clean and prevent unnecessary files from being pushed to the remote repository.    To use `.gitignore`, you need to create a file called `.gitignore` in the root directory of your Git repository and list the files and directories you want to ignore. You can use wildcards and patterns to specify multiple files or directories at once. Once you have created the `.gitignore` file, Git will automatically ignore the specified files and directories. |