**Terminal commands :**

**- `ls`:** lists all the files/folders inside the current path

**- `ls -a`:** lists all the files/folders inside the current path, including hidden files

**- `cd ..`:** go back one folder

**- `cd ../..`:** go back two folders

**- `cd ../../<folder name>`:** go back twice, then into a folder

**- `touch <file name>’:** Simple way (Git Bash) to create a file inside the current path

**- ‘cat <file name>’:** used to display the contents (even codes) of any file and it works with any .txt, .js, .py, .java, etc. file

**- ‘mv <oldfile.txt> <newfile.txt>’:** renames the file on your system

**- `rm -rf <file name>`:** Deletes the file from the directory

**Observation:**

- Hidden folders/files usually start with a `.`

- `.` refers to the current directory

- `..` refers to the parent directory (one level up)

- That's why **`cd ..`** takes you back - you're literally telling it "Go to the parent directory"!

**To run Git commands, we are going to first install Git.**

**How to configure Git after installing it?**

Run the commands:

**- ‘git config –global user.name <”Your Name”>’**

**- ‘git config –global user.email <”Your email”>’**

**- ‘git config –list’**

After setting up our user-name and email address, we can use all the git commands now.

**Git commands:**

- **`git init`**: Initialized empty Git repository

|  |
| --- |
| Creates a hidden `.git` folder |
| Initializes basic Git configuration |
| Enables Git tracking for that folder |
| Inside the `.git` folder: |
| `HEAD` (tracks current branch) |
| `config` (repository settings) |
| `objects/` (stores all our files) |
| `refs/` (tracks branches) |
| `hooks/` (custom scripts) |

|  |
| --- |
| **Before `git init`:** |
| - Just a normal folder |
| - No version control |
| - Can't track changes |
| - Can't use Git commands |
| **After `git init`:** |
| - Folder becomes a Git repository |
| - Can track changes |
| - Can create branches |
| - Can connect to GitHub |

**How can we make changes to the repository after creating a remote repository on GitHub?**

To make changes in the remote repository we have to bring a remote repository locally on our system.

**- ‘git clone <URL of the remote repository>’:** git clone makes a complete copy of a remote repository on our local machine. It:

|  |
| --- |
| Downloads all files and version history |
| Sets up local Git repository |
| Creates remote tracking branch |
| Automatically configures "origin" remote pointing to source repository |

**- `git status`:** Tracks the history of changes made in the repository. Shows which files have been added, modified, or deleted.

**- `git add .` :** Stages ALL changes in the current directory and its subdirectories

**- `git add <filename>` :** Stages changes only for the specified file

|  |
| --- |
| Example: |
| - `git add names.txt` : Only stages changes in names.txt |
| - Other modified files remain unstaged |
|  |

**- ‘git restore <file name> / git restore --staged <file name>’:** We can unstage files using this

**NOTE:**

We can always rely on `git status` to see what's staged (green) and what's not (red).

**Before moving forward let’s know what's a staging area and a commit?**

**What is Staging Area?**

|  |
| --- |
| Think of the staging area as a "preparation zone" or "packing area" |
| It's like a box where you put things before shipping them |
| Files you want to commit must go through staging first |

**Why use a Staging Area?**

|  |
| --- |
| Let’s take an example |
| - We modified 3 files: A.txt, B.txt, C.txt |
| - We want to commit A.txt and B.txt together, but C.txt separately |
| - Staging lets us: |
| 1. First stage and commit A.txt and B.txt |
| 2. Then stage and commit C.txt later |

**What is a Commit?**

|  |
| --- |
| A commit is like taking a snapshot of our project at a specific point in time |
| Each commit has: |
| A unique ID (hash) |
| A message describing what changed |
| The actual changes made to files |
| Information about when and who made the changes |

***“Think of commits like saving a video game:”***

|  |
| --- |
| Each save is a different point in our progress |
| We can go back to any save point if needed |
| Each save has a description of what we achieved |
| We can see our whole journey through all our saves |

**Why do we basically make commits?**

|  |
| --- |
| Track History: See what changed, when, and why |
| Backup Points: Can return to any previous commit if something breaks |
| Collaboration: Other developers can understand our changes |

**- `git commit -m "message"` :** Creates a commit with our staged changes (The -m flag is for adding a commit message).

|  |
| --- |
| **Example:** |
| `git commit -m "Added new student names" |

|  |
| --- |
| So, the typical workflow looks like: |
| - Firstly, make some changes to your files |
| **- `git status`** (check what's changed) |
| **- `git add <file name>`** (stage specific files or all files) |
| **- `git status`** (verify what's staged) |
| **- `git commit -m "message"`** (commit the staged changes) |

**- ‘git log’:** A commit history tracker. It gives us a detailed log of all the commits we've made in our repository, starting from the most recent one.

**- ‘git reset <UID>’:** Removes all commits that came after

|  |
| --- |
| Case Study: |
| 1. You create file.txt with content "Hello" (First commit: abc123) |
| 2. You add "World" to file.txt (Second commit: def456) |
| 3. You add "!!!" to file.txt (Third commit: ghi789) |
| When you do: |
| git reset abc123 |
|  |
| Now what happens: |
|  |
| The commit history will only show the first commit |
| BUT file.txt will still contain "Hello World!!!" in your folder |
| The changes aren't committed anymore, but the file itself keeps those changes |
| You'll see these changes as "unstaged" if you run git status |

**-‘git stash <file name>’:** Files are REMOVED from our working directory (they disappear from file manager) and the changes are saved in a hidden Git area so we can switch to other work and come back to them later.

**-‘git stash pop’:** Brings back the most recent stash

**-‘git stash clear’:** permanently deletes ALL our stashed changes.

**Now how to push your projects/folders to GitHub?**

First we’ll create a connection between our local project/folder with the remote repository on GitHub.

Create a new repository or open an existing a repository and then copy the URL of the repository present on GitHub. The URL will act as junction with the repository on GitHub with our local project/folder. To attach the URL to our project/ folder use the command.

**-‘git remote add origin <URL>’:** This command establishes a connection between your local Git repository and a remote repository (GitHub).

After the connection has been made whatever changes and whenever the changes were made to our project/folder will reflect on the remote repository.

Since the connection has been made, so whenever we make any changes to our local project/folder we’ll not use the ‘git remote add origin <URL>’ again and again after every commit instead we’ll use the:

**-‘git push origin main’:** This command uploads your local commits from the main branch of your repository to the remote repository named origin (usually GitHub).

**-‘git remote add upstream <URL of the repo to be forked>’:** By convention, the original project from where we have forked our project is known as the upstream. This command is used to link a second remote repository to your local project, typically for tracking updates from the original source repository (upstream). It is commonly used in forking workflows on GitHub.

Now you cannot make changes to anyone’s project directly. We have to first fork it, since we cannot modify anyone’s project without their consent. But since on forking we basically create a copy of that project on your account, we can modify anything with that copy.

Now to reflect to your changes to the main project you have to create a pull request, so that the owner of the original project can merge your changes.

**-‘git remote -v:** This command lists all remote repositories linked to your local Git project, showing their URLs and access methods.

**Observation:**

When we want bring a remote repository (that we created) locally we use the ‘**git remote add origin <URL>’** but when we want to add a forked remote repository locally we use the git clone command.

**Conclusion:**

|  |
| --- |
| For ANY repository (forked or not), you typically start with ‘**git clone** **<URL>’** when you're setting up the project locally for the first time. git clone does several things: |
| 1. Downloads all the code |
| 2. Sets up the local Git repository |
| 3. Automatically adds the "origin" remote |
|  |
| You DON'T usually use **‘git remote add origin <URL>’** as your first step - this command is mainly used when: |
| 1. You created a local repository first (git init) |
| 2. You now want to connect it to a remote repository |

|  |
| --- |
| **So, after learning push we can say our workflow looks like:** |
| 1. Make changes to local files |
| 2. Stage changes: `git add .` or `git add filename` |
| 3. Commit changes: `git commit -m "message"` |
| 4. Push changes: `git push <branch name>` |

**What is a Branch?**

|  |
| --- |
| A branch is a separate line of development in Git that lets you: |
| 1. Work on features/fixes isolated from main code |
| 2. Track parallel versions of code |
| 3. Experiment without affecting stable code |
| 4. Enable multiple developers to work simultaneously |

**Branch Management**

Creating & Switching Branches:

**- `git branch <branch-name>`:** Create new branch

**- `git checkout <branch-name>`:** Switch to a branch

**- `git checkout -b <feature-name>`:** Create and swich in one command

**- `git branch`:** List branches (asterisk \* shows current branch)

**Working with Branches:**

**-`git push <branch name>`:** Pushes the changes to the specified branch

|  |
| --- |
| Changes made in one branch stay in that branch |
| Files/changes unique to feature branch won't be visible when on main branch |
| Need to merge feature branch into main to get changes there |

**Steps for merging:**

|  |
| --- |
| 1. `git checkout main`: Switch to main branch |
| 2. `git merge feature`: Merge feature branch |
| 3. `git push origin main`: Push merged changes |

**Forking**

Now if we want to make changes to a repository which is not our own we have to “fork” the repository.

Forking a repository on GitHub creates a copy of that repository on our account. After we have forked the repository of another user and brough it remotely on our own account, we have to clone our fork locally by using the

**- ‘git clone <forked-repo-URL>`**

We have already learnt about this command that it makes a complete copy of a remote repository on our local machine.

Now, after cloning the remote repository we are ready to make changes to the local repository thus created. But before moving forward we must learn a way to commit while working on someone else’s project. We should always create a separate branch for our personal commits.

Working directly on the main branch isn't recommended.

|  |
| --- |
| **Here's a real-world example:** |
| Imagine a popular open-source text editor like VS Code: |
| 1. A developer notices that some characters aren't displaying correctly. |
| 2. Instead of changing VS Code's main code directly: |
| - They fork the VS Code repository |
| - Create a branch called `fix-fonts` |
| - Make and test their changes |
| - Submit a pull request (We’ll learn about this shortly) |
|  |
| **Benefits:** |
| - For VS Code team (owner): |
| - Main branch stays stable for millions of users |
| - Can review changes before merging |
| - Multiple developers can work on different fixes simultaneously |
|  |
| - For developer (contributor): |
| - Can experiment without breaking the main product |
| - Gets feedback from VS Code team |
| - Their solution can help all VS Code users globally |

And now after making changes to our fork we now push our changes to a separate branch, hence creating a Pull Request (PR):

**-`git push origin <branch name>’**

So, when we create any change in our copy of the project, how do we make sure whatever the change that we made in our copy is visible in the main project. To do that we request it from the owner via Pull Request. And after the request is accepted, means when it is merged whatever changes we made to our fork in any branch will be visible to main branch of the original repository.

**NOTE:**

We know that whatever changes we made will be committed to the new branch. The thing to notice is that when we create a pull request (PR) from a branch, any new commits we push to that same branch will automatically be added to the existing PR. We can't create a second PR from the same branch until the first one is merged/closed.

|  |
| --- |
| This is why it's recommended to: |
| 1. Create a new branch for each feature/fix |
| 2. Keep changes focused and related within one PR |
| 3. Create separate branches for unrelated changes that need separate PRs! |

**Syncing the forked repository with the original (upstream) repository**

After PR merge, the changes appear only in upstream main but not your fork.

So, Commands to use for syncing our forked repo with the original repo:

**- ‘git fetch --all –prune’:** Fetches all updates including deleted refs

**- ‘git reset --hard upstream/main’:** Forces local branch to match upstream

**- ‘git push’:** Updates our fork

Alternative approach (Shorter approach):

**- `git pull upstream main`**

**- `git push`**

This synchronizes our fork's main branch with upstream, avoiding manual branch switching on GitHub.

**NOTE:**

Sometimes while pushing/making a PR there’s an occurrence of an error. The error occurs because the remote repository has commits that you don't have locally. Git is blocking your push to avoid overwriting those changes. To resolve this error use these commands:

**-‘git fetch origin’:** Retrieves updates from the remote repository but doesn't merge them. It lets you view changes without affecting your local branch.

**-‘git pull origin main’:** Fetches updates from the main branch of the remote repository and merges them into your local branch.  
Keeps your local branch up-to-date with the remote.

**-‘git push origin main’:** Pushes your committed changes from the local main branch to the remote main branch.  
Updates the remote repository with your local changes.

**So, it's a good practice to always pull before pushing while working on a shared project!**

**The typical workflow for ‘git remote add origin <URL>’ is:**

**Purpose**: Connects your local project to a remote repository (typically empty) you created on a platform like GitHub.

**Workflow**:

|  |
| --- |
| You initialize a local Git repository (**git init**). |
| Then, you link it to the remote (**git remote add origin <URL>**). |
| After making changes (then stage and commit), you **push them** **directly** (git push origin main)—**no need for a pull request** (PR) since it's your repository. |

**The typical workflow for ‘git clone <URL>’ is:**

**Purpose**: Copies an existing remote repository to your local system, including its history and files.

**Workflow**:

|  |
| --- |
| You clone the repository (**git clone <URL>).** |
| If you're contributing (e.g., in a team project), you’ll typically work on a separate branch, make changes (then stage and commit), and push them (remember to pull to avoid conflicts) |
| To integrate your changes into the main branch, you often need to create a **pull request (PR)**, if you’re not the owner or it’s a collaborative project. |

For visual understanding and working of all the commands and workflow check out:

[**Complete Git and GitHub Tutorial by Kunal Kushwaha**](https://youtu.be/apGV9Kg7ics?feature=shared)

***-Hamzah Latif***

*(Author)*