Git & Github Notes

Git is a version control system that helps you keep track of changes in your code.

A Version Control System (VCS) is a tool that helps track and manage changes to files over time.

Types of Version Control Systems:

1. Local VCS – Simple system that stores changes on your computer (e.g., saving multiple copies of files).
2. Centralized VCS (CVCS) – A single server stores all versions, and users must connect to it (e.g., SVN).
3. Distributed VCS (DVCS) – Each user has a full copy of the project, allowing offline work (e.g., Git).

Why Use a Version Control System?

✅ Keeps Track of Changes – Saves different versions so you can revert if needed.
✅ Collaboration – Multiple people can work on the same project without conflicts.
✅ Backup & Recovery – Prevents accidental data loss.
✅ Organized Workflow – Helps manage different features or bug fixes using branches.

🔍 How Git Tracks Changes (Step-by-Step Explanation)

1️⃣ Git Stores Snapshots, Not Differences

Unlike older version control systems (like SVN), which store differences between files, Git takes snapshots of the entire project at different points in time.
📌 Snapshot = A copy of all files at a certain moment.

- If no changes are made to a file, Git does not store a duplicate—it just points to the previous version to save space.
- If a file is modified, Git stores a new snapshot of that file and links it to previous versions.

How Git Identifies Changes (Hashes & Commits)

- Git uses a powerful technology called SHA-1 Hashing to track every change.
- Every time you commit, Git creates a unique ID (hash) for that commit.

🛠️ Example:

```
git commit -m "Fixed a bug"
```

✔️ Git generates a hash like `a1b2c3d4e5f6g7h8i9j0` and links it to the previous commit.
✔️ Each commit is like a checkpoint in a video game—you can always go back!

1️⃣ How Git Thinks About Files

Before understanding how Git tracks changes, it's important to know how Git classifies files in a repository.

Git sees files in two states:

1. **Untracked** – The file exists in your working directory but is not yet tracked by Git.
2. **Tracked** – The file is being monitored by Git and will be included in commits.
   ○ Unmodified (No changes since last commit)
   ○ Modified (Changes exist but not yet staged)
   ○ Staged (Changes are marked for commit)

Example:
Imagine you have a project folder with a file named index.html.

- When you create it, Git does not track it yet.
- Once you run `git add index.html`, Git starts tracking it.

- When you modify it, Git knows there is a change but won't save it until you commit.

2️⃣ The Three Git Areas (How Changes Move Through Git)

1.Working Directory : Where you edit files (your local project).
2.Staging area :Temporary holding space for changes before they are committed.
3.Repository : The permanent storage for all commits and history.

🔹 Git Workflow (Step by Step)

1. Modify files in the working directory.
2. Stage changes with `git add` (moves them to the staging area).
3. Commit the changes with `git commit` (moves them into the repository).
4. Push to a remote repository (e.g., GitHub) if needed.

How Git Handles Branching & Merging

Git's branching system is lightweight and fast because branches are just pointers to commits.

🛠️ Creating a Branch

```
git branch feature-branch
```

This does not create a new copy of files—it simply creates a pointer to the current commit.

🔄 Merging a Branch

Once you finish working on a feature, you can merge it into the main branch:

```
git checkout main
git merge feature-branch
```

Git will try to combine the changes automatically.

🔧 Hands-on Example: Tracking Changes in Git

Step 1: Set Up a Git Repository

Open your terminal and run:

```
mkdir my-project
cd my-project
git init
```

🚀 This creates a new Git repository inside the `my-project` folder.

Step 2: Create a File & Check Git Status

Now, create a file:

```
echo "Hello, Git!" > hello.txt
```

```
Check the status of the repository:

git status
```

```
📌 Output:

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

📝 Git sees `hello.txt` but hasn't started tracking it yet.

Step 3: Start Tracking the File (Staging Area)

Add the file to the staging area:

```
git add hello.txt
```

Check the status again:

```
git status
```

```
📌 Output:

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   hello.txt
```

🔹 Now `hello.txt` is staged, meaning it's ready to be saved in the repository.

Step 4: Commit the Change (Save the Snapshot)

Now, commit the file:

```
git commit -m "Added hello.txt"
```

```
📌 Output:

[main (root-commit) d3f1a4c] Added hello.txt
 1 file changed, 1 insertion(+)
 create mode 100644 hello.txt
```

🔹 This saves a snapshot of the project with a unique commit ID.

Step 5: Modify the File & Check Changes

```
Now, edit hello.txt:

echo "Hello, Git! How are you?" >> hello.txt
```

Check what has changed:

```
git diff
```

📌 Output (example):

```
+Hello, Git! How are you?
```

🔹 Git detects that the file was modified.

Step 6: Stage & Commit Again

```
git add hello.txt
git commit -m "Updated hello.txt with a greeting"
```

📌 Now we have two commits in our history!

Step 7: Check the Commit History

Run:

```
git log --oneline
```

📌 Output (example):

```
a1b2c3d Updated hello.txt with a greeting
d3f1a4c Added hello.txt
```

🔹 Each commit has a unique hash and a message.