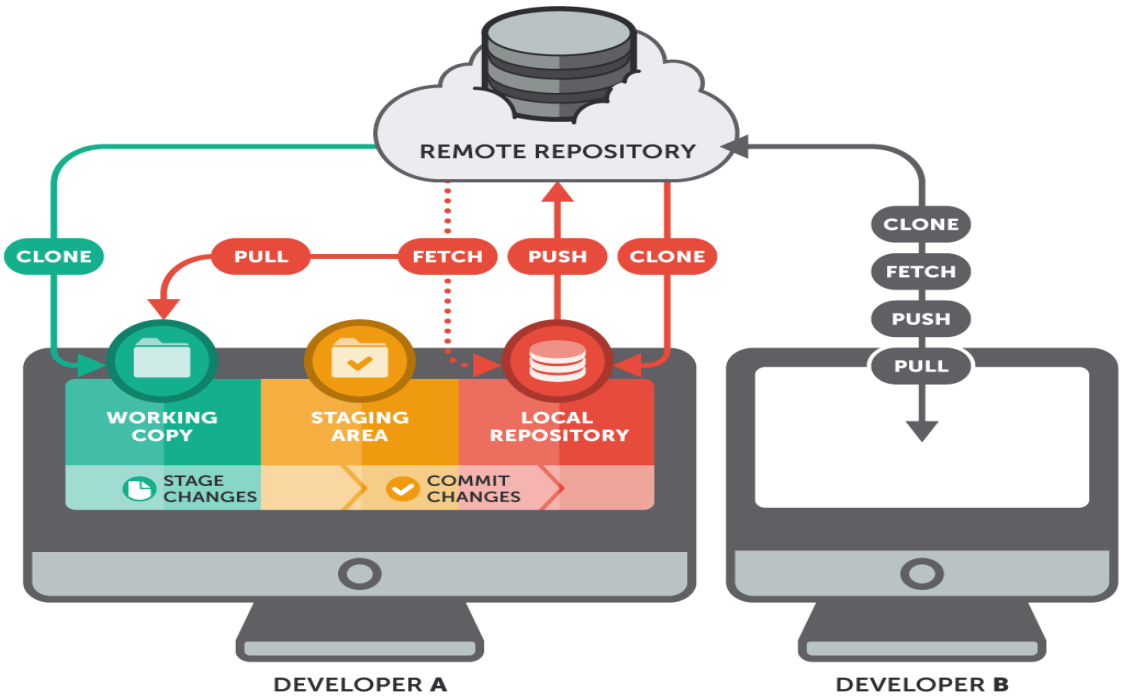
# Git (VERSION CONTROL SYSTEM)

## Introduction

### What is Git?

Git is a distributed version control system widely used for tracking changes in source code during software development. It allows multiple developers to collaborate on projects by providing mechanisms to manage and merge changes efficiently. Git stores the entire history of a project, enabling users to revert to previous versions, compare changes, and branch off to work on new features independently. It's known for its speed, flexibility, and robustness, making it a cornerstone tool in modern software development workflows.




### Why Use Git?

1. **Version Control**: Git tracks changes made to files over time, allowing developers to revert to previous versions if needed. This feature ensures that the entire history of the project is preserved.

2. **Collaboration**: Git enables multiple developers to work on the same project simultaneously. Each developer can work on their own branch, making changes independently, and then merge their work back into the main branch when ready.

3. **Branching and Merging**: Git makes branching and merging of code effortless. Developers can create separate branches to work on specific features or fixes without affecting the main codebase. Once their work is complete, they can merge their changes back into the main branch.

4. **Distributed Development**: Git is a distributed version control system, meaning that each developer has a local copy of the entire repository. This allows developers to work offline and then synchronize their changes with a central repository when they're back online.

5. **Traceability**: Git provides detailed logs of changes made to the codebase, including who made the changes and when. This traceability helps in debugging issues and understanding the evolution of the codebase over time.

6. **Backup and Disaster Recovery**: By using Git, developers have a reliable backup of their codebase. Even if a developer's local copy is lost or corrupted, they can recover the code from the central repository.

7. **Open Source and Community Suppor**t: Git is open source and has a vast community of users and contributors. This means there are numerous resources, tutorials, and tools available to help developers learn and use Git effectively.

# Getting Started

## Installing Git

**Download Git**: If Git is not installed, you can download the installer for your operating system from the official Git website: https://git-scm.com/downloads

**Verify Installation**: After the installation is complete, open a terminal or command prompt again and type:

**git --version**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice
$ git --version
git version 2.39.1.windows.1
```

This command should now display the Git version number, confirming that Git has been successfully installed.

## Configuring Git

After installing Git, you may want to configure your username and email address, which will be associated with your Git commits. You can do this by running the following commands in the terminal or command prompt:

**git config --global user.name "Your Name"**

**git config --global user.email "your.email@example.com"**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice
$ git config --global user.name "Rahul Kumar Paswan"

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice
$ git config --global user.email "rahulkumarpaswan123@gmail.com"
```

This command will display all global configuration settings for Git on your system.

**git config --global --list**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice
$ git config --global --list
user.name=Rahul Kumar Paswan
user.email=rahulkumarpaswan123@gmail.com
```

## Initializing a Repository

Initializing a repository refers to the process of setting up a directory as a Git repository, thereby enabling Git to start tracking changes to files within that directory.

When you initialize a repository, Git creates a hidden subdirectory within the directory called ".git". This subdirectory contains all the metadata and configuration files that Git uses to manage the repository, including information about commits, branches, tags, and remote repositories.

You should initialize a repository in a directory when you want to start using version control for the files within that directory. Here are some reasons why you might want to initialize a repository.

**Before Initializing a Repository**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice
$ ls -a
./  ../
```

**After Initializing a Repository**

**git init**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice
$ git init
Initialized empty Git repository in C:/Users/acer/OneDrive/Documents/Rahul_Git_Practice/.git/

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ ls -a
./  ../  .git/
```

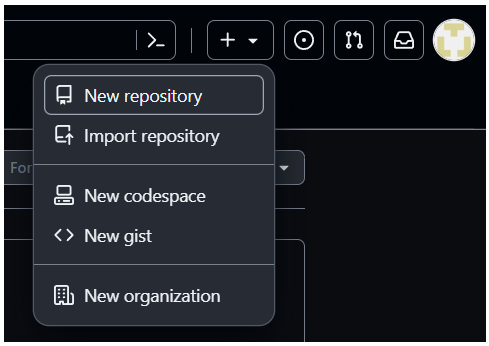
# GitHub Integration

## Creating a Repository

Log in to your GitHub account.

Click on the "+" icon in the top-right corner and select "New repository." Enter a name for your repository.

Click the "Create repository" button to finalize the creation of your new repository.



# Basic Git Commands

Before starting to work with a Git repository on your local machine, it's essential to establish a connection between your local environment and the remote repository hosted on GitHub.

**Check Existing Remotes**: It's good practice to verify if any remote repositories are already linked to your local Git project.

**git remote -v**

This command will list any existing remote repositories and their corresponding URLs.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ git remote -v
```

**Connect to Remote Repository**: If no remote repository is currently linked or you need to connect to a new one, you can do so using this command.

**git remote add <remote-name> <remote-url>**

**git remote add origin <repo-link>**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ git remote add origin https://github.com/Rahul-Kumar-Paswan/Git_Practice.git

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ git remote -v
origin  https://github.com/Rahul-Kumar-Paswan/Git_Practice.git (fetch)
origin  https://github.com/Rahul-Kumar-Paswan/Git_Practice.git (push)
```

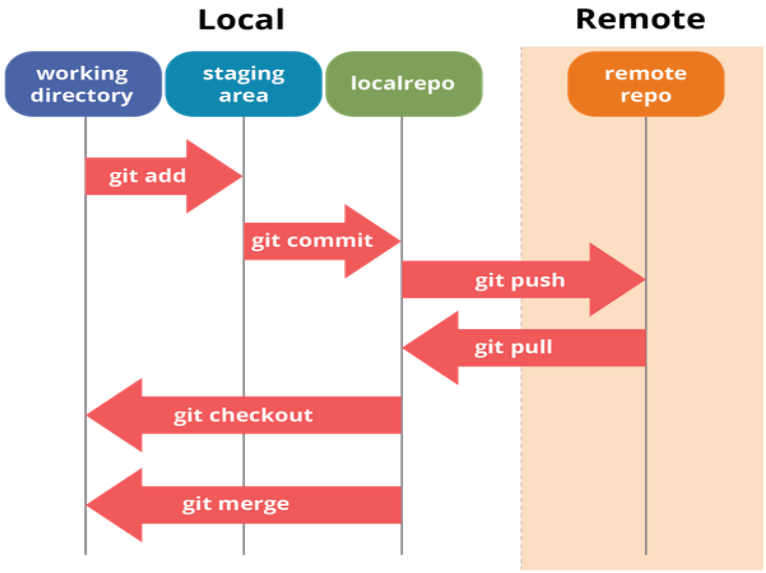
**Change Remote Repository :**

If you need to change the remote repository URL later, you can use the following command, replacing ‘new-repo-link’ with the URL of the new repository:

**git remote set-url origin new-repo-link**

**Rename the remote:** If you want to keep the existing URL but change the remote name, you can use **git remote rename <old_name> <new_name>.**

**Delete and re-add:** If you need to completely remove and re-add the remote with the new URL, you can use **git remote remove origin** followed by **git remote add origin <new_repo_link>.**




**Working Directory:** The working directory is the directory on your local machine where you're actively working on your project. It contains all the files and directories associated with your project. When you make changes to files in this directory, Git detects these modifications as "unstaged changes."

**Staging Area (Index):** The staging area is an intermediate area between your working directory and your local repository. It's essentially a snapshot of your working directory at a particular point in time. When you modify files in your working directory, you can selectively choose which changes to include in your next commit by adding them to the staging area. This allows you to review and organize your changes before making them a part of your project's history.

**Local Repository:** The local repository is where Git stores all the committed changes and project history on your local machine. When you commit changes from the staging area, they are saved in the local repository, and a new commit object is created with a unique identifier (hash). This allows you to track the history of your project and revert to previous states if needed. Each Git repository typically consists of a working directory, a staging area, and a local repository.

**Remote Repository:** The remote repository is a version of your project hosted on a remote server, such as GitHub, GitLab, or Bitbucket. It serves as a centralized location where multiple developers can collaborate on the same project. You can push your local commits to the remote repository to share your changes with others, and you can also pull changes from the remote repository to update your local copy with the latest modifications made by other collaborators. The remote repository acts as a backup and a centralized hub for collaboration in distributed version control systems like Git.

**git status:** Show the current status of the repository, including tracked/untracked files and changes to be committed.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ git status
On branch master

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Create a file main.py

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ touch main.py

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ vi main.py
```

After creating a new file, main.py, Git recognizes this as an untracked file. It means Git is not currently monitoring changes in this file.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (master)
$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)
```

Running git status again, you'll notice that main.py appears under the list of "untracked files." This indicates that Git acknowledges the existence of this new file but hasn't yet started tracking its changes.

**git add [file]**: Add a file to the staging area. You can also use **git add .** to add all files in the current directory.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git add main.py

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git status
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   main.py
```

running **git status** again will reflect that **main.py** has transitioned from being an untracked file to a tracked file. This means that Git is now monitoring changes in **main.py**, and any modifications made to it will be included in the next commit.

**git commit:** Commit staged changes with a descriptive message.

**git commit -m "commit message"**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git commit -m "first commit"
[main (root-commit) c1cf19a] first commit
 1 file changed, 1 insertion(+)
 create mode 100644 main.py
```

After executing the **git commit** command with a descriptive message like **git commit -m "first commit"**, the staged changes are committed to the repository. This means that the changes you added to the staging area using **git add** are now permanently stored in the version history of the repository.

You can verify this by using the **git log** command, which displays a chronological list of commits, including the latest one you just made.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git log
commit c1cf19a23b969d6376468922f24dfa0baabb1fb3 (HEAD -> main)
Author: Rahul Kumar Paswan <rahulkumarpaswan123@gmail.com>
Date:   Sat Mar 2 16:11:58 2024 +0530

    first commit
```

Upon running **git status** again after the commit, you'll observe that there are no longer any changes staged for commit. Git will inform you that the working directory is clean, indicating that all modifications have been committed to the repository.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git status
On branch main
nothing to commit, working tree clean
```

**git push:** Upload local repository content to a remote repository.
**git push -u origin <branch name>**
Using **git push -u origin <branch name>** allows you to upload the content of your local repository to a remote repository. This command ensures that your local changes are synchronized with the remote repository, making them accessible to collaborators.

Upon executing this command, you'll observe that your files are transferred to the remote repository, making them available to others who have access to the remote repository.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git push -u origin main
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 722 bytes | 240.00 KiB/s, done.
Total 9 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/Rahul-Kumar-Paswan/Git_Practice.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

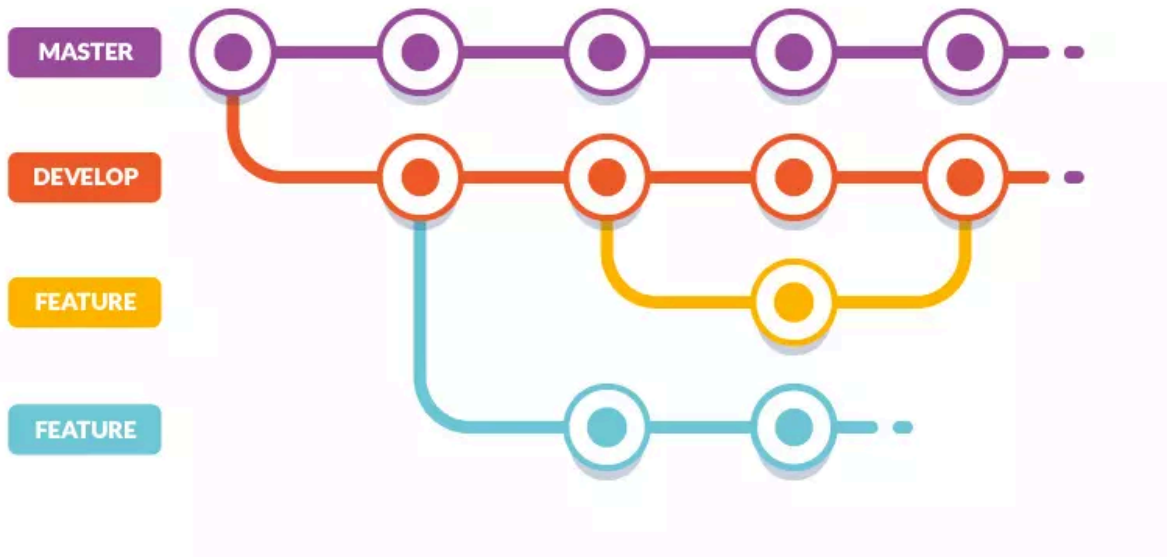
# Branching

Git branching is a powerful feature that allows you to diverge from the main line of development (usually called the “master” or “main” branch) and work on different features, fixes, or experiments in isolation.

**What is a Branch?:**

- A branch in Git is essentially a lightweight movable pointer to a commit. Each branch represents an independent line of development within a repository.

- The main branch, often named `master` (though it's common now to use `main`), is the default branch that typically represents the stable version of the project.




**Creating a Branch:**

- You can create a new branch using the `**git branch <branch-name>**` command. This creates a new branch pointing to the same commit as the current branch.

- Alternatively, you can use `**git checkout -b <branch-name>**` to create a new branch and switch to it in a single command.

**Switching Between Branches:**

- You can switch between branches using the `**git checkout <branch-name>**` command. This updates the working directory to match the version of the project stored in the specified branch.

**Working on a Branch:**

- Once you're on a branch, you can make changes to your project just like you would on the main branch (`master` or `main`).

- These changes are isolated to the specific branch you're working on, allowing you to work on multiple features simultaneously without affecting the main branch.

**Committing Changes:**

- After making changes on a branch, you can commit them using `**git commit**`. These commits are unique to the branch and do not affect other branches until you merge them.

**Deleting Branches:**

- After you've finished working on a branch and merged its changes into the main branch, you can delete the branch using `**git branch -d <branch-name>**`. This removes the branch from your local repository.

- Use `**git push origin --delete <branch-name>**` to also delete the branch from the remote repository.

**Merging Branches:**

- Merging is the process of combining changes from one branch into another. You can merge a branch into another branch using `**git merge <branch-name>**`.

- When you merge a branch into another, Git integrates the changes from the source branch into the target branch, creating a new merge commit if necessary.

**Resolving Conflicts:**

- Conflicts may arise during the merge process if changes conflict with each other. Git will prompt you to resolve these conflicts manually by editing the affected files and choosing which changes to keep.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git branch feature1

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git branch
  feature1
* main

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git checkout -b feature2
Switched to a new branch 'feature2'

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (feature2)
$ git branch
  feature1
* feature2
  main

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (feature2)
$ git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git branch -d feature2
Deleted branch feature2 (was 2de3cd8).
```

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git merge feature1
Updating 2de3cd8..e8457bf
Fast-forward
 hello.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 hello.py
```

# Collaboration

## Cloning a Repository

**git clone**: Clone a repository from a remote URL to your local machine. Simply download code to the local machine.
**git clone <repo-url>**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/git_clone_practice
$ git clone https://github.com/Rahul-Kumar-Paswan/Git_Practice.git
Cloning into 'Git_Practice'...
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 9 (delta 0), reused 9 (delta 0), pack-reused 0
Receiving objects: 100% (9/9), done.
```

After executing the command, you can confirm the successful download by using the **`ls`** command, which will display the newly cloned repository on your local machine.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/git_clone_practice
$ ls -a
./  ../  Git_Practice/

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/git_clone_practice
$ cd Git_Practice/

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/git_clone_practice/Git_Practice (main)
$ ls
main.py
```

## Pushing Changes

**git push:** Upload local repository content to a remote repository.
**git push -u origin <branch name>**

See above for explanation i.e on page no 10.

## Pulling Changes

**git pull:** Fetch changes from a remote repository and merge them into the current branch.

`**git pull**` is a command used to fetch changes from a remote repository and integrate them into the current branch of your local repository. After fetching the changes, Git automatically merges them into your current local branch.

If there are no conflicts between the changes fetched from the remote and your local changes, Git performs a fast-forward merge, updating your local branch to reflect the changes from the remote repository. However, if there are conflicting changes, Git will prompt you to resolve these conflicts manually.

Pulling changes is essential for maintaining synchronization between your local repository and the remote repository. It ensures that you're working with the latest version of the codebase and enables seamless collaboration with other team members while preventing divergence in project history.

For instance, if your team member has made changes to the same lines of code in the repository without your knowledge, attempting to push your changes could lead to conflicts. Therefore, it's necessary to pull the changes first and merge them into your local branch to resolve any conflicts before pushing your own changes. See the below image for explanation.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git push -u origin main
To https://github.com/Rahul-Kumar-Paswan/Git_Practice.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/Rahul-Kumar-Paswan/Git_Practice.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 968 bytes | 56.00 KiB/s, done.
From https://github.com/Rahul-Kumar-Paswan/Git_Practice
   9f18a80..65021c6  main       -> origin/main
Merge made by the 'ort' strategy.
 app.py | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 app.py
```

# Advance Topics

## Rebasing

**Git rebase** is a powerful command used to integrate changes from one branch into another. It works by transferring commits to a new base commit, which can help maintain a cleaner, more linear project history. Here's an overview of how rebase works.

### Understanding Git Rebase

Imagine you're working on a feature branch called `**feature1**` that diverged from the `**main**` branch a few commits ago. Meanwhile, other commits have been made to the `**main**` branch. You want to incorporate those latest changes from `**main**` into your `**feature1**` branch.

Using `**git rebase**`, you can **"replay"** your feature branch commits on top of the `**main**` branch commits. This process can potentially result in a more straightforward history than merging, as it creates the appearance that all changes were made in a linear sequence, even if they were originally developed in parallel.

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git log --graph --oneline --decorate --all
* bd17bf9 (HEAD -> main, origin/main) rebase second commit from main
| * 6744f5f (origin/feature1, feature1) rebase first commit
|/
* e8457bf first commit on feature1
*   2de3cd8 Merge branch 'main' of https://github.com/Rahul-Kumar-Paswan/Git_Practice
|\
| * 65021c6 Create app.py
* | 9891c84 second
|/
* 9f18a80 first commit
* ebd906e check
* c1cf19a first commit
```

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git rebase feature1
Successfully rebased and updated refs/heads/main.

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git log --graph --oneline --decorate --all
* 4e64f92 (HEAD -> main, feature1) rebase first commit
* bd17bf9 (origin/main) rebase second commit from main
| * 6744f5f (origin/feature1) rebase first commit
|/
* e8457bf first commit on feature1
*   2de3cd8 Merge branch 'main' of https://github.com/Rahul-Kumar-Paswan/Git_Practice
|\
| * 65021c6 Create app.py
* | 9891c84 second
|/
* 9f18a80 first commit
* ebd906e check
* c1cf19a first commit
```

**Why Use Rebase?**

**- Cleaner project history:** Rebasing can result in a more linear and readable history.

**- Avoiding unnecessary merge commits:** Rebasing can eliminate the merge commits that occur when you `**git merge**` branches, making your project history cleaner and easier to follow.

## Git diff

`**git diff**` is a versatile command used to show differences between various Git entities such as commits, branches, files, and more. It's particularly useful for reviewing changes before committing them, comparing branches, or examining the changes made between commits.

**1. Unstaged Changes:** To see the differences in files that have been modified but not yet staged, you can simply run:

**git diff**

**2. Staged Changes:** To see what has been staged (with `git add`) but not yet committed, you can use:

**git diff --cached** or **git diff --staged**

You can compare two commits by specifying their commit hashes:

**git diff <commit1> <commit2>**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git diff 8f9cfc8 2df4cf0
diff --git a/main.py b/main.py
index b27fcde..0b1b797 100644
--- a/main.py
+++ b/main.py
@@ -1,4 +1,4 @@
 print("Hello Rahul!!")
 print("trying to push without pulling from remote repo")
 print('trying to do rebase --- feature1 branch')
-print('trying to do rebase --- feature1 branch')
\ No newline at end of file
+print('trying to do rebase --- feature1 branch ')
\ No newline at end of file
```

To see the differences between two branches:

**git diff <branch1> <branch2>**

## Git stash

`**git stash**` is a command used to temporarily shelve (or stash) changes you've made to your working directory so you can work on a different task. Your changes are saved in a stack-like structure (where you can have multiple stashes), and you can apply or remove them later. This feature is particularly useful when you need to quickly switch contexts without committing incomplete work.

**1. Stashing Changes:** To stash your current changes, simply run:

**git stash**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git status
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   main.py

no changes added to commit (use "git add" and/or "git commit -a")

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git checkout feature1
error: Your local changes to the following files would be overwritten by checkout:
        main.py
Please commit your changes or stash them before you switch branches.
Aborting

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git stash
Saved working directory and index state WIP on main: 2df4cf0 changing main.py

acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git checkout feature1
Switched to branch 'feature1'
Your branch is up to date with 'origin/feature1'.
```

**2. Popping a Stash:** If you want to apply the most recent stash and remove it from the stash list, you can use:

**git stash pop**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git stash pop
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   main.py

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (a24194e4d082cc64b775a35b26acc8192578d8ea)
```

## Git revert

`**git revert**` is a command used to create a new commit that undoes the changes made by a previous commit or commits. This is a safe way to undo changes, as it doesn't alter the project's history. Instead, `**git revert**` generates a new commit that reverses the effect of one or more earlier commits.

To revert a specific commit, you need the commit's hash ID. You can find this ID via `**git log**`. Once you have the commit's hash, you can revert it as follows:

**git revert <commit-hash>**

```
acer@LAPTOP-C2T5B8T3 MINGW64 ~/OneDrive/Documents/Rahul_Git_Practice (main)
$ git revert 8b5e483
[main f48505b] Revert "abc12345"
 1 file changed, 1 insertion(+), 1 deletion(-)
```

For example, if the commit hash is `abc1234`, you would run:

**git revert abc1234**

This command creates a new commit on top of the current branch with the changes from `abc1234` undone.

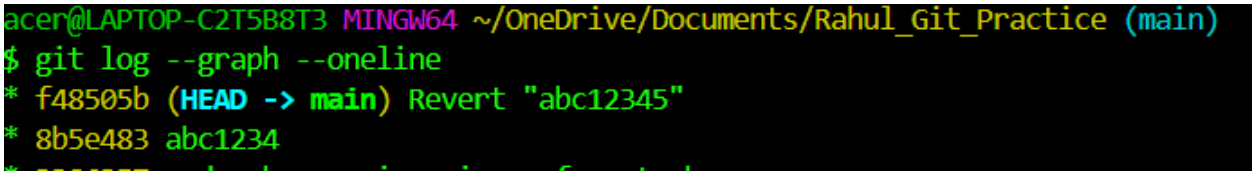
## Git reset

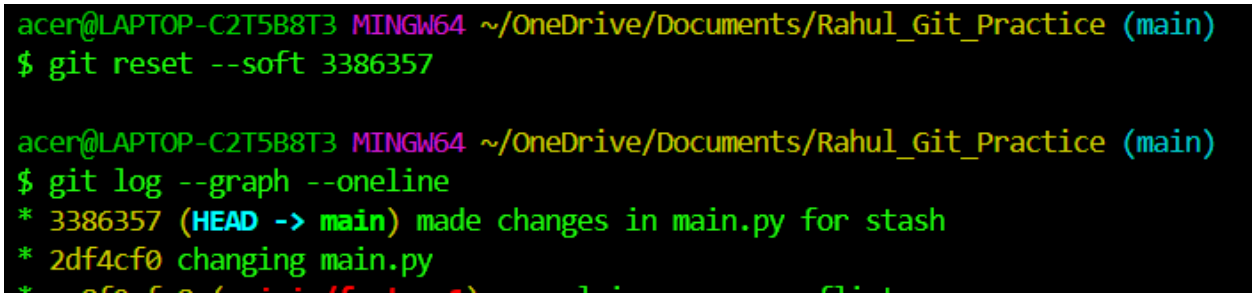
**`git reset`** is a command in Git that is used to undo changes in a repository's history to a specified state. It can affect the staging area (index), the working directory, and the commit history, depending on the options used. `git reset` operates on your current branch and can change the branch's tip to a specified commit, effectively removing commits that come after it in the history.

**`git reset`** can be used with three different modes, which affect the working directory and staging area differently:

**1. `--soft`:** This mode does not touch the staging area or the working directory. It only moves the HEAD to the specified commit. Changes from the commits ahead of the reset point will be staged.

**git reset --soft <commit>**




**2. `--mixed` (default):** Resets the index but not the working tree. Changes are kept in your working directory but not staged.

**git reset --mixed <commit>** Or **git reset <commit>**

**3. `--hard`:** Resets the index and working tree. Any changes to tracked files in the working directory since the specified commit are discarded.

**git reset --hard <commit>**

- Undo the last commit, keep changes: To undo the last commit and keep the changes in your working directory:

**git reset --soft HEAD~1**