**Git and GitHub Study**

****

**Aravind U R**

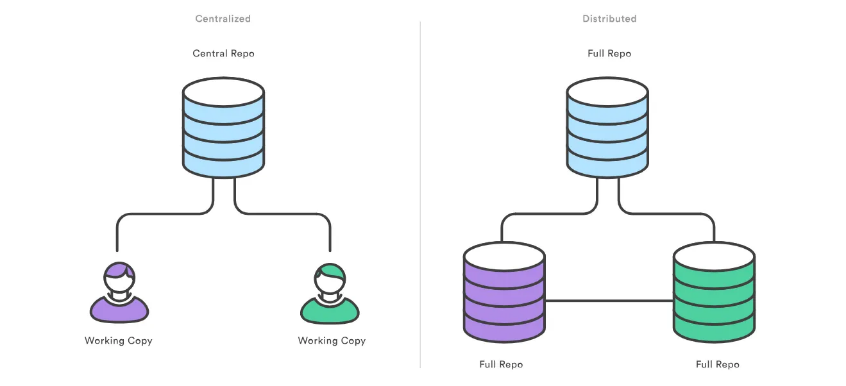
**Ref: Date with Git on Brototype youtube channel**

**Version Control System**

A **Version Control System (VCS)** is a tool used to manage changes to source code or any set of files over time, enabling multiple users to work on projects collaboratively. It tracks every modification, making it easier to revert to previous versions, track progress, and manage concurrent development without conflicts.

**Types of Version Control Systems:**

1. **Centralized Version Control Systems (CVCS):**
   * One centralized server holds the repository, and multiple clients check out files from this central place.
   * Example: Subversion (SVN)
   * **Pros**: Centralized control, easy to manage.
   * **Cons**: Server failure may cause loss of data; limited to server availability.
2. **Distributed Version Control Systems (DVCS):**
   * Every user has a complete copy of the repository, including its history.
   * Examples: Git, Mercurial
   * **Pros**: Work offline, faster operations, better handling of branching and merging.
   * **Cons**: Initial learning curve, more disk space required.



**Summary Table:**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **CVCS** | **DVCS** |
| **Repository** | Single, centralized repository | Each user has a full repository copy |
| **Work Offline** | Limited (must be connected) | Full offline support |
| **Speed** | Slower (network dependent) | Faster (local operations) |
| **Collaboration** | Centralized, linear model | Decentralized, flexible |
| **Backup/Redundancy** | Less redundant, server-dependent | High redundancy, local copies |
| **Branching and Merging** | Complex, often avoided | Easy, frequently used |
| **Security** | Centralized control | Decentralized, harder to control |

**Key Advantages of DVCS Over CVCS**

**DVCS**, especially Git, has become the go-to version control system in modern development due to its flexibility, speed, resilience, and ability to handle complex workflows. It enables better collaboration, especially in large, distributed teams and open-source projects, which is why it is preferred over **CVCS** in most development environments today.

|  |  |  |
| --- | --- | --- |
| **Aspect** | **DVCS Advantage** | **CVCS Limitation** |
| **Offline Work** | Full offline support (commits, branches) | Requires a connection for most operations |
| **Speed** | Fast local operations (commits, history, merges) | Network-dependent, slower commits and operations |
| **Branching & Merging** | Easy, efficient, and lightweight | More complex, prone to conflicts |
| **Collaboration** | Flexible, parallel workflows | Linear, restrictive workflows |
| **Resilience** | Every user has a full backup of the project | Centralized, single point of failure |
| **Large Projects** | Scales well for large teams and contributors | Less scalable, more bottlenecks |
| **Open-Source Support** | Ideal for community-driven projects | Not designed for decentralized contributions |
| **Code Review & Collaboration** | Integrated with modern platforms like GitHub | Fewer collaboration features |

**Git: **

**Git** is a distributed version control system (DVCS) that is widely used in software development to manage and track changes in source code. Developed by **Linus Torvalds** in 2005 for the development of the Linux kernel, Git has become one of the most popular and powerful tools for version control in both open-source and enterprise projects.

**Key Features of Git:**

1. **Distributed System:**
   * Every developer has a complete local copy of the entire project history, including branches, commits, and files.
   * This decentralized structure allows developers to work offline, committing and creating branches locally without needing a connection to a central repository.
2. **Fast Performance:**
   * Git is designed to be very fast, with most operations (like commits, diffs, and branches) happening locally on the developer's machine.
3. **Branching and Merging:**
   * Git makes branching and merging easy and efficient. Branching is lightweight, enabling teams to create separate branches for features, bug fixes, or experiments without affecting the main codebase.
   * Merging branches back into the main project is also streamlined, with Git providing advanced merge conflict resolution tools.
4. **Data Integrity:**
5. **Collaboration and Integration:**
   * Git allows multiple developers to work on a project simultaneously, managing changes in different branches. Tools like GitHub, GitLab, and Bitbucket are built around Git and provide features like pull requests, code reviews, and continuous integration (CI) pipelines.

**GitHub **

**GitHub** is a web-based platform built on **Git**, offering tools and services for version control, collaboration, and project management. It is one of the most popular platforms for hosting and managing Git repositories, especially in the open-source community. GitHub provides developers with a centralized space to store, track, and collaborate on code, while also integrating project management, issue tracking, and continuous integration/continuous delivery (CI/CD) features.

**Key Features of GitHub:**

1. **Git Repositories**:

* GitHub allows developers to host **Git repositories** in the cloud, where they can manage, share, and collaborate on code.
* **Public repositories** are free and visible to everyone, making GitHub a central hub for open-source projects. **Private repositories** can be used for personal or commercial projects.

1. **Collaboration Tools**:

* **Pull Requests**:

Developers can submit changes to a project by creating a pull request. This feature allows others to review, discuss, and merge changes.

* **Forking**:

Forking allows users to create a personal copy of another user's repository. This is widely used in open-source projects, where contributors fork a repository, make changes, and then submit pull requests to merge their contributions back into the main project.

* **Code Reviews**:

GitHub facilitates peer reviews of code changes, enabling teams to comment on specific lines of code, suggest improvements, or approve changes.

1. **Continuous Integration/Continuous Delivery (CI/CD)**:

* **GitHub Actions**:

GitHub provides an integrated CI/CD service called **GitHub Actions**, which allows developers to automate

workflows, run tests, build code, and deploy applications

automatically based on triggers like pull requests or commits.

**Common uses include:**

* Automatically running tests when code is pushed.
* Building and deploying code to staging or production environments.
* Integrating with external services such as Slack or Docker.

1. **GitHub Pages**:

GitHub allows developers to host static websites directly from a GitHub repository. Using **GitHub Pages**, users can deploy a personal or project website for free, making it an excellent option for documentation or portfolios.

**Advantages of GitHub:**

* **Easy Collaboration**: GitHub simplifies working with teams, enabling developers to contribute to projects, review code, and merge changes with minimal friction.
* **Open-Source Focus**: GitHub has fostered a large open-source community, allowing developers to contribute to popular projects, discover new tools, and learn from each other.
* **Integrated Tools**: With features like GitHub Actions, GitHub Projects, and GitHub Pages, GitHub offers a suite of tools that streamline software development.
* **Code Visibility**: Public repositories allow anyone to view, use, and contribute to projects, which is essential for open-source software development.

Here is a table that outlines the key differences between **Git** and **GitHub**:

|  |  |  |
| --- | --- | --- |
| **Aspect** | **Git** | **GitHub** |
| **Definition** | A distributed version control system for tracking changes in source code. | A web-based platform that hosts Git repositories. |
| **Type** | Software (Version Control System) | Cloud-based platform and service for Git repositories |
| **Purpose** | Manages local and remote code versions, enabling branching, merging, and tracking history. | Facilitates collaboration on Git repositories with features like pull requests, issue tracking, and more. |
| **Installation** | Must be installed locally on the user's machine. | No installation needed; accessed via web browser. |
| **Functionality** | Handles source code version control on the command line; users can commit, branch, and merge. | Provides hosting for Git repositories with a user interface and additional project management tools. |
| **Collaboration** | Collaboration is manual (via push/pull with remote repositories). | Allows teams to collaborate on repositories with integrated tools like code review and task management. |
| **User Interface** | Command-line interface (CLI) and third-party GUIs. | Graphical user interface (GUI) available on the web and desktop apps. |
| **Storage** | Stores repositories on local machines or remote servers. | Stores repositories on GitHub's cloud servers. |
| **Access Control** | No built-in access control (access managed by repository owner on servers). | Provides access control (public or private repositories, user permissions). |
| **Backup** | Local backups managed by the user. | GitHub automatically backs up repositories on the cloud. |
| **Pricing** | Free and open-source. | Offers free and paid plans for repository hosting and features. |
| **Community** | Not specific to Git. | GitHub has a large, active developer community with open-source projects. |
| **Additional Features** | None. | Includes extra features like GitHub Actions (CI/CD), Pages (static websites), and Packages. |

**GitBash:**

For working on Git, we have used the Git bash here.

Git Bash is a command-line interface for Git that provides a Unix-like environment on Windows.

It's particularly useful for developers familiar with Unix-style commands, as it brings a set of tools commonly found in Bash (the Bourne Again Shell) to Windows.

**How to Install Git Bash:**

1. Download Git for Windows from the official Git website.
2. During installation, select **Git Bash** as the default terminal for Git operations.
3. Once installed, you can open Git Bash from the start menu or by right-clicking on the desktop and selecting "Git Bash Here."

**Git Basic Commands**

**To set the Author configuration:**

* **Set Author Configuration Globally**

This sets the author information for all repositories on your system:

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

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

* **Set Author Configuration locally**

This sets the author information only for the current repository:

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

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

Let’s start the Git basic commands;

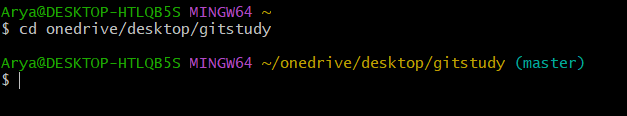
**1. git init**

**Initialize a Repository:**

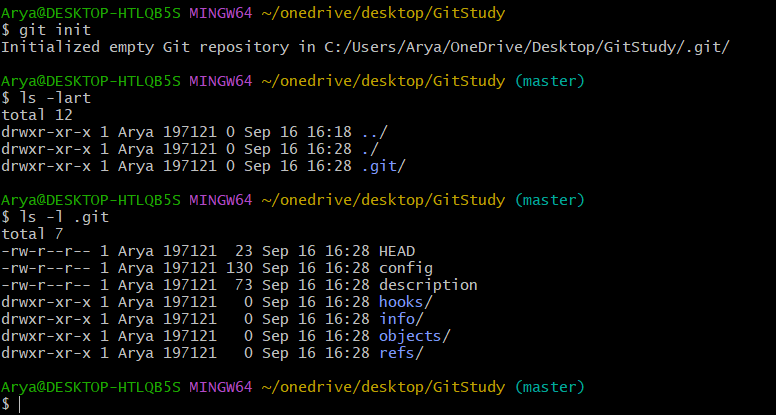
The **git init** command is used to initialize a new Git repository. When you run this command in a directory, it sets up the necessary files that Git needs to track changes in that directory.

Steps:

* **Navigate** to the directory where you want to create the repository:



* **Initialize** the Git repository by running:



While run the **git init** command, its initialised an empty repository in specified directory. And a hidden **.git** directory is created in the root of your project folder. This directory contains all of the configuration files and metadata that Git needs to track your project.

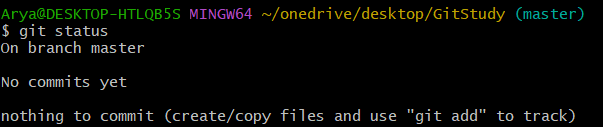
2. **git status**

**git status** is a Git command used to display the current state of the working directory and the staging area.

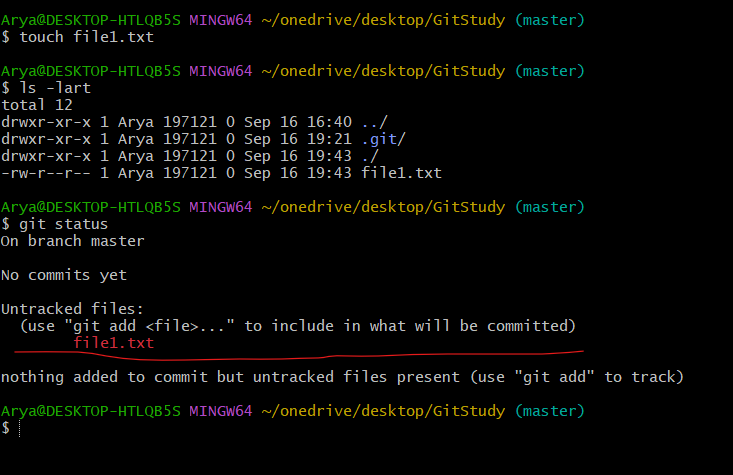
git status shows;

* **Changes to be committed** (staged changes): These are the files you’ve added to the staging area, ready to be committed.
* **Changes not staged for commit**: Files that have been modified but not yet staged for commit.
* **Untracked files**: Files that aren't tracked by Git yet.

git status before no changes:



Then we have added a file1.txt in folder;



Here we can see that; one untracked file file1.txt is added on the directory.

3. **git add**

The **git add** command adds changes in your working directory to the staging area (also called the "index") in Git. This prepares the changes to be included in the next commit.

Syntax of **git add**:

* Add a specific file:

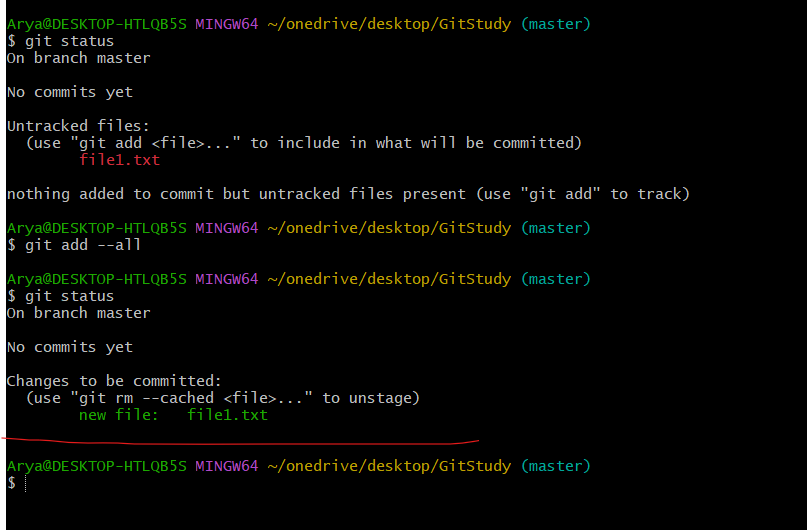
**git add <filename>**

* Add all changes in the current directory:

**git add --all**

or

**git add .**



Here we can see that see that the git status before and after git add command; the file1.txt is added to the staging area and ready for commit.

4. **git restore**

The **git restore** command is used to restore files in your working directory and staging area to a previous state.

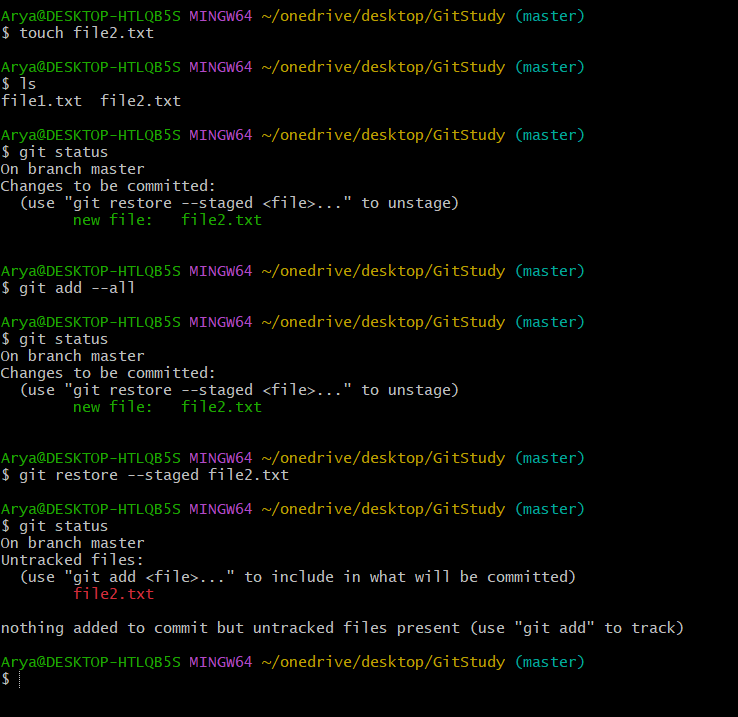
Syntax:

* **Restore a file to the staging area:**

**git restore --staged <filename>**

* **To restore a file to its state in a specific commit:**

**git restore --source <commit> <filename>**

****

Here we can see that, the file2.txt is restored from staged area.

5. **git commit**

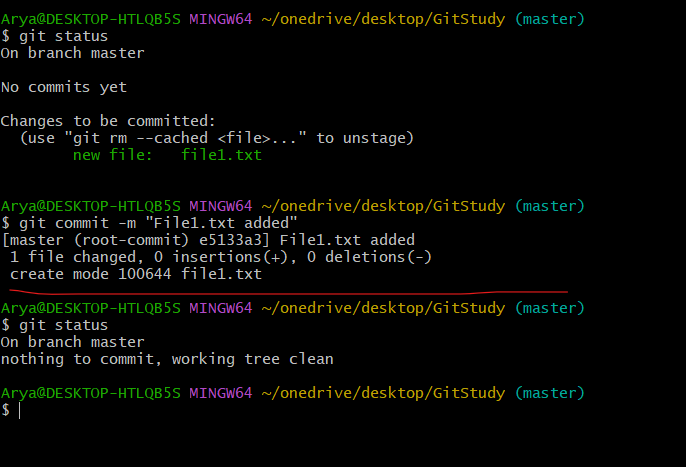
The **git commit** command saves changes that have been added to the staging area to the Git repository's history. Each commit represents a snapshot of your project at a specific point in time.

Syntax: **git commit -m "Your commit message"**

**Note:**

**Amend the previous commit**: If you made a mistake in your last commit (like forgetting to include a file or needing to change the message), you can amend it with

Syntax: **# git commit --amend -m "This is your new Git Message"**



This saves your changes in the Git history with a message explaining what the commit does. After this, the changes are saved locally and can be pushed to a remote repository.

6. **git log**

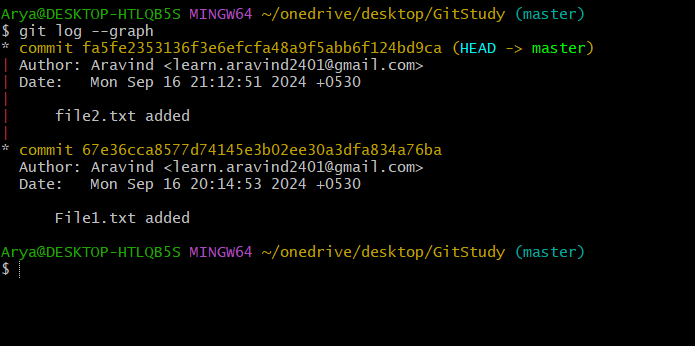
The **git log** command is used to view the commit history of a Git repository.

Output Details

* **Commit Hash (commit id)**: The unique identifier for the commit.
* **Author**: The person who made the commit.
* **Date**: The date and time when the commit was made.
* **Commit Message**: A description of the changes made in the commit.

**Graphical Representation**: Show a graphical representation of the commit history, useful for visualizing branch merges:

**git log --graph**

****

6. **git diff**

**git diff** is a powerful command that shows the differences between various states of your files in a Git repository.

To see differences between two specific commits are;

Syntax: git diff <commit1> <commit2>

**Git Branch**

In **Git**, a **branch** is essentially a lightweight pointer to a particular commit, which allows developers to work on separate versions of a project in parallel without interfering with each other's work. This enables flexibility in managing features, bug fixes, and releases while maintaining a clean and organized development workflow.



**Git HEAD and Master/Main**

**What is HEAD?**

* HEAD is a reference to the **current commit** that you’re working directory and staging area are based on. It typically points to the **latest commit** on the currently checked-out branch.

**In simpler terms, HEAD shows where you are in the commit history.**

**What is master or main?**

* master (or main) is the default branch in a Git repository. When a repository is initialized, Git used to create a branch named master by default.
* This branch typically contains the **production-ready code** or the **stable version** of the project.

**Git Merge**

Git Merge is the process of combining the changes from one branch into another.

This is a common operation in Git when developers want to integrate feature work, bug fixes, or other changes back into a stable branch like main or develop.

Types of Git Merges:

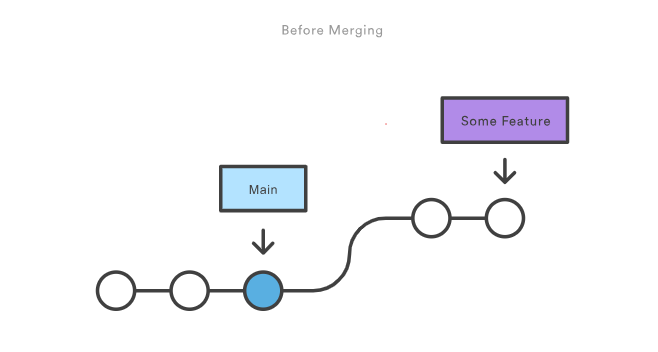
**1. Fast-Forward Merge**

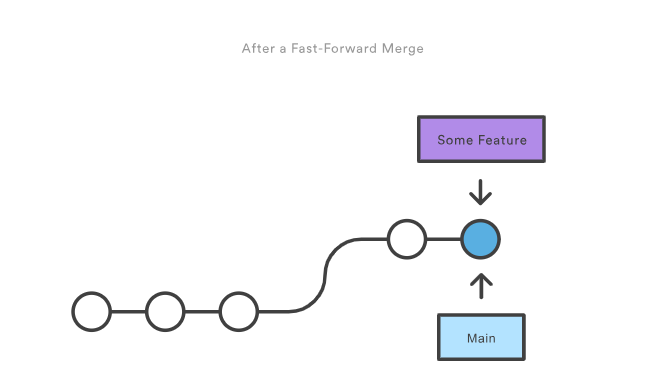
**What is it?**

* A fast-forward merge happens when the branch being merged in (e.g., a feature branch) is ahead of the target branch, and there are no divergent changes. In this case, Git simply moves the target branch pointer forward to the latest commit.

**Use Case**:

* When the main branch hasn't changed since the feature branch was created.
* It is used when there are no other changes on the base branch besides the commits from the feature branch.





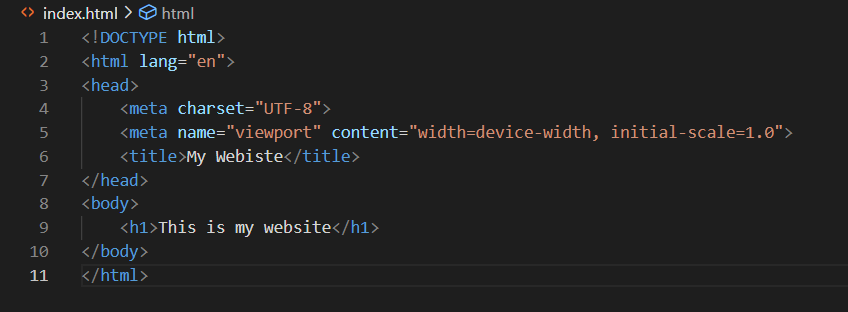
**Fast-Forward Merge** using a simple HTML code.

**Scenario:**

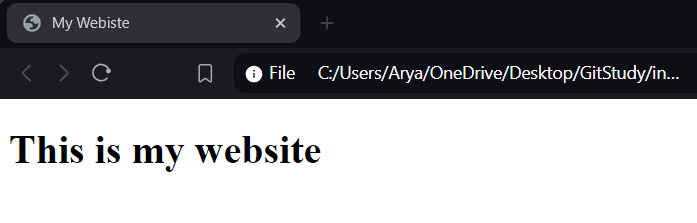
* The main branch has a basic HTML structure.
* A new feature-header branch is created to add a <header> to the HTML file.
* Since no other changes have been made to the main branch while working on the feature-header branch, Git performs a **fast-forward merge** when the feature branch is merged back into main.

**Step 1: Initial Setup (on master branch)**

Let’s start with a basic index.html file in the master branch.



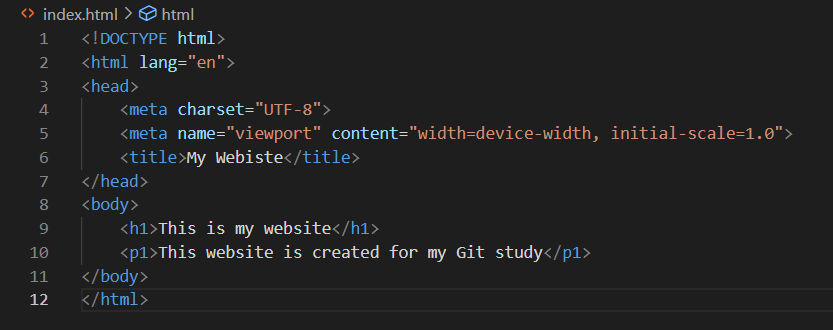
Result in browser



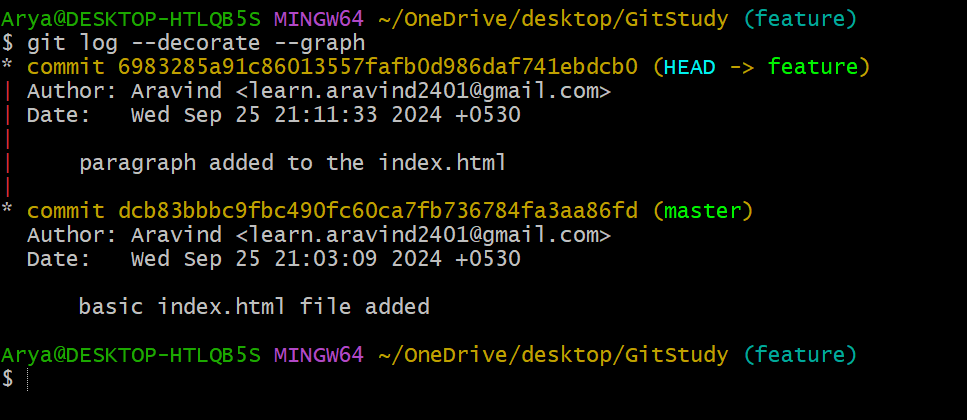
**Step 2: Create a New Branch (feature)**

Checkout to the feature branch

Edited the index.html with an addition paragraph and commited the change in the feature branch.



git log



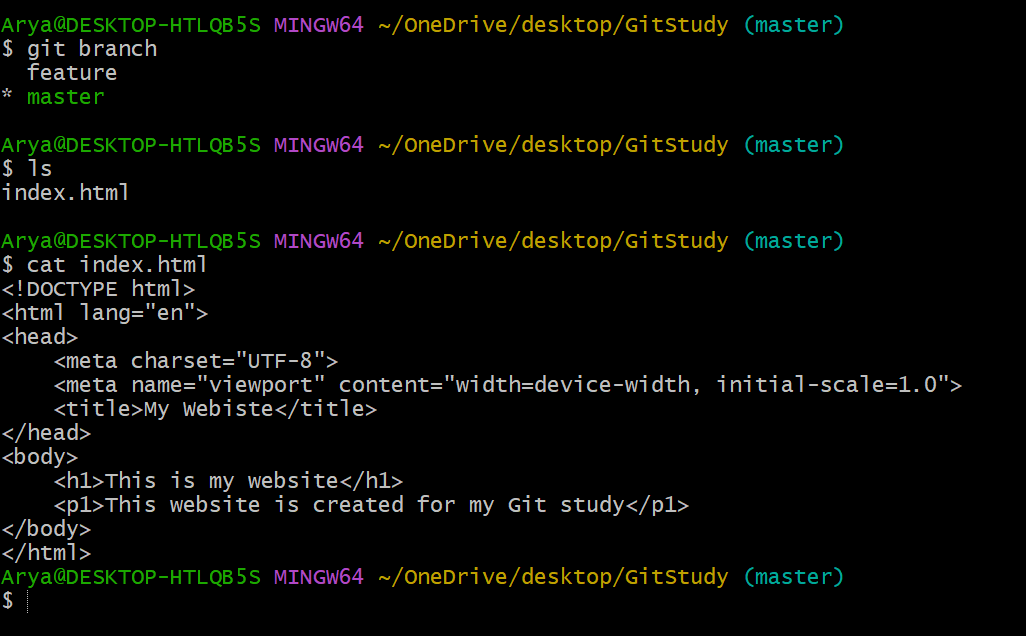
**Step 3: Perform a Fast-Forward Merge**

Since the main branch hasn't changed since you created the feature-header branch, Git can perform a fast-forward merge. No additional commits are needed.

checkout to the main branch

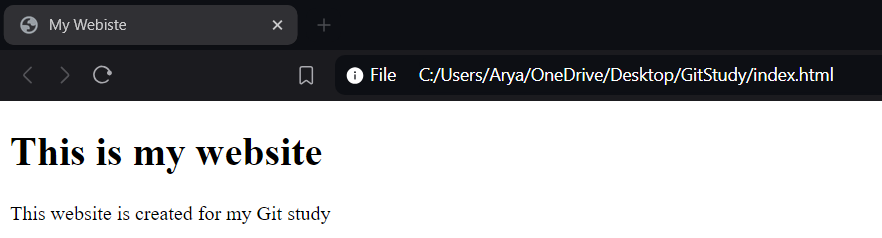
run the command **git merge feature**

This simply moves the master branch pointer to the latest commit in the feature-header branch.



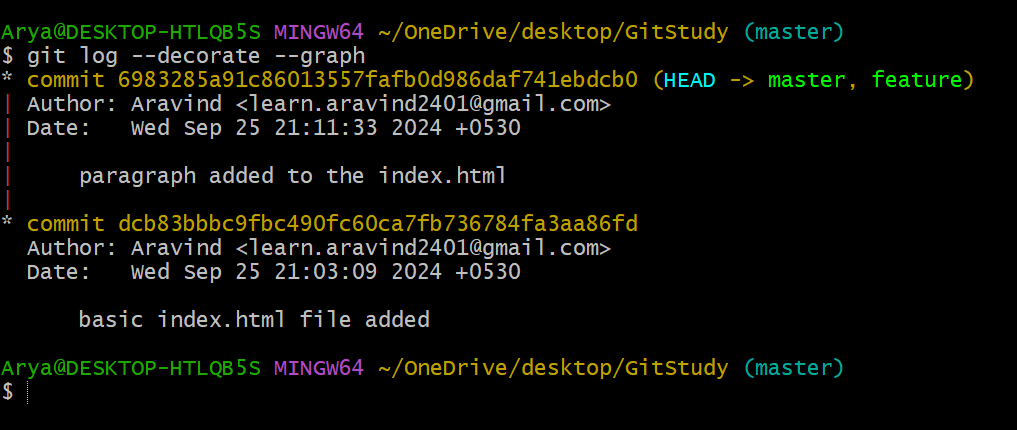
Here we can see that the changes we are commited in the feature branch merged to the master branch.

Result in browser



git log

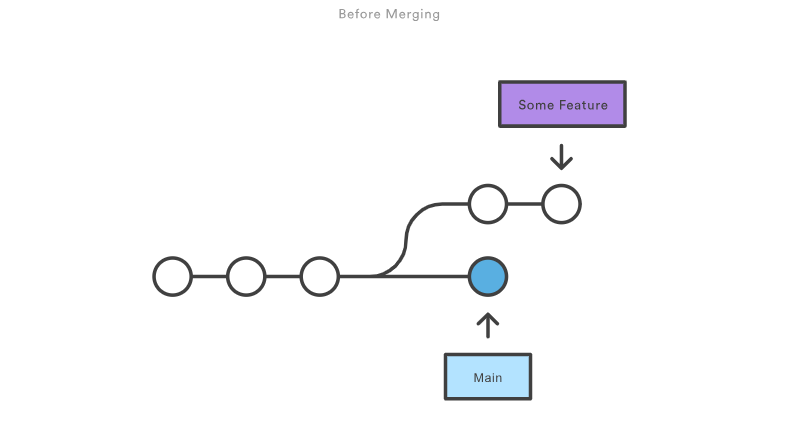
**Linear History**: The commit history remains linear, with no merge commit clutter.



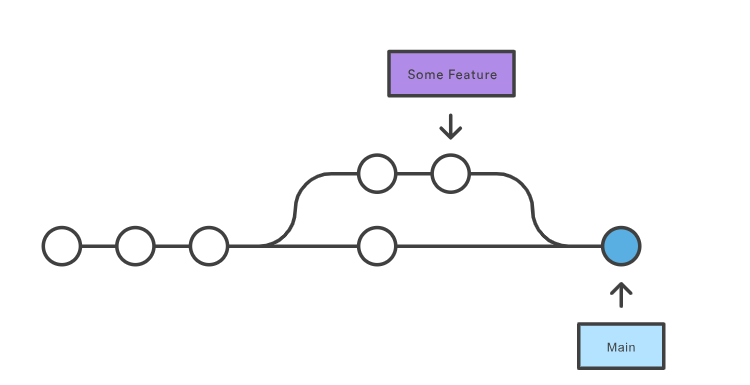
**2. Recursive Merge/ORT Merge (Optimized Recursive Tree Merge)**

A **recursive merge** is Git's default merge strategy used when two branches have diverged (i.e., both branches have new commits since they were last synced). It performs a **three-way merge**, where Git uses the common ancestor of both branches and combines changes from each branch into a new **merge commit**.

The **ort** (Optimized Recursive Tree) merge is a newer merge strategy introduced in Git 2.33 (August 2021). It’s a replacement for the recursive merge strategy, designed to improve performance and clarity in the merging process. While it works similarly to the recursive merge strategy, **ort** is more efficient and produces cleaner merges, especially in complex cases with many file changes.



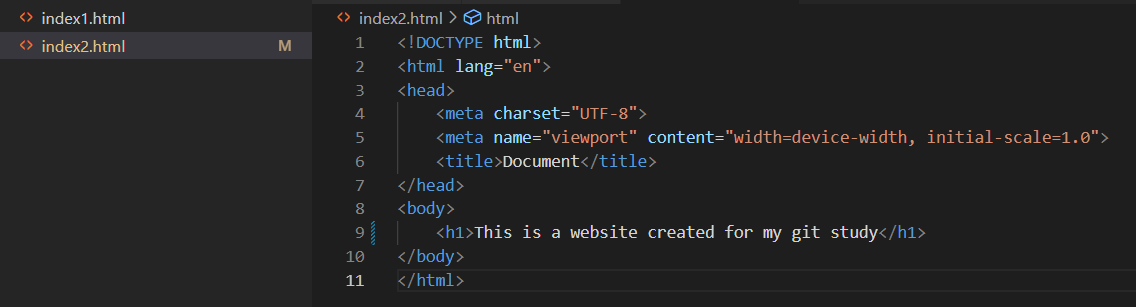
After merging

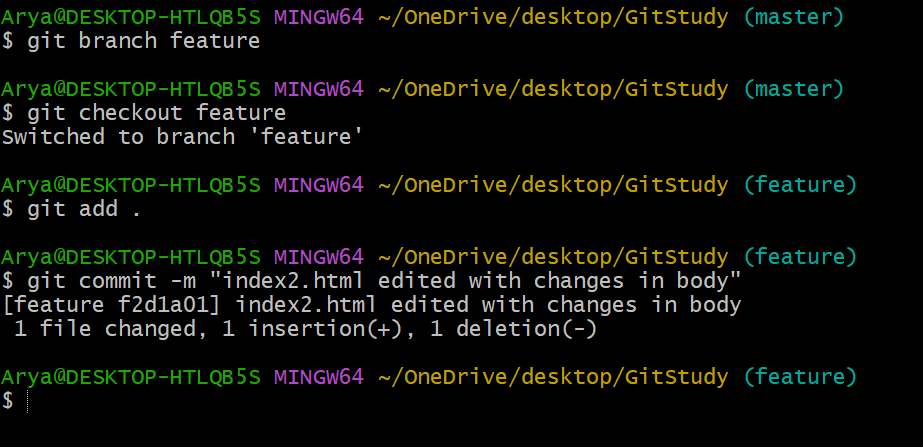


Explained the Recursive merge with two index.html files.

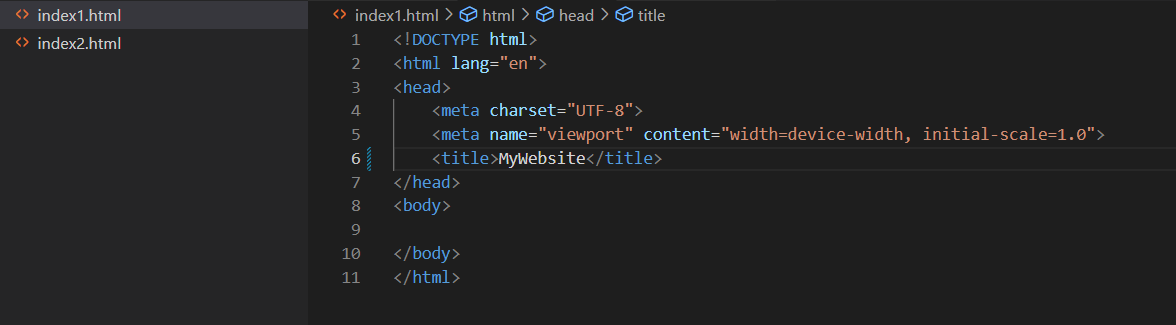
Created two html files; index1.html and index2.html.

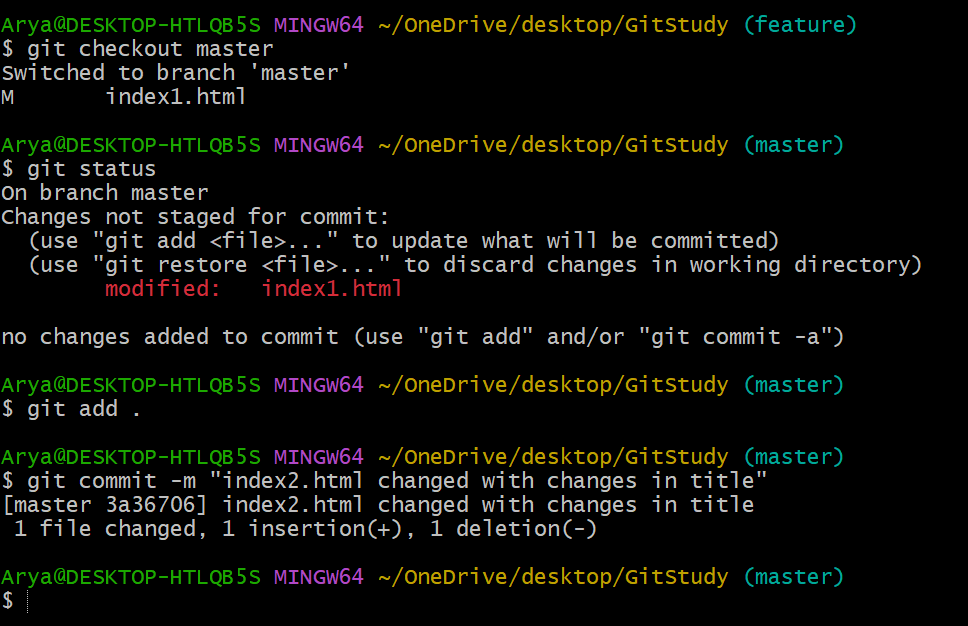
Index2.html edited with the changes in body section and commited in feature branch.



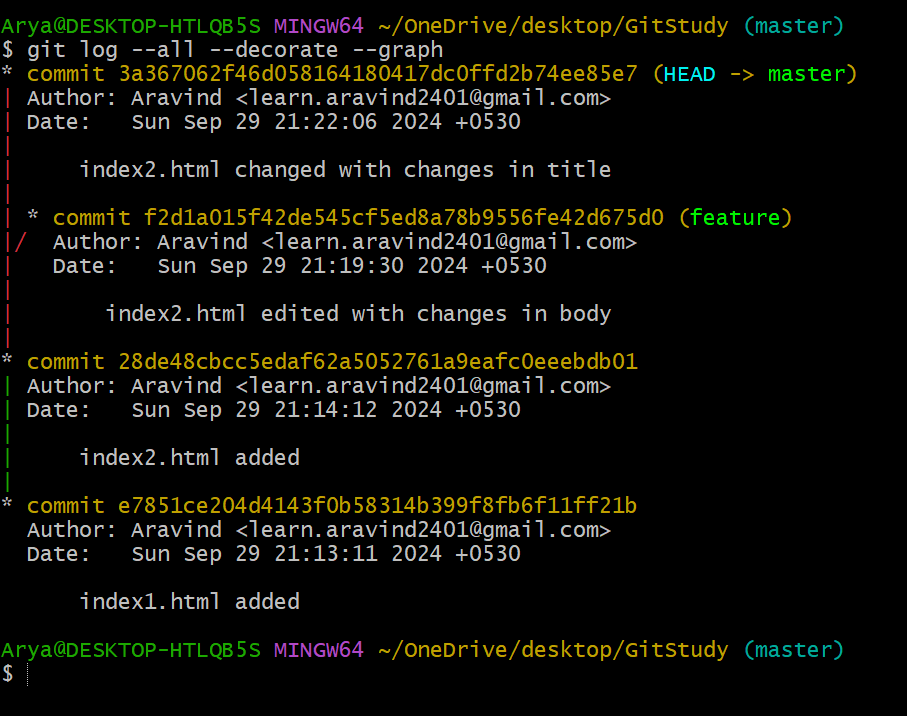


Index1.html edited with the changes in title section and commited in master branch.

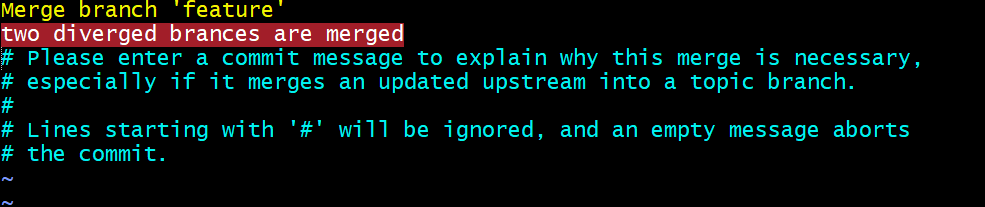




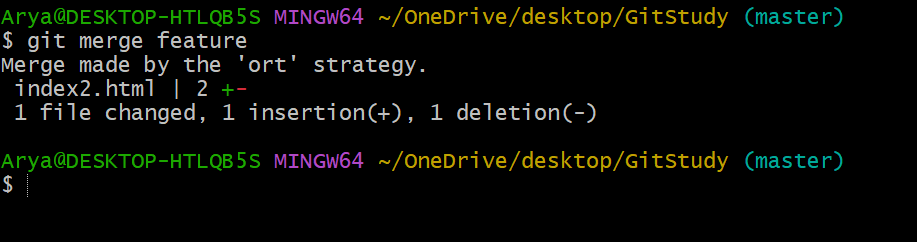
In git log, we can see that two branches are diverged.



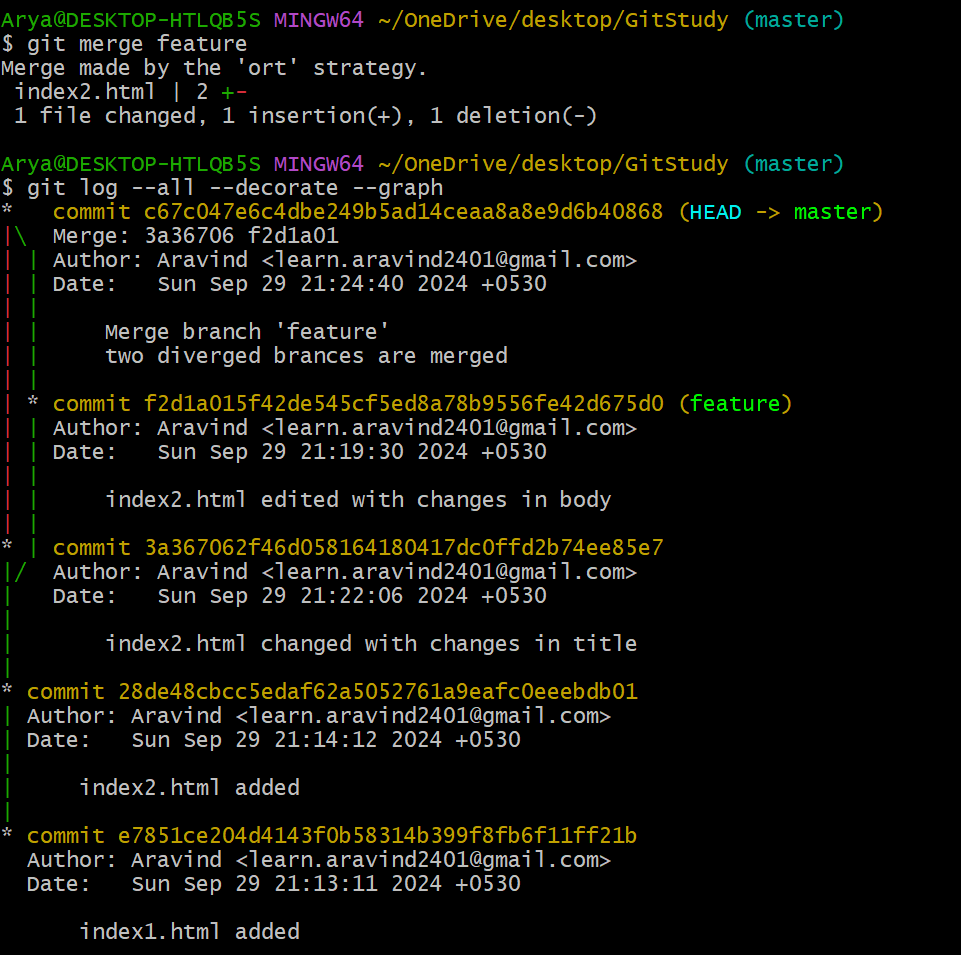
Then we commited the two branches; it will perform an ORT merge; it creates a separate commit by combining the both branches. So, we have provided a new commit message here.



here we can see that the two diverged branches are merged by Recursive/ORT strategy.



After merge, git log , two branches are merged with new commit id.

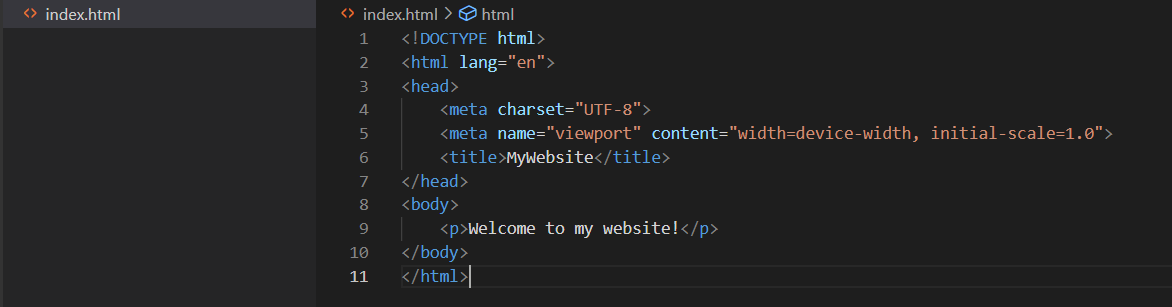


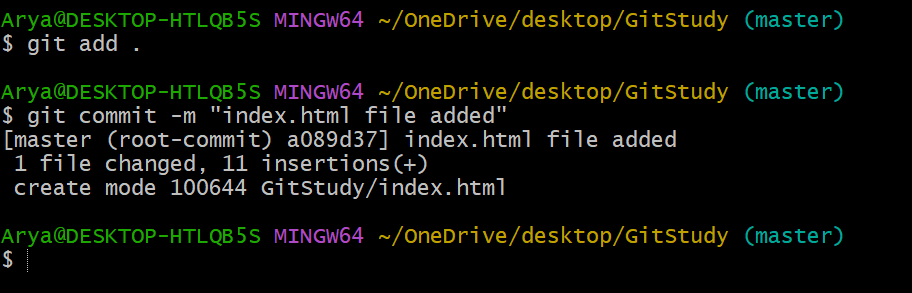
**3. Merge conflict**

A merge conflict in Git occurs when changes made in two different branches conflict with each other and cannot be automatically merged. This typically happens when the same part of a file is modified differently in two branches. In such cases, Git will require manual intervention to resolve the conflict.

Explain with a simple html file;

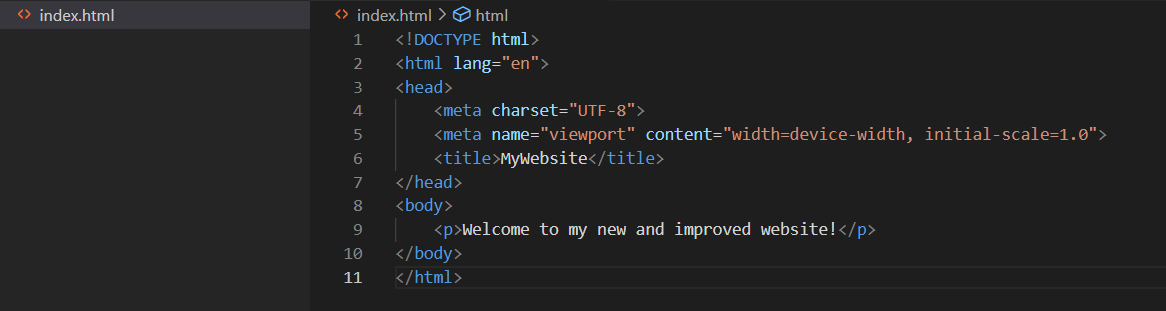
Here is the initial index.html file in the **master branch**:



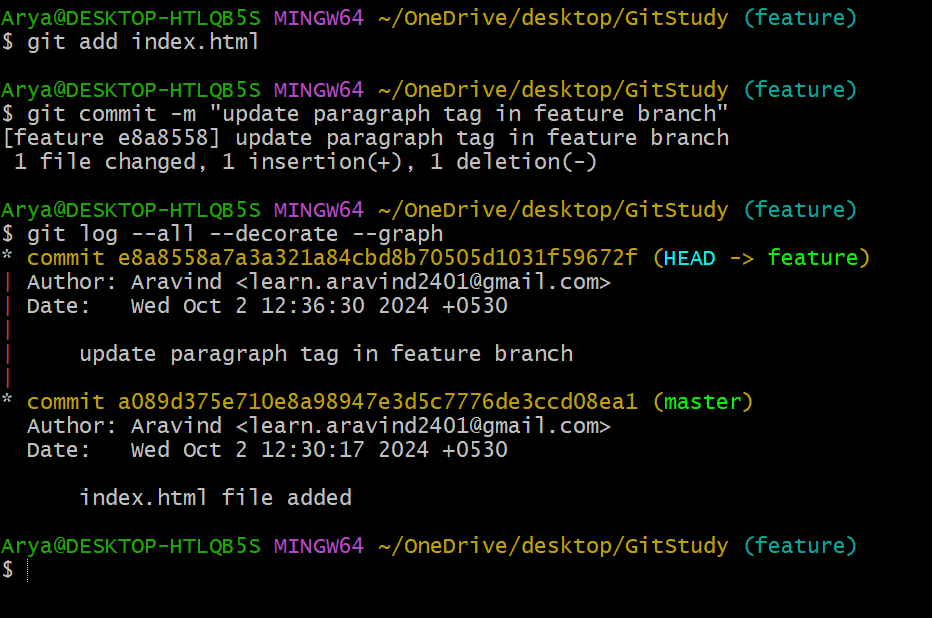


Now, create a new branch f**eature;**

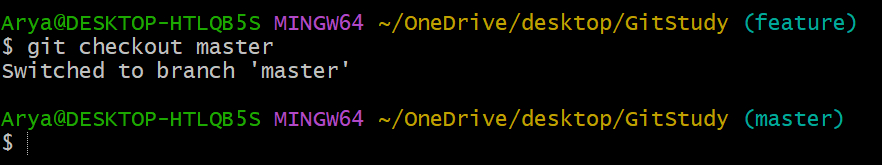
Modify index.html in the **feature branch** by changing the text inside the <p> tag.



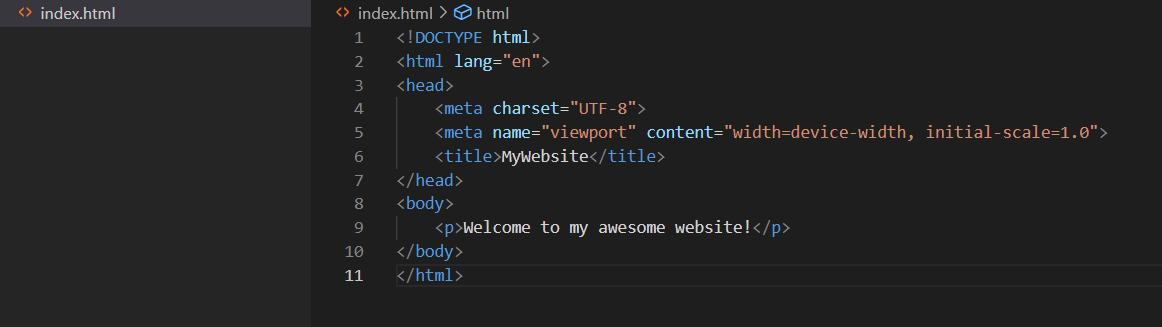
Commit the changes in the **feature branch:**



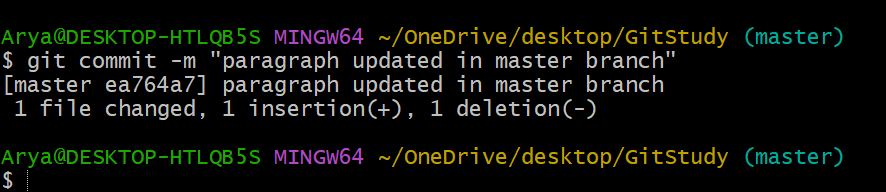
Now, switch back to the **master branch**:



edits the <p> tag in the **master branch**



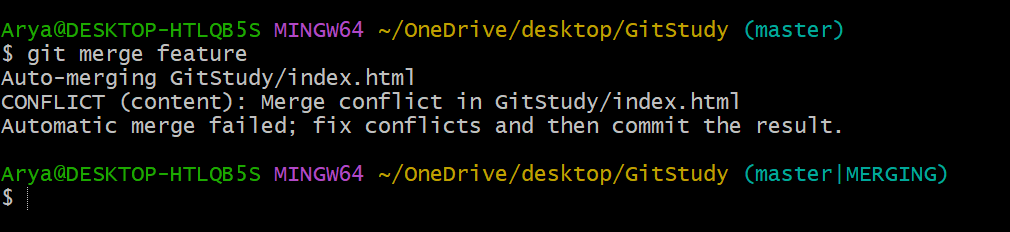
Commit the change in the **master branch**:



Now, the **main branc**h and the **feature branch** have different changes in the same <p> tag.

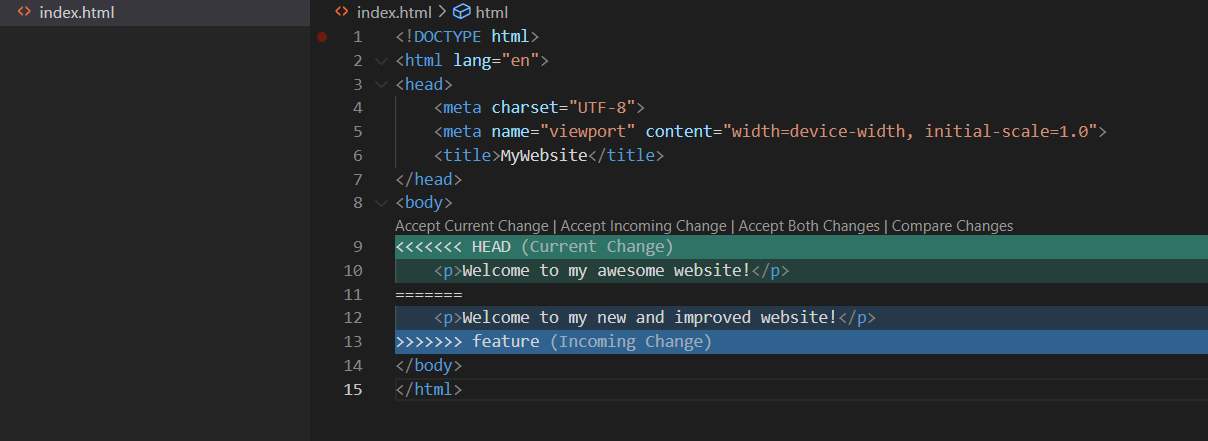
Attempt to merge the feature-header branch into the main branch:

Git will detect the conflict because both branches modified the <p> tag in different ways and will output something like this:



Git cannot automatically merge the file because the changes conflict with each other.

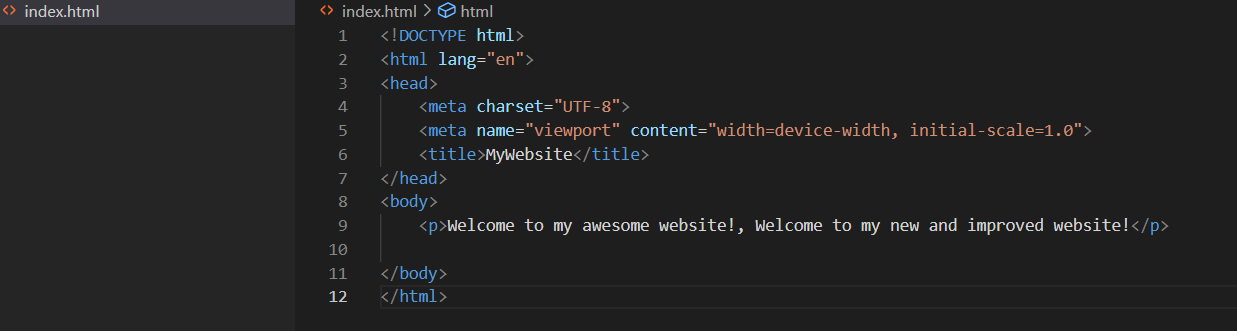
Git marks the conflicting sections in index.html like this:



* **<<<<<<< HEAD**: This shows the version from the master branch.
* **=======**: This separates the two conflicting changes.
* **>>>>>>> feature**: This shows the version from the feature branch.

To resolve the conflict, you need to manually edit the conflicting part and decide what the final text should be.

For example, you can combine both changes:



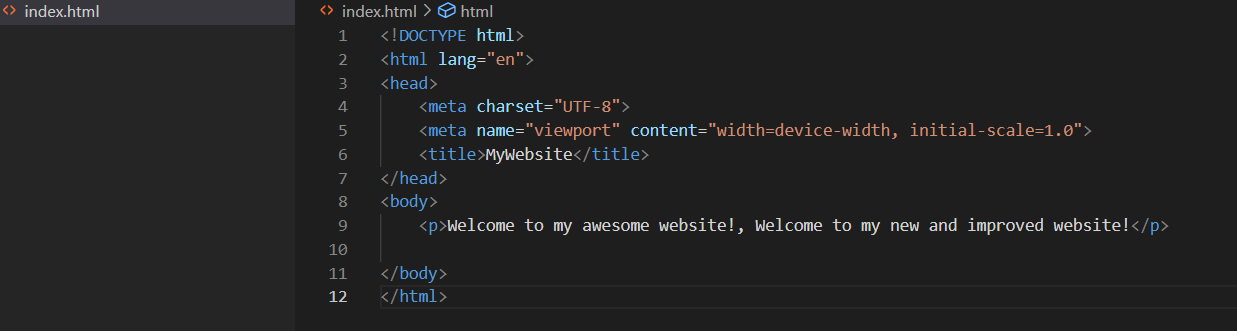
After resolving the conflict, add the file to the staging area:

**git add index.html**

Finally, complete the merge by committing the resolved file:

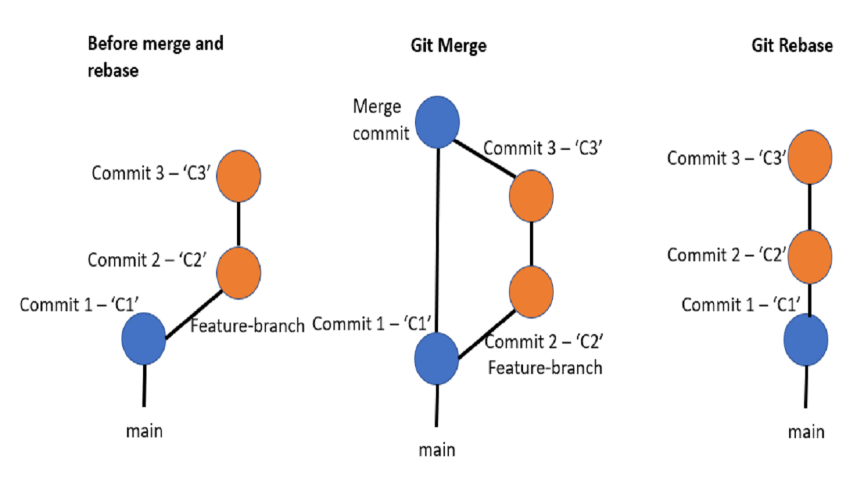
git commit -m “conflict cleared”

so, after the merge result, in the master branch;



**4. git Rebase**

**Git rebase** is a command used to integrate changes from one branch into another. It "rebases" the commits of your current branch on top of another branch. Unlike git merge, which creates a new "merge commit," git rebase rewrites the commit history, making it appear as if all changes were applied in a linear order, which can result in a cleaner project history.



**git Reset**

**git reset** is a Git command used to undo changes in a repository. It can modify the current branch's history and working directory.

* **--soft**

Moves the HEAD pointer to a specified commit but keeps the changes in the staging area (index).

git reset --soft <commit>

* --**hard**

Moves the HEAD pointer to a specified commit, clears the staging area, and resets the working directory to match the specified commit.

git reset --hard <commit>

**git Stash**

**git stash** is a Git command used to temporarily save changes that you’re not ready to commit, while keeping your working directory clean.

It allows you to "stash" your local modifications so that you can switch branches or pull in new changes without losing your work.

When you run git stash, Git saves your uncommitted changes (both staged and unstaged) in a stack and then reverts your working directory to match the last commit.

* **git stash list** - List stashes: You can view a list of all stashes
* **git stash apply -** apply the most recent stash back to your working directory.

**GitHub - Small Team Collaboration**

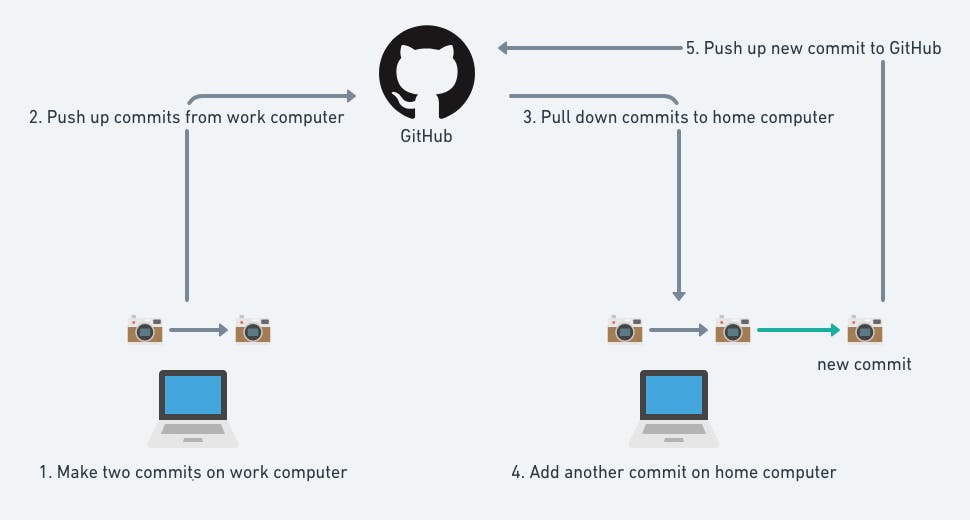
GitHub is a web-based platform that uses Git, a Distributed Version Control System (DVCS). GitHub provides a collaborative environment for developers to host, review, and manage their code.

It enhances Git’s capabilities by adding features such as issue tracking, project management, and continuous integration/deployment.

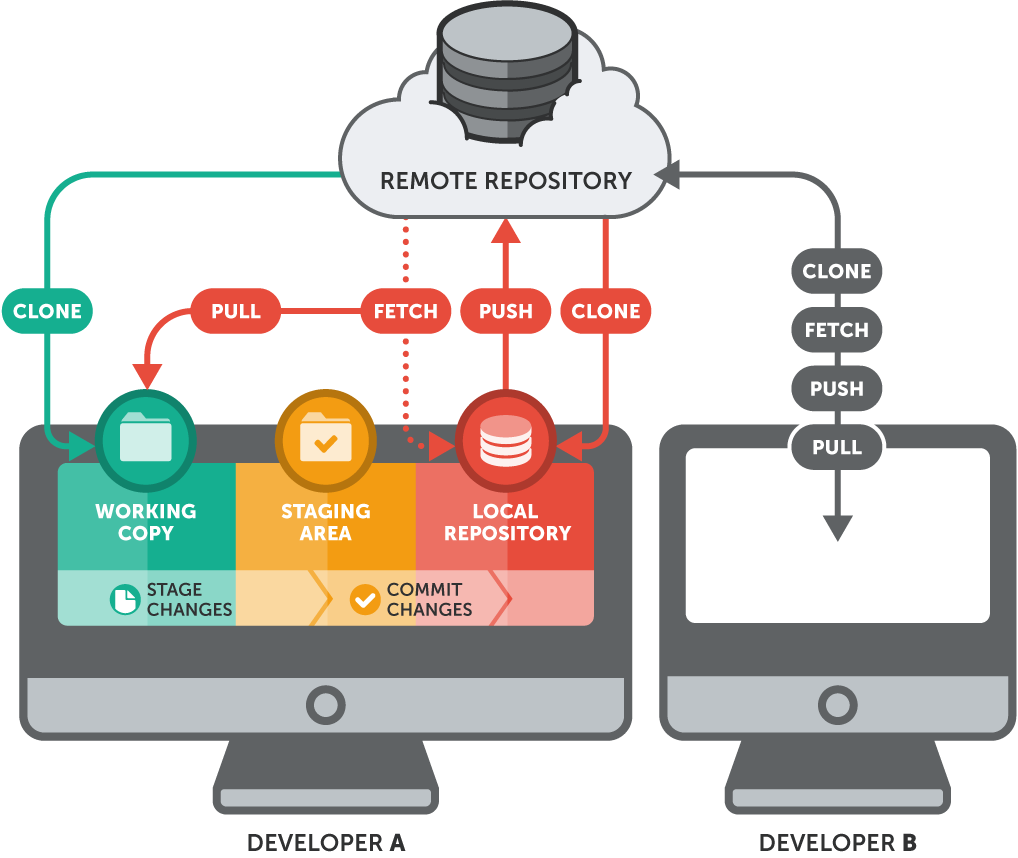
**Importance of Integrating Git with GitHub**

1. **Collaboration**: GitHub allows multiple developers to collaborate on the same project, providing tools for code review, comments, and discussion.
2. **Remote Repositories**: Hosting repositories on GitHub ensures your code is backed up and accessible from anywhere.
3. **Project Management**: GitHub offers features like issue tracking, pull requests, and project boards to manage development workflows.
4. **GitHub Actions:** Provides built-in tools for automating workflows such as code testing, building, or deploying using GitHub Actions.

**Workflow**



**Git Remote**



In Git, **git remote** is a command that allows you to manage and interact with **remote repositories**. A remote repository is typically hosted on a service like GitHub, GitLab, or Bitbucket.

**To add a new remote repository:**

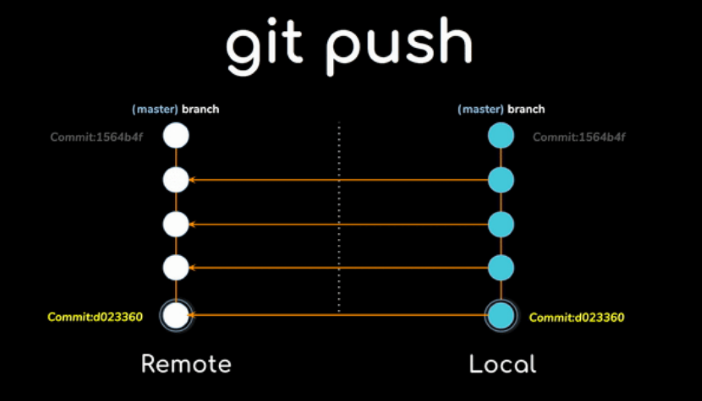
Syntax: **git remote add <name> <url>**

<name>: remote name

<url>: url for the remote repository in GitHub, GitLab, or Bitbucket.

**eg: git remote add origin https://github.com/cloud-aravind/GitStudy.git**

**Git Push**



**git push** is a Git command used to upload (or "push") the local repository's content to a remote repository. When you use git push, your commits from the local branch are sent to the corresponding branch on the remote repository.

Syntax: **git push <remote-name> <branch-name>**

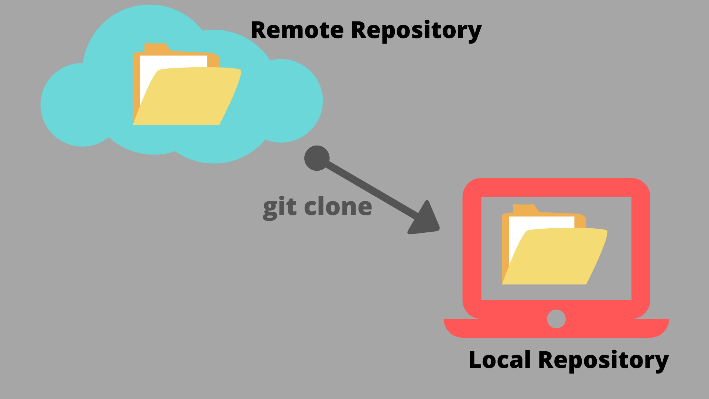
<remote-name>: The name of the remote repository (e.g., origin by default).

<branch-name>: The name of the branch you're pushing (e.g., main, develop).

Eg**: git push origin main**

This command pushes the local main branch to the origin remote repository.

**Git Clone**



**git clone** is a Git command used to create a copy of an existing repository from a remote server to your local machine. It downloads all the files, commits, and branches of the repository and sets up a local version for you to work on.

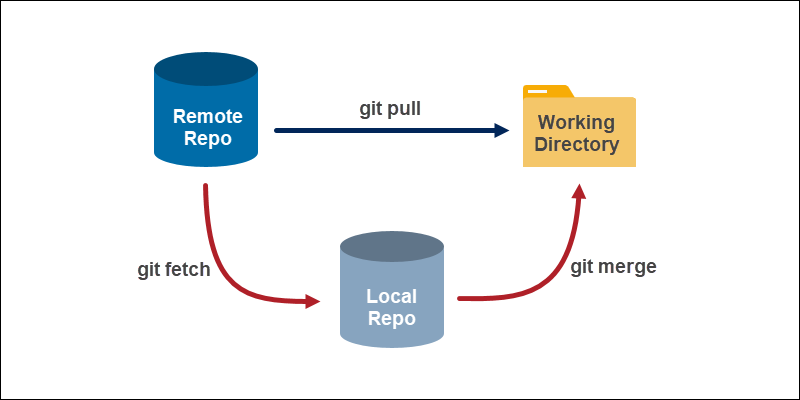
Syntax: **git clone <repository-url>**

<repository-url>: The URL of the remote repository you want to clone (e.g., a GitHub, GitLab, or Bitbucket repository URL).

Eg: **git clone** [**https://github.com/user/repo.git**](https://github.com/user/repo.git)

This command clones the repository located at https://github.com/user/repo.git to your local machine**.**

**Git Pull**



**git pull** is a Git command that is used to fetch and integrate changes from a remote repository into your current local branch. Essentially, it combines two Git operations: **fetch** and **merge**.

**Fetch**: Downloads the latest changes (commits, branches, and files) from the remote repository.

**Merge**: Merges those changes into your current branch.

Syntax: **git pull <remote> <branch>**

**<remote>**: The name of the remote repository (usually origin).

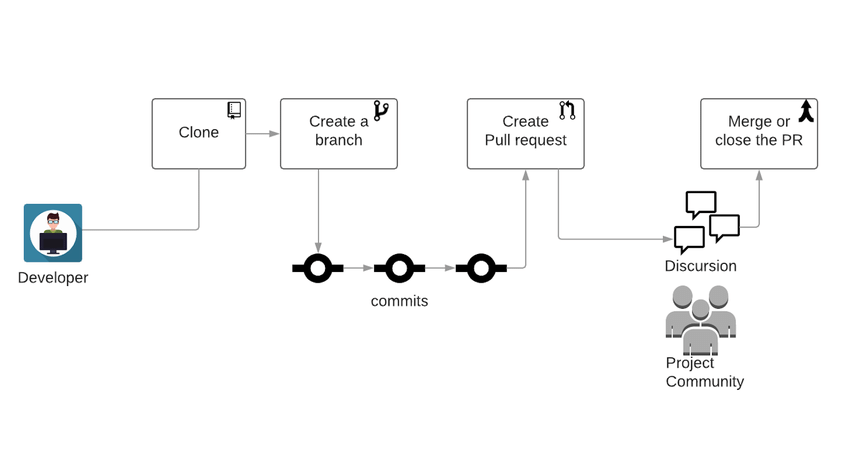
**<branch>**: The branch you want to pull from (e.g., main, develop).

Eg: **git pull origin main**

This command fetches and merges the changes from the main branch of the remote repository origin into your current branch.

**It helps you stay in sync with the team's work but may require conflict resolution if changes overlap.**

**Git Pull Request**



A **pull request (PR)** is not a direct Git command, but a feature provided by platforms like **GitHub**, **GitLab**, or **Bitbucket**.

**pull request workflow:  
0. “Pull” the changes to your local machine (get the most recent base)  
1. Create a “branch” (version)  
2. Commit the changes  
3.a Push your changes  
3.b Open a “pull request” (propose changes)  
4. Discuss and review your code  
5. Rebase and tests  
6. “Merge” your branch to the master branch**

**Why Use a Pull Request?**

* **Code Review**: Allows other team members to review your code before merging.
* **Discussion**: Provides a place to discuss changes, give feedback, and suggest improvements.
* **Collaboration**: Essential for collaboration on large projects, ensuring the integrity of the main branch is maintained.

**.gitignore file**

Keep the files names in .gitignore then that files not add and commit, just skip that file while adding and committing.

**To delete a branch**

# git branch -d <branch\_name>