Version Control with Git and GitHub

https://www.linkedin.com/in/aditi-das-42aa12246/

TABLE OF CONTENTS

GIT BASICS

Git :

A distributed version control system (VCS) created by Linus Torvalds in 2005. Git helps track changes in source code during software development, allowing multiple people to collaborate on a project. Each collaborator has a copy of the repository, which they can update, modify, and merge with the central repository as needed.

GitHub :

A web-based platform built around Git, providing a graphical interface to manage code repositories. It includes features like bug tracking, task management, and wikis, enabling developers to share and collaborate on projects more easily. GitHub hosts repositories and allows version control with Git functionalities.

Version Control System (VCS) :

Records changes to files, tracking edits over time, and allowing specific versions to be recalled. Git's distributed nature means it can operate offline or be synchronized to remote repositories like GitHub.

Repository (Repo) :

A repository is a directory or storage space where your project files and their complete history are stored. There are two types of repositories:

1. Local repository: The repository that exists on your local machine.
2. Remote repository: The version of the repository stored on a remote server (e.g., GitHub).

Commit :

A **commit** is a snapshot of your repository at a specific point in time. Commits represent the changes made to the repository, and each commit includes a unique ID (SHA-1 hash) and metadata like the author, date, and commit message.

GIT BASICS

Branch :

A branch in Git represents a separate line of development. By default, Git has a main branch (formerly master). Developers create new branches for working on different features, bug fixes, etc., without affecting the main codebase.

Merge :

Merging is the process of integrating changes from one branch into another. For example, when a feature branch is complete, it is merged into the main branch.

Staging Area :

The staging area (or index) is an intermediate area where changes are stored before they are committed. It allows you to prepare and review changes before finalizing them.

Remote :

A remote is a version of the repository that is hosted on the internet or another network. Remote repositories like GitHub or GitLab are used for collaboration and sharing changes with others.

Working Directory :

This is the directory on your local machine where your project files are stored and where you work. The working directory reflects the current state of the repository.

Fork :

A fork in Git is a personal copy of someone else's repository. It allows developers to freely experiment with changes in their own copy without affecting the original project. Forks are typically used in open-source projects, where contributors fork the original repository, make changes, and then propose those changes back to the original project via pull requests. This allows for independent development while maintaining the ability to sync with and contribute to the upstream project.

Pull Requests (PRs) :

A pull request (PR) is a way to request that changes made in a branch are reviewed and merged into another branch, typically into the main or develop branch of a repository. PRs are used primarily in collaborative environments to facilitate code review, discussion, and automatic testing before changes are merged.

Merge Conflicts :

A merge conflict occurs when Git can't automatically merge changes due to conflicting changes in the same file(s) on different branches. This typically happens when two people edit the same part of a file or when changes on two branches diverge significantly.

CONFIGURING GIT

To configure Git, set up user information (name and email) and verify that Git is correctly installed. Here are the steps and commands for configuring Git:

1. Install Git

- If Git is not installed, download and install it from git-scm.com.
- Verify the installation by checking the Git version.
- Command : *git –version*

2. Set Up Identity (Name and Email)

To configure Git with name and email. This information will be associated with commits.

Global Configuration (Applies to all repositories):

- Set your name: *git config --global user.name "ABC Name"*
- Set your email: *git config --global user.email your.email@example.com*

3. Verify Git Configuration

To check the current configuration (global and local), use:

- *git config –list*

This will display the configured username, email, and other settings.

4. Set Default Text Editor (Optional)

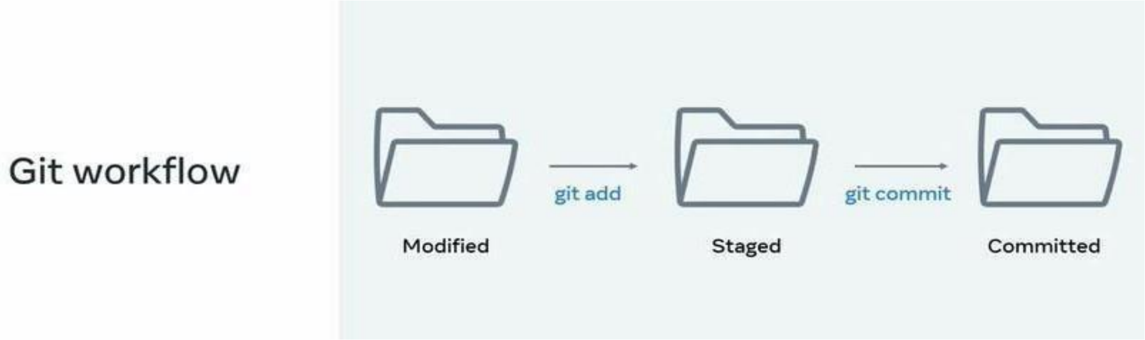
If one wants to change the default text editor for Git (used when writing commit messages), one can configure it:

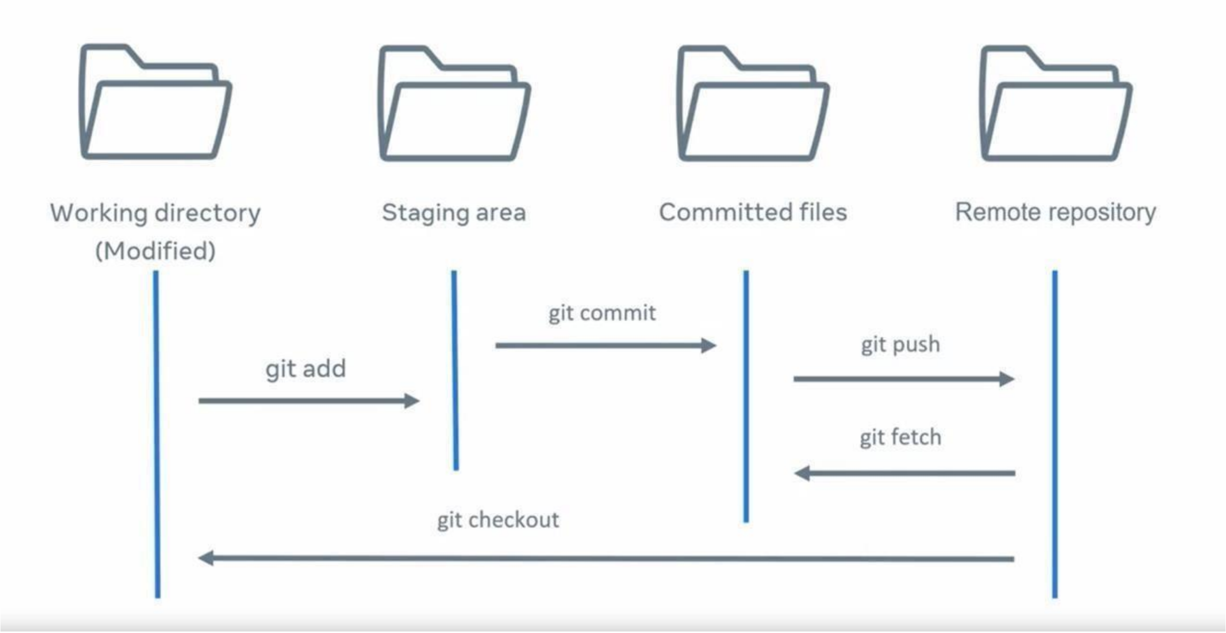
- *git config --global core.editor "code --wait"* # For VSCode

GIT WORKFLOW

The Git workflow consists of three states :

- **Modified:** A file has been changed.
- **Staged:** Changes have been added using the git add command and are ready to be committed.
- **Committed:** Changes have been finalized and saved to the repository using the git commit command.





Last stage is the committed stage . After git push its available and the files can be fetched from the remote repository by git pull or git fetch.

GIT COMMANDS

TYPE	COMMAND	DESCRIPTION
SETUP	git config --global user.name “[firstname lastname]”	set a name that is identifiable for credit when review version history
	git config --global user.email “[valid-email]”	set an email address that will be associated with each history marker
	git config --global color.ui auto	set automatic command line coloring for Git for easy reviewing
SETUP & INIT	git init	Initializes a new Git repository.
	git clone repository_link	Clones a remote repository to the local machine.
STATUS	git status	show modified files in working directory, staged for your next commit.
STAGE	git add filename	add any particular file only
	git add .	Adds all modified or untracked files to git
COMMIT	git commit -m "some message"	Records changes in the local repository .
	git commit -m “msg1” -m “msg2”	For multiple line commit message.
	. git commit -a -m “your message”	stage any changes to tracked files and commit them in one step.
	git commit --amend	replaces the previous commit with a new commit that has its own unique identifier and the amended message

REWRITE HISTORY	git reset commit hash	resets a repository to a specific commit by using the commit hash.
	git reset	Rewrites commit history
	git reset filename	unstage a file while retaining the changes in working director
	git reset --hard [commit]	clear staging area, rewrite working tree from specified commit.
	git reset HEAD-1	reset head by 1 step back.
SHARE & UPDATE	git push origin branchname	upload your local branch's changes to a remote repository, specifically the branch named branchname on the remote repository.
	git push -u remote branchname	Sets the upstream reference for the specified branch.
	git remote	Manages remote repositories.
	git remote -v	Lists all remote repositories and their URLs.
	git remote show name	Shows info. About a single remote repository.
	git remote update	Fetches updates for remotes or remote groups
	git remote add origin link	adds a remote repository named origin with the specified *link* URL to your local Git repository.
	git fetch	Retrieves remote updates.
	git fetch remote_name branchname	Fetch updates from a specific branch in a remote repository
	git branch	List branches (the asterisk denotes the current branch)

BRANCH & MERGING	git branch	List branches (the asterisk denotes the current branch)
	git branch -a	List all branches (local and remote)
	git branch -d branchname	Delete a branch
	git branch -M main	To rename a branch
	git checkout branchname	Navigating or moving from one branch to another
	git checkout -b new_branchname	Checkout from a branch and create new branch simultaneously.
	git checkout main	going from a branch to the main branch in the repo
	git merge branchname	Joins changes from one branch to another.
	git merge –abort	Can only be used after merge conflicts . This command will also abort the merge and go back to the pre-merge state.
	git merge source_branch target_branch	Merge source branch into target branch
	git stash	Temporarily shelve or stash current changes.
	git stash clear	Remove all stashed entries
BLAME	git blame filename	determine who last modified each line of that file, and when they made those changes.
	git blame -n number filename	Limits the number of lines displayed in the blame output.
	git blame -r filename	Reverses the order of the blame output, showing the

BLAME		earliest changes first.
	git blame -s filename	Outputs a summary of the blame information, including the author, commit hash, and date for each line.
	git blame -o filename	Outputs the blame information in a machine-readable format that can be easily parsed by other programs
INSPECT & COMPARE	git log	To check all the commits in all branches / history of git commits
	git log —stat	shows which flies were changed , how many lines were added and removed
	git log --graph --oneline --all	provides a concise and visual representation of your entire commit history across all branches
	git log -2	Gives details of last 2 commits
	git log –pretty=oneline	to demo all commits separately in 1 line
DIFFERENCE	git diff branchname	to compare commits, branches, files & more
	git diff --staged	shows the differences between staged changes and the last commit in Git.

.gitignore file

A .gitignore file is used in Git to specify which files or directories should be excluded from version control. This helps maintain a clean repository by preventing unnecessary or sensitive files from being tracked.

1. Purpose of .gitignore

The main purpose of the .gitignore file is to ensure that certain files and directories are not tracked by Git. This is particularly useful for:

- Temporary files: Files that are generated during development but are not part of the source code (e.g., log files).
- Build artifacts: Compiled or built code and files generated by build tools.
- Sensitive information: Files like API keys, configuration files, or credentials that should not be shared or stored in version control.

2. File Syntax of .gitignore

Specify patterns in the .gitignore file to determine which files and directories should be ignored by Git.

General Syntax:

- Use * as a wildcard to match multiple characters.
- Use / to specify directories.
- Use ! to negate a pattern and include files that would otherwise be ignored.

Examples:

- *.log ignores all files with a .log extension.
- build/ ignores the entire build directory.
- !important.log includes the file important.log, even if all .log files are ignored.

.gitignore file

3. Location of .gitignore

The .gitignore file is typically placed in the root directory of your Git repository. Also add additional .gitignore files in subdirectories to apply specific rules only to those directories.

4. How to Create and Edit .gitignore

Step 1: Create the .gitignore file

- Simply create a text file named .gitignore in the root of your repository.

Step 2: Edit the .gitignore file

- Open the file in any text editor and add patterns for files or directories you want Git to ignore.

Step 3: Apply the .gitignore changes

- If new patterns are added, commit the .gitignore file for the changes to take effect:

  ```
  git add .gitignore
  git commit -m "Add .gitignore file"
  ```

- If a file has already been tracked by Git and you now want to ignore it, you must first remove it from version control:

  ```
  git rm --cached <filename>
  ```

.gitignore file

Important Notes

- **Global .gitignore**: You can create a global .gitignore file to apply to all your Git repositories (for files like .DS_Store or .log across all projects):
  git config --global core.excludesfile ~/.gitignore_global

- **Prevents Sensitive Data Leaks:** Use .gitignore to prevent sensitive files such as .env or configuration files with API keys from being pushed to a remote repository, ensuring better security.

VERSION CONTROL

Version control is a system that tracks changes to files over time, allowing developers to:

- Keep track of code modifications.
- Access previous versions of code (revision history).
- Revert to earlier versions when necessary.
- Collaborate with others while minimizing file conflicts**.**

Types of Version Control Systems

1. Centralized Version Control Systems (CVCS)

A system where all versioned files are stored on a central server.
How It Works: Developers pull code from the server, work locally, and push changes back.

Advantages:

- Easy access control.
- Simple and easier to learn for new developers.

Disadvantages:

- Requires an active connection to the server for most actions.
- Slower performance due to reliance on a central server.

2. Distributed Version Control Systems (DVCS)

Every user has a complete copy of the repository, including its history.
How It Works: Users can work offline, make commits locally, and later sync changes with the remote server.

Advantages:

- Fast and efficient.
- Can work offline.
- Higher performance and flexibility.

Disadvantages:

- May have a steeper learning curve for beginners.

Popular Version Control Tools

- Subversion (SVN): Centralized system, successor to CVS, widely used for enterprise-level projects.
- Mercurial: A distributed version control system, similar to Git but simpler in terms of commands.
- Perforce: Another enterprise-level VCS often used for large-scale projects.

STAGING VS. PRODUCTION

Staging Environment:

A testing environment that mirrors production. Used for introducing new features, running tests (unit, integration, performance), and performing database migrations.

Production Environment:

The live environment where the application runs for users. All staging changes should be thoroughly tested before deployment to production to avoid issues.

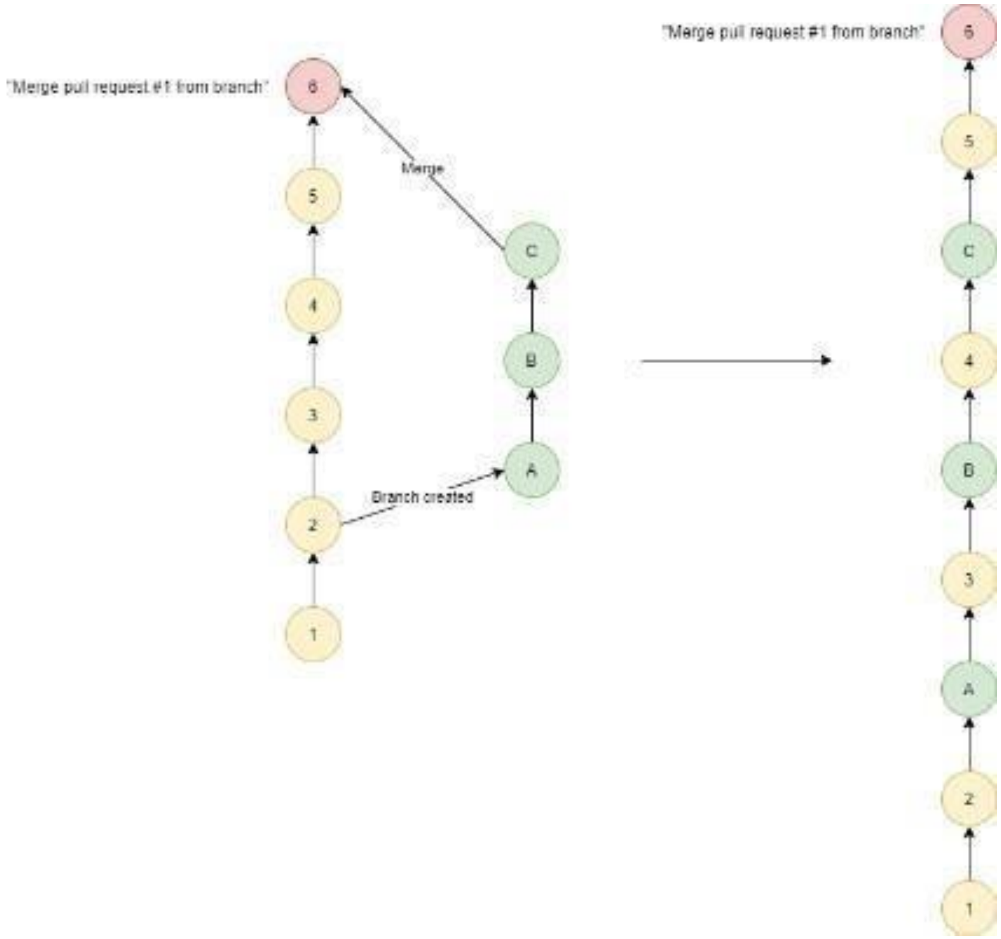
ASPECT	STAGING ENVIRONMENT	PRODUCTION ENVIRONMENT
Purpose	Testing and validation before deployment	Live application serving real users
Data	Test data or a copy of real data	Real user data
User Access	Internal users (developers, testers, stakeholders)	External, real users
Impact of issues	Low (isolated from users)	High (affects user experience and business)
Testing	Extensive testing: unit, integration, performance	Limited to monitoring and real-time bug fixing
Deployment frequency	Frequent (after each feature or fix)	Controlled and infrequent (after thorough validation)
Security	Basic security measures	Strong security protocols

MERGE TYPES

1. Merge :

A standard merge will take each commit in the branch being merged and add them to the history of the base branch based on the timestamp of when they were created.

It will also create a merge commit, a special type of "empty" commit that indicates when the merge occurred.


"Merge pull request #1 from branch"
6
5
4
3
2
1
Merge
C
B
A
Branch created
"Merge pull request #1 from branch"
6
5
C
4
B
3
A
2
1

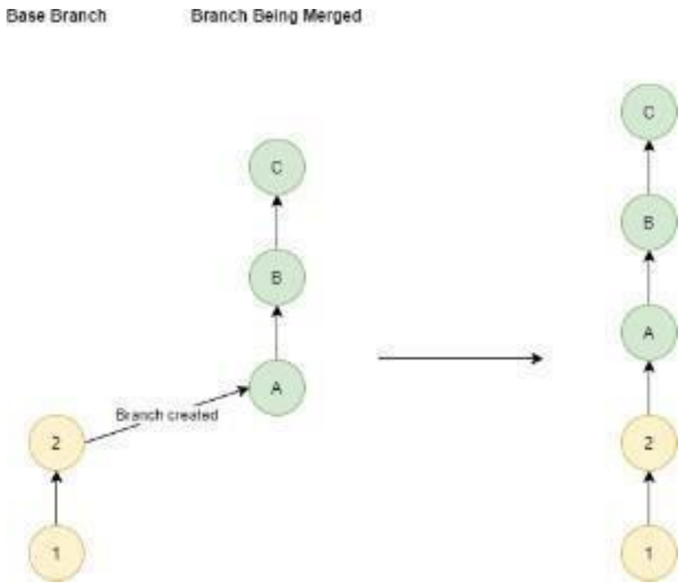
MERGE TYPES

2. Fast Forward Merge :

A fast-forward merge occurs when the branch you’re merging hasn’t diverged from the current branch. In this case, Git simply moves the pointer of the current branch forward to point to the most recent commit in the branch being merged, and no actual merging happens.

How Fast-Forward Merges Work:

- Example: You’re on main, and you want to merge a feature branch called feature2. If no other commits have been made to main since feature2 branched off, Git can simply "fast-forward" main to include all of the feature2 commits.
- Fast-forward merges result in a linear history.



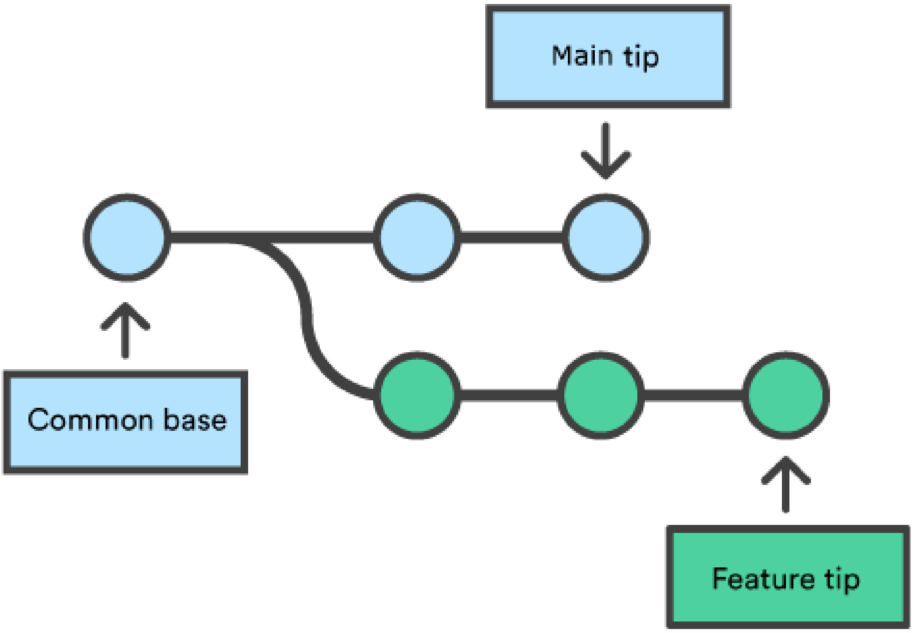
MERGE TYPES

3. Three way Merge :

A three-way merge is used when the branches have diverged (i.e., both branches have new commits that the other doesn't). In this case, Git creates a new "merge commit" that combines the histories of both branches.

How Three-Way Merges Work:

- Git looks at the common ancestor between the two branches (the commit where they diverged), compares it with the current state of each branch, and creates a new merge commit.
- This merge commit brings the histories of both branches together.



MERGE TYPES

4. Squash & Merge :

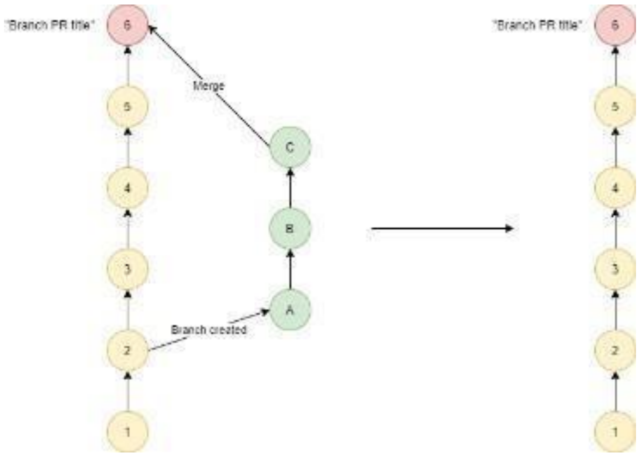
Squash takes all the commits in the branch (A,B,C) and melds them into 1 commit. That commit is then added to the history, but none of the commits that made up the branch are preserved.

How Squash and Merge Works:

1. Feature Branch with Multiple Commits:
   - When working on a feature branch, developers often make multiple small commits, such as bug fixes, tests, or incremental feature updates.
2. **Open a Pull Request (PR)**:
   - Once the work on the feature branch is complete, a PR is opened to merge the feature branch into the main branch.

3. **Squash the Commits**:
   - Before merging, you can choose the **Squash and Merge** option on GitHub. This squashes all the commits into one, summarizing the work as a single commit.
   - You will be prompted to provide a **new commit message** that summarizes all the changes in the squashed commit.
4. **Merge the Squashed Commit**:
   - The feature branch is merged into main, but instead of the original four separate commits, the main branch now contains a single, squashed commit that represents all the changes from the feature branch.

MERGE TYPES



5. Rebase & Merge :

A rebase and merge will take where the branch was created and move that point to the last commit into the base branch, then reapply the commits on top of those changes.

How Rebase and Merge Works:

1. Feature Branch Development:
   - Suppose you're working on a feature branch, and during development, other commits are added to the main branch. The commit history of both branches diverges.

2. Rebase Instead of Merging:
   - When you choose Rebase and Merge, GitHub or Git will:
     - ♣ Take the commits from the feature branch.

MERGE TYPES

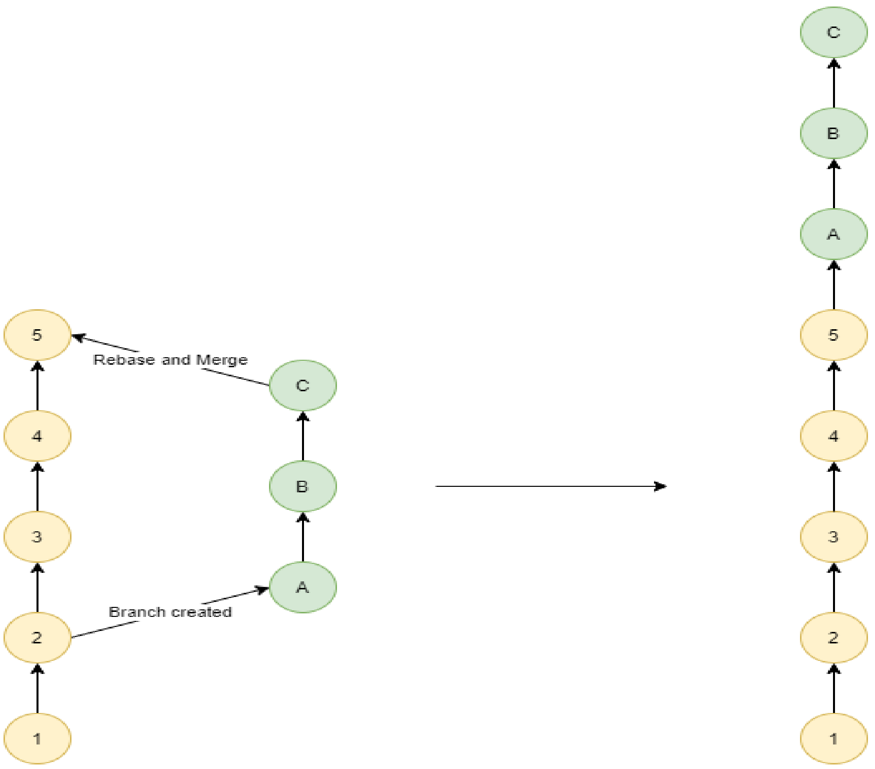
♣ Reapply or "replay" them one by one on top of the latest commits from main as if they were made after the latest changes in main.

3. No Merge Commit:

• Unlike a regular merge, no merge commit is created. The result is a linear history, where the commits from the feature branch appear as if they were added directly on top of the main branch, without the divergence that occurs in regular merges.

4. Fast-Forward Merge:

• After rebasing, the feature branch can be merged into the main branch with a fast-forward merge, creating a clean, linear history.



PULL REQUESTS

Pull Requests (PRs) :

A Git pull request (PR) is a feature commonly used in platforms like GitHub, GitLab, and Bitbucket that allows developers to propose changes to a codebase.

Creating a Pull Request:

- After committing changes in a branch, push the branch to the remote repository: **git push origin <branchname>**
- Navigate to your repository on GitHub, GitLab, or Bitbucket.
- Select the branch and click the **"New Pull Request"** button.
- Add a description of the changes, tag relevant team members for review, and submit the PR.

Reviewing a Pull Request:

- Team members can review the code, suggest changes, or approve it.
- Automated tests and CI/CD pipelines are often run at this stage to ensure the changes don't break the build.

Merging a Pull Request:

- After approval, the PR can be merged into the target branch.
- On GitHub, you can merge with different strategies like:
  - **Squash and merge**: Combines all commits into a single commit before merging.
  - **Merge commit**: Keeps all the commits and creates a merge commit.
  - **Rebase and merge**: Reapplies the changes on top of the target branch without a merge commit.

MERGE CONFLICTS

Merge Conflicts :

A merge conflict occurs when two or more developers try to edit the same file content simultaneously, or when one developer deletes a file while another is modifying it.

Common Causes of Merge Conflicts:

- Concurrent changes to the same lines of code in a file.
- Different changes made to the same file in different branches.
- Renaming or deleting files in one branch while the same file is edited in another.

How to Resolve Merge Conflicts:

- During a Pull Request: If there are conflicts between the source branch and the target branch during a PR, GitHub (or another Git platform) will highlight these conflicts.
- During a Git Merge: If you run *git merge <branch>* and Git can't automatically merge, it will pause the merge and highlight the conflict:

```
CONFLICT (content): Merge conflict in <file>
Automatic merge failed; fix conflicts and then commit the result.
```

Steps to Resolve Conflicts:

- Identify Conflicted Files: Open the files that have conflicts. Git will mark the conflicting areas with special markers:

```
<<<<<<< HEAD
Your changes in the current branch
=======
Changes in the branch you're merging into
>>>>>>> <branch-name>
```

MERGE CONFLICTS

Edit the File:

- Choose which changes to keep: yours, theirs, or a combination.
- Remove the conflict markers (<<<<<<<, =======, and >>>>>>>).

Add and Commit the Resolved Files:

- After resolving the conflicts, mark the files as resolved:
  git add <file>
- Then, commit the merge: *git commit*

Abort a Merge:

- If you don't want to continue with the merge due to conflicts or other reasons, you can abort it: *git merge --abort*