Git

Version Control System:

A Version Control System (VCS) is a software tool that helps manage changes to source code and other files in a collaborative development environment. It tracks and manages different versions of files over time, allowing multiple developers to work on the same project simultaneously without conflicts. Version control systems are crucial for software development, but they can also be used for managing changes in any type of text-based documents.

There are two main types of version control systems:

- **Centralized Version Control System (CVCS):** In a CVCS, there is a central server that stores all versions of a project's files, and developers check out files from this central repository to work on them. Examples of CVCS include CVS (Concurrent Versions System) and Subversion.

- **Distributed Version Control System (DVCS):** In a DVCS, each developer has a complete copy of the repository, including the entire project history. This allows developers to work independently and make changes locally before sharing them with others. Git and Mercurial are examples of distributed version control systems.

Key features and benefits of version control systems include:

- **History Tracking:** VCS keeps a record of changes made to files over time, allowing developers to review and revert to previous versions if needed.

- **Collaboration:** Multiple developers can work on different parts of a project simultaneously. The VCS helps merge changes from different contributors.

- **Branching and Merging:** VCS allows developers to create branches for parallel development, enabling them to work on features or fixes without affecting the main codebase. Merging integrates changes from one branch into another.

- **Conflict Resolution:** VCS provides mechanisms for resolving conflicts that may arise when two or more developers make conflicting changes to the same file.

- **Backup and Recovery:** Version control systems serve as a backup mechanism, ensuring that project history is preserved even if local copies are lost or damaged.

The most widely used version control system for modern software development is Git, due to its distributed nature, speed, and popularity in open-source projects. Many hosting services, such as GitHub, GitLab, and Bitbucket, provide platforms for hosting and collaborating on Git repositories.

Local Version Control Systems (LVCS):

A Local Version Control System (LVCS) is a type of version control system that operates on a single computer and keeps track of changes to files within a local repository. Unlike centralized or distributed version control systems that involve collaboration and multiple copies of a repository, a local system is designed for individual use on a developer's machine.

In a Local Version Control System:

- **Repository:** The version control system maintains a repository on the developer's local machine, storing different versions of files over time.

- **Version Tracking:** Changes to files are tracked within the local repository, allowing developers to revert to previous versions if needed.

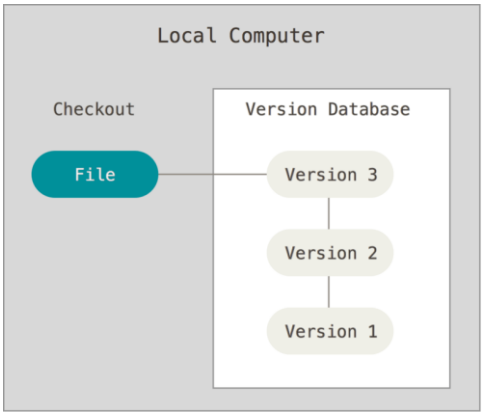
- **History:** The system records the history of changes made to files, including details such as who made the changes and when.

- **Benefits:**
  - **Simplicity:** Local Version Control Systems are straightforward and easy to set up. They are suitable for individual developers or small projects where collaboration and networked repositories are not a priority.
  - **Performance:** Operations such as commits, checkouts, and history queries are generally faster because they are performed locally, without the need for network communication.

- **Limitations:**
  - **Limited Collaboration:** Since the system is local, it doesn't support collaboration between multiple developers. It's not suitable for projects where multiple people need to work on the same codebase simultaneously.
  - **Backup Challenges:** While changes are tracked locally, there might be challenges in maintaining comprehensive backups of the version history if the local machine experiences issues.

Examples of Local Version Control Systems include early versions of **RCS (Revision Control System)** and **SCCS (Source Code Control System).** These systems were precursors to more advanced centralized and distributed version control systems that evolved to address the needs of collaborative and distributed software development.

While Local Version Control Systems are simple and effective for individual developers or small projects, they lack the features required for complex collaboration and larger-scale projects. As a result, many development teams have transitioned to using more advanced version control systems, such as Git (a distributed system) or SVN (a centralized system), to better support collaboration and manage the complexities of modern software development.



Centralized Version Control System (CVCS):

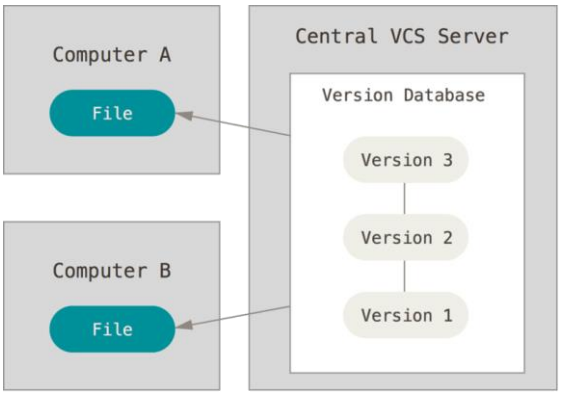
- **Central Repository:**
  - CVCS has a central repository that stores the entire version history of the project.
  - Developers check out files from this central repository to work on them.

- **Network Dependency:**
  - Requires a constant connection to the central server. Developers need access to the server to perform most version control operations.
  - Limited ability to work offline or in a disconnected environment.

- **Collaboration:**
  - Developers may face conflicts when trying to commit changes simultaneously, and these conflicts need to be resolved centrally.
  - Typically, collaboration is more sequential, as developers need to obtain locks on files before making changes to avoid conflicts.

- **Performance:**
  - Performance can be impacted when the central server is under heavy load or when network latency is high.

- **Examples:**
  - **CVS (Concurrent Versions System)** and **Subversion (SVN)** are examples of CVCS.



Distributed Version Control System (DVCS):

- **Complete Repository Copies:**
  - Each developer has a complete copy of the repository, including the entire project history.
  - Developers can work independently, making changes locally before sharing them with others.

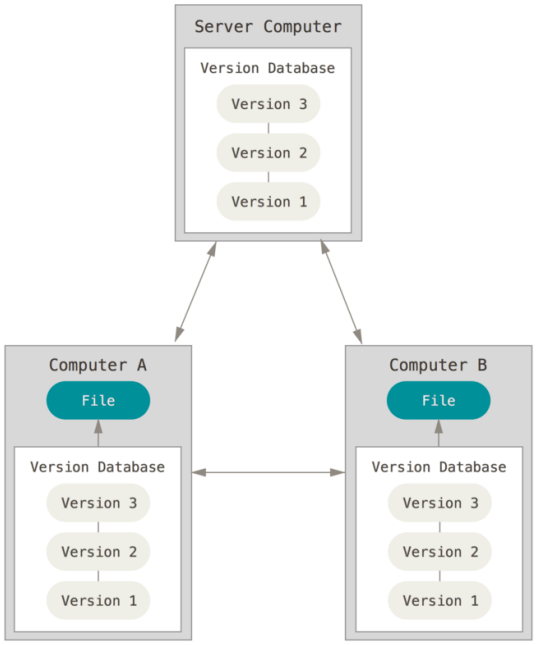
- **Network Independence:**
  - Developers can work offline and perform most version control operations locally without needing constant access to a central server.

- **Collaboration:**
  - Conflicts are resolved locally before pushing changes to the central repository.
  - Collaboration is more concurrent, as developers can work on their local copies independently and then merge changes.

- **Branching and Merging:**
  - DVCS excels at branching and merging, allowing developers to create branches for feature development or bug fixes and then merge changes seamlessly.

- **Performance:**
  - Performance is often faster, as many operations can be performed locally without network overhead.

- **Examples:**
  - **Git** and **Mercurial** are examples of DVCS. **Git**, in particular, has become widely adopted in the software development community.



Differences between Centralized Version Control System (CVCS) and Distributed Version Control System (DVCS):

- CVCS is centralized, with a single repository; DVCS is distributed, with each developer having a complete copy of the repository.
- CVCS requires constant network access; DVCS allows for more independence and offline work.
- CVCS may face conflicts more often during concurrent development; DVCS promotes concurrent development with easier conflict resolution.
- DVCS excels in branching and merging, providing flexibility in development workflows.

Distributed Version Control System (DVCS):

A Distributed Version Control System (DVCS) is a type of version control system that allows multiple developers to work on a project simultaneously and collaboratively. In a DVCS, each developer has a complete copy of the entire project's history, including all files and changes made over time. **This is in contrast to Centralized Version Control Systems (CVCS), where all changes are stored in a central server, and developers need to connect to the server to access the repository.**

In a DVCS, every developer has a local repository, which contains the full history of the project. This local repository enables developers to work offline, commit changes, create branches, and merge changes without requiring constant connectivity to a central server. Each developer can have their own branches to work on different features or fixes independently.

When developers want to share their changes with others or synchronize their work, they can push their changes from their local repository to a shared central repository or pull changes from others' repositories into their own. This allows for efficient collaboration and facilitates parallel development.

Some popular examples of Distributed Version Control Systems include:

- **Git:** Git is currently the most widely used DVCS, known for its speed, flexibility, and robust branching and merging capabilities.
- **Mercurial:** Mercurial is another popular DVCS, known for its simplicity and ease of use.

DVCS systems have become the preferred choice for many software development teams due to their flexibility, ability to work offline, and efficient collaboration capabilities, making them a significant improvement over traditional centralized version control system.

Advantages of Distributed Version Control System (DVCS):

- **Offline Work:** Developers can work offline with a complete copy of the repository, allowing them to make commits, create branches, and perform other version control operations without a network connection.
- **Flexibility in Development Workflows:** DVCS, especially Git, excels in branching and merging, providing flexibility for different development workflows. Developers can easily create branches for features, bug fixes, or experiments and merge changes seamlessly.
- **Parallel Development:** Multiple developers can work concurrently on different branches without interfering with each other. This promotes parallel development and speeds up the overall development process.
- **Local Operations:** Most version control operations, such as commit, log, and diff, can be performed locally without the need for constant network access. This results in faster performance and reduces dependence on a central server.
- **Redundancy and Backup:** Since each developer has a complete copy of the repository, the risk of data loss is reduced. Multiple copies of the project's history exist, providing a form of redundancy and backup.
- **Community Collaboration:** Platforms like GitHub and GitLab leverage DVCS, enabling collaboration on open-source projects and facilitating contributions from a global community.

Disadvantages of Centralized Version Control System (CVCS):

- **Network Dependency:** CVCS relies heavily on a central server. Developers need constant network access to perform version control operations, which can be a limitation in some environments, especially when working offline or in areas with poor connectivity.
- **Concurrency Challenges:** Concurrent development can be more challenging in CVCS, as multiple developers might face conflicts when committing changes simultaneously. This can lead to a more sequential and less parallelized development process.
- **Limited Branching and Merging:** Branching and merging can be more complex in CVCS compared to DVCS. Branches may require locks to avoid conflicts, and merging may involve more manual intervention.
- **Single Point of Failure:** The central server represents a single point of failure. If the central server goes down, developers may experience disruptions in version control operations until the server is restored.
- **Backup Complexity:** Since the entire version history is stored on the central server, ensuring regular and comprehensive backups becomes critical. Loss of the central repository can result in significant data loss.

While both CVCS and DVCS have their strengths and weaknesses, DVCS, especially exemplified by Git, has gained widespread adoption due to its flexibility, performance, and support for distributed and collaborative development workflows. Many modern development teams prefer DVCS for its advantages in handling complex and dynamic software projects.

What is Git?

Git is a modern and widely used **distributed version control** system in the world. It is developed to manage projects with high speed and efficiency. The version control system allows us to monitor and work together with our team members at the same workspace.

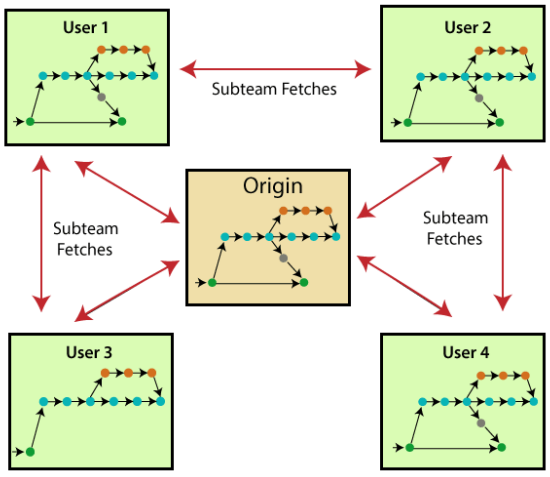
Git is foundation of many services like **GitHub** and **GitLab**, but we can use Git without using any other Git services. Git can be used **privately** and **publicly**.

Git was created by **Linus Torvalds** in **2005** to develop Linux Kernel. It is also used as an important distributed version-control tool for **the DevOps**.

Git is easy to learn, and has fast performance. It is superior to other SCM tools like Subversion, CVS, Perforce, and ClearCase.

Features of Git:

- **Open Source:** Git is an **open-source tool**. It is released under the **GPL** (General Public License) license.
- **Scalable:** Git is **scalable**, which means when the number of users increases, the Git can easily handle such situations.
- **Distributed:** One of Git's great features is that it is **distributed**. Distributed means that instead of switching the project to another machine, we can create a "**clone**" of the entire repository. Also, instead of just having one central repository that you send changes to, every user has their own repository that contains the entire commit history of the project. We do not need to connect to the remote repository; the change is just stored on our local repository. If necessary, we can push these changes to a remote repository.



- **Security:** Git is secure. It uses the **SHA1 (Secure Hash Function)** to name and identify objects within its repository. Files and commits are checked and retrieved by its checksum at the time of checkout. It stores its history in such a way that the ID of particular commits depends upon the complete development history leading up to that commit. Once it is published, one cannot make changes to its old version.

- **Speed:** Git is very fast, so it can complete all the tasks in a while. Most of the git operations are done on the local repository, so it provides a huge speed. Also, a centralized version control system continually communicates with a server somewhere.

  Performance tests conducted by Mozilla showed that it was extremely fast compared to other VCSs. Fetching version history from a locally stored repository is much faster than fetching it from the remote server. The core part of Git is written in C, which ignores runtime overheads associated with other high-level languages.

  Git was developed to work on the Linux kernel; therefore, it is capable enough to handle large repositories effectively. From the beginning, speed and performance have been Git's primary goals.

- **Supports non-linear development:** Git supports seamless branching and merging, which helps in visualizing and navigating a non-linear development. A branch in Git represents a single commit. We can construct the full branch structure with the help of its parental commit.

- **Branching & Merging:**
  Branching and merging are the great features of Git, which makes it different from the other SCM (Source Code Management) tools. Git allows the creation of multiple branches without affecting each other. We can perform tasks like creation, deletion, and merging on branches, and these tasks take a few seconds only. Below are some features that can be achieved by branching:

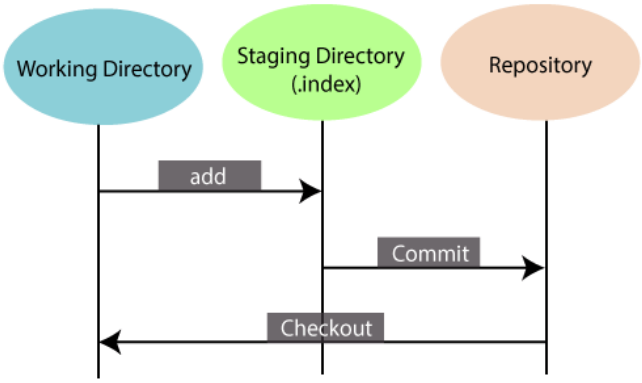
  - We can **create a separate branch** for a new module of the project, commit and delete it whenever we want.
  - We can have a **production branch**, which always has what goes into production and can be merged for testing in the test branch.
  - We can create a **demo branch** for the experiment and check if it is working. We can also remove it if needed.
  - The core benefit of branching is if we want to push something to a remote repository, we do not have to push all of our branches. We can select a few of our branches, or all of them together.

- **Data Assurances:** The Git data model ensures the **cryptographic integrity** of every unit of our project. It provides a **unique commit ID** to every commit through a **SHA algorithm**.

  We can **retrieve** and **update** the commit-by-commit ID. Most of the centralized version control systems do not provide such integrity by default.

- **Staging Area:** The Staging area is also a unique functionality of Git. It can be considered as a preview of our next commit, moreover, an intermediate area where commits can be formatted and reviewed before completion. When you make a commit, Git takes changes that are in the staging area and makes them as a new commit. We are allowed to add and remove changes from the staging area. The staging area can be considered as a place where Git stores the changes.

Although, Git doesn't have a dedicated staging directory where it can store some objects representing file changes (blobs). Instead of this, it uses a file called index.

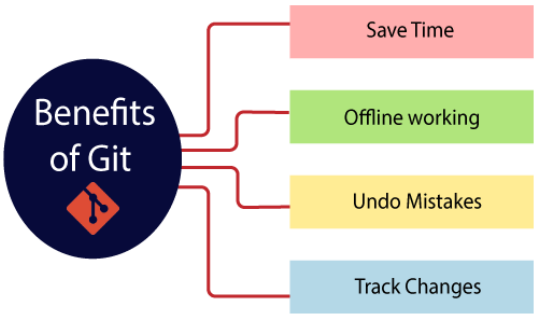


Another feature of Git that makes it apart from other SCM tools is that it is possible to quickly stage some of our files and commit them without committing other modified files in our working directory.

- **Maintain the clean history:** Git facilitates with Git Rebase; It is one of the most helpful features of Git. It fetches the latest commits from the (master or others) branch and puts our code on top of that. Thus, it maintains a clean history of the project.

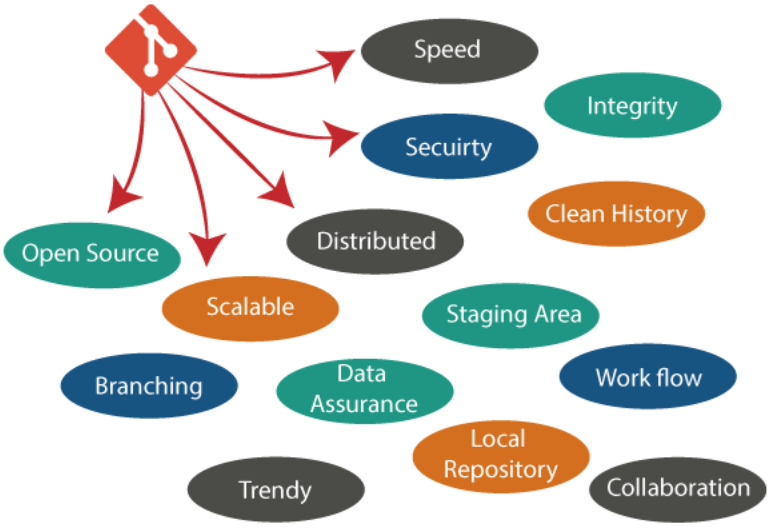
Benefits of Git

A version control application allows us to **keep track** of all the changes that we make in the files of our project. Every time we make changes in files of an existing project, we can push those changes to a repository. Other developers are allowed to pull your changes from the repository and continue to work with the updates that you added to the project files.



Why Git?

We have discussed many **features** and **benefits** of Git that demonstrate the undoubtedly Git as the **leading version control system**. Now, we will discuss some other points about why should we choose Git.



What is Git SSH Key?

An SSH key is an access credential for the SSH (secure shell) network protocol. This authenticated and encrypted secure network protocol is used for remote communication between machines on an unsecured open network. SSH is used for remote file transfer, network management, and remote operating system access. The SSH acronym is also used to describe a set of tools used to interact with the SSH protocol.

SSH uses a pair of keys to initiate a secure handshake between remote parties. The key pair contains a public and private key. The private vs public nomenclature can be confusing as they are both called keys. It is more helpful to think of the public key as a "lock" and the private key as the "key". You give the public 'lock' to remote parties to encrypt or 'lock' data. This data is then opened with the 'private' key which you hold in a secure place.

The mechanism that Git uses for this check summing is called a SHA-1 hash. This is a **40-character** string composed of **hexadecimal characters (0–9 and a–f)** and calculated based on the contents of a file or directory structure in Git. A SHA-1 hash looks something like this:

24b9da6552252987aa493b52f8696cd6d3b00373

Git Generally Only Adds Data:

When you do actions in Git, nearly all of them only **add** data to the Git database. It is hard to get the system to do anything that is not undoable or to make it erase data in any way. As with any VCS, you can lose or mess up changes you haven't committed yet, but after you commit a snapshot into Git, it is very difficult to lose, especially if you regularly push your database to another repository.

This makes using Git a joy because we know we can experiment without the danger of severely screwing things up. For a more in-depth look at how Git stores its data and how you can recover data that seems lost, see Undoing Things.

Three States of Git:

In Git, files go through three main states as developers work on them. These states are commonly referred to as the "Three States of Git." The states are:

- **Working Directory:** The working directory is where you do your work. It contains the actual files of your project. When you modify a file in your working directory, Git recognizes it as a "**modified**" file.

- **Example:** Assume you have a file called **example.txt** in your project. You open the file and make some changes, adding new content. At this point, **example.txt** is in the "modified" state in your working directory.

```plaintext
# content of example.txt in the working directory
This is some text in example.txt.
I am making changes.
```

- **Staging Area (Index):** The staging area, also known as the index, is a space where you prepare changes before committing them. Files in the staging area are snapshots of the modified files from the working directory. You use the **git add** command to move changes from the working directory to the staging area.

  - **Example:** To commit the changes, you need to stage them. You use the following command: Now, the changes you made to example.txt are in the staging area. The staging area is a snapshot of the file as it stands in your working directory at this moment.

```bash
git add example.txt
```

- **Repository (HEAD):** The repository, or HEAD, represents the commit history of your project. When you commit changes, you create a snapshot of the files in the staging area, and this snapshot becomes a part of the Git repository. The HEAD is a pointer to the latest commit in the current branch.

  - **Example:** To finalize the changes, you commit them to the repository. You use the following command: The changes are now committed, and the HEAD points to the latest commit in your branch. The working directory and staging area are now clean.

```bash
git commit -m "Added new content to example.txt"
```

Git Repository:

In Git, a repository is a data structure that stores metadata for a set of files and directories, along with a history of changes to those files. It includes the entire history of the project, starting from the initial commit up to the latest changes. The term **"HEAD"** in Git refers to the latest commit in the currently checked-out branch of the repository.

Let's break down these concepts:

- **Git Repository:**
  - A Git repository is a directory or folder in which Git version control is being used. It contains a hidden subfolder named "**.git**", which houses the internal data structure and configuration information required for version control. The "**.git**" folder is what makes a directory a Git repository.

- **HEAD:**
  - In Git, the term **"HEAD"** refers to a special pointer or reference that points to the latest commit in the currently checked-out branch. It essentially represents the current state of your working directory. The **HEAD** reference can also point to a specific commit or a branch, depending on the context.

- When you create a new commit, the **HEAD** is updated to point to that new commit, and the branch reference is moved forward. In other words, HEAD is always pointing to the tip of the branch you currently have checked out.

- If you're on a specific branch, the **HEAD** points to the latest commit on that branch. If you're in a detached **HEAD** state (not on any branch), the HEAD directly points to a specific commit.

Is GitHub and GitLab Git Repository?

GitHub and GitLab are not Git repositories themselves; instead, they are web-based platforms that provide hosting services for Git repositories and offer additional features to facilitate collaborative software development. **These platforms act as hosts for Git repositories**, providing a centralized location where developers can store, manage, and collaborate on their code.

Here's a breakdown of the terms:

- **Git:** Git is a distributed version control system used for tracking changes in source code during software development. It allows multiple developers to collaborate on a project by providing a mechanism for tracking changes, managing branches, and facilitating code collaboration.

- **GitHub: GitHub is a web-based platform that provides hosting services for Git repositories.** It includes features such as issue tracking, pull requests, code review, and project management tools. GitHub is widely used for open-source projects and facilitates social coding by allowing developers to collaborate easily.

- **GitLab:** GitLab is another web-based platform that offers similar services to GitHub. It provides a centralized location for hosting Git repositories and includes features like continuous integration, issue tracking, and project management. GitLab can be used as a self-hosted solution, allowing organizations to run their own instance of GitLab on their servers.

In summary, GitHub and GitLab are platforms built around Git, providing web-based interfaces for managing and collaborating on Git repositories. Developers use these platforms to host their Git repositories, manage project workflows, and collaborate with other team members. While Git is the version control system underlying these platforms, GitHub and GitLab offer additional tools and features to enhance the software development process.

What is GitHub:

GitHub is a Git repository hosting service. GitHub also facilitates many of its features, such as access control and collaboration. It provides a Web-based graphical user interface.

GitHub is an American company. It hosts source code of your project in the form of different programming languages and keeps track of the various changes made by programmers.

GitHub offers both **distributed version control and source code management (SCM)** functionality of Git. It also facilitates some collaboration features such as bug tracking, feature requests, task management for every project.

Features of GitHub:
GitHub is a place where programmers and designers work together. They collaborate, contribute, and fix bugs together. It hosts plenty of open-source projects and codes of various programming languages.

Some of its significant features are as follows:

- Collaboration
- Integrated issue and bug tracking
- Graphical representation of branches
- Git repositories hosting
- Project management
- Team management
- Code hosting
- Track and assign tasks

Difference between Git & GitHub:

Git	GitHub
Git is software.	It is a service.
Linux maintains Git.	Microsoft maintains GitHub.
It is a command-line tool.	It is a graphical user interface.
You can install it locally on the system.	It is hosted on the web. It is exclusively cloud-based.
It focuses on code sharing and version control.	It focuses on centralized source code hosting.
It is open-source licensed.	It has a free-tier and pay-for-use tier.

Git Environment Setup:
The environment of any tool consists of elements that support execution with software, hardware, and network configured. It includes operating system settings, hardware configuration, software configuration, test terminals, and other support to perform the operations. It is an essential aspect of any software.

It will help you to understand how to set up Git for first use on various platforms so you can read and write code in no time.

Git Configuration:
Git configuration refers to the settings and parameters that **determine how Git behaves on a specific system or for a particular project.** These configurations can be set at different levels: **system-wide**, **user-specific**, and **project-specific.** Git configuration allows users to customize various aspects of Git's behavior, such as user information, default behaviours, and external tools integration.

Git configuration refers to the settings and options that control the behavior of the Git version control system on your local machine. Git configuration allows you to customize various aspects of Git's behavior, such as user identity, default text editor, merge and diff tools, aliases, and more.

Git configuration is necessary because it allows you to tailor Git to your specific needs and preferences. Without configuration, Git would use default settings, which might not align with your desired workflow.
Here are some key reasons why Git configuration is essential:

- **User Identity:** Git configuration allows you to set your name and email, which are used to identify you as the author of commits. This is crucial for accurately tracking contributions in collaborative projects.

- **Default Text Editor:** Git needs a text editor for certain operations, such as writing commit messages. With configuration, you can set your preferred text editor for a smoother workflow.

- **Merge and Diff Tools:** You can configure external merge and diff tools to handle conflicts more efficiently when merging branches or viewing differences between files.

- **Aliases:** Git configuration allows you to create aliases, which are shorthand commands for frequently used Git operations. This saves time and reduces the need to type lengthy commands repeatedly.

- **Remote Repository URLs:** Configuring remote URLs enables Git to know where to push and pull changes when collaborating with others on a shared repository.

- **Branch Settings:** Git configuration allows you to set up default branch names and upstream branches, which determine where to push and pull changes by default.

- **Global and Local Configuration:** Git configuration can be set at different levels - **system-wide**, **user-specific**, and **repository-specific**. This flexibility allows you to customize settings globally or on a per-repository basis.

By configuring Git, you make it work the way you want, enhancing your productivity and making collaboration with others more seamless. Git's flexibility in configuration is one of its strengths, as it can adapt to different development scenarios and team preferences.

System-Level Configuration:

- **Purpose:** This level is typically used to set configurations that should be shared by all users on the system. It's less common for individual users to modify system-level configurations unless they have administrative privileges.

- **Location:** The system-level configuration is stored in a file that applies to all users and repositories on a given system. It is usually located in /etc/gitconfig or /usr/local/etc/gitconfig on Unix-like systems.

- **Command to Set System-level Config:**

```bash
git config --system
```

User-Level Configuration:

- **Purpose:** User-level configuration is where individual users set their personal preferences for Git. It overrides system-level configurations and is specific to a particular user on a given system. Settings such as user name, email, and default editor are commonly configured at this level.

- **Location:** The user-level configuration is stored in a file located in the user's home directory. The file is commonly named "**.gitconfig**" or can be stored in the .config/git/config directory.

- **Command to Set User-level Config:**

```bash
git config --global
```

Repository-Level Configuration:

- **Purpose:** This level is used to configure settings that are specific to a particular Git repository. These configurations override both system-level and user-level settings for that repository. It's useful for project-specific settings or configurations that should not be shared globally.

- **Location:** The repository-level configuration is stored in the .git/config file within a specific Git repository.

- **Command to Set Repository-level Config:**

```bash
git config
```

no –system and –global flags

Configuration Example:

- **User Information (User-level):**

```bash
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

- **Default Editor (User-level):**

```bash
git config --global core.editor "your-favorite-text-editor"
```

- **Repository Default Branch Name (Repository-level):**

```bash
git config init.defaultBranch "main"
```

View Configuration:

To view the current configuration settings at a specific level, you can use the --list option with the git config command:

```bash
git config --system --list
git config --global --list
git config --local --list
```

Understanding and managing these configuration levels allows users to tailor their Git environment based on their preferences, ensure consistency across projects, and set project-specific configurations when needed.

Git supports a command called **git config** that lets you get and set configuration variables that control all facets of how Git looks and operates. It is used to set Git configuration values on a global or local project level.

Setting **user.name** and **user.email** are the necessary configuration options as your name and email will show up in your commit messages.

The git config command can accept arguments to specify the configuration level. The following configuration levels are available in the Git config.

- **local**
- **global**
- **system**

The **--system**, **--global**, and **--local** flags in Git are used to specify the level at which you want to set or view configuration options. Each flag corresponds to a different scope, affecting the system-wide, user-wide, or repository-specific settings. Understanding and using these flags appropriately is crucial for customizing Git behaviour based on different contexts. Here's a breakdown of their importance and purposes:

--system flag:

- **Purpose:** This flag sets or views configurations at the system level. System-level configurations apply to all users and repositories on a given system.
- **Importance:** Typically, system-level configurations are managed by administrators and are shared across all users on a machine. They provide a way to enforce global settings for Git.
- **Example Commands:**

```bash
git config --system user.name "System User"
git config --system --list
```

--global flag:

- **Purpose:** This flag sets or views configurations at the user level. User-level configurations apply to a specific user and override system-level settings.
- **Importance:** Users can customize their Git experience by setting personal preferences such as user name, email, and default editor. These configurations are specific to the user and apply across all repositories for that user.
- **Example Commands:**

```bash
git config --global user.name "Your Name"
git config --global --list
```

--local flag:

- **Purpose:** This flag sets or views configurations at the repository level. Local configurations apply only to the current Git repository and override both system-level and user-level settings for that repository.

- **Importance:** Local configurations are project-specific and allow developers to customize settings based on the requirements of a particular repository. It's useful for maintaining project-specific defaults or for settings that shouldn't be shared globally.

- **Example Commands:**

```bash
git config user.email "your.email@example.com"
git config --local --list
```

Best Practices:

- **Precedence:** Configurations at the **--local** level take precedence over **--global**, and **--global** takes precedence over **--system**. In other words, local configurations in a repository override global settings, and global settings override system-wide settings.

- **Use Cases:**
  - **--system** is typically managed by administrators and is less commonly modified by individual users.
  - **--global** is useful for personalizing Git behaviour across multiple repositories.
  - **--local** is beneficial for project-specific configurations that should not affect other repositories.

Following are the Git commands which are being covered:

git config command:

The **git config** command is used to set or get configuration options for Git at different levels: **system**, **user**, and **repository**. These configurations define various settings that influence the behaviour of Git. The purpose and importance of **git config** are as follows:

- **Purpose:**
  - **Customizing Git Behaviour:** Allows users to customize how Git behaves on their system. and Configurations include user information, default editor, color preferences, and more.
  - **Setting Identity Information:** Configures user name and email, which are important for recording commit authorship.
  - **Defining Defaults:** Sets default behaviours, such as the default branch name or the default merge strategy.
  - **Configuring Aliases:** Enables the creation of aliases for Git commands, making them shorter or more intuitive.
  - **Specifying External Tools:** Configures external tools to be used for viewing, merging, and comparing changes.

- **Importance:**
  - **User Identity:** Setting user.name and user.email is crucial for accurate attribution of commits.
  - **Consistent Workflows:** Allows users to define consistent workflows across projects by setting default behaviours.
  - **Personalization:** Enables personalization of the Git environment based on individual preferences.
  - **Efficiency:** Configuring aliases helps in reducing the amount of typing required for common Git commands, improving efficiency.
  - **Collaboration:** Ensures that commit authorship information is correctly configured, which is important for collaboration and tracking contributions.
- **Level of Configuration:**
  - **System-level (--system):** Applies to all users and repositories on a given system. And typically set by administrators.
  - **User-level (--global):** Applies to a specific user on a system and overrides system-level configurations. And personalizes the Git environment for an individual user.
  - **Repository-level (no flag):** Applies to a specific Git repository and overrides both system-level and user-level configurations. And useful for project-specific settings that shouldn't be shared globally.
- **When it is Used:**
  - **First-Time Setup:** When setting up Git for the first time on a system, users configure their identity and default behaviours.
  - **Project Initialization:** When creating a new Git repository **(git init)**, users may configure repository-specific settings.
  - **Customizing Workflows:** Throughout the development process, users may adjust configurations to suit their workflow or project requirements.
  - **Collaborative Development:** Configurations are crucial for collaborative development, ensuring consistency in commit authorship and workflow.
  - **Troubleshooting and Tuning:** Users may use **git config** to troubleshoot issues, adjust settings, or customize Git to better suit their needs.
- **Example:**
  - **Set the name:**
    $ git config --global user.name "User name"
  - **Set the email:**
    $ git config --global user.email “email id”
    $ git config --global user.email “himanshudubey481@gmail.com”
  - **Set the default editor:**
    $ git config --global core.editor Vim
  - **Check the setting:**
    $ git config --list

- **Set up an alias** for each command:
  $ git config --global alias.co checkout
  $ git config --global alias.br branch
  $ git config --global alias.ci commit
  $ git config --global alias.st status

In summary, the git config command is fundamental for configuring and customizing the Git environment. It is used during initial setup, project initialization, and ongoing development to define identity, set defaults, and personalize the Git workflow. Understanding and using git config appropriately contribute to a smoother and more efficient version control experience.

git init command:

The **git init** command is used to initialize a new Git repository. When executed in a directory, it transforms that directory into a new Git repository, marking the start of version control for a project. The purpose and importance of **git init** are as follows:

- **Purpose:**
  - **Version Control Initialization:** Establishes a new repository for version control, enabling the tracking of changes to files.
  - **Creation of Hidden .git Directory:** Creates a hidden subdirectory named **.git** that contains the version control metadata, including the object database, configuration files, and other essential components.
  - **Commit History Establishment:** Initializes the commit history, allowing for the recording and tracking of changes made to files over time.
  - **Branch Initialization:** Creates the default branch, typically named "master" (though Git has moved to using "main" as a default name in more recent versions).
  - **Staging Area Setup:** Sets up the staging area, also known as the index, where changes to files are prepared before being committed.
- **Importance:**
  - **Version Control Kickstart:** Marks the beginning of version control for a project, allowing developers to track changes and collaborate effectively.
  - **Commit Tracking:** Enables the recording of changes, creating a commit history that can be used to understand the evolution of the codebase.
  - **Branching and Merging:** Initializes the default branch **(e.g., "main")** and sets up the repository for branching and merging, facilitating parallel development.
  - **Staging Area Preparation:** Sets up the staging area, providing a space where changes can be reviewed and organized before being committed.
  - **Metadata Generation:** Generates the necessary metadata and configuration files within the **.git** directory, making version control operations possible.
- **When it is Used:**
  - **New Project Creation:** When starting a new software project, developers typically use **git init** to initiate version control for the project.

- **Existing Project Conversion:** If an existing project needs to be brought under version control, developers may use **git init** in the project's root directory.
- **Repository Cloning:** When developers clone a remote repository using **git clone**, the command implicitly performs an initialization of the new repository.
- **Starting a New Feature Branch:** Developers may use **git init** when creating a new feature branch within an existing repository.
- **Temporary or Experimental Repositories:** For small, temporary projects or experimental code, developers might use **git init** to quickly set up version control.

■ **Notes:**

- After running git init, developers typically follow up with commands like **git add** to stage changes and **git commit** to make the first commit.
- For existing projects, consider using **git clone** to clone an existing repository rather than initializing a new one with **git init**.
- The default branch name has evolved from **"master"** to **"main"** in newer Git versions, reflecting a move towards more inclusive language.

■ **Examples:**

- **Create a local repository:**
  $ git init
- **Make a local copy** of the server repository.
  $ git clone <link_gihub>

In summary, **git init** is a foundational command in Git that initializes a new repository, enabling version control for a project. It is used when starting a new project, converting an existing project to version control, or when creating temporary or experimental repositories.

git add command:

The **git add** command is used to add changes in the **working directory to the staging area**. The staging area is an intermediate area where you prepare changes before making a commit. The purpose and importance of git add are as follows:

■ **Purpose:**

- **Staging Changes:** Adds modifications, new files, or file deletions to the staging area, preparing them for the next commit.
- **Selective Committing:** Allows for selective committing by choosing which changes to include in the next commit.
- **Reviewing Changes:** Provides an opportunity to review and organize changes before committing them to the version control history.

■ **Importance:**

- **Control Over Commit Content:** Grants developers control over the content of each commit by selecting specific changes to include in the next commit.
- **Logical Commits:** Facilitates the creation of logical and granular commits, each representing a specific feature, bug fix, or improvement.

- **Commit Review:** Allows developers to review changes in the staging area before committing, ensuring that only desired changes are included.

- **Versioning Control:** Establishes a clear distinction between changes in the working directory and changes that are ready to be committed, providing a structured versioning workflow.

- **When it is Used:**
  - **Before Committing:** After making changes to files in the working directory, developers use **git add** to stage those changes before committing them.

  - **Selective Staging:** When developers want to commit only specific changes, they use **git add** with specific file paths or use patterns to stage only certain changes.

  - **New File Addition:** Before committing a new file, developers use **git add** to include the new file in the staging area.

  - **Deletion Staging:** When a file is deleted, **git** add is used to stage the deletion for the next commit.

  - **Conflict Resolution:** After resolving conflicts during a merge or rebase, **git add** is used to stage the resolved changes before completing the operation.

- **Examples:**
  - **Add a file** to staging (Index) area:
    $ git add <filename.txt>

  - **Add all files** of a repo to staging (Index) area:
    $ git add .

  - **Add specific directory**
    $ git add <directory_name/>

In Summary, After using **git add**, the changes are in the staging area but not yet committed. The next step is to use **git commit** to create a commit with the staged changes. The staging area allows for a more controlled and deliberate approach to versioning, promoting a clean and organized version control history. The **git add** command is an essential part of the Git workflow, contributing to the flexibility and power of version control by enabling selective and well-organized commits.

git commit command:

The **git commit** command is used to record changes made to the files in the staging area, creating a new commit in the version control history. The purpose and importance of **git commit** are as follows:

- **Purpose:**
  - **Record Changes:** Creates a snapshot of changes in the staging area and records them as a new commit in the Git repository.

  - **Versioning:** Establishes a point in the project's history, allowing for versioning and tracking changes over time.

  - **Commit Message:** Requires a commit message that describes the changes made in the commit, providing a clear and concise summary of the modifications.

- **Importance:**
  - **Commit History:** Builds a chronological history of changes, helping developers understand the evolution of the codebase.

- **Rollback Capability:** Enables the rollback to previous states by reverting to earlier commits in the history.
- **Collaboration:** Facilitates collaboration by allowing developers to share their changes with others through commits.
- **Logical Organization:** Promotes a logical organization of changes into meaningful and atomic commits, each representing a specific feature, bug fix, or improvement.

■ **When it is Used:**

- **After Staging Changes:** Once changes are staged using **git add**, the **git commit** command is used to permanently store those changes in the version history.
- **After Conflict Resolution:** When resolving conflicts during a merge or rebase, developers use **git commit** to finalize the resolution and create a new commit.
- **After Interactive Staging:** After interactively staging changes **(git add -p)**, developers use **git commit** to commit the selected changes.
- **After Commit Amendments:** If a commit needs to be amended (changed or updated), developers use **git commit --amend** to modify the last commit.

■ **Examples:**

- **Basic Commit**
  $ git commit -m “this is my first commit message”
- **Commit with All Staged Changes:**
  $ git commit -am “commit all changes”
- **Commit with Interactive Staging:**
  $ git add -p
  $ git commit -m “commit selectively staged changes”
- **Amend the Last Commit** (modify the last commit)**:**
  $ git commit --amend -m “update the commit messages”

In summary, **git commit** is a fundamental command in Git that records changes, creating a permanent snapshot in the version history. It plays a central role in version control, facilitating collaboration, history tracking, and the logical organization of project changes.

git diff command:

The **git diff** command is used to display the differences between two states of a Git repository. It can show differences between the working directory and the staging area, differences between the staging area and the last commit, or differences between any two commits, branches, or tags. The purpose and importance of **git diff** are as follows:

■ **Purpose:**

- **Viewing Changes:** Shows the line-by-line differences between files in different states (e.g., working directory, staging area, or commits).
- **Reviewing Modifications:** Helps developers review changes before committing, ensuring that only the desired modifications are included.

- **Identifying Unstaged Changes:** Highlights changes in the working directory that are not yet staged for the next commit.
- **Understanding History:** Allows users to explore and understand the differences between different versions of the codebase.

■ **Importance:**

- **Pre-Commit Review:** Enables developers to review and understand changes before committing them to version control.
- **Conflict Resolution:** Assists in resolving conflicts during operations like merge or rebase by visualizing conflicting changes.
- **Comparing Commits:** Facilitates the comparison of different commits, branches, or tags to understand the evolution of the codebase.
- **Selective Staging:** Helps in selectively staging changes for the next commit using the interactive mode **(git add -p)**.

■ **When it is Used:**

- **Before Committing:** Developers use **git diff** to review changes made in the working directory before committing them.
- **Before Staging:** To review unstaged changes, developers can use **git diff** without any additional arguments.
- **Comparing with Staged Changes:** To see the differences between the staging area and the last commit, developers use **git diff --staged** or **git diff --cached**.
- **Comparing Commits or Branches:** Developers use **git diff** to compare different commits, branches, or tags to understand the changes made over time.
- **Conflict Resolution:** During a merge or rebase operation, developers use **git diff** to identify conflicting changes and resolve conflicts.

■ **Examples:**

- **Track the changes that have not been staged:**
  $ git diff
- **Track the changes that have staged but not committed:**
  $ git diff --staged
- **Track the changes after committing a file:**
  $ git diff HEAD
- **View the Staged changes:**
  $ git diff --staged
- **Compare the two commits:**
  $ git diff <commit_1_id> <commit_2_id>
- **Track the changes between two commits:**
  $ git diff <brance_1_name> < branch_2_name>

In summary, **git diff** is a valuable command for reviewing changes, identifying differences, and understanding the evolution of a Git repository. It plays a crucial role in the pre-commit review process, conflict resolution, and the exploration of code history.

git status command:

The **git status** command is used to display the state of the **working directory** and the **staging area** in a Git repository. It provides information about which files have been modified, which files are staged for the next commit, and which files are untracked. The purpose and importance of **git status** are as follows:

- **Purpose:**
  - **Status Overview:** Shows the current status of the working directory, indicating changes that have been made since the last commit.
  - **Staging Area Information:** Displays which changes are staged (added to the index) and ready for the next commit.
  - **Untracked Files:** Identifies files in the working directory that are not yet tracked by Git.
- **Importance:**
  - **Pre-Commit Review:** Allows developers to review changes before committing, providing a summary of modified, staged, and untracked files.
  - **Workflow Guidance:** Helps users understand the state of their repository, guiding them on which actions to take next (e.g., commit, stage, or discard changes).
  - **Conflict Identification:** Indicates if there are conflicts that need resolution after operations like merge or rebase.
  - **Untracked Files Awareness:** Alerts developers to files in the working directory that are not yet being tracked by Git, prompting them to decide whether to stage or ignore them.
- **When it is Used:**
  - **Before Committing:** Developers use **git status** to review changes made in the working directory before committing them.
  - **Before Staging:** To see which changes are unstaged, developers run **git status** without any additional arguments.
  - **After Staging:** After using git add to stage changes, **git status** provides information about the staged changes and what is ready for the next commit.
  - **Conflict Resolution:** During or after conflict resolution (e.g., after a merge or rebase), developers use **git status** to check for any remaining conflicts.
  - **Untracked Files Check:** To identify and decide how to handle untracked files in the working directory.
- **Example:**
  - **Display the state of the working directory and the staging area:**
    $ git status
  - **Status for specific path:**
    $ git status <path/to/>

In summary, **git status** is a fundamental command in Git that provides developers with an overview of the state of their working directory and staging area. It is an essential tool for understanding the status of changes, guiding the workflow, and making informed decisions about committing, staging, or ignoring files.

git show command:

The **git show** command is used to display information about a specific Git object, such as a commit, tag, or tree. It shows the details of a single Git object, including the commit message, author information, and the changes introduced in the commit. The purpose and importance of **git show** are as follows:

- **Purpose:**
  - **Display Commit Information:** Shows detailed information about a specific commit, including the commit message, author, and date.
  - **View Changes Introduced:** Displays the changes introduced in a specific commit, including additions and deletions of lines.
  - **Show Tag or Tree Information:** Can be used to view information about tags, trees, and other Git objects.
- **Importance:**
  - **Commit Review:** Facilitates a detailed review of a specific commit's content and changes.
  - **Understanding History:** Helps developers understand the evolution of the codebase by examining the details of past commits.
  - **Verification of Tags:** Assists in verifying information about tags, ensuring that the correct information is associated with a specific tag.
- **When it is Used:**
  - **Viewing Commit Information:** Developers use **git show** to view detailed information about a specific commit.
  - **Reviewing Changes in a Commit:** To inspect the changes introduced by a particular commit, developers use **git show**.
  - **Inspecting Tags:** When working with tags, **git show** can be used to display information about a specific tagged commit.
- **Examples:**
  - **View latest commit information:**
    $ git show
  - **View specific commit information:**
    $ git show <commit_id>
  - **View the information about tag:**
    $ git show <tag_name>

In summary, **git show** is a valuable command for inspecting and reviewing specific Git objects, particularly commits. It is used to display detailed information about a commit, including changes, authorship, and commit metadata. This command is essential for understanding the history of a Git repository and verifying information about specific commits or tags.

git log command:

The **git log** command is used to display a chronological log of commits in a Git repository. It provides information about the commit history, including details such as **commit hashes**, **authorship**, **dates**, and **commit messages**. The purpose and importance of **git log** are as follows:

- **Purpose:**
  - **Viewing Commit History:** Displays a detailed and chronological list of commits made in the repository.
  - **Authorship and Dates:** Shows information about each commit, including the author's name, email, commit date, and commit message.
  - **Branch Information:** Indicates the branching structure, illustrating where branches diverge and merge.
  - **Filtering Commits:** Provides options for filtering the commit history based on criteria such as author, date range, or file changes.
- **Importance:**
  - **Code Evolution Understanding:** Helps developers understand how the codebase has evolved over time by reviewing the commit history.
  - **Troubleshooting and Debugging:** Assists in identifying when specific changes were introduced, aiding in troubleshooting and debugging.
  - **Collaboration and Code Review:** Facilitates collaboration by allowing team members to review each other's work through the commit history.
  - **Branch Visualization:** Visualizes the branching and merging history, making it easier to follow the development flow of different features and bug fixes.
- **When it is Used:**
  - **Commit History Review:** Developers use **git log** to review the commit history of a repository, gaining insights into the changes made over time.
  - **Identifying Changes:** When investigating an issue or debugging, developers use **git log** to identify when specific changes were made.
  - **Branch Visualization:** To understand the branching and merging history, developers use **git log --graph** to visualize the commit graph.
  - **Code Review:** During code review, team members use **git log** to review the commits made by others and understand the context of changes.
- **Examples:**
  - **Display the most recent commits and the status of the head:**
    $ git log
  - **Display the output as one commit per line:**
    $ git log --oneline
  - **Displays the files that have been modified:**
    $ git log --stat
    $ git log --stat <commid_id>

- **Display the graphically:**
  $ git log --oneline --graph --all
  $ git log --oneline --graph

- **Display the modified files with location:**
  $ git log -p

- **Filter by author:**
  $ git log --author="dibyendubiswas1998"

- **Show the File changes:**
  $ git log -- filename.txt

In summary, **git log** is a fundamental command in Git that provides developers with a detailed view of the commit history. It is a valuable tool for understanding code evolution, troubleshooting, collaborating with team members, and visualizing the branching structure of a Git repository.

git blame command:

The **git blame** command is used to show what revision and author last modified each line of a file. It helps track changes to specific lines of code in a Git repository, showing the commit information and authorship for each line in a file. The purpose and importance of **git blame** are as follows:

- **Purpose:**
  - **Line-by-Line Annotation:** Annotates each line of a file with the commit hash and author who last modified that line.
  - **Code Accountability:** Helps identify the person responsible for specific changes or additions to the code.
  - **Historical Context:** Provides a historical context for each line, helping developers understand when and why changes were made.
- **Importance:**
  - **Code Understanding:** Facilitates a deeper understanding of the codebase by revealing the historical evolution of each line.
  - **Accountability and Collaboration:** Helps developers take responsibility for their code and fosters collaboration by providing clear authorship information.
  - **Troubleshooting and Debugging:** Aids in troubleshooting and debugging by identifying the commit associated with specific code lines.
- **When it is Used:**
  - **Identifying Changes:** Developers use **git blame** to identify when and by whom specific lines of code were last modified.
  - **Code Review:** During code reviews, **git blame** can be used to understand the origin of particular code sections and discuss changes with team members.
  - **Bug Investigation:** When investigating bugs or issues, **git blame** can help pinpoint the commit introducing the problematic code.
- **Examples:**
  - **Basic Usages:**

$ git blame <file_name>

- **Show Author Names:**
  $ git blame -L 10, 20 <file_name>

- **Show Commit Hashes:**
  $ git blame -l <file_name>

In summary, **git blame** is a powerful command for tracking changes to specific lines of code in a Git repository. It aids in understanding the history of code, promotes accountability, and is valuable for collaboration, code reviews, and troubleshooting. Suppose you are working in a collaborative development environment, where multiple developers are working on a particular project, and one developer was responsible for changing a particular file. And senior or administrator or team lead want to see which developer is responsible for changing this particular file, then using this command senior or administrator or team lead can see the changes **line by line** based on the particular file.

.gitignore file:

The **.gitignore** file is used to specify intentionally untracked files and directories that Git should ignore when tracking changes in a repository. **This file is important for controlling what files and directories are not included in version control.** The purpose and importance of the **.gitignore** file are as follows:

- **Purpose:**
  - **Ignoring Unnecessary Files:** Allows developers to specify files and directories that should not be tracked by Git, such as build artifacts, temporary files, or user-specific configurations.
  - **Reducing Repository Size:** Helps keep the size of the Git repository smaller by excluding files that are not essential to the project or are generated during the development process.
  - **Preventing Unwanted Commits:** Prevents accidentally committing files that are meant to be excluded, reducing the chances of unwanted commits.
  - **Consistent Build Environments:** Ensures that different developers working on the same project have a consistent and reproducible build environment by excluding platform-specific or user-specific files.
- **Importance:**
  - **Cleaner Repository:** Contributes to a cleaner and more focused repository by excluding files that do not belong to the versioned source code.
  - **Avoiding Confusion:** Helps avoid confusion and potential issues when collaborating with others by ensuring that only essential files are included in version control.
  - **Enhanced Collaboration:** Facilitates smoother collaboration by preventing the inclusion of files that are specific to individual developers' environments.
  - **Security and Privacy:** Protects sensitive information, such as API keys or configuration files containing credentials, from being accidentally committed and shared.
- **When it is Used:**
  - **Project Initialization:** Developers create a **.gitignore** file when initializing a new Git repository to define patterns for files and directories that should be ignored.
  - **Existing Projects:** In existing projects, developers may add or modify the **.gitignore** file to exclude files or directories that were not previously ignored.

- **Collaborative Development:** Essential for collaborative development to ensure that only necessary files are shared and that individual developers can customize their local environments without affecting the repository.

- **Examples:**

```plaintext
# Ignore build artifacts
build/

# Ignore log files
*.log

# Ignore user-specific files
config/user-specific.txt

# Ignore compiled binaries
*.exe
*.dll
```

In summary, the **.gitignore** file is a crucial component in Git repositories, allowing developers to specify files and directories that should be excluded from version control. It contributes to a cleaner repository, avoids unnecessary commits, and enhances collaboration by providing a consistent and focused development environment.

git shortlog command:

It seems there might be a slight confusion in your question. There isn't a specific "**git shortlog** file." However, I'll provide information on the **git shortlog** command, which is used to summarize and format Git logs based on authorship. If you meant something else, please provide clarification.

- **Purpose:**
  - The **git shortlog** command is used to generate a summarized version of the commit log, focusing on authorship. It aggregates commits by author, providing a compact summary of who contributed to the repository and how many commits each author made.

- **Importance:**
  - **Author Contribution Summary:** Offers a concise summary of commits, categorized by author, making it easier to see who has contributed and in what proportion.
  - **Change Attribution:** Helps in attributing changes to specific contributors, aiding in understanding the distribution of work in a collaborative project.
  - **Changelog Generation:** Useful for generating changelogs or release notes, summarizing contributions by different authors in a release.

- **When it is Used:**
  - **Changelog Creation:** Before creating a release or changelog, developers use **git shortlog** to summarize contributions by different authors.
  - **Author Contribution Analysis:** Developers and project maintainers use **git shortlog** to analyze how much each contributor has contributed to the project.

- **Examples:**
  - **Basic command** for generate overall summary of all commits:

```
$ git shortlog
```

In summary, **git shortlog** is a valuable command for summarizing Git logs based on authorship. It provides a quick overview of contributions by different authors, making it easier to understand the distribution of work in a Git repository.

git clone command:

The **git clone** command is used to create a copy of a remote Git repository on your local machine. It is used when you want to obtain a local copy of a repository hosted on a remote server, typically to start working on the project, collaborate with others, or contribute to an existing codebase. The purpose and importance of **git clone** are as follows:

- **Purpose:**
  - **Repository Copy:** Creates an identical copy of a remote Git repository on your local machine, including all branches, commit history, and files.
  - **Collaboration:** Facilitates collaboration by allowing multiple developers to work on the same codebase without directly modifying the original repository.
  - **Contribution:** Enables contributors to fork a repository, make changes locally, and then propose those changes for inclusion in the original project.
- **Importance:**
  - **Code Sharing:** Allows developers to share and collaborate on code by providing them with a local working copy of a remote repository.
  - **Version Control:** Establishes version control for the codebase, allowing developers to track changes, create branches, and make commits locally.
  - **Forking and Contributing:** Facilitates the process of forking a repository, making changes, and submitting pull requests to contribute changes back to the original project.
  - **Offline Work:** Enables developers to work on a project even when they don't have an internet connection by providing a local copy of the repository.
- **When it is Used:**
  - **New Project Setup:** When starting a new project, developers use **git clone** to create a local copy of the project's repository.
  - **Contributing to Open Source:** Contributors use **git clone** to fork a repository, create a local copy, make changes, and then submit pull requests to contribute to open-source projects.
  - **Collaborative Development:** Team members clone a shared repository to collaborate on a project, each working with their local copy and pushing changes to the remote repository.
- **Example:**
  - **Basic Example:**

```
$ git clone <repository_link>
```

In summary, **git clone** is a fundamental command in Git that allows developers to create local copies of remote repositories. It is essential for collaborative development, contributing to open-source projects, and setting up a local development environment for existing projects.

git push command:

The **git push** command is used to update a remote repository with the changes made in a local repository. It is used when you want to share your local commits with others or update a central repository. The purpose and importance of **git push** are as follows:

- **Purpose:**
  - **Share Local Changes:** Pushes local commits to a remote repository, making them accessible to other contributors.
  - **Update Central Repository:** Updates the remote repository with the latest changes from the local branch, ensuring that the remote reflects the current state of your work.
  - **Collaboration:** Facilitates collaboration by allowing team members to share their changes and collaborate on a shared codebase.
- **Importance:**
  - **Synchronization:** Ensures that changes made locally are synchronized with the remote repository, maintaining a consistent and up-to-date codebase.
  - **Collaborative Development:** Enables multiple developers to collaborate on a project by sharing their local changes with others.
  - **Continuous Integration:** Supports continuous integration workflows where automated build and test processes are triggered after a successful push to the remote repository.
  - **Remote Repository Maintenance:** Helps maintain a centralized and authoritative repository by allowing contributors to contribute their changes.
- **When it is Used:**
  - **Sharing Local Changes:** After making local commits, developers use **git push** to share those changes with other team members.
  - **Updating Remote Branch:** To update a branch in the remote repository with the latest local changes, developers use **git push**.
  - **Contributing to Open Source:** Contributors use **git push** to submit their changes to the remote repository when contributing to open-source projects.
  - **CI/CD Pipelines:** In CI/CD pipelines, **git push** is often used to trigger automated processes such as builds and tests when changes are pushed to the remote repository.
- **Example:**
  - **To push changes from a local branch to its corresponding remote branch, you would use a command like this:**
    $ git push <romote_name> <branch_name>
    $ git push origin main
  - Force push data:
    $ git push -f
  - Delete a remote branch by push command:
    $ git push <branch_name> -delete
    $ git push origin -delete

In summary, **git push** is a crucial command in Git that allows developers to share their local changes with a remote repository, making collaboration and continuous integration possible. It is a fundamental part of the Git workflow when working on projects with multiple contributors or when contributing to open-source projects.

git pull command:

The **git pull** command is used to fetch changes from a remote repository and merge them into the local working branch. It is used when you want to update your local repository with the latest changes from the remote repository. The purpose and importance of **git pull** are as follows:

- **Purpose:**
  - **Fetch and Merge:** Retrieves the latest changes from the remote repository and merges them into the local branch.
  - **Synchronize with Remote:** Updates the local branch with the changes made by others in the remote repository, ensuring synchronization.
- **Importance:**
  - **Maintaining an Up-to-Date Local Copy:** Helps keep the local repository up to date with the latest changes from the remote repository.
  - **Avoiding Divergence:** Prevents the local branch from diverging too far from the remote branch, reducing the chances of conflicts later.
  - **Collaborative Development:** Facilitates collaboration by allowing developers to pull in changes made by others and contribute their own changes.
- **When it is Used:**
  - **Before Making Changes:** Developers often use git pull before making new changes to ensure that they are working with the latest code.
  - **Synchronization:** When collaborators want to synchronize their local branches with the latest changes from the remote repository.
  - **Resolve Conflicts:** After pulling changes, developers may need to resolve conflicts if the remote changes conflict with their local changes.
- **Example:**
  - **To pull changes from the remote repository into the current local branch, you would use a command like this:**
    $ git pull <remote_name> <branch_name>
    $ git pull origin main

In summary, **git pull** is an essential command in Git for keeping the local repository up to date with the latest changes from the remote repository. It is used to synchronize the local branch, avoid divergence, and facilitate collaborative development by incorporating changes made by other contributors.

git fetch command:

The **git fetch** command is used to retrieve changes from a remote repository but does not automatically merge them into the local working branch. It is used when you want to see what changes are available in the remote repository without merging them immediately into your local branch. The purpose and importance of git fetch are as follows:

- **Purpose:**
  - **Retrieve Remote Changes:** Fetches new branches, tags, and commits from a remote repository to the local repository.
  - **Preview Changes:** Allows you to see what changes are present in the remote repository before merging them into your working branch.

- **Importance:**
  - **Maintaining Awareness:** Helps developers stay aware of changes in the remote repository without automatically applying those changes to their working branch.
  - **Reducing Conflicts:** Gives developers the opportunity to review and resolve any potential conflicts before merging remote changes into their local branch.
- **When it is Used:**
  - **Checking for Updates:** Developers use **git fetch** to check for updates in the remote repository before making changes locally.
  - **Reviewing Changes:** Before merging changes, developers fetch the latest updates to review what has changed in the remote repository.
- **Example:**
  - **To fetch changes from the remote repository, you would use a command like this:**
    $ git fetch <remote_name>
    $ git fetch origin
  - **Fetch the remote repository:**
    $ git fetch < repository_Url>
  - **Fetch a specific branch:**
    $ git fetch <branch_name>
  - **Fetch all the branches simultaneously:**
    $ git fetch -all

In summary, **git fetch** is used to retrieve changes from a remote repository, allowing developers to see what changes are available without automatically merging them. It is a useful command for staying aware of remote updates, reducing conflicts, and facilitating a controlled integration of changes into the local working branch.

Differences between "git pull" & "git fetch" command:
Certainly, here are the key differences between the **git pull** and **git fetch** commands:

- **Automatic vs. Manual Integration:**
  - **git pull:** Automatically fetches changes from the remote repository and merges them into the current local branch.
  - **git fetch:** Fetches changes from the remote repository but does not automatically integrate them into the local working branch. Manual steps (e.g., **git merge** or **git rebase**) are required to apply the changes.
- **Workflow Impact:**
  - **git pull:** Typically used when you want to quickly update your local branch with the latest changes from the remote and immediately integrate them.
  - **git fetch:** Useful when you want to check for remote updates without automatically applying them, allowing you to review changes before merging.
- **Branch Tracking:**
  - **git pull:** Automatically fetches changes from the remote branch that is tracked by the current local branch and merges them.

- **git fetch:** Fetches changes from all branches in the remote repository, but does not automatically update the local tracking branches.

- **Integration Flexibility:**
  - **git pull:** Offers less flexibility in choosing how to integrate changes; it automatically performs a merge.
  - **git fetch:** Provides more flexibility as developers can review changes, choose when and how to integrate them, and avoid immediate conflicts.
- **Safety and Awareness:**
  - **git pull:** Immediate integration may lead to unintentional conflicts, and developers may not have a chance to review changes before merging.
  - **git fetch:** Offers a safer option as developers can review fetched changes, check for conflicts, and decide when to integrate them.
- **Single vs. Multiple Commands:**
  - **git pull:** Combines the fetch and merge steps into a single command.
  - **git fetch:** Separates the fetch and merge steps, allowing developers to perform additional actions (e.g., code review) between them.
- **Efficiency:**
  - **git pull:** May be more efficient for a quick update and integration of remote changes.
  - **git fetch:** Can be more efficient when developers want to review changes before deciding to integrate them, avoiding unnecessary merges.
- **Common Use Cases:**
  - **git pull:** Commonly used for quick updates and collaborative development where immediate integration is acceptable.
  - **git fetch:** Preferred in scenarios where developers want to maintain control over when and how remote changes are integrated, such as in code review processes.

git revert command:

The **git revert** command is used to create a new commit that undoes the changes made by a previous commit. It is used when you want to revert the effects of a specific commit without removing it from the commit history. The purpose and importance of **git revert** are as follows:

- **Purpose:**
  - **Undoing Changes:** Creates a new commit that undoes the changes introduced by a specific commit.
  - **Preserving Commit History:** Retains the commit history by creating a new commit instead of removing the targeted commit.
  - **Collaborative Development:** Useful in collaborative projects where reverting a commit without affecting others is necessary.
- **Importance:**
  - **Maintaining Commit History:** Preserves the chronological order of commits, providing a clear and accurate history of changes made to the codebase.

- **Undoing Specific Changes:** Allows developers to selectively revert specific changes introduced by a commit rather than discarding the entire commit.
- **Non-Destructive Reversion:** Provides a non-destructive way to undo changes, ensuring that the commit history remains intact.

- **When it is Used:**
  - **Correcting Mistakes:** When a mistake is made in a commit, **git revert** can be used to undo the changes introduced by that specific commit.
  - **Reverting Changes in a Controlled Manner:** When it's important to maintain a clean and linear commit history while undoing changes.
  - **Collaborative Development:** In collaborative projects, where removing commits might disrupt other contributors, **git revert** is a safer option.
- **Examples:**
  - **To revert the changes introduced by a specific commit, you would use a command like this:**
    $ git revert <commit_hash_id>
  - **Undo the changes:**
    $ git revert

In summary, **git revert** is a valuable command for selectively undoing changes introduced by a specific commit while preserving the commit history. It is particularly useful in collaborative development environments where maintaining a clean and linear history is important, and it provides a non-destructive way to correct mistakes in the codebase.

git remote command:

The **git remote** command is not used to manage files, but rather to manage connections to remote repositories. It is used to view, add, and remove remote repositories associated with a local Git repository. The purpose and importance of **git remote** are as follows:

- **Purpose:**
  - **Managing Remote Connections:** Allows users to manage connections to remote repositories where the codebase is stored.
  - **Viewing Remote Repositories:** Provides information about the remote repositories associated with the local repository.
  - **Adding and Removing Remotes:** Enables users to add new remote repositories or remove existing ones.
- **Importance:**
  - **Collaborative Development:** Essential for collaborative development where code is shared among multiple contributors using a central repository.
  - **Fetching and Pushing Changes:** Required for fetching changes from or pushing changes to remote repositories.
  - **Branch Tracking:** Establishes tracking relationships between local branches and corresponding branches on remote repositories.

- **When it is Used:**
  - **Repository Initialization:** When setting up a new repository, users may use **git remote** to add a connection to a remote repository.
  - **Collaboration Setup:** Before collaborating with others, users typically add remotes to repositories where the shared code is hosted.
  - **Fetching Remote Changes:** Prior to fetching changes from a remote repository, users may use **git remote -v** to view information about the remotes.
  - **Pushing Changes:** When pushing changes to a remote repository, users may use **git remote** to ensure they are pushing to the correct destination.
- **Examples:**
  - **To view the remotes associated with a local repository, you can use:**
    $ git remote -v
  - **To add a new remote named "origin," you might use:**
    $ git remote add <remote_name> <repo_linl>
    $ git remote add origin https://github.com/dibyendubiswas1998/repo.git
  - **Remove a remote connection from the repository:**
    $ git remote rm
  - **Rename remote server:**
    $ git remote rename <old_remote_name> <new_remote_name>
  - **Show additional information about a particular remote** (It will result in information about the remote server. It contains a list of branches related to the remote and also the endpoints attached for fetching and pushing):
    $ git remote show

In summary, **git remote** is a command used to manage connections to remote repositories in a Git project. It is crucial for collaborative development, sharing code, and keeping local repositories in sync with changes made by others in a central repository.

git rm command:

The **git rm** command is used to remove files from both the working directory and the staging area. It is a way to tell Git that you want to delete a file, and it stages the removal so that it can be committed in the next commit. The purpose and importance of **git rm** are as follows:

- **Purpose:**
  - **File Deletion:** Marks files for deletion, both in the working directory and the staging area.
  - **Staging Changes:** Prepares the removal of files for the next commit by staging the deletion.
  - **Updating Repository History:** Ensures that the removal of files is reflected in the Git commit history.
- **Importance:**
  - **Version Control:** Maintains the integrity of version control by explicitly tracking file deletions in the Git repository.

- **Committing Changes:** Allows developers to commit the removal of files along with other changes in a single commit.

- **Code Cleanup:** Useful for cleaning up the repository by removing unnecessary or unwanted files.

- **When it is Used:**
  - **Deleting Files:** When you want to delete one or more files from the working directory and commit the removal.

  - **Updating .gitignore:** When updating the **.gitignore** file to exclude certain files, using **git rm** removes the files from version control.

  - **Renaming or Moving Files:** When renaming or moving files, using **git rm** on the old path and then adding the new path helps Git recognize the change.

- **Examples:**
  - **To use git rm to remove a file named "example.txt" and stage the deletion, you would use the following command:**
    $ git rm file.txt

- **Notes:**
  - The **git rm** command can be combined with the **--cached** option to remove a file from the staging area without deleting it from the working directory. This is useful when you want to stop tracking a file but keep it in your local filesystem.
    $ git rm --cached file.txt

  - After using **git rm**, you need to commit the changes to make the removal permanent. You can use **git commit** to commit the staged changes.
    $ git commit -m “removed he file.txt”

  - If a file is deleted from the working directory without using **git rm**, Git may recognize the deletion as an untracked change. In such cases, you can use **git add -u** to stage the removal.
    $ git add u

In summary, **git rm** is a command used to remove files from both the working directory and the staging area, preparing them for the next commit. It is essential for maintaining a clean version history and updating the repository to reflect changes in the project's file structure.

git mv command:

The **git mv** command is used to move or rename files and directories in a Git repository. It is a convenience command that combines the actions of renaming a file (or moving it to a different directory) and staging the change, making it ready for the next commit. The purpose and importance of **git mv** are as follows:

- **Purpose:**
  - **File or Directory Renaming:** Renames a file or directory in the working directory and stages the change for the next commit.

  - **Consistent History:** Helps maintain a consistent version history by explicitly tracking the file or directory renaming.

- **Importance:**
  - **Git Tracking:** Informs Git about the renaming/moving of files, ensuring that it tracks the changes in its history.

- **Simplified Workflow:** Simplifies the process of renaming or moving files by combining the actions into a single command.

- **When it is Used:**
  - **Renaming Files:** When you want to rename a file in your project and stage the change for the next commit.
  - **Moving Files to a Different Directory:** When you want to move a file or directory to a different location within the project and stage the change.

- **Examples:**
  - To use **git mv** to rename a file from **"old_file.txt"** to **"new_file.txt"** and stage the change, you would use the following command:
    $ git mv <old_file_name.txt> <new_file_name_.txt>
  - **Move from one location to another location:**
    $ git mv dir_2/file_.txt dir1/file_.txt

- **Notes:**
  - After using **git mv**, you still need to commit the changes to make the renaming or moving permanent. You can use **git commit** to commit the staged changes.
    $ git commit -m “Rename the old_file.txt to new_file.txt”
  - If you prefer not to use **git mv**, you can manually rename or move files in your working directory, followed by using **git add -u** to stage the changes.
    $ git mv <old_file_name.txt> <new_file_name_.txt>
    $ git add -u
    $ git commit -m “Rename the old_file.txt to new_file.txt”

In summary, **git mv** is a convenient command for renaming or moving files and directories in a Git repository. It simplifies the process by combining the renaming action with the staging of changes, helping maintain a consistent and tracked version history in your Git repository.

git clean command:

The **git clean** command is used to remove untracked files and directories from the working directory of a Git repository. Untracked files are files that are not part of the current commit or any previous commits. The purpose and importance of **git clean** are as follows:

- **Purpose:**
  - **Clean Working Directory:** Removes untracked files and directories, providing a clean state in the working directory.
  - **Freeing Disk Space:** Helps free up disk space by removing unnecessary files that are not part of the versioned history.

- **Importance:**
  - **Preventing Unintended Committing:** Prevents accidental inclusion of untracked files in commits, ensuring only intentional changes are committed.
  - **Maintenance and Cleanup:** Facilitates repository maintenance by removing temporary files, build artifacts, or other files not intended for version control.

- **When it is Used:**
  - **Before Committing Changes:** Developers may use **git clean** before committing to ensure that only necessary and intentional changes are included.

- **Cleanup after Build Processes:** Build processes may generate temporary files or build artifacts that can be removed using **git clean**.

- **Examples:**
  - **To remove untracked files and directories from the working directory, you can use:**
    $ git clean [-n] [-f] [-d]
    - The **-n** option (dry run) shows a preview of what would be deleted without actually removing anything.
    - The **-f** option (force) is required to actually remove the files and directories. Be cautious with this option as it irreversibly deletes files.
    - The **-d** option is used to also remove untracked directories.
  - **Dry run to preview what would be deleted:**
    $ git clean -n
  - **Actually remove untracked files and directories:**
    $ git clean -f -d

- **Notes:**
  - Be cautious when using the **-f** option with **git clean** because it permanently deletes untracked files. Ensure you really want to remove those files before executing the command.
  - By default, **git clean** only removes untracked files, not ignored files. To remove ignored files as well, you can use the **-x** option.
  - **git clean** doesn't remove files that are currently being tracked by Git, even if they are modified. To remove modified files, you should first commit or discard the changes.

In summary, **git clean** is a powerful command for cleaning up the working directory by removing untracked files and directories. It helps maintain a clean and focused repository, preventing unintended commits of temporary or unnecessary files. Caution should be exercised, especially when using the **-f** option, to avoid accidental data loss.

git rm -f command:

There is no specific **git rm -f** command in Git. Instead, the **-f** option is commonly used with the **git rm** command to force the removal of files from both the working directory and the staging area. The purpose and importance of **git rm -f (**or **git rm --force)** are as follows:

- **Purpose:**
  - **Forceful Removal:** Forces the removal of files from both the working directory and the staging area, regardless of whether the files are modified or have unstaged changes.
  - **Overriding Safety Checks:** Overrides safety checks, allowing the removal of files even when Git might consider it risky (e.g., when there are unstaged changes).

- **Importance:**
  - **Unconditional Removal:** Useful when you want to forcefully remove files without being prompted for confirmation, especially when dealing with versioned files.
  - **Batch Removal:** Enables the removal of multiple files in a single command without individual confirmation prompts.

- **When it is Used:**
  - **Removing Modified Files:** When you want to remove files, including modified ones, from the working directory and staging area without committing the changes.
  - **Batch Removal of Files:** When you need to remove multiple files, and you want to do so with a single command, bypassing confirmation prompts.
- **Examples:**
  - **To forcefully remove a file named "example.txt" using git rm -f, you would use the following command:**
    $ git rm -f example.txt
- **Notes:**
  - The **-f** option is optional in many cases, as **git rm** will prompt for confirmation before removing files. However, in situations where you want to bypass the confirmation prompt, the **-f** option is useful.
  - Be cautious when using the **-f** option, as it removes files without confirmation. Make sure you really want to delete the files before using it.
  - When using **git rm** to remove files, you still need to commit the changes to make the removal permanent. Use **git commit** after **git rm** to finalize the removal in a commit.
    $ git commit -m “Remove the example.txt”

In summary, **git rm -f** is used to forcefully remove files from both the working directory and the staging area, bypassing confirmation prompts. It is useful when you want to quickly and unconditionally remove files, especially in cases where versioned files need to be removed without committing their changes.

git describe command:

The **git describe** command is used to provide a human-readable description of a specific commit's identifier, typically a tag or a branch, relative to the closest annotated tag. It is particularly useful for obtaining a concise and meaningful identifier for a commit, especially in scenarios where you want to quickly understand the state of the codebase. The purpose and importance of **git describe** are as follows:

- **Purpose:**
  - **Describe a Commit:** Generates a string that describes a specified commit's relationship to the closest annotated tag.
  - **Concise Commit Identifier:** Provides a short and informative identifier that includes the closest tag name, the number of commits since the tag, and the abbreviated commit hash.
- **Importance:**
  - **Readable Version Information:** Offers a more readable and meaningful version identifier compared to a raw commit hash.
  - **Identifying Release Candidates:** Useful for identifying release candidates or understanding the state of the codebase relative to tags.
- **When it is Used:**
  - **Versioning:** When you want to obtain a human-readable version identifier for a specific commit.

- **Releases and Tags:** Commonly used in build and release processes to obtain version information for the current state of the repository.

■ **Examples:**

- To use **git describe** to describe the current commit, you can use the following command:
  $ git describe

  This will output a string that typically looks like v1.2.3-10-gabcdef, where:
  - ➢ v1.2.3 is the closest annotated tag.
  - ➢ 10 is the number of commits since that tag.
  - ➢ gabcdef is the abbreviated commit hash.

■ **Notes:**

- The output of **git describe** can be customized using various options to control the format and content of the generated description.
  $ git describe --tags --abbrev=0 --long
  - ➢ The **--tags** option restricts the search to tags only.
  - ➢ The **--abbrev** option controls the length of the abbreviated commit hash.
  - ➢ The **--long** option includes the full commit hash instead of the abbreviated form.
  - ➢ git describe is particularly useful in continuous integration (CI) and deployment pipelines, where a concise version identifier is often needed.

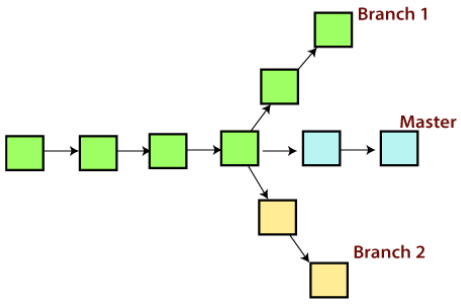
In summary, **git describe** is a helpful command for obtaining a human-readable version identifier for a specific commit, particularly when you want to understand the relationship between a commit and the closest annotated tag. It is commonly used in versioning, release management, and **CI/CD** processes to provide meaningful version information for a codebase.

Git Branching:

Git branch is a mechanism in the Git version control system that allows users to work on multiple versions of their codebase simultaneously.

A branch is essentially a separate line of development that can be used to experiment with new features or fixes without affecting the main codebase. Each branch is essentially a pointer to a particular commit in the Git repository, which allows users to switch between branches easily and quickly.

By using Git branches, developers can work on different features or bug fixes independently of each other, and then merge the changes back into the main codebase once they are complete and tested. This allows for greater flexibility and organization in the development process, and can help prevent conflicts between different sets of changes.



- **Importance of Branching:**
  - ★ **Isolation of Changes:**
    - **Importance:** Branches allow developers to work on new features, bug fixes, or experiments in isolation from the main codebase.
    - **Benefits:** This isolation prevents interference between different sets of changes, making it easier to manage and understand the development process.
  - ★ **Parallel Development:**
    - **Importance:** Branches enable parallel development by allowing multiple teams or developers to work on different features simultaneously.
    - **Benefits:** This improves efficiency and accelerates the development process, especially in larger projects with multiple contributors.
  - ★ **Feature Development:**
    - **Importance:** Feature branches are commonly used to develop new features or enhancements.
    - **Benefits:** Developers can work on a specific feature without affecting the main codebase. It also allows for collaborative development of features before they are integrated into the main branch.
  - ★ **Bug Fixing:**
    - **Importance:** Branches facilitate the creation of separate branches for bug fixes.
    - **Benefits:** This allows for the isolation of bug fixes from ongoing feature development, making it easier to identify and apply fixes without introducing new issues.
  - ★ **Release Management:**
    - **Importance:** Branches are essential for managing releases and hotfixes.
    - **Benefits:** Release branches help freeze a specific version for stabilization, while hotfix branches allow for urgent bug fixes in production without disrupting ongoing development.
  - ★ **Experimentation and Prototyping:**
    - **Importance:** Developers can create experimental branches to try out new ideas or prototype features.
    - **Benefits:** This encourages creativity and innovation, providing a safe space for testing without impacting the stability of the main codebase.
  - ★ **Code Review and Collaboration:**
    - **Importance:** Branches facilitate code review and collaboration.
    - **Benefits:** Developers can share branches with others for review before merging changes into the main branch, ensuring code quality and alignment with project goals.
  - ★ **Versioning and Historical Tracking:**
    - **Importance:** Git's branching model provides a clear historical record of changes.
    - **Benefits:** Each branch represents a version or state of the project, making it easy to track and revert changes if needed. This aids in understanding the evolution of the codebase.

★ **Continuous Integration and Deployment (CI/CD):**

- **Importance: CI/CD** processes often rely on branches to trigger automated builds and deployments.
- **Benefits:** Feature branches, release branches, and other branches can be configured to trigger specific **CI/CD** pipelines, enabling automated testing and deployment workflows.

■ **Master Branch:**

★ **Historical Context:**

- The term "master" has been historically used as the default branch name in many Git repositories.
- It originates from the early development of version control systems.

★ **Default Branch:**

- When you initialize a new Git repository, the default branch is often named **master.**
- Developers commonly create feature branches, work on changes, and then merge them into the **"master"** branch when features are completed or ready for integration.

★ **Renaming Considerations:**

- In recent years, there has been a move towards more inclusive and neutral language in software development.
- Some projects and organizations have considered changing the default branch name from **"master"** to a more inclusive term like **"main"** to address concerns associated with the historical context of the term.

■ **Main Branch:**

★ **Inclusive Terminology:**

- The term **"main"** is an alternative to **"master"** that has gained popularity, particularly in response to discussions around inclusive language.
- The use of **"main"** aims to avoid language that may carry historical connotations or be perceived as exclusive.

★ **GitHub Default:**

- GitHub, one of the most widely used Git repository hosting platforms, has adopted **"main"** as the default branch name for newly created repositories.
- This decision has contributed to the increased adoption of **"main"** as the default branch name in many projects.

★ **Gradual Transition:**

- Some projects and organizations have opted to transition from **"master"** to **"main"** to reflect more inclusive language practices.
- Existing repositories often allow both **"master"** and **"main"** as valid branch names to accommodate the transition period.

It's important to note that the choice between **"master"** and **"main"** is largely a matter of convention, and there is no functional difference between the two in terms of Git's capabilities. The decision to use one over the other often depends on project preferences, community discussions, or the adoption of specific naming conventions.

- **Examples:**
  - **Create a new branch:**
    $ git branch <branch_name>
  - **Switch to a branch:**
    $ git checkout <branch_name>
    $ git switch <branch_name>
  - **Create and Switch to new branch:**
    $ git checkout -b <new_branch_name>
    $ git switch -c <new_branch_name>
  - **List of all branches:**
    $ git branch
  - **Delete a branch:**
    $ git branch -d <branch_name>
  - **Force to delete a branch:**
    $ git branch -D <branch_name>
  - **Rename a Branch (current branch):**
    $ git branch -m <new_branch_name>
    $ git branch -m <old_branch_name> <new_branch_name>
  - **Show branch information:**
    $ git show-branch
  - **Merge Changes from Another Branch:**
    $ git merge <branch_name>
    $ git merge my-feature
  - **Rebase changes from another branch:**
    $ git rebase <branch_name>
  - **View the last commit on each branch:**
    $ git branch -v
  - **Set upstream branch:**
    $ git branch -u <upstream_branch>
    $ git branch -u origin/main
  - **Show remote branches:**
    $ git branch -r
  - **Show all branches (local and remote):**
    $ git branch -a
  - **Delete a remote Branch:**
    $ git push origin -delete <branch_name>

git checkout command:

The **git checkout** command in Git serves multiple purposes and is a versatile command that is used in various scenarios. Its primary purposes and importance include:

- **Switching Branches:**
  - **Purpose:** Changes the working directory to the specified branch.
  - **Importance:** Essential for navigating between different branches during development.
  - **Example:**
    $ git checkout <branch_name>

- **Creating and Switching to New Branches:**
  - **Purpose:** Creates a new branch and switches to it in one command.
  - **Importance:** Provides a convenient way to start working on a new feature or bug fix.
  - **Example:**
    $ git checkout -b <branch_name>

- **Checking Out Specific Commit:**
  - **Purpose:** Allows you to move the HEAD pointer to a specific commit.
  - **Importance:** Useful for inspecting historical changes or creating a new branch based on a specific commit.
  - **Example:**
    $ git checkout <commit_id>

- **Checking Out Files from a Specific Commit:**
  - **Purpose:** Restores specific files to their state in a given commit.
  - **Importance:** Helpful when you want to revert changes to specific files without changing the branch.
  - **Example:**
    $ git checkout <commit_id> -- <file_Path/file_name.txt>

- **Discarding Local Changes:**
  - **Purpose:** Discards changes in the working directory and staging area.
  - **Importance:** Useful when you want to reset the working directory to match the last committed state.
  - **Example:**
    $ git checkout -- <file_Path/file_name.txt>

- **Switching Between Branches with Uncommited Changes:**
  - **Purpose:** Allows you to switch branches even if you have uncommitted changes.
  - **Importance:** Facilitates seamless branch switching during development.
  - **Example:**
    $ git checkout -m <branch_name>

- **Creating Orphan Branches:**
  - **Purpose:** Creates a new branch with no commit history.
  - **Importance:** Useful for starting a new, unrelated project within the same repository.
  - **Example:**
    $ git checkout -- orphan <new_branch_name>

- **Checking Out Remote Branches:**
  - **Purpose:** Creates and switches to a new local branch tracking a remote branch.
  - **Importance:** Allows you to work on changes that are tracked in a remote repository.
  - **Example:**
    $ git checkout -b <local_branch_name> <remote_name>/<remote_branch_name>

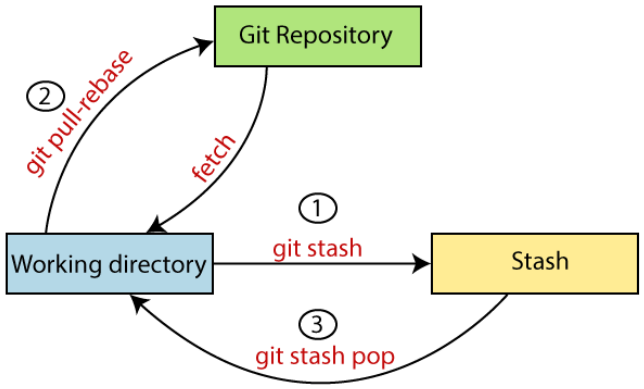
- **Checking Out Tags:**
  - **Purpose:** Moves the HEAD to the specified tag.

- **Importance:** Useful for inspecting code at a specific tagged release.
- **Example:**
  $ git checkout tags/<tag_name>

In summary, **git checkout** is a versatile command in Git that is used for switching branches, creating new branches, inspecting historical states, discarding changes, and more. It plays a crucial role in navigating the development history and managing the working directory in different scenarios.

git stash command:

In Git, **"stashing"** refers to the process of temporarily saving changes in your working directory that are not ready to be committed. Stashing allows you to set aside your current changes, clean up your working directory, and perform other operations, such as switching branches or pulling changes, without committing or discarding your in-progress work.



Generally, the stash's meaning is "**store something safely in a hidden place.**" The sense in Git is also the same for stash; Git temporarily saves your data safely without committing.

When you stash changes, Git takes a snapshot of your working directory and index (staged changes), reverts your working directory to the last commit, and then saves the snapshot in a **"stash."** The stash itself is a stack, and you can have multiple stashes, each representing a different set of changes.

- **The Typically Use Cases of Stashing include:**
  - **Switching Branches:** When you have changes in your working directory that you don't want to commit, but you need to switch to another branch.
  - **Pulling Changes:** Before pulling changes from a remote repository, especially when you have uncommitted changes.
  - **Merging or Rebasing:** Before performing a merge or rebase operation when you have uncommitted changes.
  - **Temporary Context Switching:** When you need to work on a different task or fix a bug and want to set aside your current changes.

The **git stash** command in Git is used to temporarily save changes in your working directory that are not ready to be committed. It's particularly useful when you need to switch branches, but you have uncommitted changes that you don't want to commit or discard. The purpose and importance of git stash are as follows:

- **Temporary Storage of Uncommited Changes:**
  - **Purpose:** Temporarily saves changes in the working directory without committing them.
  - **Importance:** Allows you to switch branches or perform other operations without the need to commit or discard changes that are still in progress.

- **Example:**
  $ git stash

- **Stashing Multiple Sets of Changes:**
  - **Purpose:** You can create multiple stashes, each representing a different set of changes.
  - **Importance:** Useful when you're working on multiple features simultaneously and need to switch between them.
  - **Example:**
    $ git add <stash_file_name(s)>
    $ git stash save <messages>
    $ git stash save “Feature A Changes”
    $ git stash save “Feature B Changes”

- **Stashing Untracked Files:**
  - **Purpose:** The **-u** or **--include-untracked** option allows you to stash untracked files as well.
  - **Importance:** Helpful when you want to clean up your working directory completely, including untracked files.
  - **Example:**
    $ git stash -u (stash the untrack files. Move the untrack files from working directory to stashing directory)

- **Stashing Ignored Files:**
  - **Purpose:** The **-a** or **--all** option stashes all changes, including ignored files.
  - **Importance:** Useful when you want to stash everything, including files specified in **.gitignore**.
  - **Example:**
    $ git stash -a

- **List of Stashes:**
  - **Purpose:** Shows a list of stashes and their descriptions.
  - **Importance:** Allows you to identify and choose which stash to apply or drop.
  - **Example:**
    $ git stash list

- **Applying Stashed Changes:**
  - **Purpose:** Applies the most recent stash to the working directory.
  - **Importance:** Retrieves and reapplies the changes that were stashed.
  - **Example:**
    $ git stash apply <stash_name> (it takes copy of files from stash area and then move to working area)

- **Applying a Specific Changes:**
  - **Purpose:** Applies a specific stash identified by its index.
  - **Importance:** Useful when you have multiple stashes and want to apply a particular one.
  - **Example:**
    $ git stash apply stash@{2}
- **Dropping a Stash:**
  - **Purpose:** Deletes a stash, removing it from the list.
  - **Importance:** Useful when you no longer need the stashed changes.
  - **Example:**
    $ git stash drop

- **Clearing all Stashes:**
  - **Purpose:** Remove or Clear all stashes.

- **Importance:** Useful when you want to clean up all stashes.
- **Example:**
  $ git stash clear
  $ git stash list (it shows empty list)

- **Clearing a Branch from a Stashes:**
  - **Purpose:** Creates a new branch and applies the most recent stash to it.
  - **Importance:** Enables you to continue working on the changes in a new branch.
  - **Example:**
    $ git stash branch <new_branch_name>

- **Track the stashes and their changes:**
  $ git stash show -p <stash_name>

- **Re-apply the previous commits:**
  It takes files from stash area and then move to staging area
  $ git stash pop <stash_name>
  $ git commit -m "commis_message"

- **Delete a most recent stash from the queue:**
  $ git stash drop

- **Overall:**
  $ git status
  $ git stash
  $ git status
  $ git stash list

  $ git stash -u (stash the untrack files. Move the untrack files from working directory to stashing directory)
  $ git add <stash_file_name(s)>
  $ git stash save "message" (save the stash with particular messages. If you do not mention or provide any message then it takes the first message where HEAD is pointing)

  For view the stash (it showing the detailed overview of changes):
  $ git stash show -p <stash_name> (stash_name came from stash list)
  Move from Stash area to Working area, and then perform staging & commit:
  $ git stash apply <stash_name> (it takes copy of files from stash area and then move to working area)
  $ git add <file_name(s)>
  $ git commit -m "messages"

  $ git stash pop <stash_name> (it takes files from stash area and then move to staging area)
  $ git commit -m "messages"

  To clear entire stash:
  $ git stash clear
  $ git stash list (it shows empty list)
  Create stash branch:
  $ git stash branch <branch_name> <stash_name> (based on stash_name or a particular stash move to particular stash branch)
  After that, you automatically switch to the particular branch that is created by stash branch.
  $ git commit -m "messages"

In summary, **git stash** is an important command for temporarily saving changes in your working directory without committing them. It provides flexibility and convenience when you need to switch branches or perform

other operations while keeping your work in progress. Stashes can be applied, listed, dropped, or even used to create new branches, making them a valuable tool in Git workflows.

git cherry pic command:

The **git cherry-pick** command in Git is used to apply a specific commit or a range of commits from one branch onto another branch. This command is particularly useful when you want to selectively choose and apply changes from one branch to another. The purpose and importance of **git cherry-pick** are as follows:

- **Purpose:**
  - ★ **Applying Specific Commits:**
    - **Purpose:** Copies the changes introduced by a specific commit and applies them to the current branch.
    - **Importance:** Useful when you want to bring in specific changes without merging the entire branch.
    - **Example:**
      $ git cherry-pick <commit_hash>
  - ★ **Applying a Range of Commits:**
    - **Purpose:** Applies a range of consecutive commits onto the current branch.
    - **Importance:** Efficient for copying a series of changes from one branch to another.
    - **Example:**
      $ git cherry-pick <start_commit>^..<end_commit>
- **Importance:**
  - ★ **Selectively Merging Changes:** Allows you to choose specific commits and bring them into another branch, providing a more granular and controlled approach to merging changes.
  - ★ **Avoiding Full Branch Merges:** Useful when you don't want to merge an entire branch but only specific changes from that branch.
  - ★ **Picking Bug Fixes or Features:** Enables the cherry-picking of specific commits, such as bug fixes or new features, into a stable or release branch.
  - ★ **Maintaining a Clean History:** Helps in maintaining a clean commit history by avoiding unnecessary commits or merges.
  - ★ **Combining Changes from Multiple Branches:** Facilitates the integration of changes from different branches without merging entire branch histories.
- **When it is Used:**
  - ★ **Integrating Hotfixes:** When a critical bug fix is made in a hotfix branch, you can cherry-pick the fix into the main development branch or other relevant branches.
    $ git checkout main
    $ git cherry-pick <hotfix_commit>
  - **Backporting Commits:** When you want to bring specific changes from a newer branch back into an older release branch.
    $ git checkout release-1.0
    $ git cherry-pick <new_feature_commit>
  - **Recording Commits:** When you need to reorder or modify commits before merging them into another branch.
    $ git cherry-pick <commit1> <commit2> <commit3>

- **Example:**
  - ★ Assume you have a feature branch named **feature-branch** with a commit **A** that you want to apply to the **main** branch. You can use the following command:
    $ git checkout main
    $ git cherry-pic A

    This command applies the changes introduced by commit **A** from **feature-branch** to the current branch **(main)**.

- **Notes:**
  - ★ **Potential Conflicts:** Cherry-picking may result in conflicts, especially if the changes being cherry-picked conflict with the changes in the target branch. You may need to resolve conflicts manually.

  - ★ **Commit Hashes:** Make sure to use the correct commit hashes when cherry-picking.

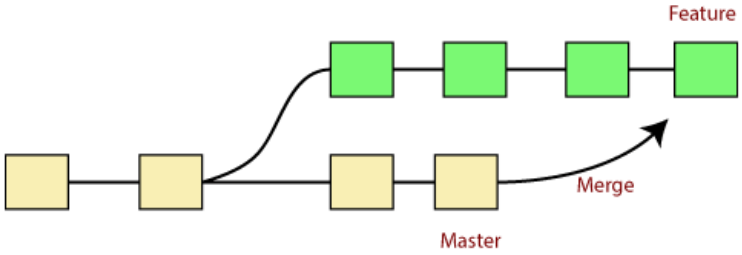
  - ★ **Avoiding Direct Commits to Release Branches:** Cherry-picking can be preferable over directly committing to release branches, as it allows for more explicit control and documentation of changes.

In summary, **git cherry-pick** is a powerful command in Git for selectively applying specific commits or a range of commits from one branch to another. It provides a controlled way to integrate changes, especially when you want to bring in specific bug fixes, features, or changes without merging entire branch histories.

Git Merging:

Git merging is a process in the Git version control system that allows you to combine changes from different branches of your codebase. When you merge two branches, Git takes the changes from both branches and applies them to a new commit, creating a new branch that incorporates the changes from both branches.

In Git, the merging is a procedure to connect the forked history. It joins two or more development history together. The git merge command facilitates you to take the data created by git branch and integrate them into a single branch. Git merge will associate a series of commits into one unified history. Generally, git merge is used to combine two branches.



In the above figure, there are two branches **master** and **feature**. We can see that we made some commits in both functionality and master branch, and merge them. It works as a pointer. It will find a common base commit between branches. Once Git finds a shared base commit, it will create a new "merge commit." It combines the changes of each queued merge commit sequence.

Merging is an important tool in Git, as it allows developers to work on separate branches without interfering with each other's work. By merging changes from multiple branches into a single branch, you can keep your codebase up-to-date and ensure that all changes are properly incorporated into the final product.

In Git, merging refers to the process of combining changes from different branches into a single branch, typically the main or master branch. Merging allows you to integrate the changes made in one branch into another,

ensuring that the codebase is up-to-date with the latest modifications from different contributors or development streams. The purpose and importance of merging in Git are as follows:

- **Purpose:**
  - ★ **Combine Code Changes:**
    - **Purpose:** Integrates changes made in one branch into another, bringing multiple lines of development together.
    - **Importance:** Enables collaboration by allowing developers to work on separate branches and later merge their changes.
    - **Example:**
      $ git merge <branch_name>
  - ★ **Incorporate Feature Branchs:**
    - **Purpose:** Allows the integration of feature branches, where new features or improvements are developed independently.
    - **Importance:** Enables a modular and collaborative development workflow.
    - **Example:**
      $ git checkout <branch_name_1>
      $ git merge <branch_name_2>
  - ★ **Resolve Conflicts:**
    - **Purpose:** Handles conflicts that arise when changes in one branch cannot be automatically merged with changes in another branch.
    - **Importance:** Enables developers to review and manually resolve conflicting changes.
    - **Example:**
      $ git checkout <branch_name_1>
      $ git commit <message>
- **Importance:**
  - **Maintaining Codebase Integrity:** Ensures that the main or master branch always contains the most up-to-date and consolidated version of the codebase.
  - **Supporting Parallel Development:** Facilitates parallel development by allowing developers to work on different features or bug fixes in isolated branches.
  - **Enabling Collaboration:** Supports collaborative development, allowing multiple team members to contribute to a project without interfering with each other's work.
  - **Facilitation Code Reviews:** Makes it easier to review changes before merging, ensuring that modifications are thoroughly examined and tested.
  - **Creating a Cohensive History:** Contributes to a clean and organized project history by consolidating related changes into a single commit or a series of commits.
- **Types of Merging:**
  - ★ **Fast -Forward Merging:**
    - **Description:** Occurs when the branch being merged can be directly fast-forwarded to include the commits from another branch.
    - **Importance:** Common in scenarios where there are no divergent changes between branches.
    - **Command:**
      $ git merge <branch_name>
  - ★ **Three-Way Merging:**
    - **Description:** Involves finding a common ancestor commit and merging changes from both branches, creating a new commit with combined changes.

- **Importance:** Handles situations where changes in both branches have occurred since their last common ancestor.
- **Command:**
  $ git merge <branch_name>

■ **Examples:**

- **Merge the branches:**
  $ git merge

- **Merge the specified commit to currently active branch:**
  $ git merge <commit_id>

  The above command will merge the specified commit to the currently active branch. You can also merge the specified commit to a specified branch by passing in the branch name in <commit>. Let's see how to commit to a currently active branch.

  See the below example. I have made some changes in my project's file **newfile1.txt** and committed it in my **test** branch.

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git add newfile1.txt

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git commit -m "edited newfile1.txt"
[test d2bb07d] edited newfile1.txt
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git log
commit d2bb07dc9352e194b13075dcfd28e4de802c070b (HEAD -> test)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 11:27:44 2019 +0530

    edited newfile1.txt

commit 2852e020909dfe705707695fd6d715cd723f9540 (test2, master)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date:   Wed Sep 25 10:29:07 2019 +0530

    newfile1 added
```

Copy the particular commit you want to merge on an active branch and perform the merge operation. See the below output:

```
HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test)
$ git checkout test2
Switched to branch 'test2'

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$ git merge  d2bb07dc9352e194b13075dcfd28e4de802c070b
Updating 2852e02..d2bb07d
Fast-forward
 newfile1.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

HiMaNshU@HiMaNshU-PC MINGW64 ~/Desktop/GitExample2 (test2)
$
```

In the above output, we have merged the previous commit in the active branch test2.

In summary, merging in Git is a fundamental operation that brings together changes from different branches, supporting collaboration and parallel development. The process ensures that the codebase remains cohesive, up-to-date, and ready for further development. Merging is crucial for maintaining a healthy and organized version control workflow in Git.

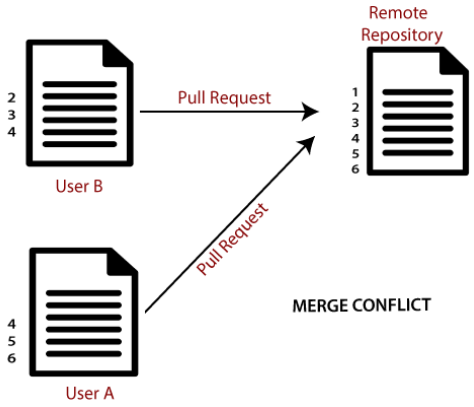
Merge Conflict:

When two branches are trying to merge, and both are edited at the same time and in the same file, Git won't be able to identify which version is to take for changes. Such a situation is called merge conflict. If such a situation occurs, it stops just before the merge commit so that you can resolve the conflicts manually.

Git track each of lines. Merge Conflict happens when exact same line change in different branch.

For example:

In the Master branch, we do some changes in a particular file and similarly we do some changes on the same file by creating a new branch at the same point of time; after that we will try to combine, then conflict occurs.



A merge conflict in Git occurs when Git is unable to automatically merge changes from different branches. This situation arises when changes made in the same part of a file in different branches cannot be reconciled automatically. In other words, Git is unable to determine which changes to accept and how to combine them because they conflict with each other.

- **Merge conflicts commonly happen in the following scenarios:**
  - ★ **Parallel Development:** When multiple developers are working on the same project simultaneously and make changes to the same file or code.
  - ★ **Branch Merging:** When merging changes from one branch into another, and there are conflicting modifications in the branches being merged.
- **How Merge Conflict Occur:**
  - ★ When you attempt to merge branches using a command like **git merge** or **git pull**, Git compares the changes made in the source and target branches. If Git detects conflicting changes in the same part of a file, it cannot automatically resolve the differences. Instead, Git marks the conflicted sections in the affected files and leaves it to the user to resolve the conflict.
- **Indicator of a Merge Conflict:**
  - ★ **When a merge conflict occurs, Git modifies the conflicted files by adding special markers to indicate the conflicting sections. These markers include:**
    - **<<<<<<< HEAD:** Marks the beginning of the conflicting changes from the current (HEAD) branch.
    - **=======:** Separates the conflicting changes from the changes in the incoming branch.
    - **>>>>>>> <branch_name>:** Marks the end of the conflicting changes from the incoming branch.

    Here's an example of what a conflicted section in a file might look like:

```
plaintext                                                    Copy code

<<<<<<< HEAD
// Changes from the current branch
Code from current branch
=======
// Changes from the incoming branch
Code from incoming branch
>>>>>>> incoming-branch
```

- **Resolving the Merge Conflict:**
  To resolve a merge conflict, you need to manually edit the conflicted files to choose which changes to keep and how to combine them. The process involves:

  - **Open the Conflicted Files:** Open the conflicted files in a text editor or an integrated development environment (IDE).

  - **Locate and Edit the Conflict Markers:**
    - Review the conflicting sections marked by Git and decide which changes to keep.
    - Remove the conflict markers (**<<<<<<< HEAD**, **=======**, and **>>>>>>> <branch_name>**).
    - Edit the code to include the desired changes or merge conflicting changes appropriately.

  - **Save the Changes:** Save the modified files after resolving the conflicts.

  - **Mark Conflicts as Resolved:** Use **git add** to mark the conflicted files as resolved.
    $ git add <conflicted_file>

  - **Complete the Merge:** Once all conflicts are resolved, complete the merge using **"git merge --continue"** or **"git commit"**.
    $ git merge --continue
    $ git commit -m "messages"

In summary, a merge conflict in Git occurs when changes from different branches cannot be automatically merged. Resolving merge conflicts involves manually editing the conflicted files, choosing which changes to keep, and marking the conflicts as resolved. It is a common part of collaborative development workflows and ensures that conflicting changes are carefully reviewed and resolved by the developers involved.

Git Rebase:
Rebasing is the process of taking one branch and adding it to the tip of another, where the tip is simply the last commit in the branch.

Rebasing changes the history of entire branch.

Git rebase is a command in the Git version control system that allows you to modify the history of your Git repository. Specifically, Git rebase is used to change the base of a branch, which means that it allows you to move the starting point of a branch to a different commit.

When you run the **git rebase** command, Git will take the commits in your current branch and apply them on top of a new base branch. This creates a new commit history that incorporates changes from both branches.

One of the main benefits of using **Git rebase** is that it can help you create a **cleaner** and **more linear commit history.** By combining multiple branches into a single branch, you can avoid creating unnecessary merge commits that clutter your commit history.

However, Git rebase can also be a potentially dangerous operation, especially if you're working with a public or shared branch. Since Git rebase modifies the history of a branch, it can lead to conflicts if other people are working on the same branch. In general, it's best to use Git rebase only for branches that you're working on alone.

If there are conflicts during a Git rebase operation, Git will prompt you to manually resolve them before continuing the rebase. Once the conflicts are resolved, Git will continue applying the remaining commits.

- **Basic Rebase Command:**
  $ git rebase <base_branch>
  This command takes the commits from the current branch that are not in the specified **<base_branch>**, applies them one by one on top of the latest commit in the **<base_branch>**, and effectively moves the branch pointer.

- **Key Components of Rebase:**
  - ★ **Base Commit:** The commit at which the rebase operation starts.
  - ★ **Upstream Branch:** The branch to which you want to rebase your changes.
  - ★ **Downstream Branch:** The branch containing the changes you want to rebase.

- **Why Rebase:**
  - ★ **Maintain a Linear History:** Creates a linear sequence of commits, making the project history cleaner and more readable.
  - **Reduce Merge Commits:** Minimizes the number of merge commits in the commit history.
  - **Facilitate Code Reviews:** Simplifies the process of reviewing changes, as each commit represents a logical and cohesive unit of work.
  - **Group and Squash Commits:** Allows you to combine, reorder, or squash commits before pushing changes to a shared repository.

- **When to Rebase:**
  - ★ **Before Pushing Changes:** It is generally safe to rebase commits that have not been pushed to a shared repository.
  - **On Feature Branches:** Useful for keeping feature branches up-to-date with changes in the main development branch.
  - **Interactive Rebase:** Used to modify commit messages, squash commits, or reorder commits before pushing.

- **When not to Rebase:**
  - ★ **On Shared or Public Branches:** Avoid rebasing commits that have been shared with others to prevent conflicts in their repositories.
  - **When Collaborating:** If multiple people are working on the same branch, rebasing can cause confusion and conflicts.

- **Basic Rebase Workflow:**
  - ★ **Checkout the branch to be rebase:**
    $ git checkout <branch_name>

- **Chose the base branch:** Choose the branch onto which you want to rebase (**main** in this example):
  $ git rebase main

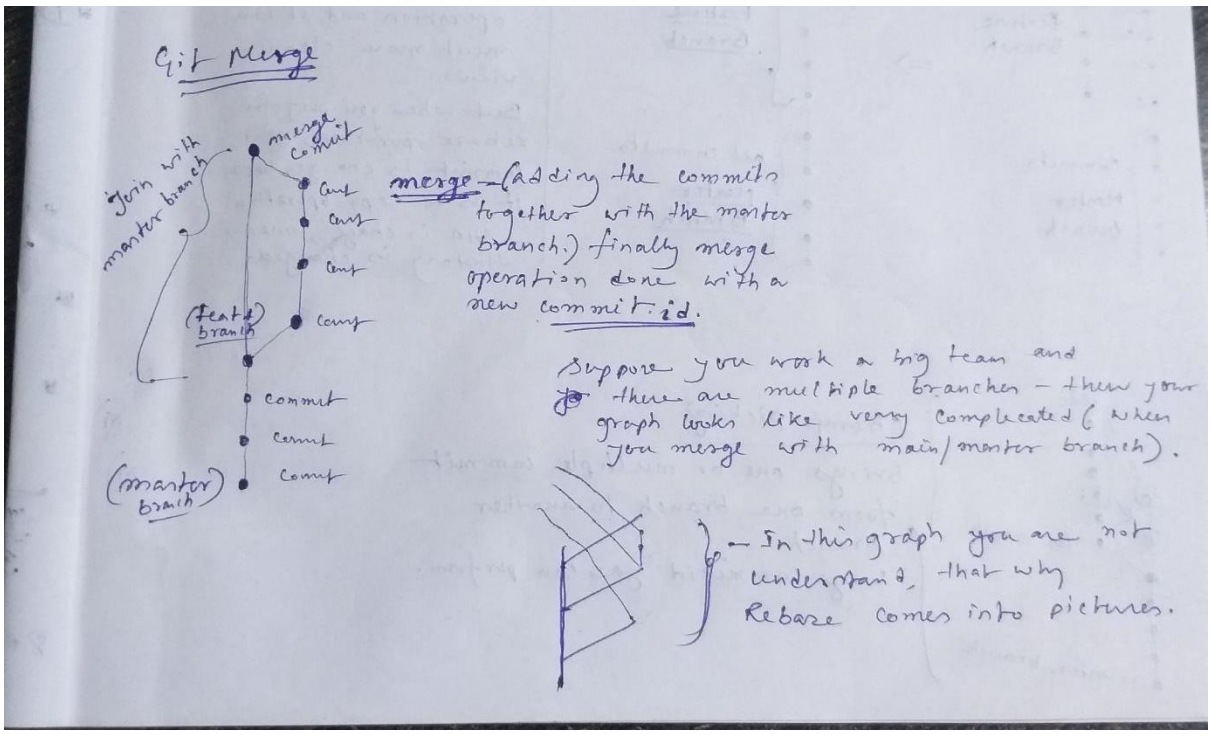
- **Resolve Conflicts (if any):** If conflicts occur, Git will pause the rebase process and ask you to resolve conflicts in the conflicted files.

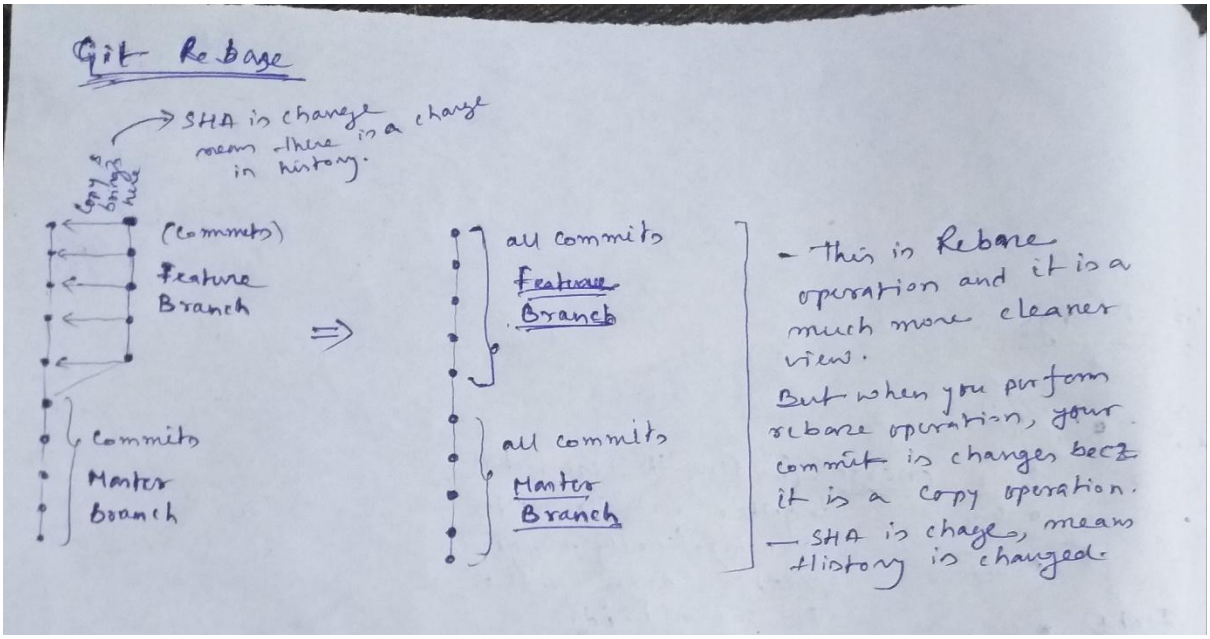
- **Continue the Rebase:** After resolving conflicts, continue the rebase process:
  $ git rebase --continue

- **Repeat if Necessary:** Repeat the conflict resolution and continuation until the rebase is complete.

- **Push the Rebase Branch:** If you have not pushed the branch to a shared repository, you can push the rebased branch.
  $ git push <remote_name> <branch_name> --force
  $ git push origin feature-branch --force

  If conflicts arise during the rebase, resolve them using the standard conflict resolution process, then continue with **git rebase --continue**.





- **Apply a sequence of commits from distinct branches into a final commit.**
  $ git rebase

- **Continue the rebasing process:**
  $ git rebase -continue

- **Abort the rebasing process:**
  $ git rebase --skip

In summary, rebasing in Git is a process that allows you to modify the commit history by moving, combining, or altering commits. It is useful for maintaining a cleaner and more linear project history, especially when incorporating changes from one branch into another. However, it should be used with caution on shared branches to avoid conflicts with collaborators.

When we perform the git rebase then commit history (hash id) is changed?

Yes, when you perform a **git rebase**, the commit history (hash IDs) is changed for the rebased branch. This is because **git rebase** involves rewriting the commit history by creating new commits.

When you rebase a branch into another branch, Git essentially takes the commits from the branch you're rebasing and applies them one by one on top of the target branch. Each applied commit gets a new hash ID because it is a new commit, even if the changes within the commit are the same.

This rewriting of commit history can have implications for collaboration, especially if the branch being rebased is shared with others. If others have already based their work on the original commits, it can cause conflicts or confusion. Therefore, it's generally recommended to use **git rebase** on branches that haven't been shared publicly or to communicate and coordinate with collaborators if rebasing is necessary on shared branches.

Git Interactive Rebase:

Git interactive rebase is a powerful feature that allows you to selectively modify and rearrange commits in a branch interactively. Unlike a regular rebase, which automatically applies commits, interactive rebase provides you with a text-based interface to choose how to modify each commit, such as editing commit messages, squashing multiple commits into one, or reordering commits. This feature is particularly useful for cleaning up commit history, combining related changes, or improving the overall readability of the commit log.

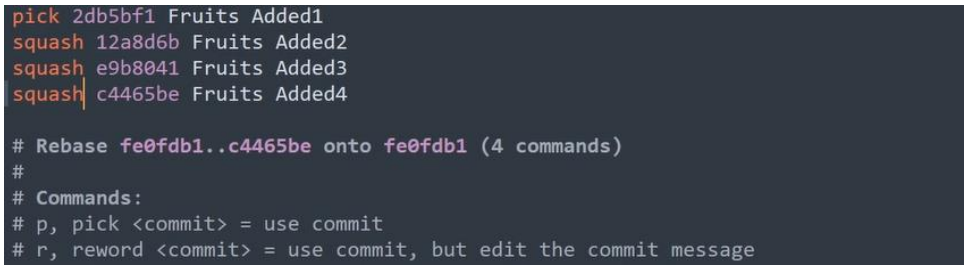
- **Basic Interactive Rebase Command:**
  $ git rebase -i <basic_commit_id>
  This command opens an interactive text file where you can specify operations for each commit in the range from the base commit to the branch's current commit.

- **Importance & Purpose:**
  - ★ **Commit Cleanup:**
    - **Purpose:** Allows you to clean up commit history by combining small, related commits into larger, more meaningful ones.
    - **Importance:** Results in a more coherent and readable history, especially before pushing changes to shared branches.
  - ★ **Commit Message Editing:**
    - **Purpose:** Enables you to edit commit messages to provide more context or adhere to project conventions.
    - **Importance:** Improves the clarity and informativeness of commit messages.
  - ★ **Reordering Commits:**
    - **Purpose:** Allows you to change the order of commits in the history.
    - **Importance:** Helps in organizing commits logically and chronologically.
  - ★ **Splitting Commits:**
    - **Purpose:** Lets you split a single commit into multiple smaller ones.
    - **Importance:** Useful when a commit contains changes related to multiple features or issues.
  - ★ **Squashing Commits:**
    - **Purpose:** Combines multiple consecutive commits into a single commit.
    - **Importance:** Reduces commit noise and maintains a cleaner history.

- **Examples:**
  - Allow various operations like edit, rewrite, reorder, and more on existing commits. Interactive Rebasing changes the history of your branch. Different options are available:
    $ git rebase -i
  - **Squash:**
    To "**squash**" in Git means to combine multiple commits into one. You can do this at any point in time (by using Git's "Interactive Rebase" feature), though it is most often done when merging branches. Please note that there is no such thing as a stand-alone git squash command. Basically, squashing means melting the previous commits.

    Suppose there are 10 commits are there but you see that one commit is useful, then you merge all the commit into one.

    $ git rebase -i HEAD~4 (it means combine 4 commits into one commit. When you run this commit then code editor is opened and then use **squash** instead of **pick,** where **pick** where you want to **squash**).



- **Amend:**
  The **git commit --amend** command is a convenient way to modify the most recent commit. It lets you combine staged changes with the previous commit instead of creating an entirely new commit. It can also be used to simply edit the previous commit message without changing its snapshot. Basically, it helps you to do modification of last commit only.
  $ git commit –amend
  $ git commit –amend -m “messages or commits”
  $ git commit –amend --no-edit (you staged some files but not to commit or use previous commit)

- **Reward:**
  You can modify any commit using particular commit id (using 7-digit SHA)
  $ git rebase -i HEAD~2 (use keyword: **reward** instead **pick;** [commit id will change])

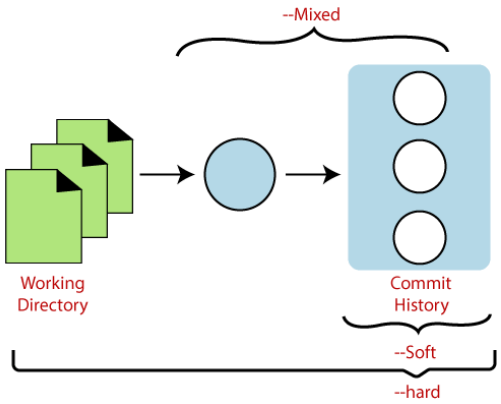
- **Delete:**
  Delete particular commit using interactive rebase.
  $ git rebase -i HEAD~4 (use keyword: **drop** instead **pick;** [commit id will change])

- **Reorder:**
  Basically, reorder the commit (means changing commit id position manually).
  $ git rebase -i HEAD~4 (re-order manually, just changing the position; here commit id will be change)

- **Performing interactive rebase between two commits:**
  $ git rebase <commit_id_A>^5 <commit_id_B>
  Replace A and B with the actual commit hashes or references you are interested in. The ^ after A is a shorthand to include commit A in the rebase. For example, if you want to rebase the last 5 commits starting from commit with hash **ABC123**:
  $ git rebase **ABC123^5 ABC123**
  After running the command, Git will open an interactive rebase file in your default text editor. The file will contain a list of commits between commits A and B.

- **Split:**
  It means, splitting the particular commit
  $ git rebase -i HEAD~4 select particular commit which you want to split. Use **edit** instead of **pick**. Then new branch will be created and do split after that:
  $ git reset HEAD^
  check the git status:
  $ git status
  Staged the files:
  $ git add <file_name_1> && $ git commit -m “message for file_1”
  $ git add <file_name_2> && $ git commit -m “message for file_2”
  Once you satisfied your changes then,
  $ git rebase –continue
  Check the commit status:
  $ git log –oneline

Git Reset:
In Git, the **git reset** command is used to reset the current branch to a specific commit, undoing changes made in subsequent commits. It is a powerful command that can be used to rewrite the commit history, move the branch pointer, and unstage changes, depending on the options provided. The **git reset** command is often used in situations where you need to make corrections or adjustments to the commit history.



- **Basic Commands:**
  $ git reset <commit>
  This command moves the branch pointer and working directory to the specified **<commit>**, discarding changes made in commits after that point.

- **Types of Reset:**
  - ★ **Soft Reset (--soft):** Moves the branch pointer to the specified commit but keeps changes from the subsequent commits staged.
    $ git reset --soft <commit>
  - ★ **Mixed Reset (Default):** Moves the branch pointer to the specified commit and unstages changes from subsequent commits.
    $ git reset <commit>
  - ★ **Hard Reset (--hard):** Moves the branch pointer to the specified commit, discards changes from subsequent commits, and resets the working directory to match the specified commit.
    $ git reset --hard <commit>

- **Importance and Purpose:**
  - ★ **Undoing Commits:**
    - **Purpose:** Allows you to undo commits by moving the branch pointer to a previous commit.
    - **Importance:** Useful for correcting mistakes in the commit history.
  - ★ **Refactoring Commit History:**
    - **Purpose:** Provides a way to refactor or reorganize the commit history.
    - **Importance:** Helps in creating a cleaner and more coherent history.
  - ★ **Staging Changes:**
    - **Purpose:** Can be used to unstage changes from the index (mixed or soft reset).
    - **Importance:** Useful when you want to modify or split commits before making a new commit.
  - ★ **Removing Unwanted Changes:**
    - **Purpose:** Discards changes made in subsequent commits, effectively removing unwanted modifications.
    - **Importance:** Helps in maintaining a consistent state of the codebase.

- **Example Workflow:**
  Let's assume you have the following commit history:

```plaintext
Hash    Message
------------------
A       Initial commit
B       Feature X
C       Feature Y
D       Bug fix
```

Suppose you want to undo the last commit **(Bug fix)** and keep the changes staged for further modifications. You can use a soft reset:

```bash
# Soft reset to commit C, keeping changes staged
git reset --soft C
```

After this command, the branch pointer will move to commit **C**, and the changes from commit **D** will be staged. You can then make additional modifications or commit the changes separately.

Alternatively, if you want to completely discard the last commit and unstage changes, you can use a mixed reset (default behavior):

```bash
# Mixed reset to commit C, unstaging changes
git reset C
```

In this case, the branch pointer moves to commit **C**, and the changes from commit **D** are no longer staged.

If you want to discard the last commit and its changes completely, you can use a hard reset:

```bash
# Hard reset to commit C, discarding changes
git reset --hard C
```

This will move the branch pointer to commit **C** and discard all changes introduced in commit **D**.

Always use caution when performing hard resets, especially when discarding changes, as it can result in permanent data loss. Soft and mixed resets are generally safer and provide more flexibility in adjusting the commit history.

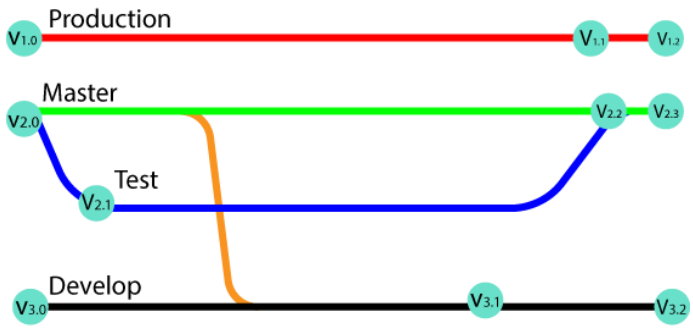
Git Tagging:

Tagging is a mechanism used to create a snapshot of a Git Repository. Tagging is traditionally used to create semantic version number identifier tags that corresponds to software release cycles.

Tags make a point as a specific point in Git history. Tags are used to mark a commit stage as relevant. We can tag a commit for future reference. Primarily, it is used to mark a project's initial point like v1.1.

Tags are much like branches, and they do not change once initiated. We can have any number of tags on a branch or different branches. The below figure demonstrates the tags on various branches.

In the below image, there are many versions of a branch. All these versions are tags in the repository.
After multiple commits, we tag or create a version.



In Git, tagging is the process of creating a reference (tag) that points to a specific commit in the repository. Tags are commonly used to mark important points in the project's history, such as releases, milestones, or significant commits. Unlike branches, tags are typically static and do not move as new commits are added to the repository. Git tags are useful for versioning and for easily referencing specific points in time.

- **Basic Command:**
  $ git tag <tag_name> <commit_message>
  This command creates a lightweight tag named **<tag_name>** that points to a specific **<commit>**. If the commit is omitted, the tag will point to the latest commit on the current branch.

- **Importance and Purpose:**
  - ★ **Release Versioning:**
    - **Purpose:** Tags are commonly used to mark releases or versions of a project.
    - **Importance:** Provides a stable reference point for a specific version of the codebase.
  - ★ **Creating Milestones:**
    - **Purpose:** Tags can be used to mark milestones or significant points in the project's history.
    - **Importance:** Offers an easy way to navigate and reference important commits.
  - ★ **Identifying Stable Points:**
    - **Purpose:** Tags help identify stable and tested points in the commit history.
    - **Importance:** Facilitates collaboration by providing clear references to known working states.
  - ★ **Releasing Notes and Changelogs:**
    - **Purpose:** Tags serve as anchors for generating release notes and changelogs.
    - **Importance:** Helps in documenting changes between different versions.
  - ★ **Collaboration and Communication:**
    - **Purpose:** Tags communicate important information to collaborators about the state of the repository.
    - **Importance:** Enhances collaboration by providing a shared understanding of project milestones.

- **Example Workflow:**

Assume you have a project with the following commit history:

```plaintext
Hash    Message
---------------
A       Initial commit
B       Feature X
C       Feature Y
D       Bug fix
```

Let's say you want to tag commit **C** as the first stable release of your project. You can create a tag named **v1.0** using the following command:

```bash
# Tagging commit C as version 1.0
git tag v1.0 C
```

Now, your commit history looks like this:

```plaintext
Hash    Message
---------------
A       Initial commit
B       Feature X
C       Feature Y
D       Bug fix
```

```java
Tags:
v1.0    Commit C (Feature Y)
```

In this example, the tag **v1.0** points to commit **C**, representing the first stable release of your project. You can use this tag to easily reference and switch to this specific version of the codebase.

To view all tags, you can use the following command:

$ git tag

To view information about a specific tag, including the commit it points to, you can use:

$ git tag <tag_name>

$ git tag v1.0

Tags provide a convenient way to mark important points in your Git history, making it easier to navigate, collaborate, and communicate with your team and users. They are especially valuable for release management and versioning in software development projects.

- **Types of Tags:**
  - **Annotated Tags**
  - **Light-Weighted Tags**

- **Who to create Tags:**
  - When you want to create a release point for a stable version of your code.
  - When you want to create a historical point that you can refer to reuse in the future.

- **Annotated tag:**
  Annotated tags are tags that store extra Metadata like developer name, email, date, and more. They are stored as a bundle of objects in the Git database.

  If you are pointing and saving a final version of any project, then it is recommended to create an annotated tag. But if you want to make a temporary mark point or don't want to share information, then you can create a light-weight tag. The data provided in annotated tags are essential for a public release of the project. There are more options available to annotate, like you can add a message for annotation of the project.
  $ git tag -a **<tag** name**>**
  $ git tag -a **<tag** name**>** <commit_id> (for specific commit, create a tag the you can use 7 digit SHA)
  $ git tag **<tag** name**>** -m "**< Tag** message**>**

- **Light-Weighted Tags:**
  Git supports one more type of tag; it is called as Light-weighted tag. The motive of both tags is the same as marking a point in the repository. Usually, it is a commit stored in a file. It does not store unnecessary information to keep it light-weight. No command-line option such as **-a, -s** or **-m** are supplied in light-weighted tag, pass a tag name.
  $ git tag **<tag** name**>**

- **List of Tags:**
  $ git tag
  $ git show <tag_name>
  $ git tag -l "**<pattern>***" (It displays the available tags using wild card pattern)

- **Delete a Tag:**
  $ git tag -d <tag_name>

- **Show the details of specific tag:**
  $ git show <tag_name>

- **Lists tags with additional information:**
  If you want to see more information about each tag, including the commit they point to and the tag message, you can use the -n option with the git tag command:
  $ git tag -n
  This will display the tag names along with the commit message associated with each tag.

- **Show most recent tag:**
  $ git describe --tags --abbrev=0

Git FORKS:
A FORK is a copy of a particular repository. Forking a repository allows you to create freely experiments with changes without affecting the original projects. There is no such command for FORK. It means that you are doing a copy operation of the entire repository or mirroring the entire repo to my account.

- **It is used when:**
  - If I want to propose some changes to someone's project;
  - Use existing project as a starting point;
  - For bug fixing or resolving issue

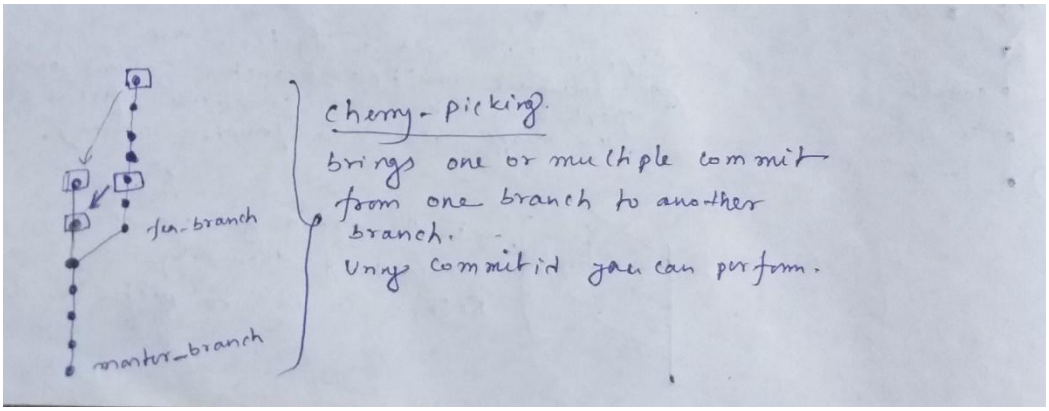
After fixed the bug, then we create a pull request to the project owner. So that he/she can see and give some review the code and then attach in its base branch.
Forking is not a git function, it is a feature of git service like GitHub or Bitbucket, etc.

- **Reasons for Forking the repository:**
  - ★ Propose changes to someone else's project.
  - ★ Use an existing project as a starting point.

Cherry Picking Git:

Cherry-picking in Git stands for applying or bringing some commits from one branch into another branch. It allows you to select a single commit and apply it to the current branch, effectively bringing changes from one branch into another.

- **When we do cherry picking:**
  - Bring the changes from a specific commit.
  - Choose one or commits.



- **Example:**
  - Switch to particular branch where you perform cherry-picking. Then file become available. And new commit id is generated.
    $ git cherry-pick <commit_id>
  - You can perform multiple cherry-picking or bring the multiple commits one branch to another branch.
    $ git cherry-pick <commit_id_1> <commit_id_2> <commit_id_3>
  - Suppose, if you want to bring the commit one branch to another branch and you don't want to commit, then you should use:
    $ git cherry-pick <commit_id> -n
  - After that you can commit:
    $ git add <file_name> && $ git commit -m "messages"
  - If there are conflicts between the changes in the commit being cherry-picked and the existing changes in the target branch, Git will pause and ask you to resolve the conflicts. After resolving conflicts, complete the cherry pick:
    $ git cherry-pick --continue

Git REFLOG:

Git reflog (reference log) is a built-in mechanism in Git that **records the history of references (such as branches and tags) and their associated commits**. The reflog keeps track of changes to branch pointers, allowing you to recover from accidental changes, reset operations, or other actions that modify the commit history. It's a safety net that helps you navigate and recover from situations where references have been moved or deleted.

- **Basic Command:**

  $ git reflog

  This command shows the history of changes to references in your Git repository, including branch movements, commit operations, and more.

- **Importance and Purpose:**
  - ★ **Recovering Lost Commits:**
    - **Purpose:** Reflog allows you to recover lost commits or branch movements caused by accidental resets, rebases, or other operations.
    - **Importance:** Acts as a safety net for undoing unintended changes to the commit history.
  - ★ **Navigation Reference History:**
    - **Purpose:** Provides a detailed history of reference changes, helping you understand how the repository evolved over time.
    - **Importance:** Offers transparency into operations that modify references, aiding in troubleshooting and understanding repository state.
  - ★ **Undoing Reset Operation:**
    - **Purpose:** Enables you to undo reset operations by referencing the commit IDs from the reflog.
    - **Importance:** Helps in recovering from accidental or undesired changes, such as a hard reset.
  - ★ **Branch Recovery:**
    - **Purpose:** Assists in recovering deleted or lost branches by providing a history of reference changes.
    - **Importance:** Acts as a safety net for branches that were mistakenly deleted or reset.

- **Example Workflow:**

  Consider the following scenario where you accidentally perform a hard reset on a branch, losing commits:

```plaintext
Hash    Message
---------------
A       Initial commit
B       Feature X
C       Feature Y
```

  Suppose you accidentally run the following command, losing the **Feature Y** commit:

```bash
# Accidental hard reset
git reset --hard HEAD^
```

Now, the branch pointer has moved back one commit:

```plaintext
Hash    Message
---------------
A       Initial commit
B       Feature X
```

To recover the lost commit, you can use the reflog:

```bash
# View the reflog
git reflog
```

The reflog output might look like:

```plaintext
HEAD@{0}: reset: moving to HEAD^
HEAD@{1}: commit: Feature Y
HEAD@{2}: commit: Feature X
HEAD@{3}: commit (initial): Initial commit
```

In this example, you can see that the lost commit **Feature Y** is recorded in the reflog at position **HEAD@{1}**. To recover the lost commit, you can reset the branch pointer to this commit:

```bash
# Recover the lost commit
git reset --hard HEAD@{1}
```

Now, the branch is restored to its previous state:

```plaintext
Hash    Message
---------------
A       Initial commit
B       Feature X
C       Feature Y
```

The reflog played a crucial role in recovering the lost commit by providing a history of reference changes, allowing you to identify and reset to the desired commit. It serves as a safety net, helping you recover from accidental operations that modify the commit history.