**GIT & GITHUB**

**Section 7: Git Commands 2**

1. **Pushing Changes to a Git Repository**

What is Git Push?

After the user is done with the modifications in the Local repository, there is a need to push these changes to the remote repository. This can be done with the git push command. Users can define which branch is to be pushed into the repository by passing its name as an argument. By default, the data will be pushed from the current branch into the same branch of the remote repository.

**Push Commits To a CodeCommit In Amazon Web Services (AWS):**

Follow the steps mentioned below to push the commit to CodeCommit in Amazon Web Services (AWS):

Step 1: Make sure that you have permission to access the CodeCommit in Amazon Web Services(AWS).

Step 2: Use the below command you can clone the repository needed to your local repository.

**git clone <HTTPS or SSH URL>**

**Step 3: Stage the modified files using the command below**[**(git add)**](https://www.geeksforgeeks.org/git/what-is-git-add/)**.**

**git add .**

**Step 4: Commit the**[**staged files**](https://www.geeksforgeeks.org/git/staging-in-git/)**into the local repository using the following command. Provide a commit message that details the changes you made and is descriptive.**

**git commit -m "message"**

**Step 5: To push changes to code commit us the following command.**

**git push <Remote URL>**

**The remote URL should be in the format shown below.**

**https://git-codecommit.[region].amazonaws.com/v1/repos/[repository-name].**

**Step 6: Repeat the above 3rd and 5th step if you face any problems like merge conflicts.**

**If you encounter any conflicts during the push, resolve them manually, and then repeat steps 3 to 5 to push the changes again.**

**Renaming Branches:**

If you want to change the name of the current working branch, first you have to switch the branch first, after that rename the branch.

**Commands:**

**git branch -m <old-branch> <new-branch>**

**By doing this, the old branch name will be changed to a new branch name.**

**Delete Branch in the Remote Repository (GitHub):**

**Command:**

git push origin :<old-branch>

(OR)

git push --delete origin my-branch

================================================================================

1. **Examining Git**
2. **git log:**

As the name suggests, this command shows commit logs. It is a command which allows you to view information about the previous commit. Unlike git status command, it only inspects the history of committed repository.

* **git log --all or git log:** This command displays all commits.
* **git log -n "limit":** This option helps to filter the commit history by applying the limit. Replacing "limit" with a number of commits will limit output as n most recently committed ones.
* **log --author="name":** This option limit the output to commits by particular author "name".
* **git log --committer="name":** This option limit result to commits that were committed by that individual.
* **git log -p:**This option shows the most detailed view of history. Here p stands for patch.
* **git log --oneline:**This option is used to get bare information in a single line per commit.
* **git log --stat:** This option helps to view the summary of changes made in each commit.
* **git log "file":** This option display commits that include specified file.
* **git log --before(or --after) "date":**This limits the commits to those within a given date range. The date is specified as string in "yyyy-mm-dd" format.
* **git log --after "date" --before "date":**This option specifies a date range.

1. **git shortlog:**

This command is "Sub-command" of git log as it summaries git log output. Each commit is grouped by author and title.

* **git log -n**  
  **--numbered:**It display the output according to number of commits per author.
* **git log -s**  
  **--summary:**This option suppresses commit description
* **git log -e**  
  **--email:**This option displays the email address of each author.
* **git log -c**  
  **--committer:m** This option shows committer identities instead of authors.
* **git log "revision range":**Displays commits in specified revision range.

1. **git diff:**  
   This command is used to compare different versions of the file or in other words, it shows the changes between the commits, working tree, branches, files, etc.

================================================================================

1. **Git Checkout and Merge**
   1. **git Checkout**

The git checkout command is used to switch between different branches or to restore files in your working directory. It’s like changing paths or opening different versions of your project.

**Key Functions of git checkout:**

**Switching Branches:** When you want to move between branches, you use git checkout. For example, if you have a feature branch and want to switch to it from the main branch, you would use this command.

git checkout feature-branch

**Creating a New Branch:**If you want to start working on a new branch, you can create and switch to it in a single command using the -b option. This creates a new branch and automatically switches to it.

git checkout -b new-feature

**Restoring Files:**If you’ve made changes to a file and want to undo them, you can restore the file to its state in the last commit.

git checkout – filename

(or)

git checkout .

**Why to use git checkout**

* **Switch Between Branches**: git checkout lets you move between different branches to work on various features or versions of your project.
* **Restore Files or View Old Commits**: It allows you to revert files to a previous state or check out an older commit to review past changes.
  1. **Git merge:**

The git merge command is used to combine changes from one branch into another. It’s like taking two paths in your project and merging them into a single path.

You can understand it as a person A who wants to become person B so what he does is he writes a command as git merge and then write person B so now A will inherit all the properties from person B and now A has become B as it has merged with B.

**Working process of git merge**

* Merges changes from one branch into another.
* If both branches have new changes, Git creates a special commit to combine them.
* If no new commits exist on the current branch, Git simply moves the branch pointer forward.
* If the same file is changed in both branches, Git requires manual conflict resolution.
* Unlike rebase, merging keeps the commit history of both branches intact.

**Why to use git merge**

* **To combine changes**: Git merge helps bring changes from one branch into another.
* **To keep history clear**: It keeps all past commits, making it easy to track changes.

**Commands of git merge:**

1. **Basic Merge:**

At the first step git checkout command is used to travel to a branch and then[git merge](https://www.geeksforgeeks.org/git/git-merge/) command with the name of the branch whose state has to be adopted by the checkout branch is written then after the current branch updates to the branch into which it is getting merged

git merge branch\_name

===================================================================================

1. **How To Revert A Commit With Git Revert?**

When working with Git, it is common to make a commit that later needs to be undone whether due to a bug, accidental changes, or a mistake in the workflow. Instead of deleting or rewriting history, Git provides the git revert command to safely undo a specific commit by creating a new one that reverses its changes.

**Git Commit**

A commit in Git represents a snapshot of your project at a specific moment. It includes:

* Changes made to the codebase
* A message describing the change
* A unique commit hash that identifies it

**Syntax**

git commit -m “Commit\_Message”

**Git Revert:**

Sometimes after committing changes, you realize there is a bug or issue, and you want to undo that specific commit **without deleting it**. The git revert command is designed for this purpose.

It **creates a new commit** that reverses the changes introduced by a previous commit, maintaining a complete and accurate commit history.

**Syntax:**

git revert <commit\_hash\_id>

**Points To Remember**

* git revert does not delete the original commit.
* It adds a new commit that undoes the changes.
* You can revert individual or multiple commits.
* Ideal for shared branches because it doesn't rewrite history.

**Useful Git Revert Options:**

**1. no-commit**

Since the git revert command adds a new commit by default. This flag applies the revert to your working directory and staging area but doesn't create a new commit. It allows you to make further modifications or additions before committing the reversion.

**Command:**

git revert --no-commit <commit-hash>

**2. -m <parent-number>**

If you are reverting a merge commit, this flag allows you to specify which parent's changes you want to revert. You provide the parent number (starting from 1) to indicate which parent's changes should be reverted.

**Command:**

git revert -m 1 <merge-commit-hash>

**3. --no-edit**

By default when we run the git revert command, a vim editor is shown where we can edit the commit message for the reverted message. However, if we do not want to change the commit message and use the default message then we can use this flag.

**Command:**

git revert --no-edit <commit-hash>

**Git Revert V/s Git Reset:**

**A black screen with white text

AI-generated content may be incorrect.**

**Git Reset V/s Git Revert V/s Git Checkout:**

**A black and white screen with white text

AI-generated content may be incorrect.**

===================================================================================

1. **Submodules in Git:**

In software development, it’s common to manage projects that depend on other projects. Git submodules are a powerful feature that allows you to include and manage external repositories within your main repository. This feature enables you to keep multiple repositories in sync and makes it easier to work on complex projects that rely on various codebases.

**What Are Git Submodules?**

Git submodules are pointers to another repository at a specific commit. They allow you to include a separate repository within your main repository, maintaining a connection between the two. This means you can manage dependencies as part of your project without merging all the code into a single repository.

**How They Work**

When you add a submodule, Git doesn't copy all the files from the external repository into yours. Instead, it records the **commit hash** of the external repository. . This means your main repository only stores a pointer to a specific commit in the submodule's repository. When someone clones your main repository, they'll also need to initialize and update the submodules to get the actual files. The submodule itself maintains its own history, branches, and tags, completely separate from the parent project.

**Benefits of Using Git Submodules**

* **Modularity**: Keep your project modular by separating dependencies into distinct repositories.
* **Isolation**: Manage changes in external dependencies without directly affecting your main project.
* **Version Control**: Pin dependencies to specific versions, ensuring stability and consistency.
* **Reusability**: Reuse submodules across multiple projects, avoiding code duplication.

**Use Cases for Git Submodules**

* **Large Projects**: Split large projects into manageable components.
* **Shared Libraries**: Include common libraries shared across different projects.
* **Third-Party Dependencies**: Manage third-party code dependencies separately from your main codebase.

**Key Commands**

1. **Adding a submodule**: Use git submodule add <URL> <path>. This command clones the external repository into the specified <path> and adds a .gitmodules file to your main repository, which keeps track of the submodule's URL and path.
2. **Cloning a repository with submodules**: After cloning the main repository, you need to run git submodule init to initialize your local configuration file, followed by git submodule update to pull the files for the submodule. A shortcut for this is git clone --recurse-submodules <URL>.
3. **Updating a submodule**: To update a submodule to its latest commit, navigate into the submodule's directory and run git pull. Then, go back to the parent repository and commit the change to record the new commit hash.
4. **Removing a submodule**: This process is a bit more involved. You need to:
   1. Remove the relevant line from the .gitmodules file.
   2. Run git rm --cached <path\_to\_submodule>.
   3. Delete the .git/modules/<submodule\_name> directory.
   4. Commit the changes.
5.  **A Submodule is a Pointer:** Your main repository doesn't actually contain the files from the submodule. Instead, it holds a **pointer** to a specific commit of that submodule's repository. This pointer is what gets tracked and versioned in your parent project's commit history.
6.  **git status Behavior:** If someone else makes changes, commits them, and pushes to the main submodule repository, your parent repository is still pointing to the **old commit**. To see the changes in your local submodule directory, you need to manually update it.
7.  **The "Modified" State:** When you update your submodule to a new commit (e.g., by running git submodule update), Git in the parent repository will recognize that the submodule's directory is now at a different commit than the one it's tracking. At this point, git status in the parent directory will show the submodule as **"modified content"**, not "untracked." This indicates that the submodule's state has changed and you need to commit this new pointer in your main repository to record the update.

**Workflows in submodules:**

As soon as you initialize the submodules and update them within a parent repository now we can use them as a different repository that can have their own branches and history. So when we are creating some changes in the submodule it's important to have a track on them and commit them properly. So let's switch to our submodule here  Submodules-in-git

A screen shot of a computer screen

AI-generated content may be incorrect.

Now let's move back to the parent repository of this submodule which is present in folder10 and let's see the status of the parent repository. As we can see that now the parent repository is aware that some changes are made in the submodules  Now if one wants to update this parent repository you can do so by using git add and git commit commands.

A screen shot of a computer

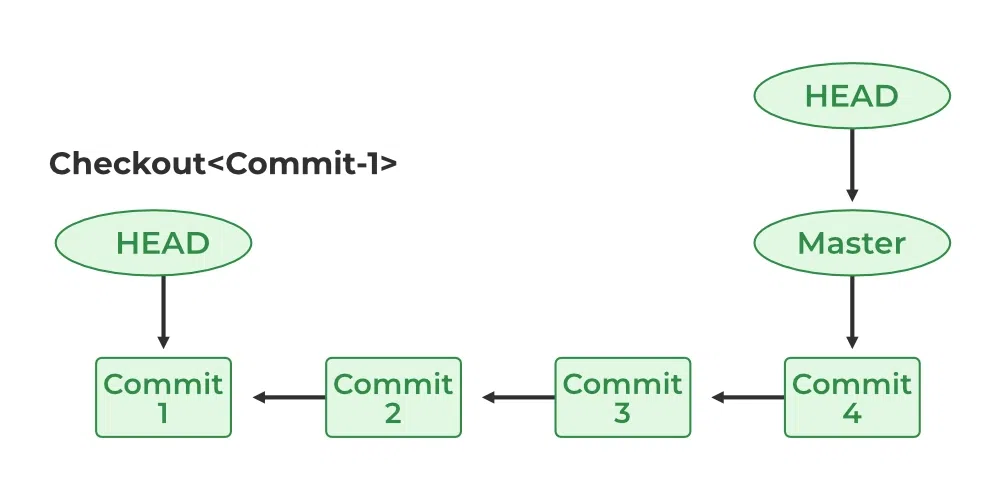
AI-generated content may be incorrect.

===================================================================================

1. **Git Head:**

In Git, HEAD is a reference to the current check-out commit in your repository. It's basically a pointer or symbolic reference to the latest commit in your branch. Every time you switch branches or check out a specific commit, HEAD moves accordingly to point to the relevant commit.

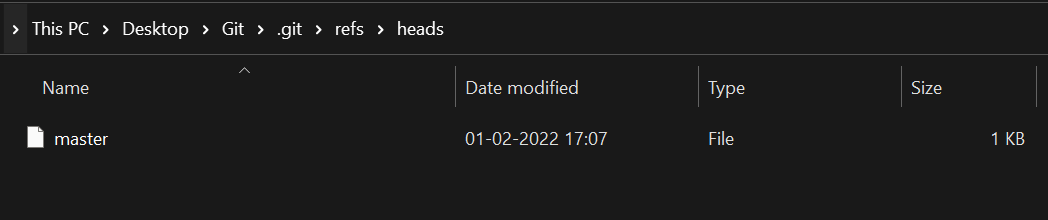
The most recent commit to the current checkout branch is indicated by the HEAD. A pointer to any reference, in a sense. The "current branch" can be thought of as the HEAD. The HEAD is moved to the new branch when you use the "checkout" command to swap branches.



**Git Refs and Heads:**

he provided text explains the concept of **HEAD** in Git, defining it as a pointer to the most recent commit in the current branch. It delves into the underlying mechanics by introducing **refs** and **heads**, which are part of the .git directory. The text explains that the .git/refs directory contains subdirectories like heads and tags.

* **HEAD** is a symbolic reference that points to the tip (the most recent commit) of the currently checked-out branch. Think of it as your project's current location marker.
* **Refs** (short for "references") are simple files inside the .git/refs directory that store a commit's full SHA-1 hash. They are human-readable pointers to specific commits. Common types include branches (refs/heads/) and tags (refs/tags/).
* **Heads** are a specific type of ref. The files within the .git/refs/heads directory represent your local branches. For example, the master or main file inside this folder contains the commit ID of the latest commit on that branch.
* The system uses this structure to keep track of the latest commit for each branch, allowing Git to know what commit a branch is currently on. When you make a new commit, the file for the current branch inside .git/refs/heads is updated with the new commit's ID.
* In essence, **HEAD** is a pointer to the **head** (the latest commit ID) of the current branch, which is, in turn, stored as a **ref** file. This hierarchical relationship is fundamental to how Git operates.



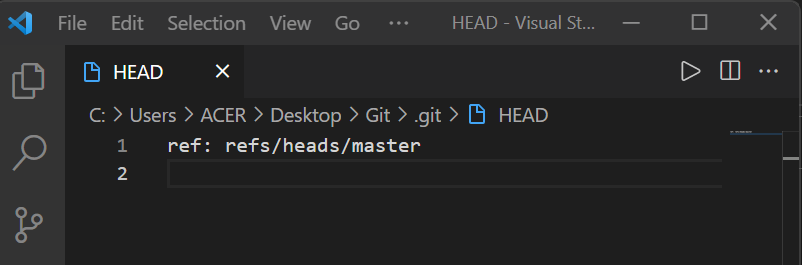
 **Commit ID Storage**: The files within .git/refs/heads/—such as the master file—are not just named after branches; they contain the specific **commit ID** of the most recent commit for that branch. You can verify this using the git log command.

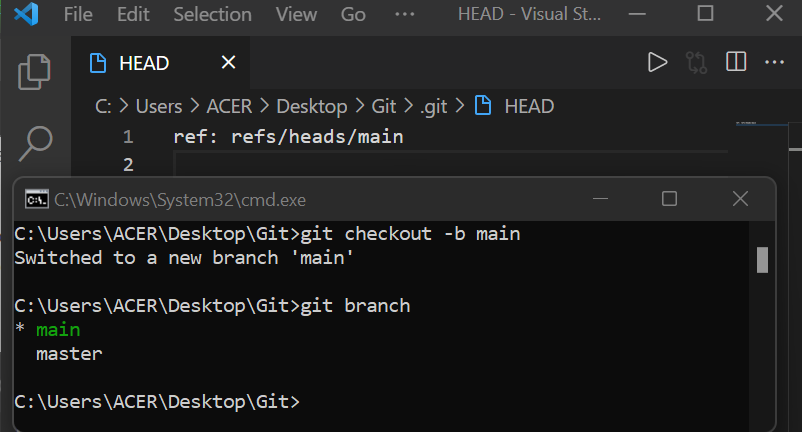
 **Pointers to Commits**: Both **refs** (references) and **heads** are essentially pointers to the latest commits. They function as lightweight files that map a branch name (the filename) to a commit's unique identifier (the file's content).

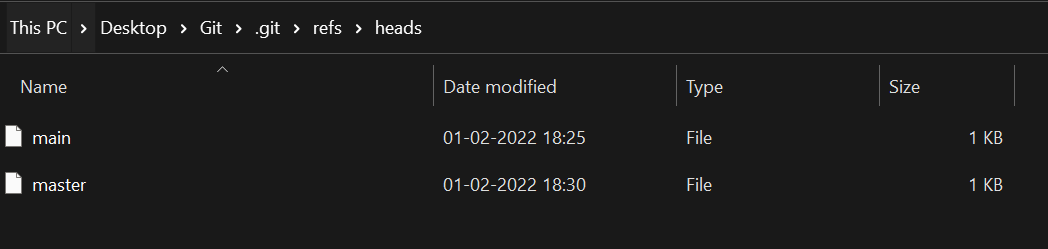
 **Tags**: Similar to how branches are tracked in .git/refs/heads/, **tags** are also stored as files in the .git/refs/tags/ directory. These files contain the commit ID that the tag points to, often used for marking specific, important points in a project's history, like a release.

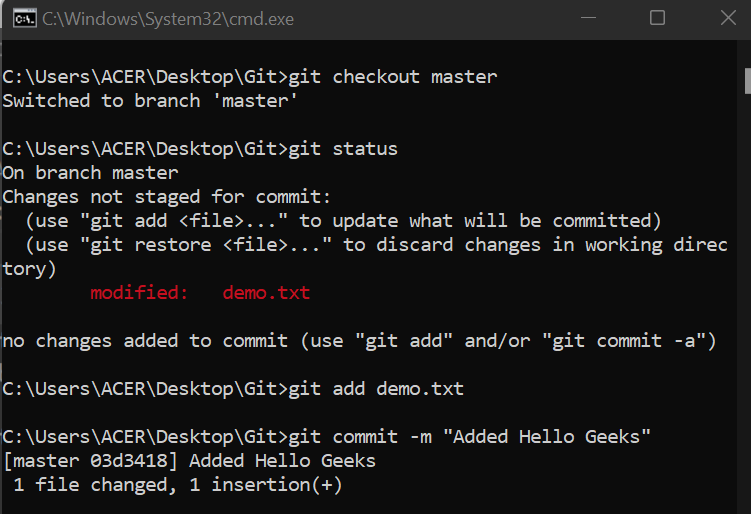
 **Simplified Tracking**: This file-based system makes it easy for Git to know the current state of a branch or tag without having to scan the entire commit history. When a new commit is made, Git simply updates the relevant branch file with the new commit's ID.

**Let us check what the master file consists of as follows:**









 **HEAD is a special pointer**: Unlike a regular branch head (which points to the last commit of a specific branch), **HEAD** is a special pointer that points to the currently active branch. It's the "you are here" marker for your working directory.

 **The .git/HEAD file**: The location of the HEAD pointer is in the .git/HEAD file. The contents of this file indicate the current branch. For instance, if you're on the master branch, the file will contain a reference to refs/heads/master.

 **HEAD changes with branch checkout**: When you switch branches (e.g., from master to main), the .git/HEAD file is updated to point to the new branch's reference. This is how Git knows which branch you are working on.

 **Commits don't directly change HEAD**: Adding new commits to a branch doesn't change what **HEAD** points to. Instead, it updates the branch's **head** (the file in .git/refs/heads/), which HEAD is already pointing to. HEAD continues to point to the same branch, but the branch itself now points to a new, more recent commit.

 **Detached HEAD state**: There's a scenario where HEAD points directly to a specific commit, not a branch. This is known as a **detached HEAD** state and can happen when you check out a tag or an older commit. In this state, any new commits won't belong to a branch.

**Git Show Head:**

To verify the HEAD status use the below command and it will also show the HEAD location.

**Syntax:**

git show HEAD

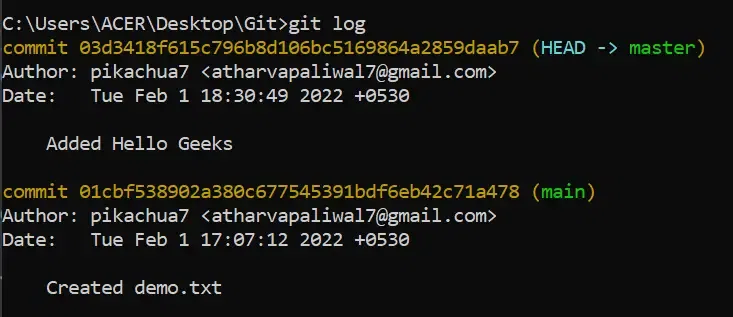


From the above image, we can observe the commit ID is given by the HEAD. It means the HEAD is on the given commit.

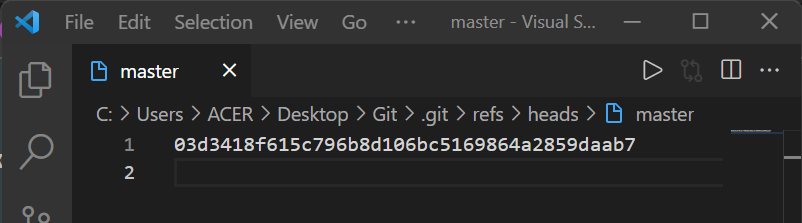
lets us check the commit [history/logs](https://www.geeksforgeeks.org/git/how-to-check-git-logs/) of the project by using the below command.

**Syntax:**

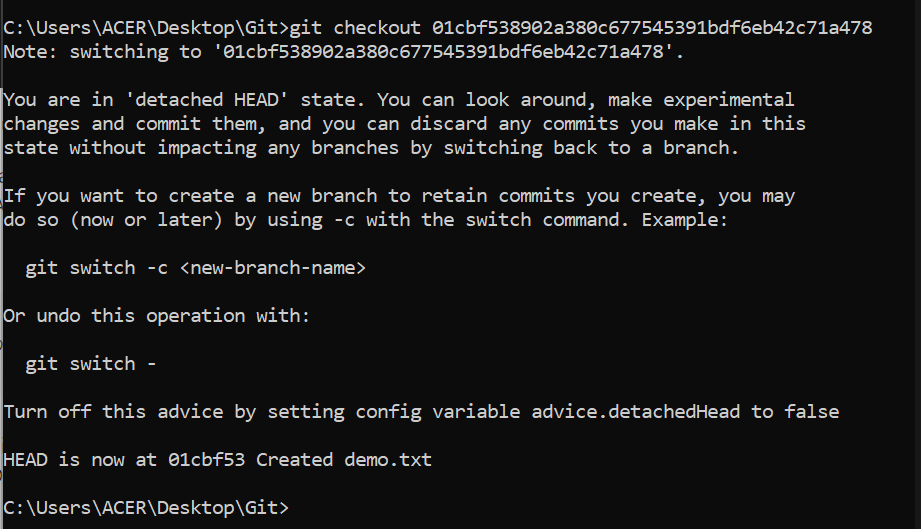
git log



We have two commits and our HEAD is now pointing to the most recent commit we have done. You can also check this in your **"**.**git/refs/heads"** folder.



**Git Detached Head:**



* **Detached HEAD State**: Normally, HEAD points to a branch (e.g., refs/heads/main), which in turn points to the latest commit on that branch. In a detached HEAD state, HEAD bypasses the branch reference and points directly to a commit's hash.
* **How it Happens**: The most common way to get a detached HEAD is by using the command git checkout <commit ID>. This moves the working directory and history back to that specific commit.
* **Checking the State**: You can confirm a detached HEAD by checking the .git/HEAD file. Instead of a branch name, it will contain the full commit hash.
* **Use Case**: A detached HEAD is useful for inspecting older versions of a project. You can check out a specific commit to view the project's state at that point in time, which can be helpful for debugging or reviewing old code.
* **Caution with New Commits**: If you make new commits while in a detached HEAD state, those commits won't be associated with any branch. They will exist but are not part of any branch's history. To save these new commits, you must create a new branch from your current HEAD position. For example, git checkout -b <new-branch-name>.

**Reverting Files**

* **Reverting a Single File**: If you only need to revert a specific file to a previous state without changing the entire working directory, you can use the command git checkout <commit ID> <file>. This command pulls just that single file from the specified commit, leaving the rest of the project's files untouched. This is a common and safe way to restore a file without triggering a detached HEAD state.

**==================================================================================**

1. **Git Clean:**

**git clean** is a Git command that helps you remove untracked files and directories from your working directory. These untracked files are those that are not being tracked by Git, meaning they are not part of your repository's version history. They might be generated during development, such as build unique, temporary files, or files you decided not to commit.

**Why Use git clean?**

Over time, a repository can accumulate many untracked files, leading to clutter and potential confusion. Using git clean helps you:

1. **Maintain a tidy Repository**: Remove unnecessary files to keep your working directory clean and organized.
2. **Avoid Mistakes**: Ensure that you don't accidentally commit unwanted files.
3. **Optimize Performance**: Improve the efficiency of your repository by eliminating excess files.

**How to Use `git clean` Safely:**

* 1. **Preview the Changes:**

git clean -n **(or)** git clean –dry-run

This command lists all the files and directories that git clean would remove, giving you a chance to review and ensure that no important files are included.

* 1. **Remove Untracked Files:**

Once you've reviewed the files, you can proceed with the actual cleanup by using the -f or --force option:

**git clean -f**

* 1. **Remove Untracked Directories:**

If you also want to remove untracked directories, use the -d option along with -f:

git clean -fd

**4. Interactively Remove Files**

For a more controlled cleanup, use the -i or --interactive option. This allows you to select which files to remove interactively:

git clean -i

**5. Remove Ignored Files**

To remove files that are listed in your .gitignore, use the -x or --ignored option:

git clean -fx

**6. Git clean directories**

By using **"--force"** will remove only the untracked files if you want to remove all the directories then the following command will help you to do that.

git clean --directories

**7.Prune remote branches:**

Once the [remote branches](https://www.geeksforgeeks.org/git/introduction-to-git-branch/) are deleted or no longer needed, by using the following command you can remove references to those branches from the local repository. Helps in keeping sync with the remote repository.

git remote prune <Alias name of remote repository>

**Delete local and remote tags:**After merging sub-branches into main branches there is no use of the tags also. You can delete those tags with the help of the following.

**For deleting local tags:**

git tag -d <tag\_name>

**For deleting remote tags:**

git push --delete origin <tag\_name>

**Git Clean Advantages:**

* Removing untracked files
* The scope is limited
* No default interaction
* Can't remove directories

**Git Clean Limitations:**

* Files will delete permanently
* The scope is limited
* No default interaction
* Can't remove directories

===================================================================================

1. **Git Rename:**

**Way 1: Rename using GitHub:**

**Step 1:**Open GitHub.

**Step 2:** Open the repository to rename any file in that repository.

**Step 3:** Open the file which we want to rename.

**Step 4:** Click the edit button and rename the file.

**Step 5:** After renaming the file, commit the changes.

**Step 6:** A file with the new name will be saved.

**Way 2: Renaming using the Command Line**

**Step 1:** Open Git Bash.

**Step 2:** Open the repository.

**Step 3:** Rename the file using the command:

git mv old\_filename new\_filename

**Step 4:**Use the **"git status"** command to check the changes.

**Step 5:** Commit the renamed file.

git commit -m "Renamed\_file"

**Step 6:**Push the changes using **"git push origin branch\_name"**

A screen shot of a computer program

AI-generated content may be incorrect.

===================================================================================

1. **Git Fork:**

When working on collaborative or open-source projects, using git clone gives you a local copy of the repository, but you don't own it and can’t directly push changes unless you have write access. This becomes a problem when contributing to someone else’s project.

Git Fork solves this by creating your own copy of the repository under your GitHub account. This allows you to make changes freely, without affecting the original project. Once your changes are ready, you can submit a pull request to propose them to the original repository. Forking is essential for safe, independent development in collaborative environments.

**What is Git Fork?**

In Git, a fork refers to a personal copy of another user’s repository. When you fork a repository, you create an independent copy that exists within your own account or organization. This copy includes

* All the files
* commit history
* branches present in the original repository at the time of forking.

**Why Use Forking?**

Forking is helpful in the following scenarios:

* You have your own copy of the project on which you may test your own changes without changing the original project.
* This helps the maintainer of the project to better check the changes you made to the project and has the power to either accept, reject or suggest something.
* When you clone an [Open-Source](https://www.geeksforgeeks.org/software-engineering/introduction-to-open-source-and-its-benefits/)project that isn’t yours, you don’t have the right to push code directly into the project.

**Step-by-Step Guide to Fork a Repository:**

**Step 1: Navigate to the Repository**

* Open the repository we want to fork on GitHub.
* We will see a Fork button in the top right corner of the repository page.

**Step 2: Click on the Fork Button**

* Click Fork to create a copy of the repository in the [GitHub](https://www.geeksforgeeks.org/git/introduction-to-github/)account.

**Step 3: Confirm the Fork**

* After forking, we will see the repository under the account.
* The repository name will appear as the-username/repository-name.
* The original repository link will still be available under the forked repository
* Find the Fork button in the top right corner.
* Fork button

**Step 4: Make Changes and Create a Pull Request**

* After forking, we can make changes in the own repository.
* To contribute to the original project, create a [Pull Request](https://www.geeksforgeeks.org/git/git-pull-request/) (PR).
* The maintainers can review and either accept or reject the changes.

**Fork Using Command Line:**

Step 1: Install and Verify GitHub CLI

* Open the terminal or [gitbash](https://www.geeksforgeeks.org/git/working-on-git-bash/) and type the below command.

gh --version

* If it returns a version number, GitHub CLI is installed. If not, install it from [GitHub CLI](https://www.geeksforgeeks.org/git/what-is-github-cli/).

Step 2: Authenticate GitHub CLI

* Before forking, log in to GitHub using the CLI:

gh auth login --web > SSH

* Follow the authentication steps in the browser.

Step 3: Fork the Repository

* Once authentication is done. Copy the Repo URL that we want to fork into our repo and use the below command.

gh repo fork <REPO URL> --clone

* The --clone flag automatically clones the repository after forking.

**Difference between Git Fork vs Git Clone:**

**A screenshot of a black and white screen

AI-generated content may be incorrect.**

**How to Decide Between Git Clone and Git Fork?**

**Use git clone if:**

* You are working within an organization where you have write access.
* You want to push/pull directly to/from the main project repository.
* You are contributing as a collaborator.

**Use git fork if:**

* You are working on an open-source project without write access.
* You want to create a private variation or experiment safely.
* You want to submit changes via pull requests.

**Configuring Git to Sync the Fork with the Upstream Repository:**

Adding an upstream remote to a Git fork allows you to track changes from the original repository you forked from.

In the context of a Git fork, the **upstream** repository is the original project that you copied. Your forked repository is considered the **origin**.

 **Origin:** Your personal copy of the repository (e.g., https://github.com/your-username/project-name).

 **Upstream:** The original repository you forked from (e.g., https://github.com/original-creator/project-name).

**The Process of Adding an Upstream Remote:**

**Adding the Remote:**

You add the original repository's URL as a new remote, typically named upstream.

**git remote add upstream** [**https://github.com/original-creator/project-name.git**](https://github.com/original-creator/project-name.git)

This command tells your local repository, "Hey, there's another repository I want to track, and I'm going to call it upstream."

**Fetching and Merging/Rebasing:**

git fetch upstream

git checkout main

git merge upstream/main

Once the remote is added, you can fetch all the branches and commits from the upstream repository.

After fetching, you can merge or rebase these changes into your local branch to sync your code. For example, to update your local main branch:

**Why is Upstream Used?**

**Pulling Latest Changes:** Projects are constantly being updated by other contributors. By having an upstream remote, you can easily pull the latest bug fixes, new features, and changes made by the original project maintainers. This ensures your local environment isn't outdated.

**Avoiding Merge Conflicts:** Regularly syncing your fork with the upstream repository minimizes the chances of significant merge conflicts when you eventually create a pull request to contribute your own changes. If your code is based on an old version, integrating it with the current project can be difficult.

**Contributing Back:** When you want to contribute your changes back to the original project, your code needs to be up-to-date with the main branch. Pulling from upstream is a standard practice before submitting a pull request to ensure your contribution is based on the latest codebase.

After forking a repository, we might want to keep it updated with the original project. Follow these steps to sync the fork,

**Step 1: Clone the Fork**

Clone the repository that we forked to the directory where we want to store the repository by using the following command.

**git clone <forked-repository-URL>**

**Step 2: Add the upstream Repository**

Add the upstream repository as a remote to the fork repository. By using the following command we can do this.

**cd <forkrepo>**

**git remote add upstream <original-repository-URL>**

**Step 3: Fetch the upstream Changes**

Use the[Git fetch](https://www.geeksforgeeks.org/git/git-fetch/) command and fetch the branch required from the upstream repository.

**git fetch upstream**

**Step 4: Merge Upstream Changes into the Branch**

Switch the branch we want to update by using the following command. Merge the branches which are fetched from upstream by using the below command.

**git checkout main**

**git merge upstream/main**

**Step 5: Push the changes to the fork repository**

 With the help of the following command, we can push the changes to the fork repository.

**git push origin main**

Now the forked repository is in sync with the original repository.

===================================================================================

1. **Git Init:**

The git init command is used to create a new Git repository. When you run this command, Git sets up the necessary structures and files in the current directory, allowing you to start tracking changes. This are the files which are generated during the .git init command.

1. **Creation of the .git Directory**: The command creates a hidden directory named .git in your project’s root directory. This directory contains all the metadata and object files that Git uses to manage the repository.
2. **Initial Setup Files**: Inside the .git directory, Git generates several important files and subdirectories, including:
   * **HEAD**: A reference to the current branch.
   * **config**: Configuration settings for the repository.
   * **description**: A brief description of the repository (used by GitWeb).
   * **objects**: Directory to store all the objects (blobs, trees, commits).
   * **refs**: Directory to store pointers to commit objects for branches and tags.

**Why Use git init?**

The git init command is important because it lays the groundwork for version control in your project. Here are some key reasons to use it:

* **Version Control**: It enables tracking of changes in your project, making it easier to revert to previous states, compare changes, and understand the history of your project.
* **Collaboration**: By initializing a Git repository, you can share your project with others, enabling collaborative development through platforms like GitHub, GitLab, or Bitbucket.
* **Backup**: Git provides a reliable way to back up your project. Each commit serves as a snapshot of your project at a specific point in time, which can be restored if needed.

**Git init Options and Usage**

Following are the someone options that can be used with the git init command.

* initialize the repository in the specific directory by giving the path.
* git --bare will initialize the bare repository it will not consist of any working directory it is used to share the file between the users.
* git init --q will remove the unnecessary outputs during the git initialization.
* git init --shared you can the repository between multiple users if you create by using this command.

**Git init vs. Git clone:**

**A screenshot of a black and white screen

AI-generated content may be incorrect.**

**Git init Bare Repositories:**

The bare git repsoitory doesn't contsain the working directory it is mainly used for the collaboration of developers to perform the push and pull operations. Bare repositories is mainly used for the storing the metadata and the version hsitory of the repositories.

Following is the command which is used for the intiliaizing the git bare repsository

***git init --bare <Name of the repository>***

Git bare repsoitories are mainly used as the central repository from where the other developers can push and pull the repsoitories.

**Git init Templates**

Before intializing the git repostory you can describe the state or you can get the custmized repository with the help of git templates. You can standaridze the repository before it is going to create which will automate you so many tasks like creating files like README file or adding a .gitignore file.

===================================================================================

1. **Git Add:**

The `**git add`** command is used to add changes in your working directory to the staging area. The staging area, also known as the index, is where you prepare a snapshot of your project’s current state before committing it to the repository.

**Why Use git add?**

* **Selective Staging**: You might not want to commit all changes at once. git add allows you to select specific files or changes to include in your next commit.
* **Organized Commits**: By staging specific changes, you can create more meaningful and organized commits. This is particularly useful when you’re working on multiple features or bug fixes simultaneously.
* **Error Reduction**: Staging your changes allows you to review what will be committed, reducing the chance of accidental or unnecessary changes being included.

git add . : Staged new and modified files without deleting.

git add -a : Staged all files to the staging area.

git add -u : Staged modified and deleted files.

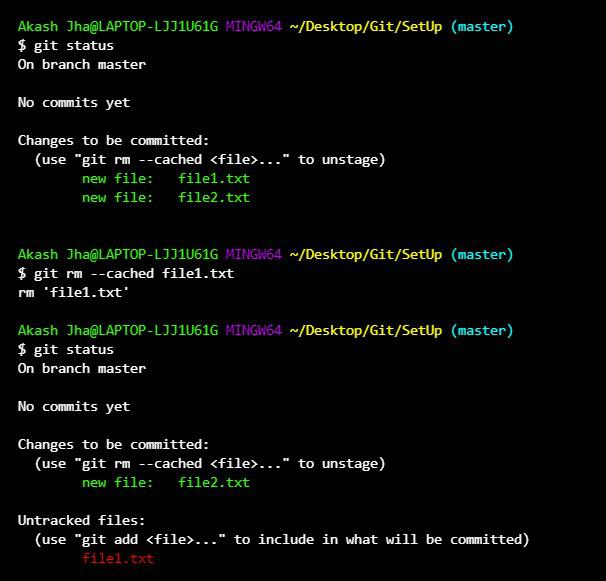
**Removing file from Staging Area:**

Command:

**git rm --cached file-name**

***Note****: Note that this will not delete the file, this will only remove the file from the staging area.*

:



===================================================================================

1. **Git Commit:**

A commit in Git is basically a snapshot of your project at a particular point in time. Each commit captures the state of the files in your working directory and includes a unique identifier (a SHA-1 hash), a commit message, and metadata such as the author and timestamp.

**Key Components of a Commit**

* **SHA-1 Hash**: A unique identifier for each commit.
* **Commit Message**: A descriptive message that explains the purpose of the commit.
* **Author**: The person who made the commit.
* **Timestamp**: The date and time when the commit was made.
* **Parent Commit(s)**: The preceding commit(s) in the project history.

**Commit Commands:**

* 1. **Common Commit:**
     + **git commit -m “Commit\_Message” 🡪 This command will not open the editor for giving commit message. We have given the inline message.**
     + **git commit 🡪 This command is like above but, it will open the editor for giving commit message.**
     + **Git commit -m “Message\_1” -m “Message\_2” 🡪 This will give the inline multi line command.**
  2. **Command with add and commit:**
     + **git commit -am “Commit\_Message”**
  3. **Amending a commit:**
     + **git commit –amend 🡪 This command will open the editor in which you can edit the commit message and overwrite the existing commit with the new commit.**
  4. **Amending a commit without change in commit message:**
     + **git commit --amend --no-edit 🡪 This command will add the commit with the existing commit without change in the existing commit message.**
  5. **Amending the Time of commit:**
* **git commit --amend --date="day\_name month\_name date time YYYY -0400"**
* **git commit --amend --date="now"**

===================================================================================

1. **Git Origin Master:**

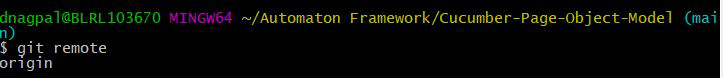
**What is git origin master?**

* Origin: This is the default name given to the remote repository from which you cloned your local repository. It acts as a shorthand reference for the URL of your remote repository.
* Master: This is the default name for the main branch in your repository. In many projects, this branch contains the stable and production-ready version of the code.

**Git – Origin**

* Origin is simply the name given to any remote repository available on GitHub.
* Whenever we need to push the changes to a remote repository, we use git push along with the remote repository "origin" and "master" branches. The term used is "**git push origin master**".
* To pull the changes from the remote repository to local, we use git pull along with remote repository "origin" and "master" branch. The term used is "**git pull origin master**".

The "**git remote**" command is used to show the remotes mapped to git remote repository



**Git remote -v**: Shows all the remote connections linked to a git repository. It shows fetch and push operations on a remote repository as below

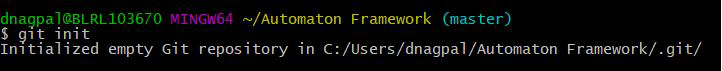
A black background with yellow text

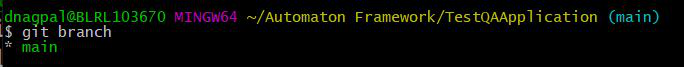
AI-generated content may be incorrect.

**Git – Master:**

**Master** is the name of a default branch in git terminology. Whenever a new repository is created in git, git gives the default name to a branch as 'Master'.

* When a new repository is initialized using "**git init**" command, git creates a single branch by default such as the "**Master**" branch.
* When multiple developers collaborate on a single feature/development work, developers create a pull request to merge the changes to master branch. After the review is done by the senior developer, changes are merged to the master branch.
* The Master branch is the most up-to-date branch and has production-ready code.





A screenshot of a computer

AI-generated content may be incorrect.

**Origin/Main in Git:**

Since Origin and Master are two different terminologies in Git but we might get confused when we see Origin/master in git context

* **Origin/master** **is a remote-tracking branch**.
* This branch exists in our local and tracks the remote repository 'origin' and branch 'master'.
* Branch in format "**remote-name/remote-branch-name**" is a remote-tracking branch

Since origin/master is a branch. Below is the process to merge the origin/master to master branch on remote origin

**Step 1:** Fetch the remote branch 'master' from remote 'origin'. Master branch would be fetched to local and local copy would be called as **origin/master.**

**git fetch origin master**

**Step 2:** Then merge the 'origin/master' to 'master'

**git merge origin/master**

**Step 3:**Finally, now push the changes from remote branch 'master' to remote 'origin'

**git push origin master**

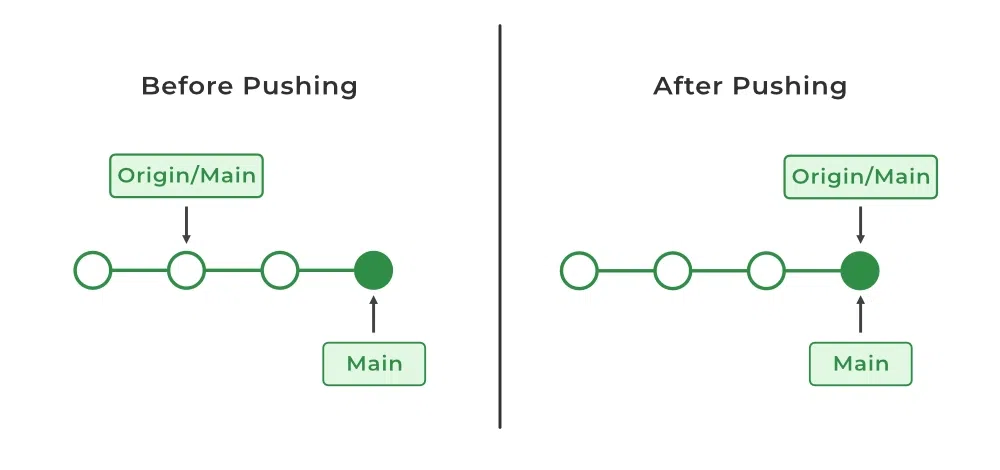
**Best Practices for Using git origin master**

1. Keep master Stable: Ensure that the master branch always contains stable and production-ready code.
2. Regular Pulls: Frequently pull changes from origin master to keep your local repository up-to-date.
3. Use Branches: Work on features and fixes in separate branches, and merge them into master only when they’re ready.

===================================================================================

1. **Git Push:**

Git push allows us to transfer files from the local repository to remote repository hosting services like GitHub, [GitLab](https://www.geeksforgeeks.org/installation-guide/how-to-download-and-install-git-lab/), etc. Other developers who want to work on the files can access them after being uploaded to a remote repository.



In the above diagram, we can observe If our local main branch is way back when compared to the central main repository after that git push origin main will publish the changes. git push is essential as same as the git merge.

git push <remote> <branch>

1. The <remote>  option refers to the remote repository to which you want to push your files it will refer to its alias name where the name is mapped with the remote repository URL
2. The <branch> option represents the branch of the GitHub repository which you want to push.

**Git Push Force To Central Repository:**

Force pushing is not a good practice it involves the risk of overwriting the existing repositories with the currently pushing repository. It will cause data loss and raise conflicts with other developers' work. Only in specific circumstances, such as when you need to delete or rewrite a string of commits that have already been sent to the remote repository, should you have to do a git force push. Here are the steps for force pushing:

**Step 1:**Commit changes.

**Step  2:**Fetch and Rebase.

Fetch the remote repository before force pushing by this we can ensure that the latest version of the repository is available, After fetching is completed rebase the local changes on top of the remote changes.

Fetch the repository.

git fetch <Repository URL or Alias name of Repository>

Rebase the changes.

git rebase <Alias name and Branch>

**Step 3:**git force push changes.

git push --force <Alias name od repository> <branch name>

**Note:** Instead of using **"--force"**you can also use the short form **"-f "**

**Git Push And Syncing:**

**Step 1:**Commit to the local repository.

**Step 2:**Push changes to the remote repository.

**Git push origin main**

**Step 3:**Pull changes from the remote repository.

**git pull**

**Step 4:**Resolve conflicts.

Git will remind you to resolve any conflicts between your modifications and those made by other collaborators if there are any. To fix the conflicts, you can manually change the conflicting files or utilize a merge tool.

**Step 5:**Re-push modifications to the remote repository.

You can push your modifications back to the remote repository after resolving the conflicts. By using the following command.

**git push**

**Pushing To Bare Repositories:**

The only one which doesn't consist of any particular working directory. The bare repository can be used by developers as a central repository in which they can push their changes and pull the updates. The bare repository only contains metadata and git objects.

**Step 1:**Create a bare repository.

git init --bare <directory>

**Step 2:**Clone the repository.

git clone

**Step 3:**Make changes and Commit the changes.

git commit -m "Commit message"

**Step 4:**Push the changes.

git push <Alias name of remote repository> <Branch name>

**Amended Force Push:**

Amended force push is a combination of commit and force push it will update the last commit and it will force push it to the remote repository.

**Step 1:** Commit changes.

**git commit --amend -m "New commit message"**

**Step 2: Force-push the amended commit.**

**git push --force-with-lease <Alias name> <Branch name>**

**using  "--force-with-lease"  will force push only if the remote branch's current commit matches the expected commit. It will prevent accidental overwriting of others' changes.**

**Git Push Jenkins:**

Once you push your code to a remote repository Jenkins will automatically get triggered. This will happen because of the configuration of any feature in GitHub, like periodically build, poll scm, and webhooks. Here are some sample steps to git push Jenkins.

**Step 1:** First commit the code to the local repository and push the code into the remote repository by using the following command.

git push <Alias name of remote

repository URL> and <branch name>

**Step 2:** Configure Jenkins in Git Hub by using WebHooks or any other feature. Whenever a new commit will happens in GitHub Jenkins build will automatically get triggered. To know more about how to integrate Jenkins with git you can refer to  [Jenkins and git integration.](https://www.geeksforgeeks.org/git/jenkins-and-git-integration-using-ssh-key/)

===================================================================================

1. **Git Status:**

The git status command is used to display the state of the working directory and the staging area. It shows which changes have been staged, which have not, and which files are not being tracked by Git. This command is important for understanding the current status of your project and for making informed decisions about your next steps.

**Command:**

**git status**

**==================================================================================**

1. **Git Remote:**

When you run git remote -v in your project, you’ll often see a remote named origin. This is Git’s default name for the remote repository URL you are connected to.

Think of it like a shortcut for the remote location where your code is stored (e.g., on GitHub or GitLab). It works like a key-value pair, where:

* origin is the key (the name),
* and the remote URL is the value (the actual address).

You will see origin used in many Git commands and messages, such as:

git push origin main

This tells Git to push your local main branch to the remote repository named origin.

**Managing Git Remote Repositories:**

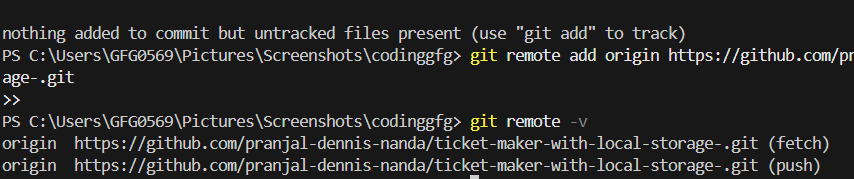
Git remote commands help you connect, manage, and interact with remote repositories like GitHub or GitLab.

**1. Add a Remote Repository**

To establish a connection between your local repository and a remote repository, use the following command:

**git remote add <name> <URL>**

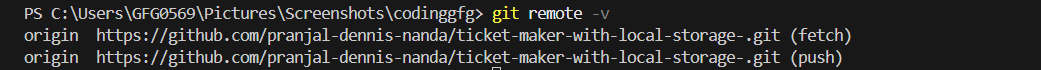
**usually, git remote add origin <URL>**



**2. View Remote Repositories**

To display a list of linked remote repositories along with their URLs for the local system user attempting to connect to a GitHub server.

**git remote -v**



**3. Rename a Remote**

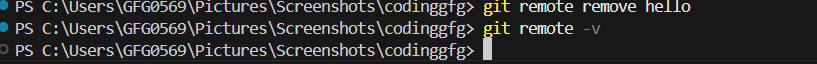
To change the name of an existing remote that you are currently connected to on the server.

**git remote rename <oldname> <newname>**

**4. Remove a Remote**

To unlink a remote repository from your local repo use the following command. If the repository name does not exist you can write origin instead of the repository name.

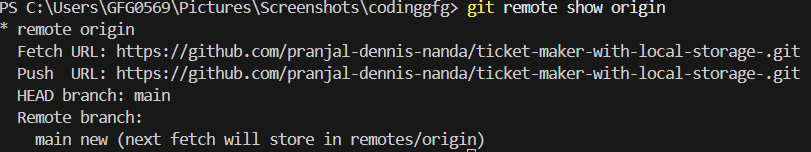
**git remote remove <repositoryname>**



**5. Show Remote Details**

To display information about a specific remote, including branches and fetch/push URLs use the following command.

**git remote show <name>**



**Important Git Remote Commands:**

**A black screen with white text

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**==================================================================================**

1. **Git Pull:**

Git pull is a command which is used to fetch and integrate the changes which are present in the remote repository to the local repository.

Git pull is basically combination of git merge and git fetch which is used to update the local branch with the changes available in the remote repository branch. By which each and every developer will have the updated code with in there local machines they can work with the newly updated code.

**Command:**

***git pull [remote-name] [branch-name]***

**When To Use Git Pull:**

Git pull is mainly used to the update the local repository of yours with the help of remote repository. This is done mainly to keep the local repository updated with remote repository or keep the sync of local repository with the remote repository following are the some of common scenarios where you would use **"git pull".**

* Collaborative Development.
* Staying Up-to-Date.
* Updating Feature Branches.
* Fetching and Merging Changes.
* Pulling Rebase Changes.

**How Git Pull Works?**

Git Pull is a command used to update the local version of a repository from a remote repository. It is a mixture of two other commands:

* git fetch
* git merge

**Git Pull and Syncing**

The main purpose of git pull command is to make the synchronization between the remote repository and the local repository. The term syncing refer to the updating the changes from remote repository.

Git pull command plays an major role in updating the local repository with the help of remote repository which will help in maintains the sync between the local and remote repository.

* **Git Fetch:**Fetches the latest changes form the remote repository to the local branch without integrating them.
* **Git merge:**It will integrates the coinages fetched from the remote repository.

**Git Pull Vs Git Fetech:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**==================================================================================**

1. **Git Clone:**

The git clone command creates a copy of an existing Git repository. This repository can be hosted on platforms like GitHub, GitLab, or Bitbucket, or it can reside on a local or remote server. Cloning a repository involves copying all the repository’s data to your local machine, including the entire history and all branches.

**Syntax:**

git clone <repository-link>

**Cloning a Specific Branch**

By default, git clone clones the main branch (often named main or master) of a repository. However, if you want to clone a specific branch of the repository, you can use the -b option.

**Syntax:**

git clone -b <branch-name> <repository-url>

This command clones the specified branch into your local repository instead of the default branch.

**Repo-to-Repo Collaboration**

Cloning repositories is a key part of collaborating on projects with other developers. When you clone a repository, you create a local copy where you can make changes. Afterward, you can push your changes back to the remote repository for others to review and merge.

To clone the remote repository use the following command.

git clone <remote\_repository\_url>

This creates a working copy of the project, allowing you to modify it, and then push the updates back to the repository.

**Git Clone -Branch:**

The git clone --branch command is used to clone a specific branch from a remote Git repository to your local machine. By default, when you clone a repository, it clones the main branch (usually called main or master). However, if you want to clone a different branch, you can specify the branch name in the command.

git clone --branch <branch\_name> <remote\_repository\_url>

<branch\_name>: This is the name of the branch you want to clone from the remote repository.

<repository\_url>: This is the URL of the remote repository (either HTTPS or SSH).

For example, if you want to clone the feature-xyz branch from a repository, you would run the following command:

git clone --branch feature-xyz https://github.com/username/repository

**Git Clone - Mirror vs Git Clone – Bare:**

Git supports two specialized cloning methods: mirror cloning and bare cloning.

1. Git Clone - Mirror:

A mirror clone creates a complete, exact copy of a remote repository, including all refs, branches, and configuration.

Use Case: Typically used for creating full backups of repositories.

git clone --mirror <repository-url>

2. Git Clone - Bare:

A bare clone only contains the repository’s version control data (e.g., .git folder) and doesn't include the working directory (the actual files). It’s commonly used for shared repositories.

Use Case: Often used for creating a central repository for collaboration or setting up a server repository.

git clone --bare <repository-url>

**Git Clone -Mirror vs. Git Clone -Bare:**

**A black and white text on a black background

AI-generated content may be incorrect.**

**Cloning a Repository into a Specific Local Folder:**

You can specify the local directory where you want the repository to be cloned. Use the following command to clone a repository into a specific folder:

**Syntax**

git clone <repository-url> <local-folder-name>

Replace the URL with the repository URL you want to clone, and specify the name of the folder where you want to copy the repository in place of <local\_folder>.

For example, to clone a repository into a folder named my-project, use:

git clone https://github.com/username/repository my-project

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Section 8: Branching & Merging:**

* 1. **Push Git branch to Remote?**

Step 1: Git init

Step 2: git status

Step 3: After changes --- git add

step 4: git commit

step 5: Now the changes are in local repository. Want to push that changes to remote repository. For that we want to connect local and remote repository by remote command.

Step 6: Copy the GitHub repo link in which you want to push this changes.

Step 7: git remote add origin <Remote Repository URL>

Step 8: git remote -v – used to check whether the origin is connected successfully or not.

Step 9: Finaly push the changes to the remote repository --- git push origin main.

**==================================================================================**

* 1. **Delete a Git Branch Locally and Remotely:**

**Delete a Git Branch Locally:**

Git won't allow you to **delete a Git branch** you are currently working on. So you must make sure to **checkout to a**

**Syntax**

git checkout <branch-name>

Now in order to **delete the test branch locally**, we use the command :

**Syntax**

git branch -d <branch-name>

**Note: The -d option will delete the branch only if it has already been pushed and merged with the remote branch. If you want to forcefully delete a branch you will have to use the -D option instead. The -D flag is synonymous with --delete --force. This will forcefully delete the branch even if it hasn't been pushed or merged with the remote. the full command is:**

Syntax:

git branch -D <branch-name>

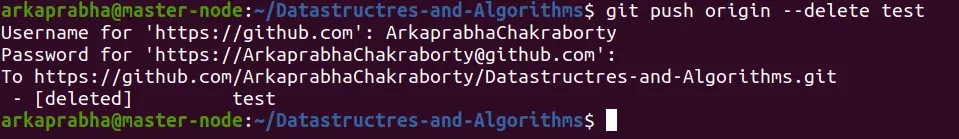
With this, we can successfully delete a local branch.

**Delete a Git Branch Remotely:**

You can’t use the git branch command to delete a remote branch. Instead, you have to use the git push command with the --delete flag, followed by the name of the branch that you want to delete. You also need to specify the remote name (origin in this case) after "git push". The command is as follows:

**Syntax**

git push <remote-name> --delete <branch-name>



This command will delete the branch remotely. You can also use the shorthand:

**Syntax**

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

**A common error faced by many in this step is:**

***error:*** *unable to push to unqualified destination: remoteBranchName The destination refspec neither matches an existing ref on the remote nor begins with refs/, and we are unable to guess a prefix based on the source ref. error: failed to push some refs to 'git@repository\_name'*

This means that someone has already deleted the branch that you want to delete. If this happens you can use the following command to synchronize your branch list in the local environment:

**Syntax**

git fetch -p

The -p flag here means "prune". After fetching the branches which no longer exist remotely will be deleted in your local working environment.

**Local Deletion vs Remote Deletion:**

**A screenshot of a computer program

AI-generated content may be incorrect.**

**==================================================================================**

* 1. **Git Merge:**

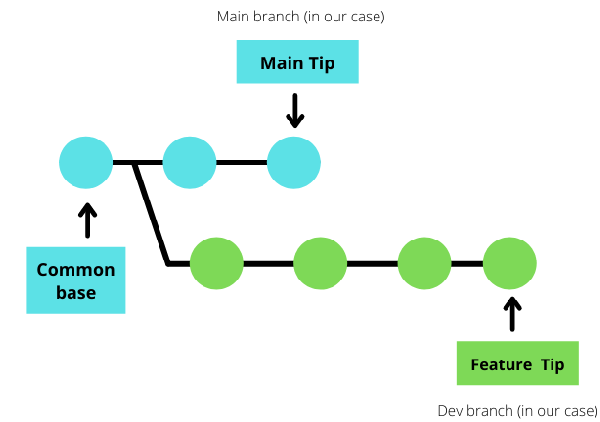
Git Merge is a command used to combine the changes from two branches into one. It integrates work from different branches into a single unified history without losing progress. For example, you can merge a feature branch into the main branch to include all recent updates.

**Preserves History:** Keeps the commit history of both branches.

* **Automatic and Manual:** Automatically merges unless there are conflicts.
* **Fast-Forward Merge:** Moves the branch pointer forward if no diverging changes exist.
* **Merge Commit:** Creates a special commit to combine histories.
* **No Deletion:** Branches remain intact after merging.
* **Used for Integration:** Commonly integrates feature branches into main branches.

**Working of Git Merge:**

**Before Merge:**



This image shows the state of the repository before the merge takes place.

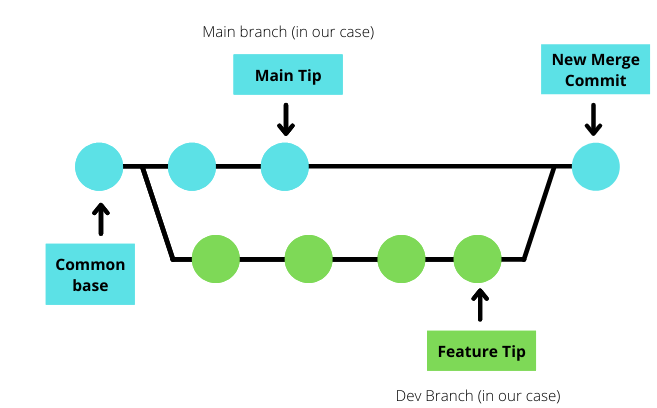
**Key Elements**

* **Common Base**: The commit from which both the main and feature (dev) branches originated.
* **Main Branch (in blue)**: This is your production or main branch where stable code lives.
* **Feature/Dev Branch (in green)**: This branch is where new development work has occurred.
* **Main Tip**: The latest commit on the main branch.
* **Feature Tip**: The latest commit on the feature branch.

**Interpretation**

* Development has happened in parallel on both the main and feature branches.
* These branches diverged from a common ancestor (the **Common Base**).
* You are now preparing to merge the feature branch back into the main branch.

**After Merge:**



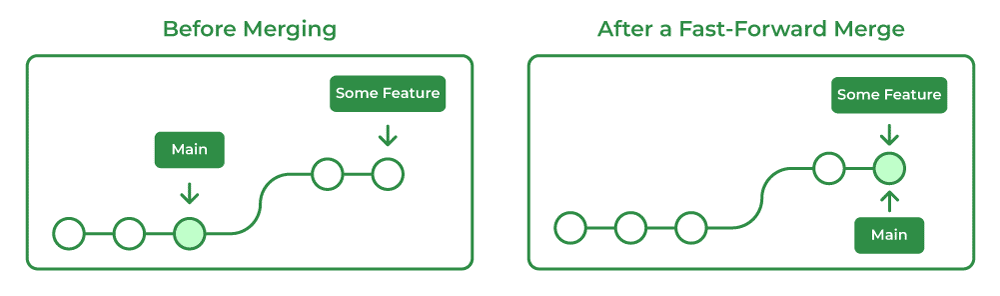
This image shows what happens after executing a git merge.

During the merge, Git compares the latest commit of the main branch (Main Tip), the feature branch (Feature Tip), and their shared ancestor (Common Base). If there are no conflicts, Git automatically creates a new merge commit combining both histories.

**Types of Merging in Git:**

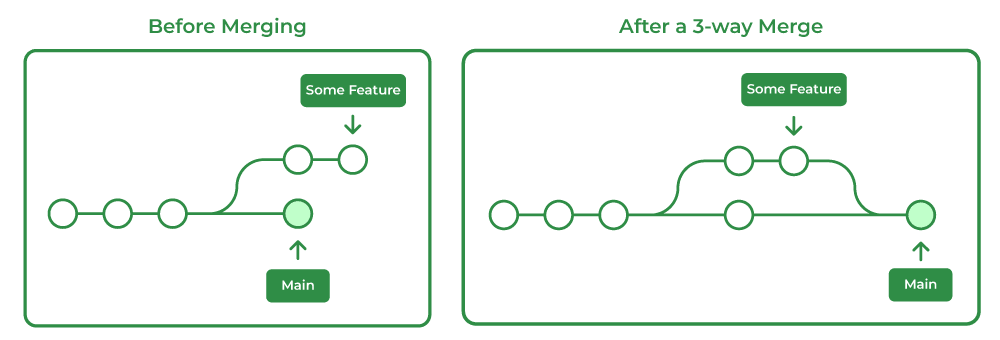
**1. Fast-forward merging:**

* This occurs when the target branch (e.g., main) is directly ahead of the feature branch (e.g., dev).
* Instead of creating a merge commit, Git simply moves the current branch’s tip to the target branch’s tip.
* [Fast-forward merging](https://www.geeksforgeeks.org/git/what-is-git-fast-forwarding/) is only possible when branches haven't diverged



**2. Three-way merging:**

* This type occurs when the base branch has changed since the branch was first created.
* Git generates a new merge commit by comparing changes in both branches with the base branch.



***Note :*** *Git also supports other types of merging like recursive and octopus merging. With the help of a single merge commit "****octopus merging"*** *can merge multiple branches at once.* ***"Recursive merging"*** *is similar to three-way merging but it can handle more complex merge operations than the three-way merging.*

**Steps to Perform Git Merge:**

**Step 1: Create a New Branch**

**git branch <new-branch-name>**

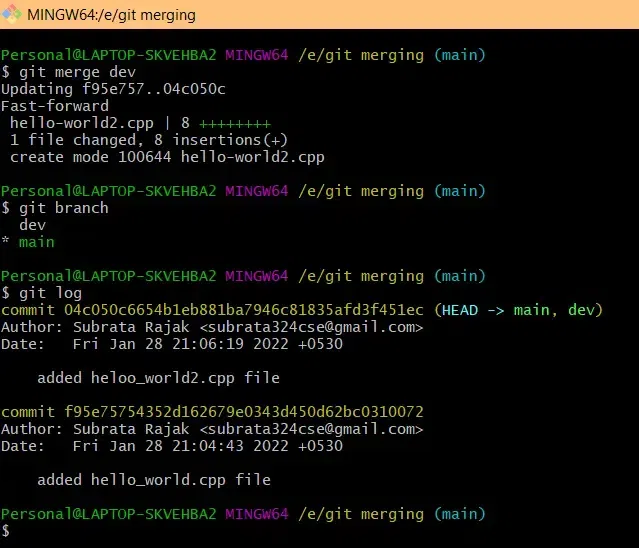
**Step 2: Pull the Latest Changes**

git checkout <target-branch>   
git pull origin <target-branch>   
  
git checkout <feature-branch>   
git pull origin <feature-branch>

**Step 3: Merge the Branch**

If any conflicts arise, Git will notify you. Resolve them manually before proceeding.

git checkout <target-branch>   
git merge <feature-branch>



**Step 4: Test the Merged Code**

# Run tests or manually test your application

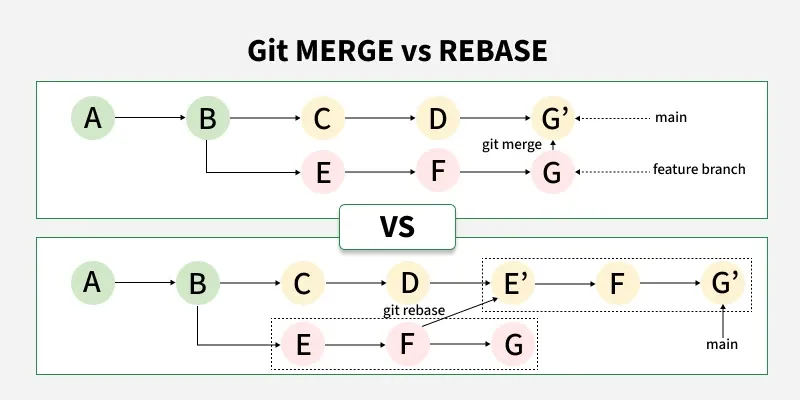
**Step 5: Commit the Merged Code**

git commit -m "Merge branch 'dev' into main"

**Step 6: Push the Merged Branch**

git push origin main

**Git Merge V/s Rebase:**

****

**A black screen with white text

AI-generated content may be incorrect.**

**=================================================================================**

* 1. **Git Merge and Merge Conflict:**

**Merging Changes:**

**Step1 : Lets check the log of master branch and dev branch.**

**A screenshot of a computer

AI-generated content may be incorrect.**

A screen shot of a computer code

AI-generated content may be incorrect.

So now as we can see from the commit present in the dev branch a file k.txt is created which means a file by the name of k.txt is created in the dev branch so now if we want that change to be reflected in the master branch for that we can use the command **git merge name\_of\_the\_branch**.

A screenshot of a computer

AI-generated content may be incorrect.

So now we can see that after using the merge command the changes are getting reflected in the master branch and we can see that a new commit is showing up in the git logs of the master branch which is having a commit message a new file k.txt is created.

A screenshot of a computer

AI-generated content may be incorrect.

**Aborting a Merge:**

A screenshot of a computer code

AI-generated content may be incorrect.

**Merge with a Commit:**

On using the simple git merge command it resolves the merge as fast-forward and only updates the branch-pointer but if you want to create a merge commit for that we have to pass --no-ff as a parameter in the below command.

**git merge branch\_name --no-ff -m commit\_message**

A screenshot of a computer program

AI-generated content may be incorrect.

**Here we can say that it is only updating the branch pointer by showing head-on master.**

A black and yellow numbers

AI-generated content may be incorrect.

A screenshot of a computer code

AI-generated content may be incorrect.

**Merge Conflicts:**

Merge Conflicts are the conflicts that occur when a developer is editing a file in a particular branch and the other developer is also editing that same file or when developer A edits some line of code and that same line of code is being edited by another developer B that leads to conflicts while merging. Now let's see how to resolve these merge conflicts.

A screenshot of a computer program

AI-generated content may be incorrect.

A screenshot of a computer error

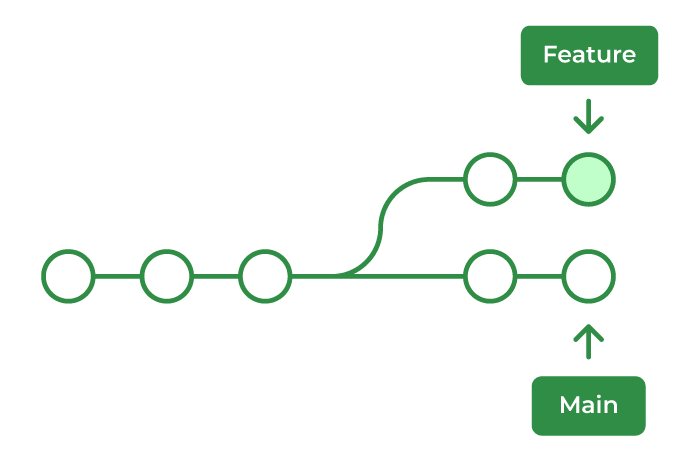
AI-generated content may be incorrect.

A screenshot of a computer program

AI-generated content may be incorrect.

* 1. **Git Rebase:**

**Git Rebase** is a Git command used to integrate changes from one branch into another by **moving your commits to the latest point (tip) of the target branch**. Unlike Git merge, which creates a new merge commit and retains the commit history as a branching tree, rebase rewrites your commit history to make it look like your work started from the most recent updates on the target branch.



**Syntax:**

git checkout <feature-branch>  
git rebase <base-branch>

In this syntax:

* **<feature-branch>** is the branch with the changes you want to rebase.
* **<base-branch>** is the branch you want to rebase your changes onto, typically main or master.

**When to Use Git Rebase**

Imagine you are working on a feature (like “adding a login button”) while your teammate adds something else (like “fixing a bug”) to the main project. Later, you want to **update your work** so it includes all the latest changes from the main project before you finish.

Think of it like this:

* You are writing a paragraph for a report.
* While you are writing, someone else updates the report with new lines.
* git rebase lets you **move your paragraph to the end** of the latest version of the report, so it looks like you wrote it after those new lines.

Example

1. You and your teammate start with this:

A → B (main branch)

2. You create your feature branch and make some changes:

A → B (main)

\

C → D (feature)

3. Meanwhile, your teammate updates the main branch:

A → B → E → F (main)

\

C → D (your branch is outdated now)

4. Now you run:

git checkout feature

git rebase main

5. Your feature branch is moved and now looks like:

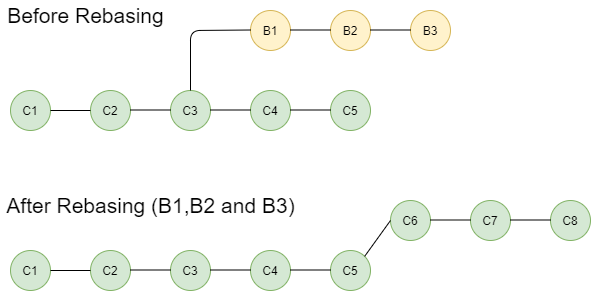
A → B → E → F → C' → D' (rebased feature branch)

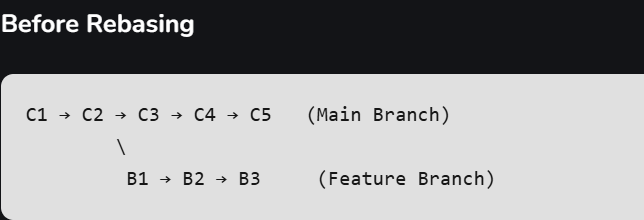
Your work (C and D) is now placed after the latest work (E and F). The code history is clean and easy to follow.

**What Happens During Rebase?**

* Git temporarily removes your changes.
* Then it adds the latest work from the branch you rebased onto.
* Then it reapplies your changes on top of that.
* It creates new commit versions of your changes (called C′ and D′).

**Visual Working of Git Rebase:**



****

* C1 to C5 are commits on the main branch (green).
* B1 to B3 are commits on the feature branch (yellow), created from C3 .
* At this point, the feature branch is outdated, because the main branch has moved ahead ( C4 and C5 were added after C3).

A black and white text

AI-generated content may be incorrect.

* The git rebase command takes commits B1 → B2 → B3 and reapplies them on top of C5, the latest commit on main.
* These become new commits: C6 → C7 → C8 (green versions of B1–B3).
* The feature branch now appears as if it was based on the latest version of the main branch.

**What Git Rebase did behind the Scenes**

1. **Found the base**: Git looked for the common ancestor (C3) between main and feature.
2. **Temporarily removed** commits B1, B2, and B3.
3. **Moved the base** of the feature branch from C3 to C5.
4. **Re-applied** the changes from B1-B3 on top of C5 as new commitsC6, C7, C8.

**Types of Git Rebase**

**1. Interactive Rebase (git rebase -i)**

* This allows you to edit, squash, reorder or delete commits in your branch. It gives you full control over the commit history, making it useful for cleaning up commit messages or combining multiple commits into one.
* It combines commits to merge them into a single commit.
* Git rebase reorders commits to reflect a more logical flow.
* Editing commit messages before pushing them to a remote repository.

**2. Non-Interactive Rebase (Standard Rebase)**

* This is the regular rebase command (git rebase <branch>), which simply applies your commits onto the target branch without allowing for manual intervention. It is ideal for straightforward rebasing where you don’t need to modify or review individual commits.
* Updating your feature branch with the latest changes from the main branch.

**3. Auto-merge Rebase**

* When rebasing, Git will automatically merge changes if there are no conflicts between the commits being rebased and the target branch. If conflicts are detected, Git will stop and require manual resolution.
* Rebasing feature branches frequently to stay up-to-date with the main branch.

**Git Standard vs Git Interactive Rebase:**

**Standard Rebase**

Standard rebasing applies all commits from the current branch to the head of the target branch without manual intervention. The following command performs a standard rebase:

git rebase master branch\_x

* This is equivalent to:

git rebase master

* Here, Git automatically takes the commits from your current branch and applies them to the specified branch (master).

**Interactive Rebase**

Interactive rebasing allows you to edit, reorder, squash, or drop commits before applying them to the new branch. This gives you full control over the branch's commit history.

git checkout branch\_x  
git rebase -i master

* This command lists all commits that are about to be moved and prompts you to edit or rearrange them based on your choices. It helps maintain a clean and structured project history.

**Common Git Rebase Commands**

The following are the most used Git rebase commands:

1. **git rebase master:**Applies the changes of the current branch onto the master branch.
2. **git rebase --continue:**Continues the rebase process after resolving conflicts
3. **git rebase --abort:**Cancels the ongoing rebase and restores the branch to its original state
4. **git rebase --skip:**Skips a commit if conflicts arise, though this is not recommended as it could damage your codebase.
5. **git rebase -I HEAD~3:**Starts an interactive rebase on the last three commits, allowing edits like commit message changes.

**Configuration Options In Git Rebase**

Customize your rebase process with options like:

* **--interactive (-i):** Enables interactive rebasing for editing commits.
* **--onto <newbase>:** Specifies a new base commit for rebase.
* **--no-verify:**Skips pre-commit hooks during rebase.
* **--auto-squash:** Automatically squashes commits marked with fixup or squash

**Best Practices for Using Git Rebase**

The following are the best practices that you should follow while using git rebase:

* **Don’t Rebase Public History:** Don’t rebase branches that are already shared with others. Rebase changes the history, which can mess things up for your teammates.
* **Use Rebase to Clean Up Commits**: Before merging your feature branch into the master branch, use rebase to tidy up your commits, combining small ones into one clear commit.
* **Rebase Regularly**: Keep your feature branch up-to-date by regularly rebasing it on the latest master or main branch. This helps avoid conflicts later.
* **Resolve Conflicts Quickly:** If conflicts happen while rebasing, fix them as soon as possible. Don’t let them pile up.

==================================================================================

1. **Git – Squash:**

**Git squash** is the process of taking a series of commits and merge them into a single commit. This technique is especially useful when a feature branch has numerous small, incremental commits that clutter the commit history. By squashing these commits, you can present a more polished and cohesive history to your collaborators.

**Git Squash-Merge Command Line**

By using the git squash-merge command you can combine multiple commits into a single commit. After using the git squash command your[repository](https://www.geeksforgeeks.org/git/what-is-a-git-repository/) will be clean and organized manner.

git merge --squash feature-branch

To merge the changes in the feature branch to your current branch we can use the above command.

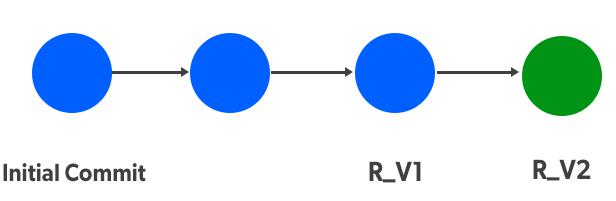
**When To Squash Commits?**

The GFG\_VIDEO project, an open-source video conferencing tool, began with an initial set of basic features like **peer-to-peer video calling** and **messaging**, which were consolidated into the first release, **R\_V1** (marked by the green tag). This initial development involved **three commits**.

Following the R\_V1 release, the development team worked on new features, including **creating groups** and **group video calls**, as well as **fixing minor bugs** (such as call drops) found in R\_V1. This work leading up to the second release, **R\_V2**, has accumulated **three more commits**.

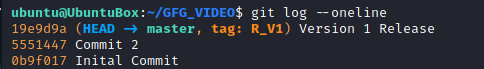
The current commit history, with three separate commits between R\_V1 and the planned R\_V2 tag, is considered **untidy** and **difficult to follow** for the R\_V2 feature set.

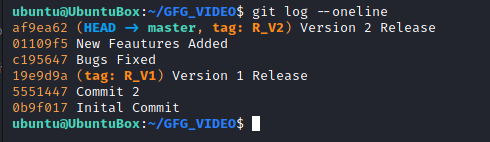




**How To Squash Commits?**

**The below image shows we have 3 commits:**Initial commit, Commit 2, and Version 1 Release. We’ve successfully released the 1st version (R\_V1) of the GFG\_VIDEO tool. After R\_V1 new features are added and minor bugs are fixed from the previous release and the tool is ready for its 2nd release R\_V2.





The above image of the GFG\_VIDEO log is after the 2nd version release. It can be observed after the Version 1 Release (tag: R\_V1) there are 3 commits for the Version 2 Release. This kind of looks untidy, to make it simpler to read we can do a squash operation.

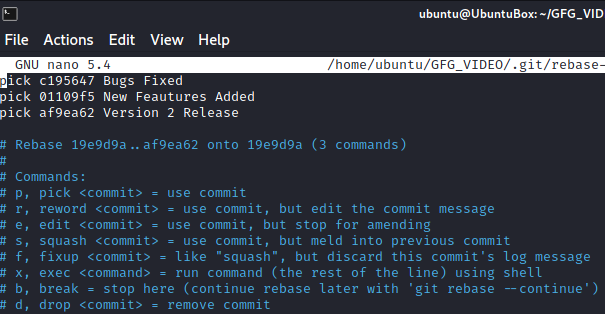
Let's perform squash now

git rebase -i HEAD~3

**Note:**Rebase is an action to rewrite commits and their history "**-i"** is to enter into an interactive mode of rebase **HEAD~n** states to perform our operation on n commits from **HEAD.**

**Squashing By Interactive Mode**

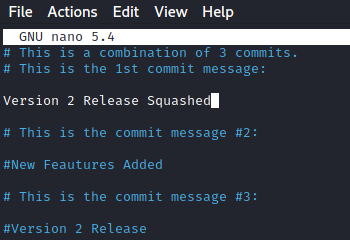
Upon entering the above command we'll get an interactive editor with all our selected commits which is where we'll be performing squash.



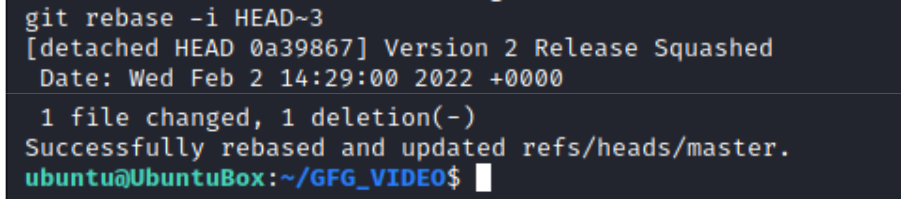
We can see we have selected 3 commits at the beginning of the interactive editor, below that we can see the commands list such as pick, reword, edit, squash, etc. To squash  2nd and 3rd commit with 1st commit, so we'll change the first word from pick to squash. whichever commits we want to squash we have to change it to squash from pick.



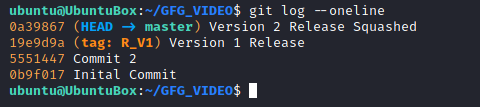
After changing commits from pick to squash save the file and close it, Immediately another editor will be opened where we have to enter the latest commit message. Enter the latest commit message and comment on the remaining old messages.



After adding the latest commit message save the file and exit the file. Now it shows rebasing is successful.



Now if we see our GFG\_VIDEO log we can observe our 3 commits after version 1 release are squashed into 1 commit.



A screenshot of a computer program

AI-generated content may be incorrect.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Section 9: File & Change Management:**

* 1. **Git Diff:**

git diff is a command-line tool in Git that shows the differences between various states of a repository. It helps developers see what changes have been made, whether they are between working directory and the staging area, between the staging area and the last commit, or between any two commits.

HEAD ---> Head will point the lastest commit we done.

HEAD^ ---> This means HEAD-1 commit.

**Git diff:**

A file say at.txt is modified here after doing a commit and here we can see that there is a difference in the file after a commit.

git difftool ----> Comparing between Wroking Area and Staging Area

A screen shot of a computer program

AI-generated content may be incorrect.

**Changes between two Commits:**

**Command:**

**git diff commit-id1 commit-id 2**

A screen shot of a computer

AI-generated content may be incorrect.

**Shows difference for Staged files:**

**Command:**

**git diff –staged (or) git diff –cached (or) git status  -v**

**git difftool --staged HEAD --> Comparing between Staging Area and Local Repository (Last Commit)**

A screen shot of a computer

AI-generated content may be incorrect.

**Using git diff for comparing branches:**

**Command:**

**git diff name\_of \_the\_branch1 name\_of\_the\_branch2**

A computer screen with text

AI-generated content may be incorrect.

**Showing Both staged and unstaged Changes:**

**Command:**

**git diff HEAD (or) git status -vv**

**git difftool HEAD ---> Comparing between Working Area and Local Repository (Last Commit)**

A screenshot of a computer

AI-generated content may be incorrect.

**Showing Differences for a Specific File or Directory:**

**Command:**

**git diff file\_name**

**git difftool -- README.md ---> Comparing single file at a time. Not multiple files.**

A screen shot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

**Showing differences between the current version and the last version:**

**Command:**

**git diff HEAD^ HEAD**

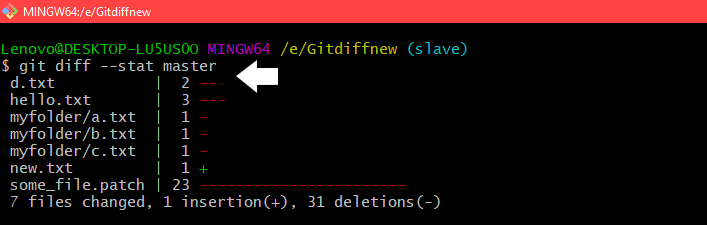
A screenshot of a computer

AI-generated content may be incorrect.

**To view the summary of changes:**

**Command:**

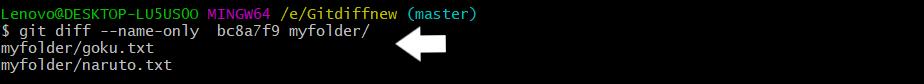
**git diff -stat commit\_id (or) git diff --stat branch**



**To view files that changed after a certain commit:**

**Command:**

**git diff --name-only commit-id**



**To view files that are different than a branch:**

**Command:**

**git diff --name-only branch\_name**

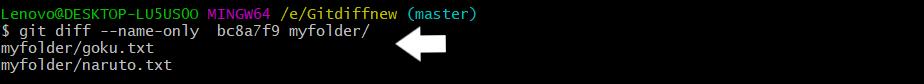
A black and red background with white text

AI-generated content may be incorrect.

**To view files in a folder that changed after a commit:**

**Command:**

**git diff --name-only commit-id folder-path**



**==================================================================================**

* 1. **Git Undo Commit:**

A commit is a snapshot of your project's files at a specific point in time. A unique SHA-1 hash identifies each commit. When you want to undo a commit, you are essentially changing the history of these images.

**Methods to Undo a Commit in Git**

**1. Using Git reset**

**Step 1: First check all your commits**

***#git log***

**Output:**commits are just examples or sample commits

commit 2: second commit   
commit 1: First commit

**Perception drawn are as follows:**

* *commit 2 is the commit you want to undo,* the *current head is here*
* commit1 is the first commit where you want to go after undo

**Step 2: To restore everything or undo all the changes we have to reset the commit.**

***#git reset --soft HEAD^***

***#git reset --hard HEAD^***

***Note:***

* *soft is used if you want to keep your changes*
* *hard is used if you don't want to keep your changes*

**Step 3***:*To check your commit is reset or not

***#git log***

**Output:**

commit 1: First commit   
*//undo the second commit, now head is at first or previous commit*

 One can clearly see last commit (i.e. second commit) is removed.

**2. Using Git revert:**

ow if we have already made your commit public then you will have to create a new commit which will "revert" the changes you made in your previous commit (current HEAD).

**Step 1: Revert your changes**

***#git revert HEAD***

We are now ready for your new commit in order to restore the file that we accidentally have remove with the below command as follows:

***#git commit -m***

**Step 2:**Now check your all commits to see the list of commits

***#git log***

**Output:**

commit 3: restoring the file that we accidentally remove   
commit 2: removing a file we don't need   
commit 1: Needed file

**Now we can revert your last commit.**

Also do note that we use the command below specified to undo the last commits in git where the head is a pointer pointing to the last commit in our branch

git reset HEAD~<no-of-commits>

* 1. **Using git checkout:**

The git checkout command can be used to create a new branch from a specific commit. This method is helpful when you want to keep the original branch unchanged.

Steps to create a new branch from a specific commit:

1. Identify the commit hash you want to create a branch from.

git log

2. Use the 'git checkout' command to create a new branch.

git checkout -b <new-branch-name> <commit-hash>

Example::

git checkout -b new-feature a1b2c3d

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Section 10: History & Debugging:**

* 1. **Git Log:**

It displays all the commits being made in that repository in multiple lines along with the commit id, author name, date and commit message. If you have multiple logs in git log so to exit from the git log command press q.

Command:

git log

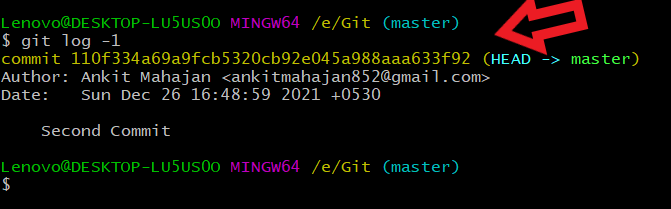
A screen shot of a computer

AI-generated content may be incorrect.

If we want to limit the log status by show only one commit or two commit we can do that by,

**Command:**

git log -n (n – Number commits to show)



**Key Terminologies:**

* Log: It is a record of all the commits done in the repository.
* Commit: A commit is a snapshot of the git repository at one point in time.
* Commit id: It is a 40-character hexadecimal value and it's a unique identifier that git generates every time we make a commit to our repository.

**Prettier log:**

To see the logs in a very decorated, neat, and clean manner we use the following command.

git log --decorate --oneline --graph

A screen shot of a computer

AI-generated content may be incorrect.

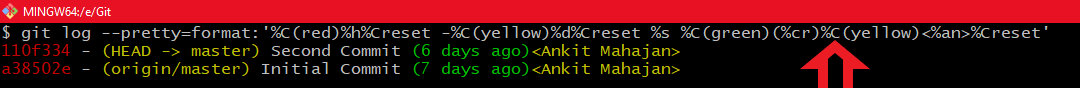
**Colorize logs:**

To see the logs in a colorized fashion or to customize the  log output format we use the below command

git log --graph ---pretty=format: '%C(red)%h%Creset -%C(yellow)%d%Creset %s %C(green)(%cr)%C(yellow)<%an>%Creset'

**A screenshot of a black screen

AI-generated content may be incorrect.**



**Oneline log**:

This will display the log outputs in oneline. But it only shows first some parts of the commit id and the commit message.

Command:

git log –oneline

A screen shot of a computer program

AI-generated content may be incorrect.

**LogSearch:**

**git shortlog:** Basically it displays the log output format in the following syntax:

Committer1(number of commits)

* Commit Message 1
* Commit Message  2
* Commit Message n

where n represents the last commit message number committed by the author

Committer2(number of commits)

* Commit Message 1
* Commit Message  2
* Commit Message n

where n represents the last commit message number committed by the author

A screen shot of a computer

AI-generated content may be incorrect.

**A screenshot of a black and white screen

AI-generated content may be incorrect.**

A screen shot of a computer program

AI-generated content may be incorrect.

**Filter logs:**

In this concept, we will see various parameters used in the command to filter logs.

* For some days: git log --after 'mention the days ago'
* for e.g:  git log --after '4 days ago'
* For specific dates: git log --after year\_number-month\_number-date\_number
* For Author name: git log --author="Author\_name"

A screen shot of a computer program

AI-generated content may be incorrect.

A computer screen with text and arrows

AI-generated content may be incorrect.

A screen shot of a computer

AI-generated content may be incorrect.

**Committed files:**

If you want to see the files in logs that are being committed by commits. we use the command

**git log --stat**

A screenshot of a computer screen

AI-generated content may be incorrect.

**Contents of a Commit:**

ow if you want to view the contents of a commit you can use a simple command

**git show commit-id**

A screen shot of a computer program

AI-generated content may be incorrect.

===================================================================================

* 1. **Git Ref and Reflogs:**

**Git Ref:**

Refs(short for references) are pointers to specific commits in a Git repository. They serve as labels for specific points in your project's history. Common refs include branches, tags, and the HEAD.

**Types of Refs**

* **Branches**: A branch is a movable pointer to the latest [commit](https://www.geeksforgeeks.org/git/what-is-git-commit/) in development. When you create a new branch, Git creates a new pointer that moves as you make new commits.
* **Tags**: Tags are static pointers, often used to mark release point. Unlike branches, tags do not move.
* **HEAD**: A special ref that points to the currently checked-out commit (usually a branch head).

**Viewing Refs in a Git Repository:**

Refs are stored as a normal file text in .git/refs directory. To explore refs in one of the project's repositories navigate to .git/refs or type the following command in Git bash in the root directory of your project.

A close-up of a computer code

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

All local branches of repository are defined in **refs/heads/** directory. Each file name matches names of the corresponding branch, and inside file, you will find a commit hash. This commit hash is the location of the tip of the branch.

**Git Reflogs:**

A **reflog** (reference log) records when the tips of branches and other refs were updated in your local repository. Every time you:

* switch branches,
* commit,
* stash,
* reset, or
* rebase,

Git logs the movement of that reference into the reflog.

A screen shot of a computer

AI-generated content may be incorrect.

**Common Git Reflog Commands**

Git reflog commands come with various subcommands for advanced usage:

A screenshot of a computer program

AI-generated content may be incorrect.

Words in square brackets such as "show", "log-options" are qualifiers, or we can say arguments to **git reflog**command.

Whether you are fixing mistakes or just understanding what happened, mastering refs and reflogs can make you much more confident using Git.

* **Git Reflog Show:**The "show" subcommand (which is also default, in absence of any subcommand) shows logs of the reference provided in command(or HEAD, by default). The reflog covers all recent actions, and in addition HEAD reflog records branch switching.
* **Git Reflog Expire:**The "expire" subcommand prunes older reflog entries. Entries older than "**expire**" time, or entries older than "**expire-unreachable"** time and not  reachable from the current tip, are removed from the reflog. This is typically not used directly by end users
* **Git Reflog Delete:**The "delete" subcommand deletes single entries from the reflog. Its argument must be an exact entry (e.g. "**git reflog delete master@{2}**"). This subcommand is also typically not used directly by end users.
* **Git Reflog Exists:**The "exists" subcommand checks whether a ref has a reflog. It exits with zero status if the reflog exists, and non-zero status if it does not.

===================================================================================

* 1. **Git Blame:**

Git blame is a Git command that annotates each line in a file with the following information:

* The last committhat modified the line.
* The author of the commit.
* The timestamp of the change.
* The line number and code content (optionally).

**Syntax:**

git blame [options] <file-name>

**Step-by-Step Guide to use git-blame:**

**Step 1: Move to an empty folder and initialize it with an empty git repository.**

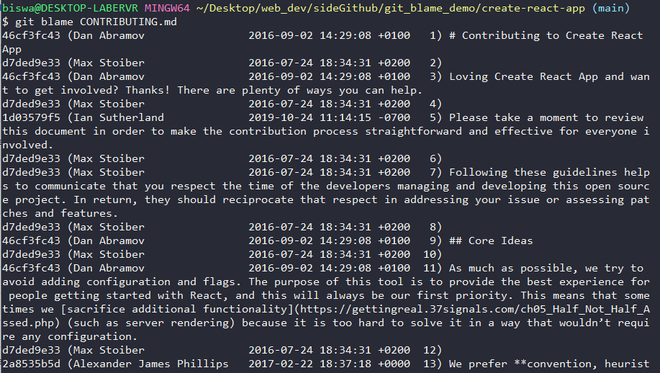
**Step 2: Clone the repository for which you want the information.**

**Step 3: To get the author name and commit information, execute the git blame command as shown below. Replace the <file-name> option with the relative path of the file from the current directory for which you want details:**

**git blame <file-name>**

**Step 4: Here I want details for the file CONTRIBUTING.md which is present in the same directory I am currently at. So, I will execute the below command:**

**git blame CONTRIBUTING.md**



***Tip: Observe that the last column of output displays the line number along with the code on that line.***

**Timestamp:It is the date and time when the commit was made expressed in standard notation. We can get the raw timestamp using the -t option as shown below:**

**git blame -t CONTRIBUTING.md**

A screenshot of a computer program

AI-generated content may be incorrect.

**Viewing the commit details:**

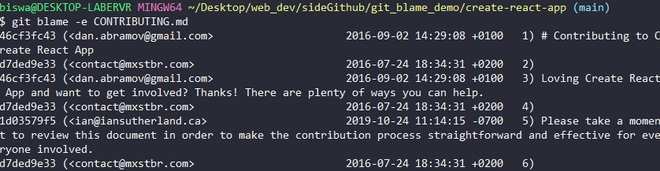
**Command:**

**git log -p <commit-id>**

**To view the author email id:**

**Command:**

**git blame -e CONTRIBUTING.md**



**==================================================================================**

* 1. **Recovering Lost Commits in Git:**

[Git](https://www.geeksforgeeks.org/git/introduction-to-github/) is a very powerful tool, and with great power comes great responsibilities. If not used and handles properly, It might cause you to lose your commits. There might be situations when you may find all of your work missing at once. If you have regularly committed your work, there is a way to recover these **lost commits**.

Commands to recover your lost commits in Git.

Git reflog

Git cherry-pick

***Note: Using the reflog will only work for a certain amount of time after the commits are lost. Git cleans the reflog periodically, so don’t wait too long!***

**Step by step procedure:**

**Step 1: The first step to recovering your lost commits is to recover the list of all your previous commits and actions done on the repository.**

**git reflog**

**Step 2: After running the command this is what you will see as the output.**

A screen shot of a computer

AI-generated content may be incorrect.

**Step 3: Find of the Hash of the commit you want to recover, For ex-12944d8. Now use the following command to bring your lost commit back.**

**git cherry-pick 12944d8**

**Step 4: Recovered Commit. That's it! Your work should be recovered soon with the following success commands:**

A screen shot of a computer

AI-generated content may be incorrect.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Section 11: Advanced Commands:**

* 1. **Git Aliases:**

A Git alias is a shorthand for a Git command or series of commands. By creating an alias, you can replace a lengthy command with a short and memorable abbreviation. This not only saves time but also reduces the risk of errors when typing commands.

**Why Use Git Aliases?**

* **Efficiency:** Save time by reducing the amount of typing needed.
* **Consistency:** Ensure you use the same commands consistently across different projects.
* **Customization:** Tailor your Git experience to suit your specific workflow.
* **Ease of Use:** Simplify complex commands into straightforward ones.

**Creating Git Aliases:**

**Method 1: Using Git Config**

Here aliases co, br, ci, and st were created globally for the commands [checkout,](https://www.geeksforgeeks.org/html/git-checkout-and-merge/)[branch](https://www.geeksforgeeks.org/git/introduction-to-git-branch/), commit, and status respectively.

If the tag "**--global**" isn't used then the aliases will be local by default.

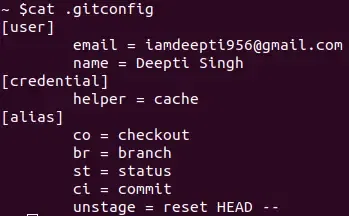
*git config --global alias.<custom\_command\_name> <original\_command Name>*

$ git config --global alias.co checkout  
$ git config --global alias.br branch  
$ git config --global alias.ci commit  
$ git config --global alias.st status

**Method 2: Editing Git Config Files Directly**

You can manually edit the .gitconfig files to create aliases. Global aliases are stored in the global .gitconfig file (located in the user’s home directory on Linux), while local aliases reside in the .git/config file within the repository.

The contents of the config file are as shown below.  The file contains various information like user email, name, alias, etc. After creating aliases they will be listed under the alias header. One can insert aliases under the [alias] header manually.



**Method 3: Using Aliases To Create New Git Commands**

**Create a new branch with a customized name:**

You can use the alias name that you want to give to the command and the name of the command.

*alias  <name> ='<command name>'*

*alias gcp='git commit -m "Work in progress." && git push origin HEAD'*

**Examples of Useful Git Aliases:**

git config --global alias.st 'status'

git config --global alias.lg 'log --oneline --graph --all'

git config --global alias.co 'checkout'

git config --global alias.ci 'commit -m'

git config --global alias.a 'add'

git config --global alias.psh 'push'

**Benefits Git Aliases:**

1. **Git aliases can make you a faster and more efficient developer as they can save you a lot of keystrokes in the long run. For example, git commit is a frequently used command, using git ci every time instead of git commit is going to save a few keystrokes.**
2. **It makes the command look simpler and clearer.**

**==================================================================================**

**2. Git – Subtree:**

Git Subtree is a strategy that lets you nest one Git repository inside another as a subdirectory. Unlike Git Submodules, which link to external repositories, Subtrees allows you to directly integrate and maintain a separate repository within your main repository. This approach makes it easier to manage dependencies and simplifies the workflow for collaborative projects.

**Benefits of Using Git Subtree**

* **Integration:** Seamlessly integrates external repositories into your main project.
* **Simplicity:** Simplifies the workflow compared to Git Submodules by avoiding additional metadata and simplifying commands.
* **Autonomy:** Each subtree can be managed independently, making it easier to track changes and updates.
* **Flexibility:** Allows for merging and splitting subtrees without affecting the main repository.

**Advantages of Subtrees**

1. Supported by Git's previous version. It supports versions older than 1.5 as well.
2. Workflow management is simple.
3. After the super-project is completed, there will be an available sub-project code.
4. You don't need newer Git knowledge.
5. Content modification without the need for a different repo of the dependence.

**Disadvantages of Subtrees**

1. It's not immediately obvious that a subtree is used to build the main repo.
2. Your project's subtrees are difficult to list.
3. You can't, at least not simply, list the subtrees' remote repositories.
4. When you change the main repository with subtree commits, then submit the subtree to its main server, and then pull the subtree, the logs can be a little misleading.

**Adding a Subtree to Parent Repository:**

To bring a new subtree to a parent repository, you first remotely add it, then use the subtree add command, which looks like this:

$ git remote add remote-name <URL to Git repo>  
$ git subtree add --prefix=folder/ remote-name <URL to Git repo> subtree-branchname

The commit log of the entire child project gets merged into the main repository.

**Pushing and pulling changes to and from the subtree**

$ git subtree push-all

or the below command performs the same as follows:

$ git subtree pull-all

**Git Submodule**

* A submodule is a **pointer** to a specific commit in another repository. It does not copy the external repository's history into your project.
* The parent repository remains **smaller** because it only stores the reference, not the full code.
* It's ideal for **component-based development** and when you need to **push changes** back to the external repository.
* The submodule's repository must be **accessible from a server** for others to clone your project.

**Git Subtree**

* A subtree **merges** the external repository's code and history directly into a subdirectory of your project.
* The parent repository size **increases** because it contains the complete history of the added repository.
* It's better for **system-based development** or when using third-party code you're unlikely to modify and push back.
* It's **easier to pull** new changes from the external repository than with submodules.

**Key Differences**

* **Data Storage:** A submodule points to a repo; a subtree contains the full repo history.
* **Repository Size:** Submodules keep the parent repo small; subtrees make it larger.
* **Pushing:** Pushing changes back to the source is easier with a submodule.
* **Pulling:** Pulling updates from the source is easier with a subtree.
* **Accessibility:** A submodule needs the source repository to be accessible; a subtree does not after the initial pull.

===================================================================================

1. **Git Patch Operation:**

A **Git patch** is a way to share code changes between different branches without merging or rebasing.

A patch file is created from the differences between two Git commits, branches, or a feature branch and the master branch.

The patch file contains a record of the changes—additions, deletions, and modifications—made to the code.

This file can be applied to another branch to replicate those specific changes, essentially "patching" the new branch with the changes from the original.

This is useful for sharing a small, specific set of changes with a teammate or applying a feature to a different project without a full merge.

**Creation and Working of Patches:**

**Step 1: There is a repository named 'Pat'. The user created a file "text.txt" in the master.**

A screenshot of a computer

AI-generated content may be incorrect.

**Step 2: Then it is added to the master branch and the change is committed with a message "initial".**

A screen shot of a computer

AI-generated content may be incorrect.

**Step 3: Checking logs**

A computer screen shot of a computer code

AI-generated content may be incorrect.

**Step 4: Now, let create a new branch named feature and checkout to our feature branch.**

A computer screen shot of text

AI-generated content may be incorrect.

**Step 5: Change the "text.txt" by adding a few words.**

A black background with white text

AI-generated content may be incorrect.

**Step 6: Next, we will save these changes, add and commit the changes in our feature branch with a message "this is in feature branch".**

**Step 7: Checking logs**

A computer screen with white and blue text

AI-generated content may be incorrect.

**Step 8: Now comes the main part- Creating a patch file. For creating a patch file we will write**

***Command:***

***git format-patch {branch in which we will apply our patch file} -o {folder in which we will save our patch file}***

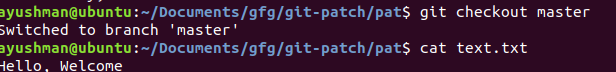
**Currently, we are in the feature branch. We need to compare our changes to the branch where we want to apply our changes(here master branch). Then we use "-o" and specify the folder in which the patch file will be created (this is optional).**

A computer screen shot of text

AI-generated content may be incorrect.

**We can also create a patch file using a particular commit's hash key. For example:-**

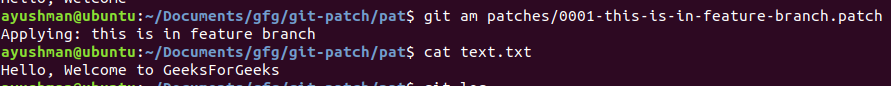
**git format-patch -1 012fe999affd9b92f6c96eb72a7260273411ea81 -o patches**

**Step 9: Now, we come back to our master branch and check our "text.txt".**

**Step 10: Now, we finally apply our patch file in the master branch.**

**Command:**

**git am {path to patch file}**



**Step 11: Checking logs**

A computer screen shot of a computer code

AI-generated content may be incorrect.

**==================================================================================**

1. **Git Prune:**

Git prune is used to remove the unreachable objects and clear up the local repository.

**git prune** is a Git command that removes objects that are no longer referenced by any branch or tag. These unreferenced objects, often called **"dangling objects",** can accumulate over time as you make changes to your repository. **Pruning** these objects helps free up disk space and keeps your repository organized.

**Syntax:**

git prune [options]

**Command options:**

* **-n or --dry-run:**is used to just display the removable items in the repository.
* **-v or -verbose:**provide details of pruning process.

**What are Git Objects?**

Git stores data as objects within its database. These objects include:

* **Commits**: Representing snapshots of your project at different points in time.
* **Trees**: Representing directories and their structure.
* **Blobs**: Representing individual files.

Over time, as you **create, modify, and delete** files and commits, Git’s internal database can accumulate objects that are no longer referenced by any branch or tag. These objects become **“unreachable” or “dangling”** and can consume unnecessary disk space.

**Why Usegit prune?**

1. **Free Up Disk Space**: Over time, your Git repository can become cluttered with unused objects, which consume disk space. Pruning helps to remove these objects and free up space.
2. **Improve Performance**: A cleaner repository with fewer dangling objects can lead to better performance when running Git commands.
3. **Maintain Organization**: Regular pruning helps keep your repository clean and well-organized, making it easier to manage and navigate.

**When to Use git prune**

git prune is particularly useful in the following scenarios:

1. **After Deleting Branches**: When you delete branches, the commits associated with those branches might still exist in your repository as dangling objects. Pruning helps remove these unreferenced commits.
2. **After Rewriting History**: If you've used commands like git rebase or git filter-branch, you may end up with orphaned commits that need to be pruned.
3. **Regular Maintenance**: Periodically running git prune as part of your repository maintenance routine helps keep it clean and efficient

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Section 12: Git Advanced Concepts:**

1. **Git Branch:**

A Git branch is like a separate workspace where you can work on a new changes without disturbing the main project. The primary code is usually stored in main or master branch. When you are done with your work on the branch, you can merge your changes back into the main project.

Branches make it easy to:

* Manage different tasks or features independently
* Test changes without affecting live code
* Collaborate with others efficiently

**Types of Branches in Git:**

* **Main (or Master) Branch**: This is the main branch where the final stable code is stored. It should always be production-ready.
* **Feature Branch**: This branch is created to work on a new feature. Once done, it is merged back into the main or develop branch.
* **Develop Branch**: It is used to combine all new features before adding them to the main branch. It helps in testing and development.
* **Hotfix Branch**: Created to quickly fix urgent issues in production. After fixing, it is merged back into the main branch.
* **Release Branch**: Used to prepare for a new version release. Only final testing and small fixes are done here before merging into main.
* **Bugfix Branch**: Created to fix specific bugs in the code. Once fixed, it is merged into the develop or main branch.

===================================================================================

1. **Exporting a Git Project:**

We are going to learn about git export but before we continue let’s understand what actually gIt export means. Git export performs a git clone and clones it in a different location but without the .git file, i.e., it won't be git repo anymore, but you would have all its files.

There is no such command as git export in Git bash, but we can perform it using a different command known as git archive.

**Source Code:**

**git archive [--format=<fmt>] [--list] [--prefix=<prefix>/] [<extra>]**

**[-o <file> | --output=<file>] [--worktree-attributes]**

**[--remote=<repo> [--exec=<git-upload-archive>]] <tree-ish>**

**[<path>…​]**

**Example:**

**git archive master | tar -x -C /somewhere/else**

By default, the output is produced in "tar" format through the git archive command, so it would be better to store it in a zipped file like "gzip" or "b2zip" format.

git archive master | bzip2 >source-tree.tar.bz2

*Here, we have renamed our file as source-tree.tar and used it in "b2zip" format.*

We can also use a .zip format using the following command:

git archive --format zip --output /full/path/to/zipfile.zip master

We can clone the entire git repository using this command but we must be careful as even though it doesn’t include the .git directory, however, it will contain other get hidden files like .gitignore  .gitattributes, etc. There are separate commands if you wish to remove them on the go. Commit a .gitattributes file with the following code as shown below as follows:

/test export-ignore

.gitattributes export-ignore

.gitignore export-ignore

If we are interested in exporting the index the command for exporting it is

git checkout-index -a -f --prefix=/destination/path/

===================================================================================

1. **How to Move the Most Recent Commit(s) to a New Branch With Git?**

Why Move Commits to a New Branch?

1. **Correcting Mistakes**: You may have accidentally committed changes to the wrong branch and need to move them to the correct one.
2. **Isolating Changes**: You might start working on a feature and realize it's better suited as a separate branch for better organization and testing.
3. **Collaborative Workflow**: Moving commits can help maintain a clean and organized workflow, especially in collaborative environments where each branch represents a specific task or feature.

**Steps to Move Recent Commits to a New Branch:**

**Step 1: Check Current Status**

**git status**

**Step 2: Create a New Branch**

**git checkout -b new-branch-name**

**Step 3: Move the Commits**

1. **Moving the Most Recent Commit:**

**Step 1: If you only need to move the most recent commit, you can use the git reset command. Switch back to the original branch:**

**git checkout original-branch-name**

**Step 2: Command to remove the changes from original branch,**

**git reset --hard HEAD~1**

**Step 3: Now, switch back to the new branch:**

**git checkout new-branch-name**

**Step 4: And cherry-pick the removed commit onto the new branch or you can use the commit hash id also to pick the deleted commit:**

**git cherry-pick original-branch-name**

1. **Moving Multiple Recent Commits:**

**Step 1: Command to move more than one commit,**

**git checkout original-branch-name  
git reset --hard HEAD~3  
git checkout new-branch-name  
git cherry-pick original-branch-name..HEAD@{1}**

**Step 4: Verify Changes**

**git log**

**==================================================================================**

1. **Git - Pack Objects:**

Git objects are compressed and available in the git object pack file. Git can use a pack file to store the objects which will help you to reduce the size of the repository. Git will update and create pack files to optimize the storage efficiency which will reduce redundancy and disk space.

Git uses **pack objects** to store repository data efficiently, compressing multiple files into a single archive to save space and improve performance.

 **What they are:** Pack objects are compressed archives containing multiple Git objects (files, commits, branches), reducing repository size.

 **Why they're important:** They **save space** by compressing objects, **improve speed** by making operations like cloning faster, and **avoid redundancy** through delta compression.

 **How they work:** Git stores objects individually and then automatically **packs them** into compressed files to optimize storage. This packing process occurs during operations like git push or git pull. When you push or pull changes to or from a Git repository, Git automatically creates or updates pack files. This process is called packing. Git tries to make sure that objects are packed as efficiently as possible, reducing their size and making the repository quicker to work with.

 **Dealing with large objects:** You can manage large files by using **Git LFS**, removing unnecessary commits with git rebase, cloning with --depth 1, or running git gc to clean up the repository.

 **Corruption:** If pack files become corrupted, you can use git fsck to check for issues, git reflog to roll back, or a fresh git fetch or git clone to restore the repository.

 **Pack size factors:** The size of a pack file is influenced by the size of the objects themselves, compression settings, and the overall complexity of the repository.

===================================================================================

1. **How to Handle Big Repositories With Git?**

There are several techniques to efficiently clone and manage large Git repositories, especially those with extensive history or large binary files.

 **Shallow Clone**

* Clones only the latest n commits, rather than the entire history.
* Significantly reduces cloning time and repository size.
* Command: git clone --depth [n] [url]
* Best for: When you only need the recent history to work on a project.

 **Cloning a Single Branch**

* Clones only a specified branch, ignoring all others.
* Useful when a repository has many branches and you only need one to get started.
* Command: git clone [url] --branch [branch\_name] --single-branch
* Best for: Focusing on a specific feature branch without cloning the entire project structure.

 **Git Filter**

* Allows you to rewrite the repository's history by filtering or removing specific files.
* Commonly used to remove large binary files from the entire history.
* **Caution:** This changes commit IDs, which may require other collaborators to re-clone the repository.
* Command: git filter-branch --tree-filter 'rm -rf [path-to-asset]'

 **Handling Large Binary Files**

* **Git LFS (Large File Storage):** A Git extension that stores large files in a separate location, leaving only small pointers in the main repository. This is the recommended solution.
* **Submodules:** A repository within another repository. This allows you to manage binary files in a separate project, keeping the main codebase clean and modular.
* **Garbage Collection (git gc):** This command optimizes the repository by cleaning up unnecessary files and turning loose objects into a single, compressed pack file.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* 1. **Stashing:**

1. **Git Stash Command:**

Git's stash command is a powerful tool for temporarily saving your uncommitted changes so you can work on something else, then reapply them later. This is useful when you need to switch branches or perform a quick, high-priority fix without committing incomplete work.

**git stash**: This command saves your modified, uncommitted changes to a temporary location, known as the stash. Your working directory and staging area are then reset to a clean state, as if you hadn't made any changes.

**git stash apply**: This command takes the most recently stashed set of changes and reapplies them to your current working directory. The changes are restored, but they remain in the stash, so you can reapply them if needed.

**git stash list**: This command shows you a list of all your current stashes. Each stash is given an index, starting with stash@{0} for the most recent one.

**git stash drop**: This command removes a specific stash from your list. By default, it removes the most recent one (stash@{0}). It's a good practice to drop a stash after you've applied its changes and committed them.

1. **Stash command for untracked files:**

Continuing with Git's stash command, this example demonstrates how to handle untracked files and introduces the more efficient git stash pop command. By default, git stash only saves changes to files that Git is already tracking. Untracked files, like a brand new file you've created, are ignored and remain in your working directory.

**git stash -u**: The -u flag, short for --include-untracked, tells Git to also stash any **untracked files** that are in your working directory. This is essential for achieving a completely clean working directory when you have new files.

**git stash pop**: This command is a combination of git stash apply and git stash drop. It applies the most recent stash to your working directory and then immediately drops it from your list of stashes. This is a more streamlined workflow when you know you won't need the stash again.

1. **Managing Multiple Stash:**

Working with multiple stashes is a common scenario in Git, and it's essential to understand how to manage and differentiate them effectively. Instead of just using git stash, you can save each stash with a unique message, similar to a commit message. This makes it easier to remember what each stash contains.

**git stash save <message>**: This command saves your changes to the stash and allows you to add a descriptive message. This is highly recommended when you're working with multiple stashes.

**git stash list**: This is your go-to command for viewing all your stashes. The stashes are indexed in a Last-In, First-Out (LIFO) order, where the most recent stash is at **stash@{0}**. The indices are counter-intuitive at first, so it's a good idea to remember that 0 is always the newest.

**git stash show <stash\_ref>**: Use this command to inspect a specific stash without applying it. The <stash\_ref> is the stash's index (e.g., stash@{1}). This helps you confirm what's in a stash before you apply it.

git stash show stash@{1}

**git stash apply <stash\_ref>**: This command applies a specific stash to your working directory. If you don't provide a reference, it will apply the most recent stash (stash@{0}).

**git stash drop <stash\_ref>**: This command removes a specific stash from your list. After you apply a stash and commit the changes, it's a good practice to drop the stash to keep your list clean. When a stash is dropped, the remaining stashes are re-indexed.

**git stash clear**: This command removes **all** stashes from your list at once. Use this with caution, as it permanently deletes all stashed changes.

1. **Stashing changes to new branch:**

The main command used in this workflow is git stash branch <new\_branch\_name>.

This command is a shortcut that streamlines the process of moving a stash to a new branch, and it's particularly useful for correcting the common mistake of starting work on the wrong branch.

**How it Works**

1. **Creates a new branch**: It creates a new branch with the specified name (in this case, newchanges) based on the commit where the stash was originally created.
2. **Checks out the new branch**: It immediately switches your working directory to the newly created branch.
3. **Applies and drops the stash**: It applies the contents of your most recent stash (stash@{0}) to the new branch and then, immediately after a successful application, it deletes that stash from your stash list.

**Why It's Powerful**

Without this command, you would have to perform the same actions manually, which would look like this:

1. git checkout -b <new\_branch\_name> (to create and switch to the new branch)
2. git stash pop (to apply the stash and remove it from the stash list)

===================================================================================

**A blue and black circle with black text

AI-generated content may be incorrect.**

**A screenshot of a computer screen

AI-generated content may be incorrect.**

**A diagram of a diagram

AI-generated content may be incorrect.**

**A screenshot of a computer

AI-generated content may be incorrect.**

git config --global core.editor "notepad++.exe -multiInst -nosession"

git config --global diff.tool p4merge

git config --global difftool.p4merge.path "<Original\_Path\_to\_p4merge>"

git config --global difftool.p4merge.prompt flase

git config --global merge.tool p4merge

git config --global mergetool.p4merge.path "<Original\_Path\_to\_p4merge>"

git config --global mergetool.p4merge.prompt flase

git config --global alias.<alias\_name> "<original\_command>"

git config --global user.name ""

git config --global user.email "'

git config --global --list

git config --global -e