01 git Notes - Nikhil Sharma - KirkYagami

Git is a distributed version control system commonly used for tracking changes in source code during software development. It allows multiple developers to work on a project simultaneously, keeping track of changes made to files over time, and enables collaboration and efficient management of code repositories. Here are some of the most useful Git commands:

1. `git init` : Initializes a new Git repository in the current directory.
2. `git clone` : Creates a copy of a remote repository on your local machine.
3. `git add` : Adds a file to the staging area, preparing it for a commit.
4. `git commit -m “”` : Records changes to the repository, creating a new commit with a descriptive message.
5. `git status` : Displays the current state of the repository, including modified, staged, and untracked files.
6. `git pull` : Fetches and merges changes from a remote repository into the current branch.
7. `git push` : Pushes commits from a local branch to a remote repository.
8. `git branch` : Lists existing branches in the repository.
9. `git checkout` : Switches to a different branch.
10. `git merge` : Merges changes from one branch into the current branch.
11. `git log` : Shows a history of commits, including commit messages and other details.
12. `git diff` : Displays the differences between the current state and the last commit. These are just a few of the many commands available in Git.

Git provides a powerful set of tools for managing version control, branching, merging, and collaboration in software development projects.

2. Why git is most popular ?

Git is the most popular version control system for several reasons:

1. **Distributed architecture:** Git is a distributed version control system, meaning that every developer has a complete copy of the entire repository, including its history. This allows developers to work offline and independently, and it provides redundancy and backup in case of server failures. The distributed nature of Git also makes branching and merging fast and efficient.
2. **Speed and efficiency:** Git is designed to be fast and perform well even with large repositories and extensive histories. It utilizes advanced algorithms and data structures to optimize operations such as committing, branching, merging, and retrieving historical information.

3. **Branching and merging:** Git provides powerful branching and merging capabilities. Branching is lightweight and easy to create, allowing developers to create separate lines of development for features, bug fixes, or experiments. Merging branches in Git is usually straightforward and results in minimal conflicts, thanks to Git's advanced merge algorithms.
4. **Collaboration and teamwork:** Git enables efficient collaboration among developers working on the same project. Multiple developers can work on different branches simultaneously, and changes can be easily shared and merged. Git also provides tools for code review and managing contributions from multiple developers.
5. **Large community and ecosystem:** Git has a vast and active user community, which contributes to its popularity. Many popular hosting platforms, such as GitHub and GitLab, support Git repositories and provide additional features like issue tracking, pull requests, and project management tools. The availability of numerous third-party tools and integrations further enhances Git's ecosystem.
6. **Flexibility and extensibility:** Git is highly flexible and can be adapted to various workflows and development processes. It provides hooks, custom scripts, and configuration options, allowing developers to tailor Git to their specific needs. Git also supports various workflows, including centralized, feature branching, and Gitflow, among others.
7. **Stability and maturity:** Git was initially developed by Linus Torvalds, the creator of the Linux kernel, and it has been widely adopted and extensively tested over the years. It has proven to be stable and reliable for managing version control in both small and large-scale projects.

These factors, along with its rich feature set and ease of use, contribute to Git's popularity and make it the preferred choice for version control in the software development community.

3. Explain the difference between 'git pull' and 'git fetch' ?

Both `git pull` and `git fetch` are used to update your local repository with the latest changes from a remote repository. However, they have different behaviors and purposes:

`git pull`:

The `git pull` command combines two operations into one: fetching the latest changes from the remote repository and merging them into the current branch.

1. It fetches the latest commits from the remote repository, similar to the `git fetch` command.
2. It automatically merges the fetched commits into the current branch, creating a new merge commit if necessary. In other words, `git pull` fetches the changes and immediately applies them to your local branch. This can result in a fast-forward merge (if the branch has not diverged) or a merge commit (if there are diverging changes).

`git fetch`:

The `git fetch` command retrieves the latest commits and updates the remote-tracking branches in your local repository. It doesn't modify your current branch or merge the changes automatically.

1. It fetches the latest commits from the remote repository and updates the corresponding remote-tracking branches, such as `origin/master` or `origin/feature-branch`.
2. It does not modify your working directory or current branch. After running `git fetch`, you can examine the fetched changes, compare them with your local branch, and decide how to integrate them. This allows you to review and understand the changes before merging them.

The main differences between `git pull` and `git fetch` can be summarized as follows:

— `git pull` combines the `git fetch` and `git merge` commands into one convenient operation, updating your current branch automatically.

— `git fetch` only retrieves the latest commits and updates the remote-tracking branches, allowing you to review and merge the changes at your own discretion.

It's worth noting that when using `git pull` or `git fetch`, you may need to handle merge conflicts if there are conflicting changes between the local and remote branches. It's a good practice to review the changes before merging or rebasing to avoid unexpected conflicts.

4. What command is used in git to change your working branch ?

```
git checkout "other-branch"
```

5. What is the difference between git and GitHub ?

Git and **GitHub** are related but serve different purposes:

Git:

— Git is a distributed version control system (VCS) designed to manage source code and track changes in files.

— It allows developers to work offline and independently on their local repositories, recording changes and creating branches to work on different features or experiments.

— Git provides functionalities like committing changes, branching, merging, reverting commits, and managing repositories.

GitHub:

— GitHub is a web-based hosting service that provides a centralized platform for Git repositories.

— It adds additional features on top of Git, such as a web interface, collaboration tools, issue tracking, pull requests, and code reviews. — GitHub allows developers to store, manage, and collaborate on code projects in a cloud-based environment.

— It provides a platform for team collaboration, open-source contributions, and project management.

Key differences:

1. **Functionality:** Git is a command-line tool that provides core version control functionalities, while GitHub is a web-based platform built around Git that offers additional collaboration and project management features.
2. **Hosting:** Git is primarily a local VCS, where repositories reside on individual computers. In contrast, GitHub provides a centralized hosting service for remote repositories accessible from anywhere.
3. **Collaboration:** GitHub facilitates collaboration among developers by offering features like pull requests, code reviews, issue tracking, and project boards. These features make it easier for teams to work together, review and discuss code changes, and manage project workflows.
4. **Social and Open Source:** GitHub has a strong social aspect, enabling developers to showcase their projects, contribute to open-source repositories, discover and follow other projects, and participate in the larger developer community.
5. **Workflow Integration:** GitHub integrates with Git, allowing developers to use Git commands locally and then push changes to a remote GitHub repository. This integration enables seamless synchronization and collaboration between local and remote repositories.

It's worth noting that while GitHub is the most popular and widely used hosting service for Git repositories, there are alternative hosting platforms available, such as GitLab and Bitbucket, which offer similar functionalities and services.

6. How do you rename a branch in Git ?

To **rename a branch in Git**, you can use the following steps:

1. **Check the current branch:** Before renaming the branch, ensure that you are not currently on the branch you want to rename. If you are on the branch, you can switch to a different branch using the `git checkout` command.

```
git checkout <other-branch>
```

2. Rename the branch: To rename a branch, you can use the `git branch` command with the `-m` option followed by the current branch name and the new desired name.

```
    git branch -m <current-branch-name> <new-branch-name>
```

For example, if you want to rename the branch named **"feature-branch"** to **"new-feature-branch,"** you would use the following command:

```
git branch -m feature-branch new-feature-branch
```

Note that if you are renaming the branch you are currently on, you need to switch to a different branch first.

3. Push the renamed branch (optional): If the branch you renamed was already pushed to a remote repository, the remote repository will still have the old branch name. To update the remote repository with the new branch name, you can use the `git push` command with the `--set-upstream` or `-u` option.

```
git push -u origin <new-branch-name>
```

Replace `` with the new name of the branch you just renamed.

This command sets the upstream branch and pushes the changes to the remote repository. Subsequent pushes or pulls can be done using the new branch name.

```
git push origin --delete <old-branch-name>
```

Replace `` with the previous name of the branch you renamed.

4. Delete the old remote branch (optional): If you have renamed the branch and pushed the changes to the remote repository, you might want to delete the old branch with the previous name from the remote repository. You can use the `git push` command with the `--delete` option followed by the old branch name to delete it from the remote repository.

Conclusion:

By following these steps, you can easily rename a branch in Git, locally and optionally update the remote repository with the new branch name. Remember to update your local repositories and inform your collaborators if you have renamed a branch that was already pushed to a remote repository to ensure a smooth workflow.

7. Describe the purpose and functionality of the '.gitignore' file ?

The `.gitignore` file is a text file used in Git to specify intentionally untracked files or directories that should be ignored by Git. Its purpose is to exclude certain files and directories from being tracked and version-controlled, ensuring they are not included in commits or pushed to a remote repository. The `.gitignore` file helps to maintain a clean and focused version control history by filtering out files that are not relevant or should not be shared.

Here's an overview of the purpose and functionality of the `.gitignore` file:

1. **Ignoring untracked files:** When you initialize a Git repository or clone an existing one, all files and directories in the repository's root directory are potentially tracked by Git. However, there are often files and directories that you don't want to include in version control, such as compiled binaries, log files, editor-specific files, or sensitive information like API keys.
2. **Filtering out files from being tracked:** By listing file patterns and directories in the `.gitignore` file, you can explicitly tell Git to ignore them. Git uses the patterns in the `.gitignore` file to determine which files and directories should be excluded when performing operations like `git status`, `git add`, or `git commit`.
3. **Syntax of `.gitignore` patterns:** The patterns in the `.gitignore` file can include wildcards, directories, and negations to define exclusion rules.

Some examples of patterns include:

— `*.log`: Ignores all files with the extension `.log`.
— `build/`: Ignores the `build` directory and all its contents.
— `config.ini`: Ignores a specific file named `config.ini`.
— `!important.log`: Negates the previous pattern and ensures that `important.log` is not ignored, even if `*.log` is listed.

1. **Placement of `.gitignore` file:** The `.gitignore` file is typically placed in the root directory of a Git repository. It can also be added to specific subdirectories to define exclusion rules for those directories and their subdirectories.
2. **Handling tracked files:** It's important to note that the `.gitignore` file only affects untracked files. If a file is already being tracked by Git, modifying the `.gitignore` file does not automatically stop tracking it. To remove a file from version control while keeping it on disk, you need to use `git rm – cached` command.
3. **Versioning the `.gitignore` file:** The `.gitignore` file itself can be version-controlled, allowing collaborators to share and update the exclusion rules. By committing and pushing the `.gitignore` file, you ensure that all team members have consistent rules for ignoring files and directories.

Conclusion:

The `.gitignore` file is a valuable tool for maintaining a clean and organized Git repository. It helps prevent unnecessary or sensitive files from being included in commits, reduces clutter in version control history, and ensures that only relevant files are shared with collaborators. By using exclusion rules in the `.gitignore` file, you can tailor the version control process to your project's specific needs.

8. What things to be considered while reviewing a pull request ?

Following points should be taken into consideration :

1. Look into all individual commits

2. Look first into the files which have relevant changes.
3. We need to make sure that the PR's code should be working properly as per the new feature for which the PR is raised.
4. Code should not affect any previously existing feature. 3. Functions should not be too long.
5. There should not be redundant code.
6. Code should be clean i.e. there should be proper indendentation , no unnecessary blank lines , semicolon after each javascript statement.
7. Variables and function names should be meaningful .
8. If the application contains unit tests than the test coverage criteria should be met.

9. What is the purpose of the 'git stash' command ?

The `git stash` command is used in Git to temporarily save changes that you have made to your working directory but are not yet ready to commit. It allows you to store your changes in a "stash" and revert your working directory to a clean state, as if you hadn't made any modifications. The purpose of the `git stash` command is to provide a way to save your work without committing it, so you can switch to a different branch, pull changes from a remote repository, or perform other operations that require a clean working directory.

Here are the main use cases and benefits of the `git stash` command:

1. **Switching branches:** When you have made changes to your working directory but want to switch to a different branch, Git may prevent the branch switch if your changes conflict with the branch you are trying to switch to. In such cases, you can use `git stash` to save your changes, switch branches, and then later apply the stash to the new branch.
2. **Pulling changes:** Before pulling changes from a remote repository, it's recommended to have a clean working directory. If you have uncommitted changes that would conflict with the incoming changes, you can stash your changes, pull the remote changes, and then reapply your stashed changes.
3. **Temporary storage:** The `git stash` command provides a convenient way to store changes temporarily without creating a commit. This can be useful when you want to set aside your changes temporarily to work on a different task or to test something in your codebase without affecting your current work.
4. **Apply your stashed changes:** You can apply back your stashed changes using command **'git stash apply'** The `git stash` command works by creating a new stash object that includes the changes in your working directory and staged changes (if any). The stash is stored separately from commits, and it doesn't affect the commit history. You can create multiple stashes, apply or drop stashes as needed, and even store untracked files in a stash.

Here are some commonly used `git stash` commands:

— `git stash save` : Creates a new stash with the changes in your working directory.

— `git stash list` : Lists all the stashes you have created.

— `git stash apply` : Applies the most recent stash to your working directory.

— `git stash pop` : Applies the most recent stash and removes it from the stash list.

— `git stash drop` : Removes a specific stash from the stash list.

— `git stash branch` : Creates a new branch and applies a stash to that branch.

Conclusion:

Overall, `git stash` provides a flexible way to temporarily save changes, switch branches, and maintain a clean working directory without committing unfinished or experimental work.

10. How do you resolve a merge conflict in Git ?

To **resolve a merge conflict in Git**, you can follow these steps:

1. **Checkout the branch:** Start by checking out the branch where the merge conflict occurred. Typically, this is the branch you were merging into, such as your feature branch or the main branch.

```
git checkout <branch-name>
```

2. Pull the latest changes: It's essential to have the latest changes from the remote repository before resolving the conflict. Use the following command to pull the changes:

```
git pull origin <branch-name>
```

3. Identify the conflicted files: Git will indicate the conflicted files in your project. You can use the `git status` command to see which files have conflicts. The conflicted files will have markers indicating the conflict within the file. For example:

```
$ git status
 ...
 Unmerged paths:
   (use "git add <file>..." to mark resolution)
         both modified:   file.txt
```

4. Open the conflicted file: Open the conflicted file(s) in a text editor and locate the conflict markers. The conflict markers look something like this:

```
<<<<<<< HEAD
   This is the content on your branch.
   =======
```

```
    This is the content from the branch you are merging.
    >>>>>>> branch-name
```

```
The content between `<<<<<<< HEAD` and `=======` is from your branch, and the content between `=======` and `>>>>>>> branch-name` is from the branch you are merging.
```

5. Resolve the conflict: Edit the conflicted file to resolve the conflict. Decide which changes to keep, modify the content as needed, and remove the conflict markers. For example, you might choose to keep one version of the code or combine the changes manually. After making the necessary changes, save the file.

6. Stage the resolved file: Once you have resolved the conflict in a file, you need to stage it to mark it as resolved. Use the following command for each resolved file:

```
    git add <file>
```

7. Commit the changes: After staging all the resolved files, commit the changes with a commit message describing the conflict resolution.

```
  git commit -m "Resolve merge conflict"
```

8. Push the changes: Finally, push the changes to the remote repository to complete the merge.

```
    git push origin <branch-name>
```

After following these steps, the merge conflict should be resolved, and your branch should be up to date with the changes from the other branch.

11. What is the difference between Git's "merge" and "rebase" operations ?

In Git, both **"merge"** and **"rebase"** are operations used to integrate changes from one branch into another. However, they differ in how they accomplish this and the resulting commit history. Let's explore the differences between the two operations with examples:

Merge:

— Merge combines the changes from one branch into another while preserving the individual branch histories.

— It creates a new commit that represents the combination of changes from both branches.

— The commit history shows a merge commit that serves as a point where the two branches converge.

— Merging is useful for integrating feature branches, bug fixes, or other branches that should maintain their independent histories.

Example scenario:

1. Start with two branches, `main` and `feature/xyz`, each with their own commit history.
2. Switch to the `main` branch: `git checkout main`
3. Execute the merge operation: `git merge feature/xyz`
4. Git combines the changes from `feature/xyz` into `main` and creates a new merge commit.
5. The commit history now includes the merge commit, representing the integration of changes from both branches.

Rebase:

— Rebase integrates the changes from one branch by incorporating them onto another branch as if they were developed linearly, resulting in a cleaner and more linear commit history.

— It moves or "replays" the commits from one branch to the tip of another branch.

— The commit history appears as if the changes from the rebased branch were made directly on top of the branch being rebased onto.

— Rebasing is useful for incorporating the latest changes from a base branch into a feature branch, maintaining a cleaner and more straightforward history.

Example scenario:

1. Start with two branches, `main` and `feature/xyz`, each with their own commit history.
2. Switch to the `feature/xyz` branch: `git checkout feature/xyz`
3. Execute the rebase operation: `git rebase main`
4. Git takes the commits from `feature/xyz` and replays them on top of `main`.
5. The commit history shows the commits from `feature/xyz` placed on top of the latest `main` commit, resulting in a linear history.

Key differences:

— Merge preserves the individual branch histories, while rebase creates a more linear history.

— Merge creates a merge commit, while rebase does not.

— Merge is suitable for integrating independent branches, while rebase is useful for keeping a cleaner history when incorporating changes from one branch to another. It's important to note that the choice between merge and rebase depends on the specific requirements, project workflow, and collaboration considerations.

12. How do you remove a file from a Git repository ?

To **remove a file from a Git repository**, you can use the following steps:

1. **Check the status:** First, it's a good practice to check the status of your repository to see which files are staged for commit and which ones are not. Use the command:

```
git status
```

This will show you the current status of your repository.

2. Unstage the file (optional): If the file you want to remove is already staged for commit, but you want to keep it in your working directory, you can unstage it using the following command:

```
git reset <file>
```

3. Remove the file: To remove the file completely from your repository, including from your working directory, you can use the `git rm` command followed by the file's path:

```
git rm <file>
```

This command will remove the file from both the Git repository and your working directory.

If the file has already been deleted from your working directory manually and you want to remove it from the repository, you can use the `- cached` option with `git rm`:

```
git rm --cached <file>
```

This will remove the file from the Git repository while keeping it in your working directory.

4. Commit the changes: After removing the file, you need to commit the changes to make the removal permanent in the repository's history. Use the `git commit` command with an appropriate commit message:

```
git commit -m "Remove "
```

Replace `` with the name or path of the file you removed.

5. Push the changes (if necessary): If you have a remote repository and you want to update it with the changes, use the `git push` command to push your local commits to the remote repository:

```
git push origin <branch-name>
```

Replace `` with the name of the branch you are working on.

Conclusion:

By following above steps, you can **remove a file from both the Git repository** and your working directory, ensuring it is no longer tracked by Git.

13. How do you undo the last Git commit ?

To **undo the last Git commit**, you can use the following steps:

1. **Check the commit history:** First, verify the commit history and ensure that you want to undo the most recent commit. Use the `git log` command to view the commit history:

```
git log
```

The log will display the commit messages and commit hashes, allowing you to identify the commit you want to undo.

2. Undo the commit and keep changes: If you want to undo the commit but keep the changes from that commit in your working directory, you can use the `git reset` command with the `--soft` option. This will move the branch pointer of the current branch to the previous commit, effectively undoing the last commit:

```
git reset --soft HEAD~1
```

The `HEAD~1` specifies the commit you want to go back to, which in this case is the previous commit. This command will leave the changes from the undone commit in your working directory as uncommitted changes.

3. Undo the commit and discard changes: If you want to completely undo the last commit and discard the changes, you can use the `git reset` command with the `--hard` option. This will remove the last commit entirely and discard any changes associated with it:

```
git reset --hard HEAD~1
```

Similar to the previous command, `HEAD~1` specifies the commit to go back to, which is the previous commit. Be cautious when using `--hard` as it permanently removes the changes from the undone commit.

4. Push the changes (if necessary): If you have already pushed the commit to a remote repository and want to update it with the undone commit, you will need to force-push the changes. However, it's generally not recommended to force-push changes that have already been shared with others unless you have a specific reason to do so.

```
git push origin <branch-name> --force
```

Replace `` with the name of the branch where you want to force-push the changes.

Conclusion:

By following these steps, you can effectively undo the last Git commit and either keep the changes as uncommitted or discard them entirely. Remember to use caution when using `--hard` or force-pushing, as they can **permanently remove or modify the commit history**.

14. How do you view the Git commit history ?

To **view the Git commit history**, you can use the following commands:

1. **View a summarized log:** The most basic way to view the commit history is by using the `git log` command. It displays the commit history in reverse chronological order, with the most recent commit appearing first. By default, it provides information such as the commit hash, author, date, and commit message.

```
git log
```

Press the spacebar to scroll through the log, and press 'q' to exit the log view.

2. View a summarized log with graph: To get a more visually informative representation of the commit history, you can use the `git log` command with the `--graph` option. This displays the commit history as a text-based graph, showing the branches, merges, and commits.

```
git log --graph
```

The graph view provides a clearer picture of how branches and commits are related.

3. View a summarized log with one-line commit messages: If you prefer a condensed view of the commit history, you can use the `--oneline` option with `git log`. This displays each commit on a single line, showing only the commit hash and the first line of the commit message.

```
git log --oneline
```

This format is useful when you want a quick overview of the commit history.

4. View a detailed log: If you want to see more detailed information about each commit, you can use the `--stat` option with `git log`. This displays the summary of changes made in each commit, showing the modified files and the number of insertions and deletions.

```
git log --stat
```

This is helpful when you need a more comprehensive understanding of the changes made in each commit.

5. Limit the number of commits: By default, `git log` shows the entire commit history. However, you can limit the number of commits displayed by using the `-n` or `--max-count` option, followed by the desired number of commits.

```
git log -n 5
```

This command will show only the last five commits.

6. Filter the commit history: You can filter the commit history based on various criteria, such as the author, specific files, or a date range. Some examples include:

— Filter by author:

```
git log --author="John Doe"
```

— Filter by specific file:

```
git log -- <file-path>
```

— Filter by date range:

```
git log --since="2023-01-01" --until="2023-06-30"
```

These filters allow you to narrow down the commit history based on your specific requirements.

Conclusion:

These commands provide different ways to view the Git commit history, allowing you to examine the changes made, the commit relationships, and the associated metadata. Choose the option that best suits your needs for understanding and analyzing the commit history.

15. What is tagging in git ?

Tagging in Git is a way to mark specific commits in a repository as important milestones or versions. A tag is simply a pointer to a specific commit in the Git history, and it provides a convenient way to reference a particular version of the code.

In Git, tags can be created and managed using the git tag command. There are two types of tags in Git: lightweight and annotated.

A lightweight tag is simply a name that points to a specific commit, similar to a branch. It is created using the git tag command, and can be used to reference a specific version of the code.

An annotated tag, on the other hand, is a full Git object that contains additional metadata such as a tagger name and email, a tag message, and a timestamp. Annotated tags are created using the `git tag -a <tagname>` command, and are useful for creating release versions of the code that include additional information about the release.

Tags can also be pushed to remote repositories using the git push command, with the — tags option to push all tags, or the `git push origin <tagname>` command to push a specific tag.

Conclusion:

In summary, ***tagging in Git*** is a way to mark specific commits in a repository as important milestones or versions, and provides a convenient way to reference a particular version of the code.

16. What is rollback in git ?

In Git, a **rollback** typically refers to the act of reverting a repository or a specific commit to a previous state. It allows you to undo changes and return to a previous point in the commit history.

There are a few different ways to perform a rollback in Git, depending on the specific situation and your requirements. Here are a couple of common scenarios:

1. **Rollback a single commit:** If you want to undo the changes introduced by a specific commit while keeping the subsequent commits intact, you can use the `git revert` command. This command creates a new commit that undoes the changes made by the specified commit.

```
git revert
```

The `` represents the identifier of the commit you want to revert, such as the commit hash or a reference like a branch or tag name. Git will create a new commit that applies the inverse of the changes introduced by the specified commit.

2. Rollback multiple commits:

If you want to remove a series of commits and make it as if they never happened, you can use the `git reset` command. This command allows you to move the branch pointer to a previous commit, effectively discarding the commits that follow.

```
git reset
```

The `` specifies the commit to which you want to roll back. Git will move the branch pointer to this commit and discard the subsequent commits. By default, this command preserves the changes introduced by the discarded commits as unstaged changes in your working directory.

If you want to completely discard the changes introduced by the rolled-back commits, you can use the `– hard` option:

```
git reset --hard
```

Note: Exercise caution when using the `git reset` command with the `– hard` option, as it permanently removes the changes from your repository.

Conclusion :

It's important to note that rolling back commits modifies the commit history, so it's generally recommended to avoid rolling back commits that have already been shared with others. Instead, consider creating a new commit that introduces the desired changes or communicating with collaborators to coordinate any necessary changes.

17. What is git bisect command ?

The `git bisect` command in Git is a helpful tool for finding the commit that introduced a bug or issue. It allows you to perform a binary search through the commit history to pinpoint the specific commit where the problem was introduced. The process involves marking specific points in the commit history as "good" or "bad" and using this information to narrow down the range of commits to search.

Here's an overview of how the `git bisect` command works:

1. **Start the bisect process:** Begin the bisect process by running the following command:

```
git bisect start
```

2. Mark the current commit as "bad": Identify the commit at which the bug is present and mark it as "bad" using the following command:

```
git bisect bad
```

3. Mark a known good commit: Specify a commit known to be free of the bug and mark it as "good". You can provide a commit hash, a branch name, a tag, or any other valid reference to indicate a known good commit:

```
git bisect good
```

4. Git performs a binary search: Git will automatically check out a commit halfway between the known good and bad commits. It then prompts you to test and determine if the bug is present in that commit. Based on your feedback, Git will mark it as "good" or "bad".

5. Repeat the process: Git will continue performing binary searches, checking out commits based on your feedback, until it finds the exact commit where the bug was introduced.

6. Finish the bisect process: Once the problematic commit has been identified, you can end the bisect process by running the following command:

```
git bisect reset
```

This command returns your repository to the original state, discarding any temporary changes made during the bisect process.

Conclusion :

The `git bisect` command helps you efficiently identify the commit that introduced a bug, making it easier to analyze, understand, and resolve the issue. It's a powerful tool for debugging and finding the root cause of problems in your codebase.