Contents

[Version 2](#_Toc186274496)

[Version control: 2](#_Toc186274497)

[What is staging in git? 2](#_Toc186274498)

[Comparing a file with previous commit using git diff: 2](#_Toc186274499)

[Commit, Tree and Blob 3](#_Toc186274500)

[Git Hash 3](#_Toc186274501)

[Git show~HEAD: 3](#_Toc186274502)

[Checking changes between two specific commits: 4](#_Toc186274503)

[When we want to see which made specific changes: 4](#_Toc186274504)

[Unstagging a file: 5](#_Toc186274505)

[Undoing changes from unstagged file: 5](#_Toc186274506)

[Unstagging and then Undoing: 5](#_Toc186274507)

[Getting commits info of specific number: 6](#_Toc186274508)

[Getting commit details on a specific date: 6](#_Toc186274509)

[Restoring File to a Specific Commit: 7](#_Toc186274510)

[Cleaning Repository 7](#_Toc186274511)

[Level of Setting: 8](#_Toc186274512)

[Changing Global Settings: 9](#_Toc186274513)

[Replacing git commands with short alias: 10](#_Toc186274514)

[Tracking aliases which we have made: 11](#_Toc186274515)

[Shell: 11](#_Toc186274516)

# Version

The contents of a file at a given point in time. It also includes metadata, or information associated with the file, such as the author, where it is located, the file type, and when it was last saved.

# Version control:

Version control is a group of systems and processes to manage changes made to documents, programs, and directories.

Version control isn't just for software. Anything that changes over time or needs to be shared can benefit from using version control. Version control allows us to track files in different states and let multiple people work on the same files simultaneously, a concept known as continuous development.

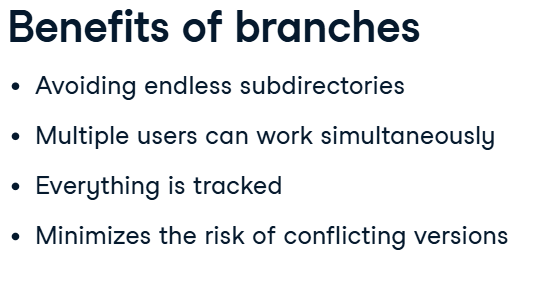
# Branches in Git:

In Git, a **branch** is a way to work on different versions of a project simultaneously. Think of a branch as a separate path where you can make changes without affecting the main project. The **main branch** (often called main or master) is the primary branch where the stable version of the project is usually stored.

When you create a new branch, you're essentially making a copy of the project at a specific point in time. You can make changes, test new features, or fix bugs in the new branch without worrying about breaking the main project. Once you're satisfied with your changes, you can merge the branch back into the main branch, combining the work.

Branches are helpful for teamwork and managing project updates. For example, if two people are working on different features, each can create a branch to work independently. Later, their changes can be merged together. This makes branches a powerful tool for organizing and experimenting in Git.

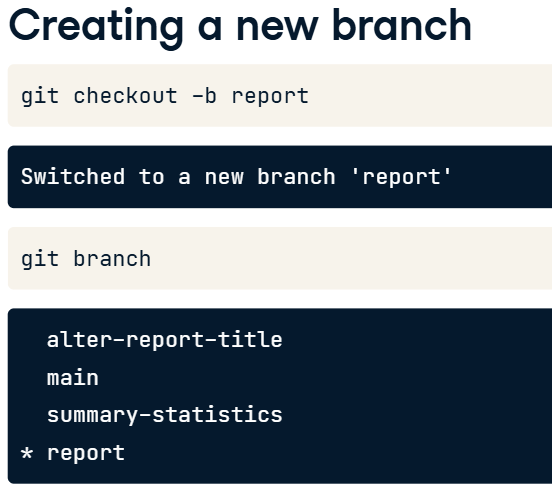
# Benefits of Branches:



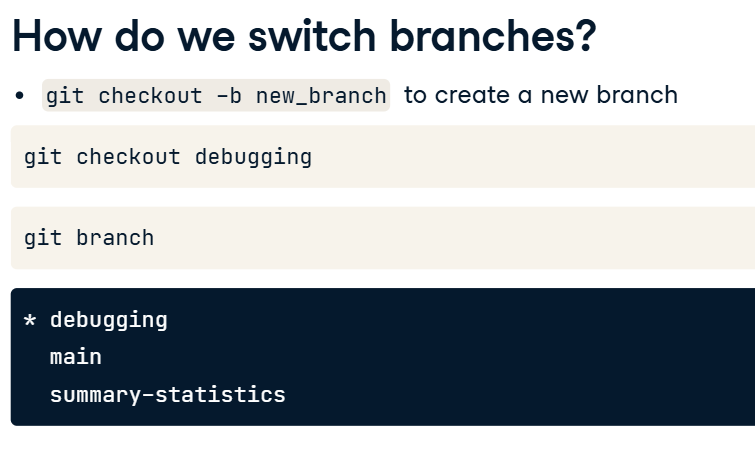
Identifying Branches:

We can see what branches exist for our project by executing the git branch command in the terminal. For this purpose, “Git branch” code is run. We see three branches in the output: alter-report-title, main, and summary-statistics. The summary-statistics branch has an asterisk next to it, which is how Git tells us we are currently in this branch.

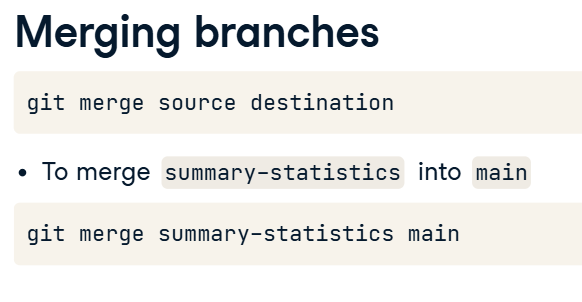
Making a new Branch:



Switching between branches:



Merging Branches into Main:



# What is staging in git?

In Git, **staging** refers to the process of preparing your changes (modifications, additions, deletions) to be included in the next commit. The staging area, also known as the **index**, acts as an intermediate step where changes are reviewed and organized before they are permanently recorded in the repository's history.

Git add . this command is used to put files in staging before permanent commit.

# Comparing a file with previous commit using git diff:

This command compares the current state of a specific file (file\_name) with the last committed version of that file.

**What It Shows:**

* **Changes made to the file** that have not been staged (added using git add) yet.
* Lines added, removed, or modified in the file.

**When to Use:**

* To review your edits before staging or committing them.
* To ensure the file changes are as expected.

# Commit, Tree and Blob

A **Commit** in Git is a snapshot of your project at a specific moment in time. It records the current state of all tracked files and the changes made since the previous commit. Each commit has a unique identifier (hash), a commit message describing the changes, and a reference to its parent commit(s), except the very first commit. Commits allow you to track, review, and revert changes as needed.

A **Tree** in Git represents the structure of your project (files and folders) at a particular commit. It acts like a directory listing, organizing the hierarchy of files and subdirectories. Each tree object points to file blobs and other subtrees, showing how the project is organized at that moment. Commits always point to a tree object that defines the state of the repository.

A **Blob** (Binary Large Object) in Git is the object that stores the actual content of a file. It contains the raw data of the file but doesn’t store its name or location within the project. Each unique version of a file’s content is stored as a separate blob. This ensures efficient storage by reusing blobs for unchanged content while saving only the differences when files are modified.

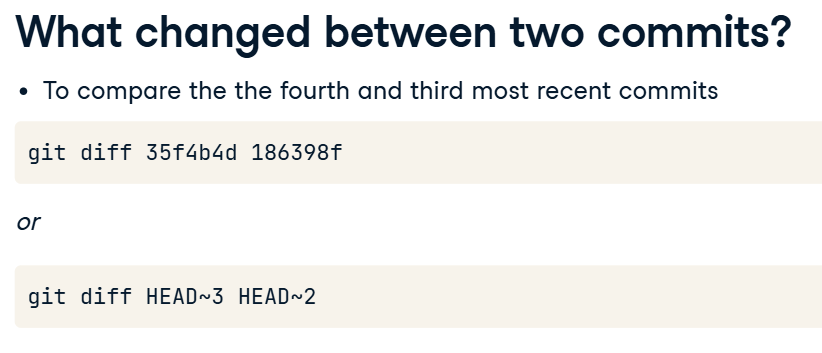
# Git Hash

A **Git Hash** is a unique identifier (also called a SHA-1 hash) generated for every commit in Git. It is a 40-character string that looks like a random mix of letters and numbers, but it uniquely represents the specific state of your project at the time of the commit. Git uses this hash to track changes and locate commits in the repository. The hash ensures data integrity, meaning any changes to the commit's content would result in a completely different hash.

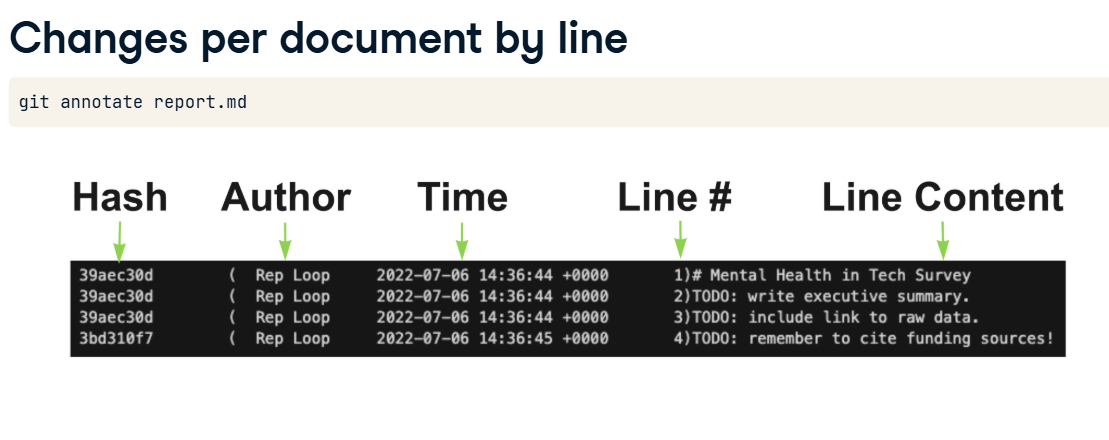
# Git show~HEAD:

The git show ~HEAD command is used to view details about the parent commit of the current HEAD (the most recent commit on the active branch). It displays information such as the commit hash, author, date, and the commit message, along with the differences introduced by that commit. This command is helpful for inspecting changes made in the immediate prior commit to the current state of your repository.

# Checking changes between two specific commits:



# When we want to see which made specific changes:



# Unstagging a file:



We can use git reset head to reset all files from stagging area without specifying any of them.

# Undoing changes from unstagged file:

The command git checkout -- file\_name is used to discard changes made to a specific file (file\_name) in your working directory. It restores the file to the state of the last commit (or the version from the staging area if it's staged). This means any uncommitted changes made to the file will be lost, and the file will revert to how it was when it was last committed. This is useful when you want to undo modifications to a file and return it to its last committed state.

But if we want to undo changes from all files, we have to run “git checkout .”.

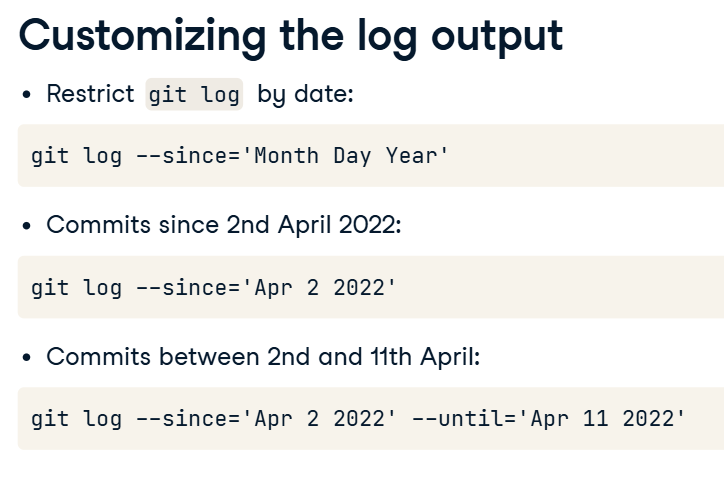
# Unstagging and then Undoing:

The command git reset HEAD is used to unstage files that have been added to the staging area, effectively removing them from the next commit. It does not modify the working directory or the commit history; it only resets the index (staging area) to match the last commit. After running this command, any changes that were staged (but not committed) will remain in your working directory, but they will no longer be staged for commit. This is useful if you mistakenly added files to the staging area and want to remove them without losing the changes in the files.

# Getting commits info of specific number:



# Getting commit details on a specific date:



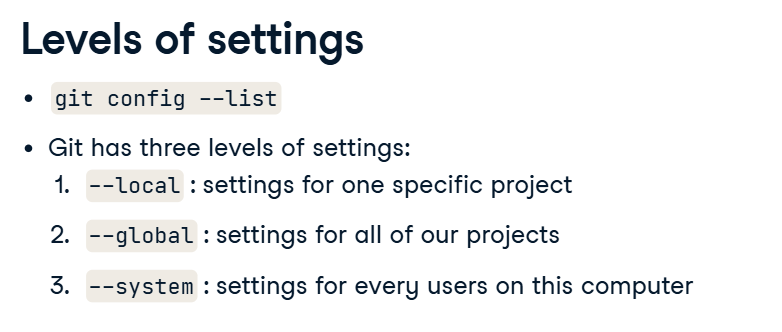
# Restoring File to a Specific Commit:



# Cleaning Repository



# Level of Setting:

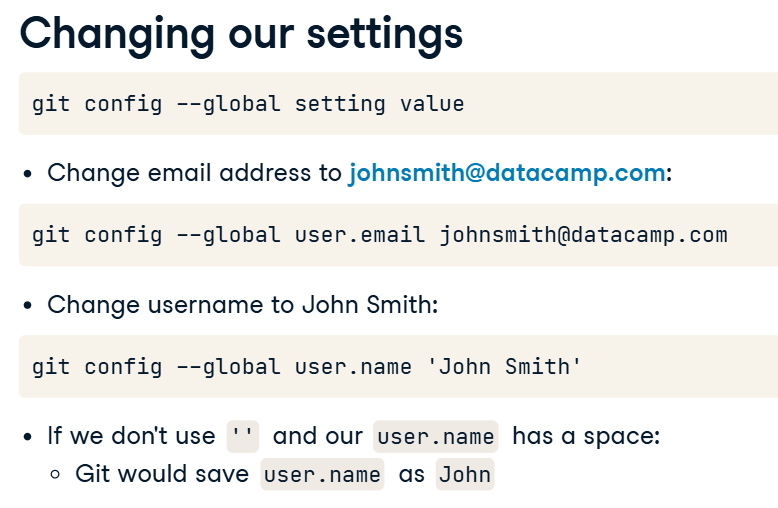


The git config command allows users to view and manage Git configuration settings. Using the --list option, it lists all effective settings by merging configurations from different scopes: system, global, and local. The output displays key-value pairs, such as the user name, email, and other repository-specific settings. This command is useful for checking all active configurations at a glance.

The --local option confines configuration settings to the current repository. These settings are stored in the .git/config file located within the repository. For example, the command git config --local user.name "Your Name" sets the user name exclusively for the repository it is executed in. Using git config --list --local displays only the settings specific to the local repository, making it ideal for repository-specific customizations.

The --global option applies configuration settings to all repositories of the current user, with settings stored in the ~/.gitconfig or ~/.config/git/config file. For instance, git config --global user.name "Your Name" sets the user name across all repositories managed by the user. The --system option, on the other hand, modifies the system-wide configuration file located at /etc/gitconfig. This applies settings like the default editor for all users and repositories on the system. Commands like git config --list --system require administrative privileges and provide a broader scope for configuration.

# Changing Global Settings:



# Replacing git commands with short alias:



# Tracking aliases which we have made:



# Shell:

A common method to use Git is via the shell, also known as the terminal. The shell is a program for executing commands.

Commands of shell:

Pwd to print in which directory we are

Ls list everything that is in the directory

Cd to change directory

Echo to create a file

Git –version to know the version of git we have installed

Git status to know how many files are in stagging area, how many are modified etc