**GIT/GITHUB**

**Source Code Management (SCM)** is the practice of managing and controlling changes to the source code of software projects. It's a subset of the broader **Software Configuration Management** (SCM), which not only tracks code but also manages other project artifacts such as documentation, configuration files, and build scripts.

**Key Functions of Source Code Management:**

1. **Version Control**:
   * SCM allows you to track changes made to the source code over time. Every change is stored as a new version, and you can always go back to previous versions of the code when necessary. This ensures that developers can work without the fear of losing work or breaking things permanently.
2. **Collaboration**:
   * Multiple developers can work on the same codebase simultaneously. SCM systems help manage the integration of changes from various contributors, ensuring that conflicts between different developers’ work are resolved in an organized way. For example, when two people change the same line of code, the system flags this conflict and asks the developer to resolve it.

**SCM (source code management)**

SCM stands for Source Code Management (also known as Version Control System or VCS). It refers to the tools and practices used to manage changes to a software project's source code over time.

SCM allows developers to track revisions, collaborate on code, and manage different versions of their codebase in an organized and efficient manner.

**Key Concepts in SCM:**

* **Version Control:**

Version control allows tracking of changes to files over time, providing a history of who made what changes and when.

This enables developers to roll back to previous versions, compare changes, and ensure that multiple developers can work on the same codebase without interfering with each other.

* **Repositories:**

A repository is where the source code and its history are stored. It can either be stored locally (on a developer’s machine) or remotely (on a server or cloud platform like GitHub, GitLab, or Bitbucket).

In a repository, code is organized into branches, which are separate lines of development.

* **Commits:**

A commit is a snapshot of changes made to the code. It contains the actual code changes along with a message describing the modification.

Each commit is identified by a unique hash and serves as a point in the history of the project.

* **Branching and Merging:**
  + **Branching** allows developers to create isolated versions of the code to work on features, fixes, or experiments without affecting the main project. Once work is completed, changes can be **merged** back into the main codebase.

1. **History and Audit**:
   * SCM keeps a detailed log of who made what changes, when, and why. This is useful for tracking the evolution of the code, troubleshooting issues, and understanding the decisions behind certain code changes.
2. **Backup and Recovery**:
   * Because every version of the code is stored in the SCM system, it acts as a backup for the source code. If anything goes wrong (e.g., accidental deletion of files, corrupted code), you can easily recover to a previous state.

Branching allows developers to create separate lines of development, making it easier to work on different features or bug fixes without disrupting the main codebase (often called main or master branch).

Merging is the process of combining changes from different branches back into one. This is commonly used when developers finish working on a feature and want to integrate it with the main project.

* **Pull Requests (or Merge Requests):**

A pull request is a way to propose changes to a codebase. A developer creates a pull request to request that their changes be merged into another branch (usually the main branch).

The pull request is reviewed by other developers, who can comment, suggest changes, and approve or reject the changes before merging.

* **Conflict Resolution:**

Merge conflicts occur when multiple developers make changes to the same part of a file in different branches. SCM tools help identify conflicts and allow developers to manually resolve them before merging changes.

* **Distributed vs. Centralized SCM:**

Distributed SCM (e.g., Git) means each developer has a local copy of the entire repository, including its history. This allows developers to work offline and perform operations like commits, branching, and merging locally before syncing with a central server.

Centralized SCM (e.g., Subversion, CVS) means there is a central repository that stores the code, and developers check out copies of the code to work on it. All commits are made directly to the central repository.

we have two types, they are

**1.DVCS: Distributed Version Control System**

In this we can get the working copy of the code into our local system

Example for DVCS: GitHub

**2.CVCS: Centralised Version Control System**

In this we cannot get the working copy of the code into our local system, but we can do the changes directly by connecting to the server

Example for CVCS: SVN (subversion)

**Popular Source Code Management Systems:**

1. **Git**: A distributed version control system that enables developers to work on their local machines and synchronize with a central repository when needed. It’s the most widely used SCM tool, often paired with platforms like **GitHub**, **GitLab**, or **Bitbucket** for collaborative projects.
2. **Subversion (SVN)**: A centralized version control system, where the source code is stored in a central repository. Developers check out code from the central server, make changes, and commit them back to the repository.
3. **Mercurial**: Similar to Git in that it is a distributed version control system. It’s another option for managing source code, though less commonly used than Git.
4. **Perforce**: Often used for larger projects or enterprise environments, particularly those with large files (e.g., game development, multimedia) due to its efficiency in handling large-scale repositories.

**Benefits of Source Code Management:**

* **Collaboration and Teamwork**: Multiple developers can work on the same project, even from different locations.
* **Consistency**: It ensures the team is working on the latest version of the code and can manage different environments and releases.
* **Traceability**: You can always track who made specific changes and why (e.g., through commit messages), which is valuable for debugging, auditing, and understanding the evolution of the project.
* **Backup**: Changes are safely stored, so there’s no risk of losing work.

In essence, **Source Code Management** is an essential practice for organizing and controlling the development of software, ensuring teams can collaborate effectively and manage the lifecycle of their codebase.

**Benefits of SCM:**

* **Collaboration:** Multiple developers can work on the same project simultaneously without overwriting each other’s work, thanks to branching and merging.
* **Version History:** SCM keeps a history of every change, so you can track who made what changes and when. This is essential for debugging, auditing, and understanding the evolution of a project.
* **Code Integrity:** By using version control, you can ensure the integrity of the codebase, rollback to a stable state if something goes wrong, and manage code conflicts effectively.
* **Backup:** SCM systems provide a built-in backup of the project’s history, so if a mistake happens, it’s easy to retrieve previous versions.
* **Traceability:** SCM enables traceability of changes and decisions. You can associate specific changes with bugs, features, or user stories, making it easier to manage and maintain software projects.

**SCM Workflow Example:**

* **Clone:** You clone a remote repository to get a local copy of the project.
* **Create a Branch:** Create a new branch to work on a feature or bug fix.
* **Make Changes:** Modify the code in your branch.
* **Commit:** Commit the changes to your local repository with a descriptive message.
* **Push:** Push your changes to the remote repository (e.g., GitHub).
* **Pull Request:** Create a pull request to merge your changes into the main branch, so other team members can review and approve it.
* **Review and Merge:** After review, your changes are merged into the main branch.
* **Deploy:** The new code is deployed to the production environment.

**Git/GitHub**

Git and GitHub are both tools used for version control, primarily in software development. Here's a breakdown of each:

**Git:**

Git is a **version control system** that helps developers manage changes to their codebase over time. It tracks the history of changes made to a project, allowing you to:

* **Collaborate**: Multiple developers can work on the same codebase without conflicting with each other.
* **Versioning**: You can keep track of different versions of the code, making it easy to go back to previous versions if needed.
* **Branching and Merging**: Git allows you to create branches (separate versions of the code) to work on features or fixes independently, and then merge them back into the main codebase.

Git is a **command-line tool**, but there are also graphical interfaces available for it, such as GitKraken or SourceTree.

**GitHub:**

GitHub is a **cloud-based platform** that uses Git for version control and adds collaboration features on top of it. GitHub is a place where developers can host their Git repositories, share code, and collaborate on projects with other developers. Some key features of GitHub include:

* **Remote Repositories**: GitHub provides cloud hosting for Git repositories, meaning your code is stored remotely and can be accessed from anywhere.
* **Collaboration**: GitHub makes it easy for multiple developers to work on the same project, track issues, review code, and even contribute to other people’s projects.
* **Pull Requests**: This is a key GitHub feature, where one can propose changes to a project. Others can review, comment on, and approve those changes before merging them into the main codebase.
* **GitHub Actions**: GitHub provides automation tools (like CI/CD) to automate tasks like testing, building, and deploying code.

In short, **Git** is the tool that tracks changes in your code, while **GitHub** is the platform where you can store, share, and collaborate on Git-managed projects.

**Git**

GitHub is a web-based platform for version control and collaboration that allows developers to store, manage, and track their code projects.

It is built around Git, a version control system created by Linus Torvalds (the creator of Linux). GitHub enhances Git by providing an

easy-to-use interface and a variety of collaborative tools for software development teams. Here's a breakdown of what GitHub is and what it does:

**Key Features of GitHub:**

* **Version Control with Git:**

GitHub hosts repositories (repos) that track changes made to code over time. Each change (commit) is recorded and can be revisited, helping teams collaborate and maintain a history of their project.

Git allows developers to create branches, make changes, and merge them back into the main project. This ensures that changes don’t interfere with each other, which is especially useful for collaborative work.

* **Repository Hosting:**

A repository is a storage space for your project. It includes all the project files, history of changes, and other configuration settings.

GitHub allows users to create public or private repositories. Public repositories are open to everyone, while private ones are restricted to specific users or teams.

* **Collaboration:**

GitHub is designed for collaboration, allowing multiple people to work on the same project. Users can fork repositories (create their own copy) and propose changes using pull requests.

Pull requests are how developers propose changes to a project. They allow others to review code, discuss changes, and merge them into the main project if accepted.

* **Issue Tracking:**

GitHub provides a built-in issue tracker where users can report bugs, request features, or track tasks. This helps teams organize work, prioritize tasks, and keep track of progress.

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

GitHub integrates with CI/CD services (e.g., GitHub Actions, Travis CI) to automatically build, test, and deploy code when changes are pushed to a repository.

This ensures that code is thoroughly tested and deployed in an automated and efficient manner.

* **Code Review and Discussions:**

GitHub allows for code reviews, where team members can review each other's code, suggest changes, and ensure quality before merging it into the main codebase.

Discussions enable teams to communicate about project features, bugs, or any aspect of the development process in a more structured way.

* **GitHub Pages:**

GitHub allows users to host static websites directly from a GitHub repository using GitHub Pages. This feature is often used for project documentation or personal portfolios.

* **Actions and Workflows:**

GitHub Actions enable users to automate their workflows, such as automatically testing code or deploying software when certain events (e.g., push, pull request) occur in the repository.

* **Security:**

GitHub provides tools for security, including features like dependency scanning, vulnerability alerts, and secret management. It helps ensure that your project remains secure and up to date.

* **Community and Open-Source Projects:**

GitHub hosts millions of open-source projects, making it a hub for community-driven software development. Anyone can contribute to public projects by forking, making changes, and submitting pull requests.

It also supports social features like stars (similar to "likes"), followers, and activity feeds, which help users discover and engage with popular or interesting projects.

**Why GitHub Is Popular:**

* **Ease of Use:**

GitHub simplifies Git by offering a web interface that makes version control, collaboration, and project management easier, even for people new to Git.

* **Collaboration:**

It provides powerful tools for teams to collaborate on projects, even if they are geographically dispersed.

* **Community and Open Source:**

GitHub is home to millions of open-source projects, making it a vital platform for the open-source community. It also allows anyone to contribute to these projects.

* **Integration:**

GitHub integrates with various tools and services, including CI/CD pipelines, cloud providers, and IDEs (Integrated Development Environments).

**Example of GitHub Workflow:**

**Create a Repository:** A user creates a repository on GitHub to store the project.

**Clone Repository:** The user clones the repository to their local machine to work on it.

**Make Changes Locally:** The user makes changes to the code on their local machine (e.g., adding a new feature, fixing a bug).

**Commit Changes:** The user commits changes locally with a message describing what was done.

**Push Changes:** The changes are pushed back to the GitHub repository so that they are stored remotely.

**Open a Pull Request:** If working on a collaborative project, the user opens a pull request to propose merging their changes into the main codebase.

**Code Review:** Team members review the changes, suggest edits, and approve the pull request.

**Merge and Deploy:** Once approved, the changes are merged into the main branch and deployed to production (if configured with CI/CD).

**GitHub** is a source code management tool, it will store the developed code.

In SCM (source code management) we have two types, they are

1. CVCS: Centralised Version Control System

In this we cannot get the working copy of the code into our local system, but we can do the changes directly by connecting to the server

Example for CVCS: SVN (subversion)

2. DVCS: Distributed Version Control System

In this we can get the working copy of the code into our local system

Example for DVCS: GitHub

URL of GitHub is: https://guthub.com/

In GitHub code is stored in Repositories

Creating GitHub account:

**GitBash:**

Git Bash is a command-line interface (CLI) application for Windows that provides a Bash emulation environment where you can run Git commands and use Bash scripts.

It combines the capabilities of Git with a Unix-like shell environment, giving developers access to the familiar Bash commands and tools (common in Linux and macOS) on a Windows machine.

**Git Bash Working:**

* **Installation:**

You can download and install Git Bash as part of the Git for Windows package from git-scm.com.

During installation, you’ll be able to choose the components you want, including Git Bash.

* **Using Git Bash:**

Once installed, you can open Git Bash by either searching for it in the Start menu or right-clicking inside a folder and selecting Git Bash Here.

This opens the command line interface where you can enter Git and shell commands.

* **Using Git Bash for Git Operations:**

Once Git Bash is open, you can perform all standard Git operations, like initializing repositories, cloning repositories, making commits, and pushing/pulling from remotes.

Since Git Bash supports Unix commands, you can also perform tasks like navigating your file system, editing files, and scripting.

* **Accessing Remote Repositories:**

Git Bash integrates with services like GitHub or GitLab and allows you to interact with your repositories over SSH or HTTPS.

You can set up your SSH keys (using ssh-keygen) and use them to authenticate with remote repositories securely.

* First, we need to configure the user name

->syntax: git config --global user.name "user\_name"

* To configure the email for the user

->syntax: git config --global user.email "mail\_id"

* To check the configured name, we can user

->syntax: git config --global user.name

* To check the configured email, we can user

->syntax: git config --global user.email

* To install the git commands

->syntax: git init

* To clone the repository from the github, ->click on the "Code" dropdown and copy the git url, then we can use the below command

->syntax: git clone git\_url

->example: git clone https://github.com/user\_name/repo.git

* then the repo will download into our local machine
* then change the directory to downloaded repository and check the files present in that repository

->syntax: ls

touch newfile

git add .

git commit -m "commit message"

git push origin main

git status

In Git, a repository (often abbreviated as repo) is a storage space where your project files and their revision history are kept. There are generally two main types of repositories: local repositories and remote repositories. Additionally, repositories can be either public or private depending on their accessibility.

Here’s a detailed explanation of these types:

**1. Local Repository**

A local repository is the repository on your local machine (computer). When you clone a repository from a remote server (like GitHub), the local repository contains a working directory and a .git directory that stores the version control information.

**Key Characteristics:**

**Your working copy**: The local repository is where you make changes to your files, create commits, and perform version control tasks.

**Version history:** It contains a full history of your project, which you can access and interact with.

No internet connection required: You can work in a local repository even without an internet connection.

**Git metadata**: It stores all the version control information and metadata in the .git folder.

**Example:**

You can create a local Git repository on your computer with the following command:

git init

This will create a .git directory in the current folder, turning it into a Git repository.

**2. Remote Repository**

A remote repository is a repository hosted on a server or online platform such as GitHub, GitLab, Bitbucket, or a self-hosted Git server. Remote repositories are typically used for collaboration, allowing multiple developers to contribute to the same project from different locations.

**Key Characteristics:**

**Collaboration:** Remote repositories allow multiple developers to collaborate by pushing and pulling code.

**Backup:** They serve as a backup of your code, ensuring that your work is safe in case your local machine fails.

**Centralized storage:** Remote repositories are the central point where the team can share and sync their work.

**Internet connection required:** To interact with remote repositories, an internet connection is required to push or pull changes.

**Example:**

To clone a remote repository from a platform like GitHub:

: git clone https://github.com/username/repository.git

Once cloned, the repository will be both a local copy and a remote connection. You can push your changes to and pull updates from the remote repository.

**3. Public Repository**

A public repository is a type of repository that is visible to anyone. Anyone with the URL can access, clone, and sometimes contribute to the repository (depending on the permissions). Public repositories are often used for open-source projects where developers from around the world can contribute to the project.

**Key Characteristics:**

**Open access:** Anyone can view, clone, and contribute to the project (if permissions allow).

**Open-source:** Commonly used for open-source projects, making it easy for developers to collaborate.

**Visibility:** Your code is visible to everyone, which can be great for showcasing work or collaborating with others.

**Example:**

A project on GitHub like React or Node.js is often open and public, allowing anyone to fork and contribute to the project.

**4. Private Repository**

A private repository is a repository that is only accessible to specific users or teams. Unlike public repositories, private repositories restrict access to the code, allowing you to control who can view, clone, or contribute to your project.

**Key Characteristics:**

**Restricted access:** Only users with explicit permissions (such as collaborators or team members) can access the repository.

**Confidentiality:** Private repositories are useful for storing sensitive code or internal projects that should not be made public.

**Access control:** The repository owner or admin can control who has read, write, or admin access to the repository.

**Example:**

On platforms like GitHub, you can create a private repository and invite specific collaborators to work on it.

**Summary of Git Repository Types:**

Type Description Access

Local Repository A repository stored on your local machine for managing code and version history. Only accessible locally.

Remote Repository A repository hosted on a remote server (e.g., GitHub, GitLab) that can be accessed by multiple users. Accessible online via the internet.

Public Repository A repository that is open for anyone to view, clone, and contribute to. Often used for open-source projects. Accessible by anyone.

Private Repository A repository that restricts access to authorized users only. Typically used for private or confidential projects. Accessible only by authorized users.

**Workflow with Repositories:**

Local and Remote: Most modern development workflows involve working with both local and remote repositories. You typically clone a remote repository to your local machine, make changes locally, and then push those changes to the remote repository.

**Example Workflow:**

Clone a repository from GitHub (or other platforms):

: git clone https://github.com/username/repository.git

Make changes locally to the files in your working directory.

Commit changes locally:

: git add .

: git commit -m "Made changes"

Push changes to the remote repository:

: git push origin main

By using a combination of local and remote repositories, Git enables efficient collaboration and version control, making it easier to manage and track changes over time.

we have two types of repositories in github, they are

**1.Public:** In this anyone on the internet can see this repository and you choose who can commit to the Repositories

We can change the public repository to private repository

->open the settings in the repository and in the Danger Zone "change visibility" to "change to private"

**2.Private:** In this you choose who can see and commit to the repositories

We can change the private repository to public repository

-> open the settings in the repository and in the Danger Zone "change visibility" to "change to public"

there are two ways to do changes in repository

**1. We can do the changes directly in the GitHub**

we can create a new file and add the code in it

we can upload the file from local system to the repository

**2. We can do the changes from the local system**

Working Directory -> Staging Area -> Commit Stage -> Repository

Clone the repo from github to local system (working directory) through gitbash

copy the git url from the code dropdown box

then open the gitbash and clone the repository

->syntax: git clone git\_url

->example: git clone https://github.com/username/repo.git

To check the repository in the local system

-> syntax: ls

then change the directory to downloaded repository

->syntax: cd repository\_name

This is the working directory, we can make the required changes to the code(changes not staged for commit)

In the working directory we have three concepts, they are

i.modified: updated the existing file

ii.deleted: deleted the existing file in github

iii.untracked: newly created file

->example: adding a file

->syntax: touch file2

after this we need to move to the Staging area(changes to be committed)

->syntax: git add filename

->example: git add . or git add file2

Then we move to the Commit stage

->syntax: git commit -m "commit\_message"

->example: git commit -m "file2 added"

then we move to the Repository

->syntax: git push origin branch\_name

->example: git push origin main

To check the commit history

->syntax: git log

To see the commits for specific time, we can use

->syntax: git log --since=number\_of\_days

->example: git log --since=3days

To see the last specific number of commit logs, we can use

->syntax: git log --max-count=number\_of\_commits

->example: git log --max-count=3

To see the commits for a specific file, we can use

->syntax: git log file\_name

->git log file1

To see the commits of a specific user, we can use

->syntax: git log --author=user\_name

->example: git log --author=user1

To check the logs for specific period fo time, we can use

->syntax: git log --since=starting\_date(yyyy-mm-dd) --until=ending\_date(yyyy-mm-dd)

->example: git log --since=2024-01-01 --until=2024-12-31

To see the logs from specific date to present date, we can use

->syntax: git log --since=starting\_date(yyyy-mm-dd) --until=today

->example: git log --since=2024-01-01 --until=today

To check the status of the changes, we can use

->syntax: git status

To check the status of particular file, we can use

->syntax: git status file\_name

->example: git status file1

To see the used commands, we can use 'history' command

->syntax: history

**git fetch:** It will fetch the changes from the GitHub but it will not disturb the working directory

->syntax: git fetch

**git merge:** It will merge the updated code in the github with the local system

->syntax: git merge

**git pull:** git fetch + git merge

It will pull the changes from the GitHub and saves the changes in local system or Working Directory

To update the changes made in the github to local system, we can use

->syntax: git pull

**Restore:** this command is used to restore the file from staging area to working directory

->syntax: git restore --staged file\_name

->example: git restore --staged file1

**Reset:** this command is used in commit stage, the Reset stage will delete from the logs

we have three types in this

**i. soft:** the changes will come from 'commit stage' to 'staging area'

->syntax: git reset --soft previous\_commit\_id

->example: git reset --soft a2626e2932

**ii. mixed:** the changes will come from 'commit stage' to 'working directory'

->syntax: git reset --mixed previous\_commit\_id

->example: git reset --mixed 35a00e72de2a

**iii. hard:** is used to remove the later commits

->syntax: git reset --hard previous\_commit\_id

->example: git reset --hard b2d7bd921e0dc

**Branch:**

In GitHub (or Git in general), a **branch** is a separate line of development that allows you to work on features, fixes, or experiments independently of the main codebase (often called main or master). Branching helps to isolate changes so that the main codebase remains stable.

**1. Creating a Branch:**

* **In GitHub (Web Interface):**
  1. Navigate to the repository.
  2. At the top of the repository, you'll see a button that says "main" (or whatever the default branch is).
  3. Click the dropdown and type the name of your new branch, then hit "Create branch."
* **In Git (Command Line):** To create a new branch locally:

: git checkout -b new-branch-name

**2. Switching Between Branches:**

* **In GitHub (Web Interface):** Simply go to the repository, click the branch dropdown, and select the branch you want to switch to.
* **In Git (Command Line):** To switch to an existing branch:

: git checkout branch-name

**3. Committing Changes to a Branch:**

After creating or switching to a branch, you can make changes and commit them:

: git add .

: git commit -m "Your commit message"

**4. Pushing Changes to GitHub:**

Once you’ve committed your changes locally, push the branch to GitHub:

: git push origin branch-name

**5. Merging a Branch:**

After finishing your work on a branch, you can merge it back into the main branch (or any other branch).

* **In GitHub (Web Interface):**
  1. Go to the "Pull Requests" tab in your repository.
  2. Click "New Pull Request" and select the base branch and the compare branch.
  3. Click "Create Pull Request" to initiate the merge process.
* **In Git (Command Line):** First, switch to the branch you want to merge into (e.g., main):

: git checkout main

Then merge the changes from the feature branch:

: git merge branch-name

**6. Deleting a Branch:**

After the merge is complete and you no longer need the branch, you can delete it:

* **In GitHub (Web Interface):** After a pull request is merged, GitHub gives you an option to delete the branch directly.
* **In Git (Command Line):** To delete a branch locally:

: git branch -d branch-name

To delete it remotely:

: git push origin --delete branch-name

**Why Use Branches?**

* **Isolated Work:** You can develop features or fix bugs without affecting the main codebase.
* **Collaboration:** Teams can work on different features or fixes at the same time without stepping on each other's toes.
* **Code Review:** Branches allow for pull requests, enabling other team members to review and discuss changes before merging them.

**Branch:**

To see the branch list, we use

->syntax: git branch

To create new branch, we use

->syntax: git branch branch\_name

->example: git branch feature1

To change the branch in git bash, we can use

->syntax: git checkout branch\_name

->example: git checkout branch-1

To merge the branch, we can use

->git merge branch\_name

->example: git merge feature1

To push the changes into github, we can use

->syntax: git push origin branch\_name

To delete the branch, we can use

->syntax: git branch -d branch\_name

->example: git branch -d branch-1

->syntax: git push origin --delete branch\_name

->example: git push origin --delete branch-1

To create Repository from git bash

->syntax: git remote add origin url (https://github.com/github\_user\_name/Repository\_name)

->example: git restore add origin https://github.com/dinesh-4136/Repo-git.git

To push the repository to Github

->syntax: git push -u origin branch\_name

->example: git push -u origin master

**Tags:**

In GitHub (and Git in general), tags are references to specific points in your repository's history, usually representing a release or a notable milestone. Tags are similar to branches, but they don’t change over time, meaning they are typically used to mark specific commits for important events like software releases or other major changes.

**Types of Tags:**

1. **Lightweight Tag:**
   * It's like a bookmark to a commit. It doesn't have any extra information (like the tagger's name, date, or message). It's simply a pointer to a commit.
2. **Annotated Tag:**
   * This is more useful for marking releases because it stores extra metadata, like the name of the person who created the tag, the date, and a tag message. Annotated tags are often signed with GPG to ensure their authenticity.

**How to Use Tags in GitHub (or Git):**

**1. Creating a Tag:**

You can create tags either locally via the command line or through the GitHub web interface for releases.

* **Creating a Lightweight Tag (Command Line):**

: git tag tag-name

* **Creating an Annotated Tag (Command Line):**

: git tag -a tag-name -m "Your tag message"

The -a flag creates an annotated tag, and -m allows you to include a message.

* **Pushing Tags to GitHub (Command Line):** After creating a tag, you need to push it to GitHub:

: git push origin tag-name

To push **all** tags at once:

: git push origin –tags

**2. Viewing Tags:**

* **In GitHub (Web Interface):** Tags are listed under the "Releases" section of your repository. If you click on "Releases," you will see a list of tags and the associated release notes.
* **In Git (Command Line):** To list all tags:

: git tag

**3. Checkout a Tag:**

You can check out a tag, which will put your working directory in a "detached HEAD" state (meaning you're not on any branch):

: git checkout tag-name

This is useful when you want to view or work with the state of the code at the time of the tag.

**4. Deleting a Tag:**

* **Locally (Command Line):**

: git tag -d tag-name

* **Remotely (Command Line):**

: git push origin --delete tag tag-name

**5. Creating a Release from a Tag (GitHub Interface):**

* Go to the "Releases" section of your GitHub repository.
* Click on "Draft a new release."
* In the "Tag version" dropdown, choose the tag you created, or type a new tag name.
* Fill in release notes and click "Publish release."

**Why Use Tags?**

* **Marking Releases:** Tags are perfect for marking stable release versions (e.g., v1.0.0, v2.1.1).
* **Milestones:** Tags can be used to mark major points in a project, like the completion of a feature or a bug fix.
* **Referencing a Specific Commit:** If you want to refer to the code at a specific point in time (e.g., a bug occurred in a version), you can use tags to easily check out the exact commit.

**Example Workflow with Tags:**

1. **Create a tag for a new version:** After finishing development, you might want to tag the commit as v1.0.0:

: git tag -a v1.0.0 -m "Release version 1.0.0"

1. **Push the tag to GitHub:**

: git push origin v1.0.0

1. **Create a release on GitHub:** You can then go to the "Releases" section on GitHub and create a new release, selecting the v1.0.0 tag, and adding release notes describing the new version.

**Tags:** To store the particular release code in zip (like versions)

To create a tag, when we create tag then it will create a zip (tar.gzip or zip) file up to the particular code(version)

->syntax: git tag tag\_name

->example: git tag v1.1.1

Then we need to push the tag,

->syntax: git push origin tag\_name

->example: git push origin --tags (to push all the tags at a time)

->example: git push origin v1.1.1

To see the list of tags,

->syntax: git tag --list

To delete the tag in local system

->syntax: git tag --delete tag\_name

->example: git tag --delete v2.2.2

To delete the tag in github

->syntax: git push origin tag\_name

->example: git push origin --delete v1.1.1

**Amend:**

To amend a commit in Git, you use the git commit --amend command. This allows you to modify the most recent commit by adding new changes or updating the commit message.

Here’s how to use it:

**1. Amend the most recent commit with changes:**

* If you want to amend your last commit by adding or modifying files, first stage the changes using git add:

: git add <file> # Or use 'git add .' to stage all changes

* Then, run the amend command:

: git commit –amend

* This opens the default editor to modify the commit message. You can either keep the original message or change it.

If you don’t want to modify the commit message but only want to add the staged changes, you can use:

: git commit --amend --no-edit

**2. Amend the commit message only:**

* If you don't want to change any files, just modify the commit message:

: git commit –amend

* This opens the editor to change the commit message of the most recent commit.

**Important Notes:**

* **Amending a commit rewrites history**, so if you’ve already pushed the commit to a remote repository, you'll need to force-push the changes with git push --force or git push -f (this can be dangerous if others have pulled from the remote).
* Be careful with amending commits that have already been pushed to shared repositories, as it can cause issues for others working on the same branch.

**Rabase:**

The git rebase command in Git is used to integrate changes from one branch into another. It's particularly useful when you want to "reapply" your changes on top of another branch, usually to maintain a linear history.

Here are some common use cases and examples for git rebase:

**1. Rebase your current branch onto another branch**

If you're working on a feature branch and want to incorporate changes from main (or any other branch) into your feature branch, you can rebase your feature branch onto main.

: git checkout feature-branch # Switch to your feature branch

: git rebase main # Rebase your feature branch onto main

This moves your feature branch commits on top of the main branch. If there are conflicts, Git will pause the rebase and prompt you to resolve them.

**2. Rebase interactively (for squashing or reordering commits)**

The -i flag allows you to perform an interactive rebase, which is helpful for squashing commits, reordering them, or editing commit messages.

: git rebase -i HEAD~3 # Rebase the last 3 commits interactively

This will open your default text editor with a list of the last 3 commits. You can change the command for each commit (e.g., pick, squash, edit, reword, drop) to modify how they are applied.

* **Pick**: Keep the commit as is.
* **Squash**: Combine the commit with the previous one.
* **Reword**: Edit the commit message.
* **Edit**: Modify the commit itself.
* **Drop**: Remove the commit.

Once you've made the necessary changes and saved the file, Git will apply the changes.

**3. Rebase a branch onto another branch and resolve conflicts**

If there are conflicts during a rebase, Git will pause and prompt you to resolve them.

* First, resolve the conflicts in the files.
* After resolving the conflicts, stage the changes with:

: git add <file>

* Continue the rebase with:

: git rebase –continue

* If you decide you don’t want to continue the rebase, you can abort it:

: git rebase –abort

**4. Rebase a branch onto another branch and update the remote**

Once you have successfully rebased your branch, you’ll likely need to force-push to the remote repository (since the commit history has changed).

: git push –force

**Warning**: Be cautious when using --force because it rewrites history on the remote branch, which can affect other collaborators.

**5. Rebase onto a specific commit**

If you want to rebase a branch starting from a specific commit (not just the latest commit), you can specify the commit hash:

: git rebase <commit-hash>

This rebases the current branch onto the commit you specify.

**Key Differences: merge vs. rebase**

* **Merge** preserves the commit history, adding a merge commit.
* **Rebase** rewrites history, applying your changes on top of another branch’s history for a cleaner, linear history.

**Summary**

* **Regular rebase**: git rebase <branch> to move your commits on top of another branch.
* **Interactive rebase**: git rebase -i HEAD~n to edit, squash, reorder, or drop commits.
* **Force push after rebase**: git push --force to update the remote after a rebase.

**Stash:**

The git stash command in Git is used to temporarily save changes in your working directory that you are not ready to commit yet. This is useful when you need to switch branches but don't want to commit unfinished work or have it interfere with the work on the other branch.

Here are the common git stash commands and how to use them:

**1. Stash your changes**

To stash the changes, you have made (both staged and unstaged), you can use:

: git stash

This will save your changes (including untracked files if specified) and revert your working directory to match the last commit.

**2. Stash changes with a message**

You can provide a message to help identify the stash later:

: git stash save "Your message here"

This helps you keep track of multiple stashes, especially when working on several tasks at once.

**3. Stash only staged or unstaged changes**

* To stash only the **staged** changes (those added with git add), you can use:

: git stash -k # Or --keep-index

* To stash only the **unstaged** changes (those not added with git add), you can use:

: git stash -p # Or –patch

**4. List all stashes**

If you have multiple stashes, you can view a list of them with:

: git stash list

This will show a list of all the stashes you have saved, along with their index (e.g., stash@{0}, stash@{1}, etc.) and any message you included.

**5. Apply a stash**

To apply the most recent stash to your working directory, use:

: git stash apply

If you want to apply a specific stash from the list, use its index:

: git stash apply stash@{2}

**6. Pop a stash**

git stash pop is similar to git stash apply, but it also **removes the stash** from the list after applying it.

: git stash pop

If you want to pop a specific stash, use:

: git stash pop stash@{1}

**7. Drop a stash**

If you want to delete a specific stash without applying it, you can use:

: git stash drop stash@{2}

To remove all stashes at once, you can use:

: git stash clear

**8. Show the contents of a stash**

To view the changes saved in a stash, you can use:

: git stash show stash@{0}

If you want a more detailed view (e.g., the diff), use:

: git stash show -p stash@{0}

**9. Stash untracked files**

By default, git stash only stashes tracked files. If you want to stash **untracked** or **ignored** files as well, you can use the -u or -a option:

* To stash untracked files:

: git stash -u

* To stash both untracked and ignored files:

: git stash -a

**Summary of key commands:**

* **Stash changes**: git stash
* **Stash with a message**: git stash save "message"
* **List stashes**: git stash list
* **Apply most recent stash**: git stash apply
* **Apply specific stash**: git stash apply stash@{n}
* **Pop the most recent stash**: git stash pop
* **Pop a specific stash**: git stash pop stash@{n}
* **Drop a stash**: git stash drop stash@{n}
* **Clear all stashes**: git stash clear
* **Show stash contents**: git stash show stash@{n}
* **Stash untracked files**: git stash -u
* **Stash all (including ignored)**: git stash -a

**Collaborator:**

In Git, a **collaborator** typically refers to someone who has been given access to a repository to contribute to its codebase. This can include different levels of access, such as read, write, or admin privileges, depending on how the repository owner configures the permissions.

Here's a breakdown of how collaboration in Git works:

1. **Access Control**:
   * **GitHub** (or other Git hosting services like GitLab, Bitbucket) allows repository owners to invite collaborators.
   * Collaborators can be added with different permission levels, such as:
     + **Read**: Can view and clone the repository.
     + **Write**: Can push changes to the repository (edit the code).
     + **Admin**: Can manage the repository settings, add or remove collaborators, etc.
2. **Adding Collaborators on GitHub**: If you own a repository on GitHub, you can add collaborators by:
   * Going to your repository's page.
   * Clicking on the **Settings** tab.
   * Selecting **Manage access** on the left sidebar.
   * Clicking on **Invite a collaborator**, and then entering their username or email to send the invitation.
3. **Fork and Pull Request Workflow**: If you're working with a public repository or don’t have direct write access, the usual collaboration method involves:
   * **Forking** the repository: This creates a personal copy of the repository in your GitHub account.
   * **Cloning** your fork: This copies the repository to your local machine.
   * Making changes: After making changes on your local machine, you push them to your fork.
   * **Pull request**: You create a pull request to propose changes to the original repository.
4. **Branches**: It's common for collaborators to work in separate **branches** instead of working directly on the main branch. This helps in:
   * Isolating features or bug fixes.
   * Reducing the risk of breaking the main codebase.
5. **Pull Requests**:
   * When collaborators make changes, they usually create a **pull request** (PR) so that the repository owner can review and discuss the changes before merging them into the main branch.

**Collaborator:** The Collaborator who can access the others repo, with the Collaborator access

we can give the contributor access to the other github user for our repository

->settings->Collaborator->add people->add to repository

>then contributor user needs to accept the collaborator request

>after adding, when the user made any changes to the repo, then only the user becomes as the contributor

**Personal Access Token (PAT):**

A **Personal Access Token (PAT)** in Git is an authentication method used to securely access your Git repository over HTTPS instead of using your GitHub or GitLab password. It’s commonly used as a replacement for passwords when interacting with Git hosting services like GitHub, GitLab, and Bitbucket.

Since August 13, 2021, GitHub has discontinued the use of account passwords for accessing Git repositories over HTTPS. Now, you must use a **Personal Access Token (PAT)** to authenticate when pushing or pulling changes from your repository.

**How to Use a Personal Access Token (PAT) with Git:**

**1. Create a PAT on GitHub:**

* Go to GitHub and log in.
* Click on your **profile picture** in the upper-right corner and select **Settings**.
* On the left sidebar, click **Developer settings**.
* Under **Developer settings**, click **Personal access tokens**.
* Click **Generate new token**.
* You’ll be prompted to give your token a name, select its expiration (can be set to never expire), and choose the **scopes** (permissions) for the token (e.g., repo, workflow, etc.).
* Once you've selected the appropriate permissions, click **Generate token**.
* **Important**: Make sure to copy and save the token immediately, as you won’t be able to see it again after you leave the page.

**2. Use the PAT with Git:**

When performing Git operations that require authentication (like git push or git pull), you'll be asked for your credentials:

* For **username**, enter your GitHub **username**.
* For **password**, enter your **Personal Access Token** (not your GitHub password).

**Example of using PAT with Git:**

1. **Clone a repository** (for the first time):

: git clone <https://github.com/username/repository.git>

When prompted for your password, enter the PAT instead of your password.

1. **Push changes to the repository**:

: git push origin main

Again, you'll be prompted for your GitHub username and the PAT as the password.

**3. Store the PAT securely:**

To avoid having to enter the PAT every time you perform a Git operation, you can store it using **Git credential helpers** (such as Git’s credential cache or GitHub's credential manager). This securely saves your credentials for future use.

* On **macOS** or **Linux**, you can enable Git’s credential caching:

: git config --global credential.helper cache

* On **Windows**, Git’s credential manager will handle it automatically, but you can also configure it manually:

: git config --global credential.helper manager

**4. Revoke or Modify PAT:**

If you ever need to revoke or regenerate a token, you can do this through GitHub’s developer settings (where you created the token). This is helpful if you suspect your token has been compromised or if you no longer need it.

**Why Use a PAT?**

* **Security**: Passwords are less secure than tokens, especially when interacting with APIs or over HTTPS. PATs are more secure and can be scoped to specific actions, limiting the potential damage in case of compromise.
* **Password Deprecation**: Since GitHub no longer allows the use of passwords for Git operations, PATs are the recommended method for authentication.
* **Granular Permissions**: You can control exactly what each PAT can do, which helps in creating secure workflows.

**GitHub Interview Questions**

1. **What is the procedure of pushing the changes from local to git hub?**
2. **What is the difference between git pull and git fetch?**
3. **What is the difference between DVCS and CVCS?**
4. **Why you are using branching strategy in git-hub?**
5. **What is git tagging?**
6. **What is git rebase?**
7. **What is merge conflict in github?**

**1. How do you push changes from local to GitHub?**

To push changes to GitHub from your local repository, follow these steps:

1. Initialize Git (if not already initialized)
2. git init
3. Add a remote repository (only once per project)
4. git remote add origin <repository\_url>
5. Check the current branch
6. git branch
7. Add changes to the staging area
8. git add .
9. Commit changes with a meaningful message
10. git commit -m "Commit message"
11. Push changes to GitHub (main branch example)
12. git push origin main

If pushing for the first time:

git push -u origin main

**2. Difference between git pull and git fetch?**

| Command | Function | When to Use? |
| --- | --- | --- |
| git fetch | Retrieves changes from a remote repository but does not merge them into your working branch. | When you want to see changes before merging. |
| git pull | Fetches updates and automatically merges them into the current branch. | When you want to update your local branch immediately. |

Example:

git fetch origin

git merge origin/main # Merge fetched changes manually

git pull origin main # Fetch + merge in one step

**3. Difference between DVCS and CVCS?**

| Feature | CVCS (Centralized Version Control System) | DVCS (Distributed Version Control System) |
| --- | --- | --- |
| Repository Type | Single central repository | Each user has a full repository copy |
| Internet Dependency | Requires internet to commit changes | Can commit offline |
| Examples | SVN, Perforce | Git, Mercurial |
| Speed | Slower | Faster due to local commits |
| Backup | Risky (single point of failure) | Safe (multiple copies exist) |

**4. Why use branching strategy in GitHub?**

Branching allows developers to work independently on different features without affecting the main code. Common strategies:

* Feature Branching – Develop new features in separate branches (feature-login, feature-ui)
* Git Flow – Uses develop, feature, release, and hotfix branches
* GitHub Flow – Uses main and short-lived feature branches
* Trunk-Based Development – Developers commit directly to main

Example:

git checkout -b feature-new-ui

git add .

git commit -m "New UI changes"

git push origin feature-new-ui

**5. What is Git Tagging?**

Tags are used to mark specific points in the Git history, usually for releases.

* Creating a tag:
* git tag -a v1.0 -m "Version 1.0 release"
* Listing tags:
* git tag
* Pushing a tag to GitHub:
* git push origin v1.0

**6. What is Git Rebase?**

Rebase moves a branch to a new base commit, keeping a linear history.

Example:

git checkout feature-branch

git rebase main

This applies the changes from feature-branch on top of main.

**7. What is a merge conflict in GitHub?**

A merge conflict occurs when Git cannot automatically merge changes due to overlapping modifications.

Fixing a Conflict:

1. Identify conflicts using:
2. git status
3. Open the conflicting file and resolve conflicts manually.
4. Mark conflict as resolved:
5. git add .
6. git commit -m "Resolved merge conflict**"**