Git Basics

What is Git?

Git is a distributed version control system (VCS) that is widely used for tracking changes in source code during software development. It was created by Linus Torvalds in 2005 and has since become the de facto standard for version control in the software development industry.

Git allows multiple developers to collaborate on a project by providing a history of changes, facilitating the tracking of who made what changes and when. Here are some key concepts andfeatures of Git:

1. Repository (Repo): A Git repository is a directory or storage location where your project's files and version history are stored. There can be a local repository on your computer and remote repositories on servers.
2. Commits: In Git, a commit is a snapshot of your project at a particular point in time. Each commit includes a unique identifier, a message describing the changes, and a reference to the previous commit.
3. Branches: Branches in Git allow you to work on different features or parts of your project simultaneously without affecting the main development line (usually called the "master" branch). Branches make it easy to experiment, develop new features, and merge changes back into the main branch when they are ready.
4. Pull Requests (PRs): In Git-based collaboration workflows, such as GitHub or GitLab, pull requests are a way for developers to propose changes and have them reviewed by their peers. This is a common practice for open-source and team-based projects.
5. Merging: Merging involves combining changes from one branch (or multiple branches) into another. When a branch's changes are ready to be incorporated into the main branch, you can merge them.
6. Remote Repositories: Remote repositories are copies of your project stored on a different server. Developers can collaborate by pushing their changes to a remote repository and pulling changes from it. Common remote repository hosting services include GitHub, GitLab, and Bitbucket.
7. Cloning: Cloning is the process of creating a copy of a remote repository on your local machine. This allows you to work on the project and make changes locally.

8. Forking: Forking is a way to create your copy of a repository, typically on a hosting platform like GitHub. You can make changes to your fork without affecting the original project and later create pull requests to contribute your changes back to the original repository.

Git is known for its efficiency, flexibility, and ability to handle both small and large-scale software projects. It is used not only for software development but also for managing and tracking changes in various types of text-based files, including documentation and configuration files. Learning Git is essential for modern software development and collaboration.

Why we need git?

Git is an essential tool in software development and for many other collaborative and version-controlled tasks. Here are some key reasons why Git is crucial:

1. **Version Control:** Git allows you to track changes in your project's files over time. It provides a complete history of all changes, making it easy to understand what was done, when it was done, and who made the changes. This is invaluable for debugging, auditing, and collaboration.
2. **Collaboration:** Git enables multiple developers to work on the same project simultaneously without interfering with each other's work. It provides mechanisms for merging changes made by different contributors and resolving conflicts when they occur.
3. **Branching:** Git supports branching, which allows developers to create isolated environments for developing new features or fixing bugs. This is essential for managing complex software projects and experimenting with new ideas without affecting the main codebase.
4. **Distributed Development:** Git is a distributed version control system, meaning that every developer has a complete copy of the project's history on their local machine. This provides redundancy, facilitates offline work, and reduces the reliance on a central server.
5. **Backup and Recovery:** With Git, your project's history is distributed across multiple locations, including local and remote repositories. This provides redundancy and makes it easy to recover from accidental data loss or system failures.
6. **Code Review:** Git-based platforms like GitHub, GitLab, and Bitbucket

provide tools for code review and collaboration. Developers can propose changes, comment on code, and discuss improvements, making it easier to maintain code quality.

7. **Open Source and Community Development:** Git has become the standard for open- source software development. It allows anyone to fork a project, make contributions, and create pull requests, which makes it easy for communities of developers tocollaborate on a single codebase.
8. **Efficiency:** Git is designed to be fast and efficient. It only stores the changes made to files, rather than entire file copies, which results in small repository sizes and faster operations.
9. **History and Documentation:** Git's commit history and commit messages serve as a form of documentation. It's easier to understand the context and reasoning behind a change by looking at the commit history and associated messages.
10. **Customizability:** Git is highly configurable and extensible. You can set up hooks and scripts to automate workflows, enforce coding standards, and integrate with various tools.

In summary, Git is essential for tracking changes in your projects, facilitating collaboration among developers, and ensuring the integrity and version history of your code. Whether you'reworking on a personal project or as part of a large team, Git is a fundamental tool for modern software development and version control.

What is Version Control System (VCS)?

A Version Control System (VCS), also commonly referred to as a Source Code Management (SCM) system, is a software tool or system that helps manage and track changes to files and directories over time. The primary purpose of a VCS is to keep a historical record of all changesmade to a set of files, allowing multiple people to collaborate on a project while maintaining the integrity of the codebase. There are two main types of VCS: centralized and distributed.

Centralized Version Control Systems (CVCS): In a CVCS, there is a single central repository that stores all the project files and their version history. Developers check out files from this central repository, make changes, and then commit those changes back to the centralrepository. Examples of CVCS include CVS (Concurrent Versions System) and Subversion (SVN).

Distributed Version Control Systems (DVCS): In a DVCS, every developer has a complete copy of the project's repository, including its full history, on their local machine. This allows developers to work independently, create branches for experimentation, and synchronize their changes with remote repositories. Git is the most well-known and widely used DVCS, but other DVCS options include Mercurial and Bazaar.

Key features and benefits of Version Control Systems include:

1. **History Tracking:** VCS systems maintain a complete history of changes, including who made the change, what was changed, and when it was changed. This makes it easy to review and understand the evolution of a project.
2. **Collaboration:** VCS allows multiple developers to work on the same project simultaneously. It provides mechanisms for merging changes made by different contributors and resolving conflicts when they occur.
3. **Branching and Isolation:** VCS systems support branching, allowing developers to create isolated environments for new features or bug fixes. This isolates changes and helps manage complex development tasks.
4. **Revert and Rollback:** If a mistake is made, it is possible to revert changes to a previous state or commit. This is essential for error correction and maintaining code quality.
5. **Backup and Recovery:** Project data is stored in multiple locations, providing redundancy and facilitating data recovery in case of accidental data loss or system failures.
6. **Documentation:** Commit messages and history serve as a form of documentation, explaining why a change was made, who made it, and when it was made.
7. **Efficiency:** VCS systems are designed to be fast and efficient. They typically store only the changes made to files, rather than entire file copies, which results in small repository sizes and faster operations.

VCS is a fundamental tool in software development and is used not only for source code but also for tracking changes in documentation, configuration files, and other types of text-based files. It is especially crucial for collaborative projects, allowing teams of developers to work together on the same codebase with confidence.

Git Life Cycle

The Git lifecycle refers to the typical sequence of actions and steps you take when using Git to manage your source code and collaborate with others. Here's an overview of the Git lifecycle:

1. **Initializing a Repository**:
   - To start using Git, you typically initialize a new repository (or repo) in your project directory. This is done with the command git init.
2. **Working Directory**:
   - Your project files exist in the working directory. These are the files you are actively working on.
3. **Staging**:
   - Before you commit changes, you need to stage them. Staging allows you to select which changes you want to include in the next commit. You use the git add command to stage changes selectively or all at once with git add ..
4. **Committing**:
   - After you've staged your changes, you commit them with a message explaining what you've done. Commits create snapshots of your project at that point in time. You use the git commit command to make commits, like git commit -m "Add new feature".
5. **Local Repository**:
   - Commits are stored in your local repository. Your project's version history is preserved there.
6. **Branching**:
   - Git encourages branching for development. You can create branches to work on new features, bug fixes, or experiments without affecting the main codebase. Use the git branch and git checkout commands for branching.
7. **Merging**:
   - After you've completed work in a branch and want to integrate it into the main codebase, you perform a merge. Merging combines the changes from one branch into another. Use the git merge command.
8. **Remote Repository**:
   - For collaboration, you can work with remote repositories hosted on servers like GitHub, GitLab, or Bitbucket. These repositories serve as a central hub for sharing code.
9. **Pushing**:
   - To share your local commits with a remote repository, you push them using the git push command. This updates the remote repository with your

changes.

10. **Pulling**:
    - To get changes made by others in the remote repository, you pull them to yourlocal repository with the git pull command. This ensures that your local copy isup to date.
11. **Conflict Resolution**:
    - Conflicts can occur when multiple people make changes to the same part of a file. Git will inform you of conflicts, and you must resolve them by editing theaffected files manually.
12. **Collaboration**:
    - Developers can collaborate by pushing, pulling, and making pull requests in a shared remote repository. Collaboration tools like pull requests are commonly used on platforms like GitHub and GitLab.
13. **Tagging and Releases**:
    - You can create tags to mark specific points in the project's history, such as version releases. Tags are useful for identifying significant milestones.
14. **Continuous Cycle**:
    - The Git lifecycle continues as you repeat these steps over time to manage the ongoing development and evolution of your project. This cycle supports collaborative and agile software development.

The Git lifecycle allows for effective version control, collaboration, and the management of complex software projects. It provides a structured approach to tracking and sharing changes, enabling multiple developers to work together on a project with minimal conflicts and a clear history of changes.

2. Git Installation

To install Git on your computer, you can follow the steps for your specific operating system:

1. Installing Git on Windows:

a. Using Git for Windows (Git Bash):

- Go to the official Git for Windows website: https://gitforwindows.org/
- Download the latest version of Git for Windows.
- Run the installer and follow the installation steps. You can choose the default settingsfor most options.

b. Using GitHub Desktop (Optional):

- If you prefer a graphical user interface (GUI) for Git, you can also install GitHub Desktop, which includes Git. Download it from https://desktop.github.com/ and followthe installation instructions.

2. Installing Git from Source (Advanced):

- If you prefer to compile Git from source, you can download the source code from the official Git website (https://git-scm.com/downloads) and follow the compilation instructions provided there. This is usually only necessary for advanced users.

After installation, you can open a terminal or command prompt and verify that Git is correctlyinstalled by running the following command:

```
$ git --version
```

If Git is installed successfully, you will see the Git version displayed in the terminal. You can now start using Git for version control and collaborate on software development projects.

How to Configure the Git?

Configuring Git involves setting up your identity (your name and email), customizing Git options, and configuring your remote repositories. Git has three levels of configuration: system, global, and repository-specific. Here's how you can configure Git at each level:

1. System Configuration:

- System-level configuration affects all users on your computer. It is typically used for site-specific configurations and is stored in the /etc/gitconfig file.
- To set system-level configuration, you can use the git config command with the --system flag (usually requires administrator privileges). For example:

```
$ git config --system user.name "Your Name"
$ git config --system user.email "your.email@example.com"
```

2. Global Configuration:

- Global configuration is specific to your user account and applies to all Git repositories on your computer. This is where you usually set your name and email.

To set global configuration, you can use the git config command with the --global flag. Forexample:

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

You can also view your global Git configuration by

using:

```
$ git config --global --list
```

3. Git Commands List

Git is a popular version control system used for tracking changes in software development projects. Here's a list of common Git commands along with brief explanations:

1. **git init**: Initializes a new Git repository in the current directory.
2. **git clone <repository URL>**: Creates a copy of a remote repository on your localmachine.
3. **git add <file>**: Stages a file to be committed, marking it for tracking in the next commit.
4. **git commit -m "message"**: Records the changes you've staged with a descriptivecommit message.
5. **git status**: Shows the status of your working directory and the files that have beenmodified or staged.
6. **git log**: Displays a log of all previous commits, including commit hashes, authors, dates,and commit messages.
7. **git diff**: Shows the differences between the working directory and the last committedversion.
8. **git branch**: Lists all branches in the repository and highlights the currently checked-out branch.
9. **git branch <branchname>**: Creates a new branch with the specified name.
10. **git checkout <branchname>**: Switches to a different branch.
11. **git merge <branchname>**: Merges changes from the specified branch into thecurrently checked-out branch.
12. **git pull**: Fetches changes from a remote repository and merges them into the currentbranch.
13. **git push**: Pushes your local commits to a remote repository.
14. **git remote**: Lists the remote repositories that your local repository is connected to.
15. **git fetch**: Retrieves changes from a remote repository without merging them.
16. **git reset <file>**: Unstages a file that was previously staged for commit.
17. **git reset --hard <commit>**: Resets the branch to a specific commit, discarding allchanges after that commit.
18. **git stash**: Temporarily saves your changes to a "stash" so you can switch brancheswithout committing or losing your work.
19. **git tag**: Lists and manages tags (usually used for marking specific points in history, likereleases).

20. **git blame <file>**: Shows who made each change to a file and when.
21. **git rm <file>**: Removes a file from both your working directory and the Git repository.
22. **git mv <oldfile> <newfile>**: Renames a file and stages the change.

These are some of the most common Git commands, but Git offers a wide range of features and options for more advanced usage. You can use git --help followed by the command name to get more information about any specific command, e.g., git help commit.

Experiments On

Project Management with Git

Experiment 1.

Setting Up and Basic Commands:
Initialize a new Git repository in a directory. Create a new file and add it to the staging area and commit the changes with an appropriate commit message.

Solution:

To initialize a new Git repository in a directory, create a new file, add it to the staging area, and commit the changes with an appropriate commit message, follow these steps:

1. Open your terminal and navigate to the directory where you want to create the Gitrepository.
2. Initialize a new Git repository in that directory:
   $ git init

3. Create a new file in the directory. For example, let's create a file named "my_file.txt"You can use any text editor or command-line tools to create the file. (touch my_file.txt).
4. Add the newly created file to the staging area. Replace "my_file.txt" with the actualname of your file:

$ git add my_file.txt
This command stages the file for the upcoming commit.

5. Commit the changes with an appropriate commit message. Replace "Your commitmessage here" with a meaningful description of your changes:

$ git commit -m "Your commit message here"
Your commit message should briefly describe the purpose or nature of the changes you made.
For example:
$ git commit -m "Add a new file called my_file.txt"

After these steps, your changes will be committed to the Git repository with the provided commit message. You now have a version of the repository with the new file and its history stored in Git.

Experiment 2

Creating and Managing Branches:
Create a new branch named "feature-branch." Switch to the "master" branch.Merge the"feature-branch" into "master."

Solution:
To create a new branch named "feature-branch," switch to the "master" branch, and merge the"feature-branch" into "master" in Git, follow these steps:

1. Make sure you are in the "master" branch by switching to it:
   $ git checkout master

2. Create a new branch named "feature-branch" and switch to it:
   $ git checkout -b feature-branch

   This command will create a new branch called "feature-branch" and switch to it.

3. Make your changes in the "feature-branch" by adding, modifying, or deleting files asneeded.
4. Stage and commit your changes in the "feature-branch":

   $ git add .
   $ git commit -m "Your commit message for feature-branch"

   Replace "Your commit message for feature-branch" with a descriptive commit message for thechanges you made in the "feature-branch."

5. Switch back to the "master" branch:
   $ git checkout master

6. Merge the "feature-branch" into the "master" branch:
   $ git merge feature-branch

   This command will incorporate the changes from the "feature-branch" into the "master" branch.

Now, your changes from the "feature-branch" have been merged into the "master" branch. Yourproject's history will reflect the changes made in both branches

Experiment 3

Creating and Managing Branches:

Write the commands to stash your changes, switch branches, and then apply the stashed changes.

Solution:

To stash your changes, switch branches, and then apply the stashed changes in Git, you can usethe following commands:

1. Stash your changes:

$ git stash save "Your stash message"

This command will save your changes in a stash, which acts like a temporary storage for changes that are not ready to be committed.

2. Switch to the desired branch:

$ git checkout target-branch

Replace "target-branch" with the name of the branch you want to switch to.

3. Apply the stashed changes:

$ git stash apply

This command will apply the most recent stash to your current working branch. If you have multiple stashes, you can specify a stash by name or reference (e.g., git stash apply stash@{2})if needed.

If you want to remove the stash after applying it, you can use git stash pop instead of git stashapply.

Remember to replace "Your stash message" and "target-branch" with the actual message you want for your stash and the name of the branch you want to switch to.

Experiment 4.

Collaboration and Remote Repositories:
Clone a remote Git repository to your local machine.

Solution:

To clone a remote Git repository to your local machine, follow these steps:

1. Open your terminal or command prompt.

2. Navigate to the directory where you want to clone the remote Git repository. You can use the cd command to change your working directory.

3. Use the git clone command to clone the remote repository. Replace <repository_url> with the URL of the remote Git repository you want to clone. For example, if you were cloning a repository from GitHub, the URL might look like this:

```
$ git clone <repository_url>
```

Here's a full example:

```
$ git clone https://github.com/username/repo-name.git
```

Replace https://github.com/username/repo-name.git with the actual URL of the repository youwant to clone.

4. Git will clone the repository to your local machine. Once the process is complete, youwill have a local copy of the remote repository in your chosen directory.

You can now work with the cloned repository on your local machine, make changes, and pushthose changes back to the remote repository as needed.

Experiment 5.

Collaboration and Remote Repositories:
Fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch.

Solution:

To fetch the latest changes from a remote repository and rebase your local branch onto the updated remote branch in Git, follow these steps:

1. Open your terminal or command prompt.

2. Make sure you are in the local branch that you want to rebase. You can switch to the branch using the following command, replacing <branch-name> with your actual branch name:

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

3. Fetch the latest changes from the remote repository. This will update your local repository with the changes from the remote without merging them into your local branch:

```
$ git fetch origin
```

Here, origin is the default name for the remote repository. If you have multiple remotes, replace origin with the name of the specific remote you want to fetch from.

4. Once you have fetched the latest changes, rebase your local branch onto the updated remote branch:

```
$ git rebase origin/<branch-name>
```

Replace <branch-name> with the name of the remote branch you want to rebase onto. This command will reapply your local commits on top of the latest changes from the remote branch, effectively incorporating the remote changes into your branch history.

5. Resolve any conflicts that may arise during the rebase process. Git will stop and notify you if there are conflicts that need to be resolved. Use a text editor to edit the conflicting files, save the changes, and then continue the rebase with:

```
$ git rebase –continue
```

6. After resolving any conflicts and completing the rebase, you have successfully updated your local branch with the latest changes from the remote branch.
7. If you want to push your rebased changes to the remote repository, use the git push command. However, be cautious when pushing to a shared remote branch, as it can potentially overwrite other developers' changes:

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

Replace <branch-name> with the name of your local branch. By following these steps, you cankeep your local branch up to date with the latest changes from the remote repository and maintain a clean and linear history through rebasing.

Experiment 6.

Collaboration and Remote Repositories:
Write the command to merge "feature-branch" into "master" while providing a customcommit message for the merge.

Solution:

To merge the "feature-branch" into "master" in Git while providing a custom commit messagefor the merge, you can use the following command:

```
$ git checkout master
$ git merge feature-branch -m "Your custom commit message here"
```

Replace "Your custom commit message here" with a meaningful and descriptive commit message for the merge. This message will be associated with the merge commit that is created when you merge "feature-branch" into "master."

Experiment 7.

Git Tags and Releases:
Write the command to create a lightweight Git tag named "v1.0" for a commit in your local repository.

Solution:

To create a lightweight Git tag named "v1.0" for a commit in your local repository, you can use the following command:

```
$ git tag v1.0
```

This command will create a lightweight tag called "v1.0" for the most recent commit in your current branch. If you want to tag a specific commit other than the most recent one, you can specify the commit's SHA-1 hash after the tag name. For example:

```
$ git tag v1.0 <commit-SHA>
```

Replace <commit-SHA> with the actual SHA-1 hash of the commit you want to tag.

Experiment 8.

Advanced Git Operations:
Write the command to cherry-pick a range of commits from "source-branch" to the current branch.

Solution:

To cherry-pick a range of commits from "source-branch" to the current branch, you can use the following command:

$ git cherry-pick <start-commit>^..<end-commit>

Replace <start-commit> with the commit at the beginning of the range, and <end-commit> with the commit at the end of the range. The ^ symbol is used to exclude the <start-commit> itself and include all commits after it up to and including <end-commit>. This will apply the changes from the specified range of commits to your current branch.

For example, if you want to cherry-pick a range of commits from "source-branch" starting from commit ABC123 and ending at commit DEF456, you would use:

$ git cherry-pick ABC123^..DEF456

Make sure you are on the branch where you want to apply these changes before running the cherry-pick command.

Experiment 9.

Analysing and Changing Git History:
Given a commit ID, how would you use Git to view the details of that specific commit,including the author, date, and commit message?

Solution:

To view the details of a specific commit, including the author, date, and commit message, you can use the git show or git log command with the commit ID. Here are both options:

1. Using git show:bash

```
$  git show <commit-ID>
```

Replace <commit-ID> with the actual commit ID you want to view. This command will display detailed information about the specified commit, including the commit message, author, date,and the changes introduced by that commit.

For example:

```
$ git show abc123
```

2. Using git log:

```
$ git log -n 1 <commit-ID>
```

The -n 1 option tells Git to show only one commit. Replace <commit-ID> with the actual commit ID. This command will display a condensed view of the specified commit, including its commit message, author, date, and commit ID.

For example:

```
$ git log -n 1 abc123
```

Both of these commands will provide you with the necessary information about the specific commit you're interested in.

Experiment 10.

Analysing and Changing Git History
Write the command to list all commits made by the author "JohnDoe" between "2024- 01-01"and "2024-12-31."

Solution:

To list all commits made by the author "JohnDoe" between "2024-01-01" and "2024-12-31" in Git, you can use the git log command with the --author and --since and --until options. Here'sthe command:

```
$ git log --author="JohnDoe" --since="2024-01-01" --until="2024-12-31"
```

This command will display a list of commits made by the author "JohnDoe" that fall within the specified date range, from January 1, 2024, to December 31, 2024. Make sure to adjust the author name and date range as needed for your specific use case.

Experiment 11.

Analysing and Changing Git History
Write the command to display the last five commits in the repository's history.

Solution:

To display the last five commits in a Git repository's history, you can use the git log commandwith the -n option, which limits the number of displayed commits. Here's the command:

```
$ git log -n 5
```

This command will show the last five commits in the repository's history. You can adjust the number after -n to display a different number of commits if needed.

Experiment 12.

Analysing and Changing Git History
Write the command to undo the changes introduced by the commit with the ID "abc123".

Solution:

To undo the changes introduced by a specific commit with the ID "abc123" in Git, you can use the git revert command. The git revert command creates a new commit that undoes the changes made by the specified commit, effectively "reverting" the commit. Here's the command:

```
$ git revert abc123
```

Replace "abc123" with the actual commit ID that you want to revert. After running this command, Git will create a new commit that negates the changes introduced by the specified commit. This is a safe way to undo changes in Git because it preserves the commit history and creates a new commit to record the reversal of the changes.