Git and Github

What is Git?

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

What is GitHub?

GitHub is a web-based platform used for version control and collaboration on projects primarily based on Git. It offers a range of features including source code management (version control), issue tracking, pull requests, code review, and project management tools.

GitHub is widely used by developers and teams for

- managing and sharing code,
- collaborating on projects, and
- contributing to open-source software.

Git commands

- **SETUP**

Configuring user information used across all local repositories

# set a name that is identifiable for credit when review version history

git config --global user.name "[firstname lastname]"

# set an email address that will be associated with each history marker

git config --global user.email "[valid-email]"

# set automatic command line coloring for Git for easy reviewing

git config --global color.ui auto

- **INIT**

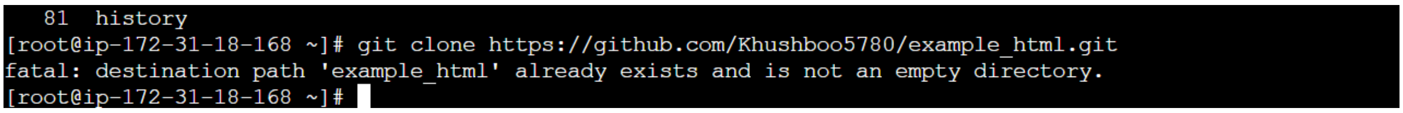
Configuring user information, initializing and cloning repositories

# initialize an existing directory as a Git repository

git init

# retrieve an entire repository from a hosted location via URL

git clone [url]



```
   81  history
[root@ip-172-31-18-168 ~]# git clone https://github.com/Khushboo5780/example_html.git
fatal: destination path 'example_html' already exists and is not an empty directory.
[root@ip-172-31-18-168 ~]#
```

- **STAGE & SNAPSHOT**

Working with snapshots and the Git staging area

# show modified files in working directory, staged for your next commit

git status

# add a file as it looks now to your next commit (stage)

git add [file]

# unstage a file while retaining the changes in working directory

git reset [file]

# diff of what is changed but not staged

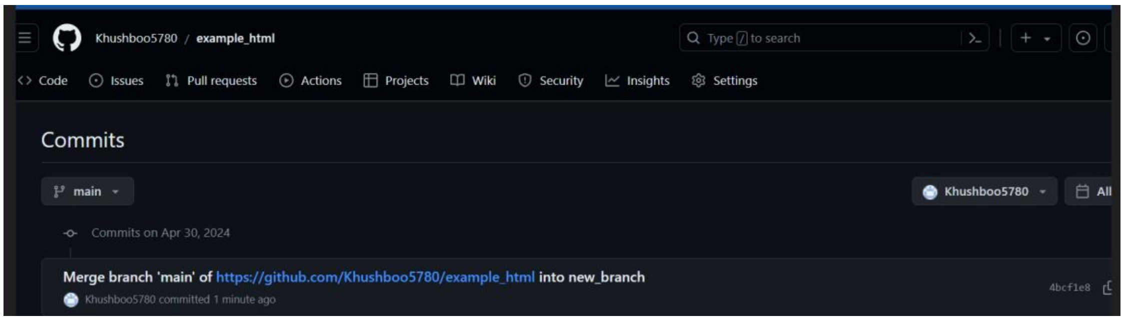
git diff

# diff of what is staged but not yet committed

git diff --staged

# commit your staged content as a new commit snapshot

git commit -m "[descriptive message]"



Commits

Commits on Apr 30, 2024

Merge branch 'main' of https://github.com/Khushboo5780/example_html into new_branch

Khushboo5780 committed 1 minute ago

4bcf1e8

- **BRANCH & MERGE**

Isolating work in branches, changing context, and integrating changes

# **list** your branches. a * will appear next to the currently active branch

git branch

# create a new branch at the current commit

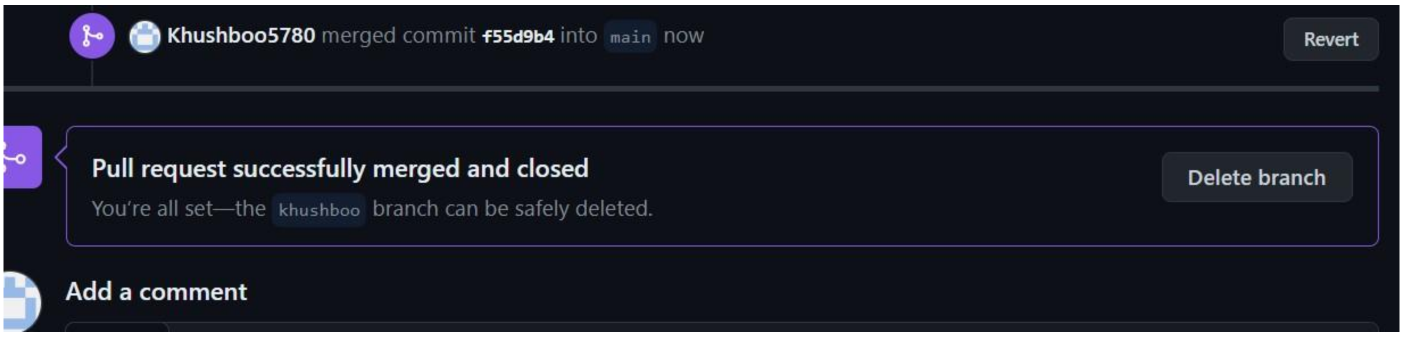
git branch [branch-name]

# switch to another branch and check it out into your working directory

git checkout

# merge the specified branch's history into the current one

git merge [branch]


Khushboo5780 merged commit f55d9b4 into main now

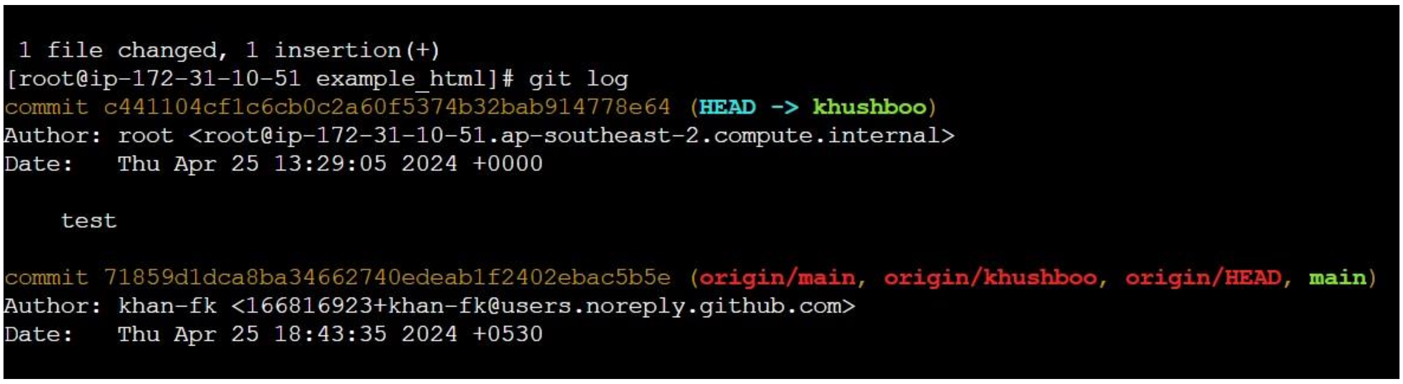
Pull request successfully merged and closed

You're all set—the khushboo branch can be safely deleted.

Add a comment

# show all commits in the current branch's history

git log


```
 1 file changed, 1 insertion(+)
[root@ip-172-31-10-51 example_html]# git log
commit c441104cf1c6cb0c2a60f5374b32bab914778e64 (HEAD -> khushboo)
Author: root <root@ip-172-31-10-51.ap-southeast-2.compute.internal>
Date:   Thu Apr 25 13:29:05 2024 +0000

    test

commit 71859d1dca8ba34662740edeab1f2402ebac5b5e (origin/main, origin/khushboo, origin/HEAD, main)
Author: khan-fk <166816923+khan-fk@users.noreply.github.com>
Date:   Thu Apr 25 18:43:35 2024 +0530
```

- **INSPECT & COMPARE**

Examining logs, diffs and object information

# show the commit history for the currently active branch

git log

# show the commits on branchA that are not on branchB

git log branchB..branchA

# show the commits that changed file, even across renames

git log --follow [file]

# show the diff of what is in branchA that is not in branchB

git diff branchB...branchA

# show any object in Git in human-readable format

git show [SHA]

- **TRACKING PATH CHANGES**

Versioning file removes and path changes

# delete the file from project and stage the removal for commit

git rm [file]

# change an existing file path and stage the move

git mv [existing-path] [new-path]

# show all commit logs with indication of any paths that moved

git log --stat -M

- **IGNORING PATTERNS**

Preventing unintentional staging or commiting of files

# system wide ignore pattern for all local repositories

git config --global core.excludesfile [file]

# Save a file with desired patterns as git ignore with either direct string matches or wildcard globs.

logs/

*.notes

pattern*/

- **SHARE & UPDATE**

Retrieving updates from another repository and updating local repos

# add a git URL as an alias

git remote add [alias] [url]

# fetch down all the branches from that Git remote

git fetch [alias]

# merge a remote branch into your current branch to bring it up to date

git merge [alias]/[branch]

# Transmit local branch commits to the remote repository branch

git push [alias] [branch]

# fetch and merge any commits from the tracking remote branch

git pull

- **REWRITE HISTORY**

Rewriting branches, updating commits and clearing history

# apply any commits of current branch ahead of specified one

git rebase [branch]

# clear staging area, rewrite working tree from specified commit

git reset --hard [commit]

- **TEMPORARY COMMITS**

Temporarily store modified, tracked files in order to change branches

# Save modified and staged changes

git stash

# list stack-order of stashed file changes

git stash list

# write working from top of stash stack

git stash pop

# discard the changes from top of stash stack

git stash drop

Advantages and Disadvantages

Git and GitHub are often used together, but they serve different purposes and offer distinct advantages and disadvantages:

Advantages of Git:

Distributed Version Control: Git is a distributed version control system, meaning every developer has a complete copy of the repository on their local machine. This allows for offline work and faster operations.

Branching and Merging: Git makes branching and merging workflows easy and efficient. Developers can work on features or fixes in parallel branches and merge them back into the main branch when ready.

Performance: Git is designed to be fast, even with large repositories and long histories. Most operations are performed locally, which reduces the need for constant communication with a central server.

Flexibility: Git is flexible and can be used for any type of project, from small personal projects to large enterprise applications. It's not tied to any specific development methodology or workflow.

Disadvantages of Git:

Steep Learning Curve: Git has a steep learning curve, especially for beginners. Concepts like branching, merging, and rebasing can be challenging to understand initially.

Command Line Interface: While Git does have GUI tools available, the command line interface is the most powerful and widely used. This can be intimidating for developers who are not comfortable with the command line.

Lack of Built-in Collaboration Features: Git itself does not provide built-in collaboration features like issue tracking, project management, or code review. These features are often handled by third-party tools or platforms like GitHub.

Advantages of GitHub:

Centralized Hosting: GitHub provides a centralized platform for hosting Git repositories. This makes it easy to share code with others, collaborate on projects, and contribute to open-source projects.

Code Review and Collaboration Tools: GitHub offers features like pull requests, code reviews, issue tracking, and project management tools. These features facilitate collaboration among team members and help maintain code quality.

Community and Ecosystem: GitHub has a large and active community of developers. It's a popular platform for open-source projects, making it easy to discover and contribute to a wide range of projects.

Integration with Third-Party Tools: GitHub integrates with a wide range of third-party tools and services, such as continuous integration platforms, code quality analysis tools, and deployment services.

Disadvantages of GitHub:

Dependency on External Service: GitHub is a third-party service, so there is a dependency on its availability and reliability. If GitHub experiences downtime or issues, it can disrupt workflows for developers and teams.

Limited Control: While GitHub provides a lot of features and flexibility, developers and organizations have limited control over the platform compared to hosting their own Git repositories.

Cost for Private Repositories: While public repositories are free on GitHub, there is a cost associated with hosting private repositories. This can be a barrier for individuals or organizations with budget constraints.

Git repositories types

In GitHub, **repositories** (repos) can be classified based on their **accessibility and purpose**. Here are the main types of repositories you'll encounter:

Public Repositories: These repositories are publicly accessible and viewable by anyone on the internet. They are often used for open-source projects where the code is meant to be freely available for collaboration and contributions from the community.

Private Repositories: Private repositories are accessible only to specified collaborators who have been granted access by the repository owner. They are commonly used for proprietary projects or sensitive code that should not be publicly available.

Forked Repositories: Forking a repository creates a copy of the original repository under your GitHub account. Forked repositories are typically used when you want to contribute to someone else's project or experiment with changes without affecting the original codebase. You can submit pull requests from your forked repository to contribute changes back to the original project.

Template Repositories: Template repositories are designed to be used as starting points for new projects. They allow you to create a new repository based on the structure and content of the template repository, which can include files, directories, and even code.

Archived Repositories: Archived repositories are read-only and no longer actively maintained. They are often used for projects that have been completed or discontinued but are kept for reference purposes.

Enterprise Repositories: Enterprise repositories are part of GitHub Enterprise, a version of GitHub designed for use by organizations. These repositories may have additional features and access controls tailored for enterprise use cases, such as security and compliance requirements.

<u>Goals</u>

GitHub serves several goals that contribute to its role as a central hub for collaborative software development:

Facilitating Collaboration: GitHub provides a platform where developers can collaborate on projects regardless of their location. It offers tools like pull requests, issues, and project boards that facilitate communication and coordination among team members.

Version Control: GitHub is built on top of Git, a distributed version control system. It enables developers to track changes to their codebase, manage different versions, and collaborate on code with ease.

Open Source Development: GitHub is a popular platform for hosting open-source projects. It allows developers to share their code with the community, accept contributions from others, and build software collaboratively.

Code Hosting: GitHub provides a centralized hosting service for Git repositories. Developers can store their code on GitHub's servers, making it accessible to themselves and others from anywhere with an internet connection.

Code Review and Quality Assurance: GitHub offers features like pull requests and code reviews that help maintain code quality and ensure that changes are thoroughly reviewed before being merged into the main codebase.

Project Management: GitHub provides tools for managing projects, including issue tracking, project boards, and milestones. These features help teams organize and prioritize their work, track progress, and plan releases.

Community Building: GitHub fosters a vibrant community of developers who share code, collaborate on projects, and learn from each other. It provides social features like following other users, starring repositories, and participating in discussions to facilitate community building.

Integration Ecosystem: GitHub integrates with a wide range of third-party tools and services, including continuous integration (CI) systems, code analysis tools, deployment services, and more. This integration ecosystem enhances developers' workflows and enables seamless automation.

Review Process

Creating a Pull Request (PR): When a developer wants to merge their changes into the main branch of a repository, they create a pull request. This pull request contains the proposed changes, along with any relevant context or information.

Assigning Reviewers: The pull request author can assign one or more reviewers to review their changes. Reviewers are notified of the pull request and can begin reviewing the code.

Reviewing Changes: Reviewers examine the proposed changes in the pull request diff, looking for potential issues, bugs, or improvements. They may leave comments directly on specific lines of code, providing feedback or asking questions.

Discussion and Iteration: Reviewers and the pull request author can engage in discussions within the pull request comments, addressing feedback, clarifying questions, and iterating on the proposed changes as necessary.

Approval or Requests for Changes: Reviewers can approve the pull request if they are satisfied with the proposed changes, or they can request additional changes if they believe further work is needed. Pull request authors make the requested changes and update the pull request accordingly.

Merge: Once all reviewers have approved the pull request (or if the project's merge requirements are met), the pull request author can merge their changes into the main branch of the repository.

By involving reviewers in the pull request process, GitHub helps maintain code quality, fosters collaboration among team members, and ensures that changes are thoroughly reviewed before being incorporated into the project'**s** codebase.

Interview Questions

1. What is Git?

Git is a decentralized version control system engineered for rapid and effective management of projects ranging from modest to vast in scale. It supports collaborative efforts among developers by monitoring file modifications and enhancing teamwork.

2**. What is a repository in Git?**

A Git repository is a location where your project resides, acting as a storage area. This repository can exist locally within a directory on your computer or on a cloud-based platform like GitHub. It encompasses all the files associated with the project, along with a record of the modifications made to these files over time.

3. What is the difference between Git and GitHub?

Git is a tool for version control that enables you to monitor and record the evolution of your source code. GitHub is an online hosting service that facilitates the management of Git repositories. GitHub offers a user-friendly web interface along with functionalities such as permission management, task organization, error tracking, and the capability to handle feature suggestions.

4**. How does Git work?**

Git works by taking snapshots of a project's files. Unlike other version control systems, Git records the entire contents of each file and its changes every time a commit is made. This makes operations like branching, merging, and reverting changes more efficient.

5**. What is a commit in Git?**

In Git, a commit is a process that records a version of the project's presently prepared modifications. This record includes details on the modifications implemented, a distinctive identifier (a SHA-1 hash), the creator's identity, and the timestamp of the commit.

6**. What is branching in Git?**

In Git, branching allows you to veer off from the primary development path and proceed with separate tasks without impacting the main workflow. This technique enables the isolated development of features, bug fixes, or experimentation within a specific section of the repository, ensuring that each process remains distinct from the others.

7**. What is a merge in Git?**

Merging is a Git operation that integrates changes from one branch into another. It can be a fast-forward merge, where the target branch is updated to the latest commit of the source branch, or a three-way merge, where divergent branch histories are combined into a new commit.

8**. What is a conflict in Git?**

A conflict in Git occurs when two branches have made edits to the same line in a file or when one branch deletes a file while the other branch modifies it. Git cannot automatically resolve these changes; the developer must manually resolve the conflicts.

9**. What is a pull request?**

A pull request serves as a mechanism for contributing to a project, typically utilized in projects hosted on GitHub. In this process, a developer implements modifications within their own branch, uploads these changes to a repository, and then initiates a pull request. This action prompts the project's maintainers to examine the proposed changes, engage in discussions about possible adjustments, and ultimately integrate the pull request into the primary branch.

10**. What is `git fetch` vs. `git pull`?**

`git fetch` downloads updates from a remote repository to your local repository without integrating them. On the other hand, `git pull` not only fetches the updates but also incorporates them into your active branch.

11**. How do you revert a commit that has already been pushed and made public?**

To undo the modifications introduced by a previous commit while ensuring safety for public commits, employ `git revert <commit_hash>`, which generates a fresh commit that reverses the earlier changes. On the other hand, `git reset` allows reverting to an earlier state; however, exercise caution when applying it to public commits, as it alters the commit history

12**. What is a `.gitignore` file?**

A `.gitignore` file is a textual document instructing Git on the files or directories to exclude from a project. This commonly encompasses files created during the build phase, local setup files, or files with confidential data.

13**. How do you clone a repository?**

To create a local copy of a repository on your machine, execute the command `git clone <repository_url>`. This action duplicates the repository locally.

14. **What is `git stash`?**

The `git stash` command temporarily stores your working directory's modifications, allowing you to switch branches without discarding your current progress.

15. **How do you view the commit history?**

Use `git log` to view the commit history. There are many options to customize the output, such as `git log --oneline` for a condensed view.

16. **What is a remote in Git?**

A Git remote refers to a shared repository utilized by all team members for the purpose of exchanging their updates. Typically, this repository is on a server or a cloud-based hosting service like GitHub.

17. **How do you create a new branch?**

Use `git branch <branch_name>` to create a new branch. Use `git checkout -b <branch_name>` to create and switch to it in one step.

18. **What is `git merge --squash`?**

The `git merge --squash` command consolidates all commits from a specified feature branch into a single commit when merging into the target branch, resulting in a tidier project history.

19**. How do you resolve a merge conflict?**

To resolve a merge conflict, edit the files to fix the conflicting changes. Then, use `git add` to stage the resolved files and `git commit` to commit the resolved merge.

20**. What is `git rebase`?**

`git rebase` transfers modifications from one branch to another, enabling the creation of a streamlined project timeline by relocating the updates of a feature branch to the forefront of the main branch.

21**. What is the difference between `git merge` and `git rebase`?**

The main difference is in how the branch history is presented. `git merge` preserves the history of a feature branch by creating a new merge commit. `git rebase` rewrites the feature branch's history to appear as if it was developed from the latest main branch, creating a linear history.

22**. How do you change the last commit?**

Use `git commit --amend` to modify the most recent commit. This can change the commit's message or include new changes.

23**. What is `git push`?**

`git push` is used to upload local repository content to a remote repository. It transfers commits from your local repo to the remote.

24. **How do you delete a branch?**

Use `git branch -d <branch_name>` to delete a local branch. If the branch is not fully merged, you may need to use `-D` instead. To delete a remote branch, use `git push <remote_name> --delete <branch_name>`.

25. **What is `git checkout`?**

`git checkout` allows navigating between different branches or reverting working tree files to a previous state. However, in the latest versions of Git, it is advised to use `git switch` for changing branches and `git restore` to revert files, each designated for these specific functions.

26. **How do you list all the remote repositories configured?**

Use `git remote -v` to list all the remote repositories configured for your local repository

27. How do you remove a file from Git but not delete it from your file system?

Use git rm --cached <file_name> to remove a file from Git without deleting it from your filesystem.

29**. What is git diff?**

git diff shows the differences between files in the working directory and the index, or between commits.

30. **How do you rename a Git branch**

To rename the current branch, use git branch -m <new_name>. To rename a different branch, use git branch -m <old_name> <new_name>.

31**. What does git reset do?**

git reset is used to undo changes. It has three main modes: --soft, --mixed, and --hard.

32. **How do you amend a commit message.**

Use git commit --amend to change your most recent commit message.

33. **What is the HEAD in Git?**

HEAD is a reference to the last commit in the currently checked-out branch.

34**. How do you find a commit by a message?**

Use git log --grep=<search-pattern> to search through commit messages.

35. **What is the difference between git stash pop and git stash apply?**

git stash pop applies stashed changes and removes them from the stash. git stash apply applies stashed changes but keeps them in the stash.

36. How do you list all branches that contain a specific commit.

Use git branch --contains <commit>.

37. What is a fast-forward merge in Git?

A fast-forward merge happens when the target branch's tip is behind the merged branch's tip, allowing the target branch to "catch up" by just moving forward to the merged branch's tip.

38. **How do you create an empty commit?**

Use git commit --allow-empty to create a commit with no changes.

39. **How do you switch branches in Git?**

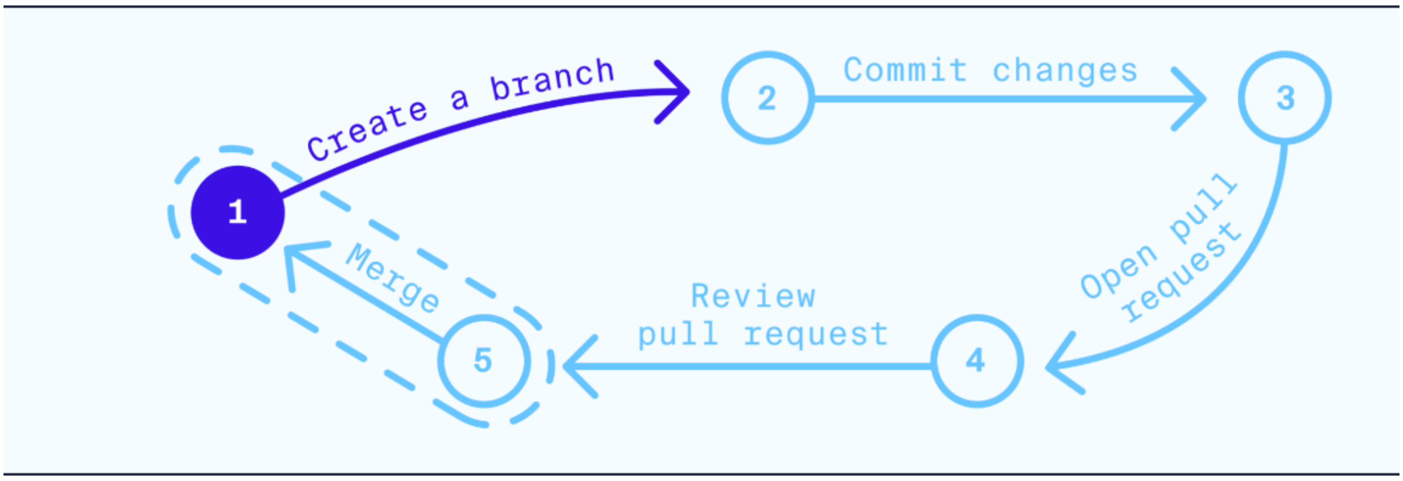
Use git checkout <branch-name> to switch branches.

40. **What are branches, and what command creates a new branch?**

C1 – – – C2 – – – C3

Above is an example of a simple commit history, in which C1 is the first commit (version), C2 is the second commit, and C3 is the third commit. This is an example of a master branch. Any new branches that get created are called the feature branch.

To create a new branch, you use the `git branch <branch-name>` command, and to view all the local branches, use the `git branch` command. The current branch is marked with an asterisk (*).



The basic GitHub workflow goes like this.

41. How often do you commit when using Git, and do you review files before committing?

You should be committing as frequently as possible. A good practice is to commit after every new feature as long as it can be done cleanly without breaking anything that already exists. Remember to leave a detailed message about your commit.

42. What's the difference between `git merge` and `git rebase`? Which method do you prefer?

The two commands allow developers to incorporate changes from one branch into another. `git merge` allows for the creation of a new "merge commit" occurring between the master and feature commit. The command joins the two (or more if needed) together. For example, `git merge` will add changes to the current branch. Whereas `git rebase` can reapply the feature commit on top of the main branch.

So the major difference between the two is how the history gets managed. With `git merge`, the history of branches remains the same but can subsequently leave clutter from unnecessary old commits. And with `git rebase`, the main history is partially forfeited in favor of the feature history.

As for answering your technique preference, there's no correct answer — just as long as you can prove your point. However, there's some scrutiny with rebasing when working in large teams because you may inadvertently rewrite public commits. But both methods can be effective. Look into the "golden rule of rebasing" for more information.

43. When should you use `git push` compared to when you use `git pull`?

`git push` is used to upload content from one repository to another. For example, you can `push` content from a local repository into a remote repository.

`git pull` is used to retrieve and integrate content from another repository. For example, you can `pull` content from a remote repository into a local repository.

44**. What command do you use to return to a previous commit?**

To return to a previous commit on a private, local repository, you'll first want to run `git log` to pull up the branch's history. Then, after identifying the version's hash you want to go to, use the `git reset` command to change the repository to the commit.

For a public, remote repository, the `git revert` command is a safer option. It'll create a new commit with the previous edits rather than delete commits from the history.

45. Has there ever been a time when you worked from multiple machines or with multiple developers? How did you like it? What did you find most challenging?

To answer this question, you can talk about your workflow. For example, explain how you've used Git in the past as your version control system for working between machines or sharing with multiple developers.

46**. What methods do you use to clean up your feature branches?**

For this question, you can mention these three commands.

1. `git merge --squash` is a command that can merge multiple commits of a branch.
2. `git commit --fixup` marks the commit as a fix of the previous commit.
3. `git rebase -i --autosquash` is a rebase type of squash for merging multiple commits.

47.**What are Git attributes used for?**

Git attributes allow programmers to try different merge strategies within the files or directories of a project. Attributes are set on paths so that when certain settings are applied along the path, the attribute can perform requests. Git attributes can also perform requests when exporting your project.

48. Have you ever done debugging with Git? How did you do this?

To debug with Git, you use `git bisect` to identify the first commit to introduce a bug by using automatic binary search. And `git blame` identifies the author of a commit, while `git grep` searches for any string or regular expression in any of the files in the staging area.

49. Have you ever encountered a merge conflict? How did you go about resolving it?

Merge conflicts happen when there are competing changes within the commits, and Git is unable to automatically determine which changes to use. Merge conflicts can be resolved by editing the file that has the conflict. And once the edits get made, add and commit the file.

You can read more about merge conflicts in this GitHub article.

50**. Explain Git hooks.**

Git hooks are custom scripts that can run when certain actions occur. There are two types: client-side hooks and server-side hooks. Git hooks can be written in any scripting programming language. They should be located within the hooks subdirectory of the .git directory.