**GIT&GITHUB**

**What is Git ?**

Git is a distributed version control system widely used for tracking changes in source code during software development. It allows multiple people to work on the same project simultaneously without interfering with each other's changes.

**Git Installation:**

installing Git on your local laptop is straightforward and depends on your operating system. Here’s a step-by-step guide for Windows, mac, and Linux:

**URL:** [**https://git-scm.com/downloads**](file:///C:\Users\Lenovo\Desktop\WPS%20Office.lnk) #it support OS like Windows,Linux,Mac.

**For Linux:**

To install git on Linux machine use the below provided commands.

sudo apt update

yum install git -y

**Configure Git:**

After completed of installation go to cmd prompt check git installed or not using this cmd: git --version

After installation, set up your Git configuration with your name and email:

git config --global user.name --- to check configured name

git config --global user.email --- to check configured email

Example:

git config --global user.name "veer"

git config --global user.email "cloud87777@gmail.com"

**Git workflow step by step:**

**Initialize a Repository:**

git init -----This creates a new Git repository in the current directory.

**Clone a Repository:** git clone <repository-URL>This copies an existing repository from a remote server to your local machine.

Touch filename ----- to create a file need to use touch cmd then followed bye filename.

git add (filename) or git add . -----to add file into staging area from local working directory.

git status ----to see the status of the file where it is .

git commit -m "Your commit message".

git log --**View Commit History.**

**Git cmd for branch :**

git push origin <branch-name>**---------**Push Changes to a Remote Repository.

git pull origin <branch-name>**----------**Pull Changes to a Remote Repository.

git branch <branch-name>---------------Create a New Branch.

git checkout <branch-name>------------Switch to a Different Branch.

git branch -d <branch-name>----------- Remove a Branch.

git checkout main ------------------------First, switch to the branch you want to merge into (e.g., main)

git merge <branch-name>-----------------Then merge the desired branch.

**Git changes:**

git diff ------------------ ----compare changes working directory to staging area

git diff --staged ------------compare changes staging to local repo

git diff head------------------compare changes working directory to local repo

git diff commit id to commit id -----compare changes in to two commits

**Revert changes on git :**

# Revert changes from working directory #

git restore <file\_name>

or

git checkout -- <file\_name> like undo command

# Revert changes from Staging Area #

git restore --staged <file\_name> #to revert changes from Staging area to working directory

# Revert changes from Local Repository #

git reset HEAD~1 # to revert changes from local repo to working directory

#if The git reset HEAD~2 command moves the current branch backward by two commits

Note: if you can give git status or any git commit command everything clear no untracked file no staging files

**Git reset we have three modes (optional)**

Mixed—its is default mode discard the commits both local repo and staging area

Git reset --mixed commit id

Soft -- discard the commits in local repo only file will be available both staging and working directory

Git reset --soft commit id

Hard : discard the commits in local repo , staging and working directory aswell

Git reset --hard commit id

#########################################

**Git HUB:**

GitHub is one of the most popular resources for developers to share code and work on projects together. It’s free, easy to use, and has become central in the movement toward open-source software.

git clone ---- to clone entire repository in first time

git pull ------to pull the updates and changes from repository

git push ------to push the local updates into remote repository.

**git push using SSH method:**

# push the changes by using SSH #

### To generate SSH Key

ssh-keygen (give this command into your terminal)

copy the public key id\_rsa.pub

paste into git hub ssh keys

1.path :go to git hub account click on your profile picture in the upper-right corner, then go to settings-->ssh and GPG keys--> Add new ssh key and give any name and paste it your public key over there and save it.

2.git local to remote sync (how to add that locally created folder into git hub)

after creating repos locally and remote we have to give following commands

git remote set-URL origin git@github.com:User/UserRepo.git

now it will sync we can start pull push commands.

**Git push using Git Hub Token # (personal access token PAT)**

###To generate Token

settings-->Developer settings-->personal access token -->Token classic --> generate token --->classic

Example :

ghp\_TWH7bFT1UbCdn3wBbQwUCgwh0nN6T83GmPno #generated one

git remote set-url origin https://ghp\_XgChtyKBqOlllqQSFG8GhEIkimGfek2aCOze@github.com:/user/repository.git

Examples:

git push https://<token>:/user/repository.git

Replace with your created token id in the name of <token > ex. for both https or ssh url clone

gut push [https://<token>@github.com:/CloudTechDevOps/practice.git](https://%3ctoken%3e@github.com:/CloudTechDevOps/practice.git)

git remote set-url origin <https://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@github.com/CloudTechDevOps/rebase-merge.git>

git remote set-url origin git@ghp\_TWH7bFT1UbCdn3wBbQwUCgwh0nN6T83GmPnogithub.com:CloudTechDevOps/rebase-merge.git

git push <https://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@github.com/CloudTechDevOps/practice.git>

git push <https://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@github.com:/user/repository.git>

git push https://${git\_token}@github.com/${GIT\_USER\_NAME}/${GIT\_REPO\_NAME} HEAD:main

gut push <https://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@github.com:/CloudTechDevOps/test.git>

git push git:ghp\_OAaYOMFDPyBsiM6r536Z9akhQwOrxg0dQmJM@github.com@github.com:CloudTechDevOps/test.git

git remote set-url origin <https://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@github.com:/CloudTechDevOps/local.git>

**Mapping local repo to remote repository without cloning :**

####### create a new repository on the command line############

touch file1, file2, file3

git init

git add .

git commit -m "first commit"

git remote add <username> <ssh url of repo>

ex:

git remote add CloudTechDevOps git@github.com:CloudTechDevOps/different.git

git remote -v (optional) if we get an error: remote origin already exists.

git remote remove origin remove existing origin map and add git remote

git branch command ton check which branch you are in.

git branch -M main #by default locally created directories updated with master branch but if the remote is main branch we need to rename the branch or create new branch local or create same branch in remote finally we need to map local and remote same branch .

(or)

create same branch in remote "git push --set-upstream <CloudTechDevOps(username)> master"

git push -u origin main

**Git Branching:**

BRANCHES ?

It's an individual process of development for code.

we create individual branch in real-time to do test and developed.

each developer will work on their own branch.

At the end we will merge all branches into actual branch.

Default branch is Master or Main.

git branch : to list the branches

git branch dev : to create a new branch

git checkout dev : to switch from one branch to another.

git checkout -b dev : to create and switch from one branch to created branch.

git branch -m oldname new name : to rename a branch

git branch -D dev : to delete a branch

**GIT Merge:**

What is Git Merge?

Git marge will help to combine the changes from two or more branches into a single branch. Developers will work on different branches to improve code or to develop the code after completion we can merge them into a single version of the code.

**With Practical example:**

Clone the sample repository from git hub remote --main branch (git clone <repo url>)

Create branch (dev) from branch (main) --- (git branch dev)

checkout to dev ---(git checkout dev)

Now We can observe all main branch files in dev branch also

We can do few changes in dev branch and push to dev git hub remote branch( git add . , git commit -m "message" git push origin dev)

(note :after push we can see the pull request raised from dev msg in git hub console)

we can assign and merge manually )

---or git command process ---

create branch from main

git add git commit git push to dev branch

if we want to merge from command process

now switch to main branch (git checkout main)

git merge dev (merge the changes into main from dev)

and push to remote repository (GitHub)

now we can see the changes reflected into main branch also because of merging into main from dev.

**Issues in git:**

# unable to push ?

if two developers are working with same repository in same branch?

This error occurs when you attempt to push your local changes to the remote repo without updating your local repo with new changes made to the remote repo.

your push request will fails when the difference will come local to remote commit id's

so first pull updates changes into local and then push it .

**Solution:**

**case--1**

## git pull ##

git pull origin <branch name> if multiple branches are there in remote

git pull ---if only one branch is there

after that merge the changes into local and push your changes in to remote

**case --2**

## git fetch and merge ##

use git fetch and then git merge also will work

git fetch origin <branch name> ---if multi branches are there

git fetch ----if you are in same branch single

**case --3**

git pull --rebase (if single branch is there)

git pull origin <branch name> --rebase (if multi branches are there )

**####how to update the developer branch which is already existing from main branch after git pull to main branch ?##**##

checkout to master or main (to which branch you need to be updated)

git pull

git checkout to your own branch

git merge main

ls (we can see list of files same like main into your branch)

**### How To Resolve Merge Conflicts? ### ---(-conflicts raises due to in ""same branch while push time"" and ""different branches while merging time"" )**

Trying to update different data int o same file git conflicts occurred

While merging the two branches if changes are made to two different branches then git will not merge automatically it prompts the user to resolve the merge conflicts manually. Below are the steps to resolve the merge conflicts in git:

**Step 1:** Identify the conflict files.

Git will automatically display a message by indicating the file to be resolved from merge conflicts. You have to resolve the conflicts manually.

**Step 2:** Open the conflict files by using vi

Open the merge conflict files by using editors whatever you are convenient with like (IDE). After opening we can conflict markers as shown below it will indicate where the conflicts are located.

**Conflict markers u can see below**

(<<<<<<<, =======, and >>>>>>>)

Step 3: Resolve the conflicts.

Remove the unnecessary changes after examining them carefully and keep the changes that are more important.

after

**Step 4:** Moving to the staging.

Use the git add command to add the updated files to the staging area after the conflicts have been resolved.

git add .

**Step 5:** Commit and Push the changes

After resolving conflicts commit the changes by using the below command. Including the message which gives information about changes made while resolving the conflicts.

git commit -m "message"

Push the changes made to the remote repository by using. Below command.

git push

-------------------------------------------------------------------------------------------

Where other developers can access the code. And perform any changes that are required.

After resolving the conflicts, it is crucial to carefully analyze and test the merged code to make sure that the modifications are functioning as intended and that no new problems have been introduced. These procedures can help developers resolve merge disputes in Git and maintain a dependable and stable code base.

**GIT Fork:**

Forking a repository means creating a copy of the repo. When you fork a repo, you create your own copy of the repo on your GitHub account. When several developers want to work on a project but need to make changes that are inappropriate for the original repository, forking is frequently used.

**Git ignore :**

What is Git Ignore?

When sharing your code with others, there are often files or parts of your project, you do not want to share.

Examples

log files

temporary files

hidden files

personal files

etc.

Git can specify which files or parts of your project should be ignored by Git using a .gitignore file.

Git will not track files and folders specified in .gitignore. However, the .gitignore file itself IS tracked by Git.

Create .gitignore

To create a .gitignore file, go to the root of your local Git, and create it:

Example

touch .gitignore

Now open the file using a text editor.

We are just going to add two simple rules:

Ignore any files with the .log extension

Ignore everything in any directory named temp

Example

# ignore ALL .log files

\*.log

# ignore ALL files in ANY directory named temp

temp/

Now all .log files and anything in temp folders will be ignored by Git.

**Diff between git pull vs git pull --rebase :**

**git pull :**

git pull fetches the latest changes of the current branch from a remote and applies those changes to your local copy of the branch. Generally this is done by merging, i.e. the local changes are merged into the remote changes. So git pull is similar to git fetch & git merge.

**git pull --rebase :**

You can pull using rebase instead of merge. The local changes you made will be rebased on top of the remote changes, instead of being merged with the remote changes.so here local changes will be added on top of remote changes without merging extra commit looks like linear changes .

particular commit if you want push

git log

take git commit want to push

git push origin b753622b12789c:refs/heads/main

**Git cherry pick:**

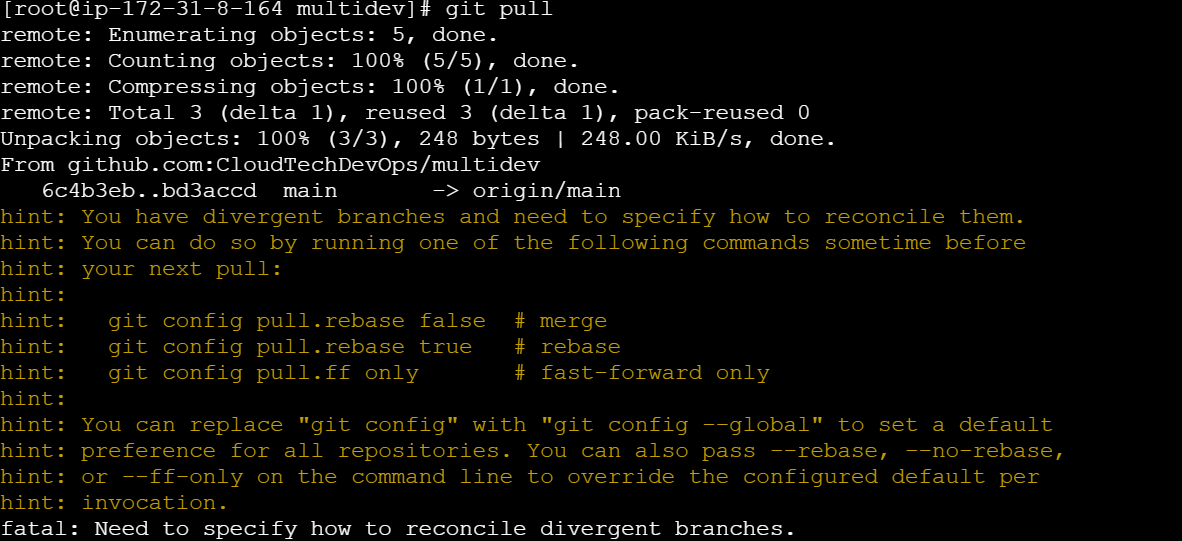
if we want merge specific commit into upstream branch (like main ) cherry pick will help us

git checkout to required branch where you want add specific commit

git cherry-pic <committed>

git push origin <branch name>

**Challenges:** if your local branch has diverged from the remote branch, meaning that both your local branch and the remote branch have unique commits that aren't in the other.



To reconcile this, you need to choose how you want to integrate these changes. You have three main options:

**1. Merge (Default)**

* Merges the remote changes into your local branch, creating a merge commit.
* Command:

**git config pull.rebase false**

**git pull**

**2. Rebase**

* Rebases your local changes on top of the remote changes. This avoids a merge commit but can lead to conflicts that you'll need to resolve.
* Command:

**git config pull.rebase true**

**git pull**

**3. Fast-Forward Only**

* Only pulls if your branch can be fast-forwarded. This means it will only work if your local branch is behind the remote branch and has no new commits. If there are local changes, it will fail.
* Command:

**git config pull.ff only**

**git pull**

You can also specify these options directly in the pull command to avoid changing the configuration:

* For **merge**:

**git pull --no-rebase**

* For **rebase**:

**git pull --rebase**

* For **fast-forward only**:

**git pull --ff-only**

Choose the method that best suits your workflow and the current state of your repository. If you're unsure, the merge option (--no-rebase) is generally a safe default.

**Conditions:**

Git pull" is a common Git command that fetches changes from a remote repository and merges them into your current branch. However, it can present some challenges, especially in collaborative environments. Here are some common issues and how to address them:

**1. Merge Conflicts**

* **Problem**: When changes in the remote branch conflict with your local changes, Git can't automatically merge them.
* **Solution**: Git will mark the conflicting files. You must manually resolve the conflicts by editing the files and then commit the resolved changes.

**2. Fast-Forward vs. Merge Commits**

* **Problem**: By default, git pull tries to fast-forward the current branch, which can lead to a linear history. If you want to maintain a history of merge commits, this might not be desirable.
* **Solution**: Use git pull --no-ff to force a merge commit, or configure your branch to always create merge commits with git config branch.<branchname>.mergeOptions --no-ff.

**3. Overwriting Local Changes**

* **Problem**: If you have uncommitted changes, git pull might fail, or it could overwrite your work.
* **Solution**: Commit or stash your changes before running git pull. If you want to apply the remote changes over your local modifications, you can use git stash before the pull, then git stash pop afterward.

**4. Diverged Branches**

* **Problem**: When your local branch has changes that are not on the remote branch, git pull might result in a "diverged branches" situation.
* **Solution**: Use git pull --rebase to reapply your local commits on top of the pulled changes. This keeps a cleaner history by avoiding unnecessary merge commits.

**5. Authentication Issues**

* **Problem**: You might encounter authentication failures, especially when using HTTPS with credentials that are no longer valid.
* **Solution**: Ensure that your credentials (username and password or token) are up to date. For SSH, check that your SSH key is correctly configured and loaded.

**1. List Your Stashes**

First, confirm that your stash was created and note its identifier:

**git stash list**

You should see a list of stashes, such as:

**stash@{0}: WIP on main: 65484b4 Merge branch 'main' of github.com:CloudTechDevOps/multidev**

**2. View the Stash Contents**

To see what was included in the stash, especially to check for file200k, use:

**git stash show -p stash@{0}**

Replace stash@{0} with the appropriate stash reference if it's different. This command shows a detailed diff of the stashed changes

**3. Apply the Stash**

To recover the stashed file and return it to your working directory, you can apply the stash:

**git stash apply stash@{0}**

**or**

**git stash pop stash@{0}**

The apply command restores the changes without removing the stash, while pop applies the changes and removes the stash.

**Drop Stash**

* To remove a specific stash without applying it:

**git stash drop stash@{n}**

* **To remove all stashes:**

**git stash clear**

Revert

**Option 1: Create a New Commit to Revert the Changes**

If you want to keep the commit history clean without rewriting it, you can create a new commit that undoes the changes introduced by the previous commit.

1. **Find the Commit Hash**:

git log

Identify the commit hash of the commit you want to revert.

1. **Revert the Commit**:

git revert <commit-hash>

This will create a new commit that undoes the changes made by the specified commit.

1. **Push the Revert Commit**:

git push origin <branch-name>

**Option 2: Reset the Branch to a Previous Commit (Rewriting History)**

This option is more aggressive and rewrites the commit history. Be careful with this approach if others have already pulled the commit.

1. **Find the Commit Hash**:

git log

Identify the commit hash you want to revert to.

1. **Reset the Branch**:

git reset --hard <commit-hash>

This will reset your branch to the specified commit, discarding any commits after it.

1. **Force Push the Changes**:

git push --force origin <branch-name>

Force pushing will overwrite the remote branch with your local branch.

**git reset --soft <commit-hash>**

This moves the HEAD to the specified commit but keeps the changes in the working directory and staging area. It’s useful if you want to keep your changes but remove the commits.

**Example:**

git reset --soft 03eb7f6d

* This resets to 03eb7f6d but leaves all changes made after this commit in the staging area.

**5. git reset --mixed <commit-hash>**

This is the default behavior of git reset. It resets the HEAD to the specified commit and keeps the changes in the working directory but removes them from the staging area.

**Example:**

git reset --mixed 03eb7f6d

* This resets to 03eb7f6d, unstages the changes made after this commit, but keeps the changes in the working directory.

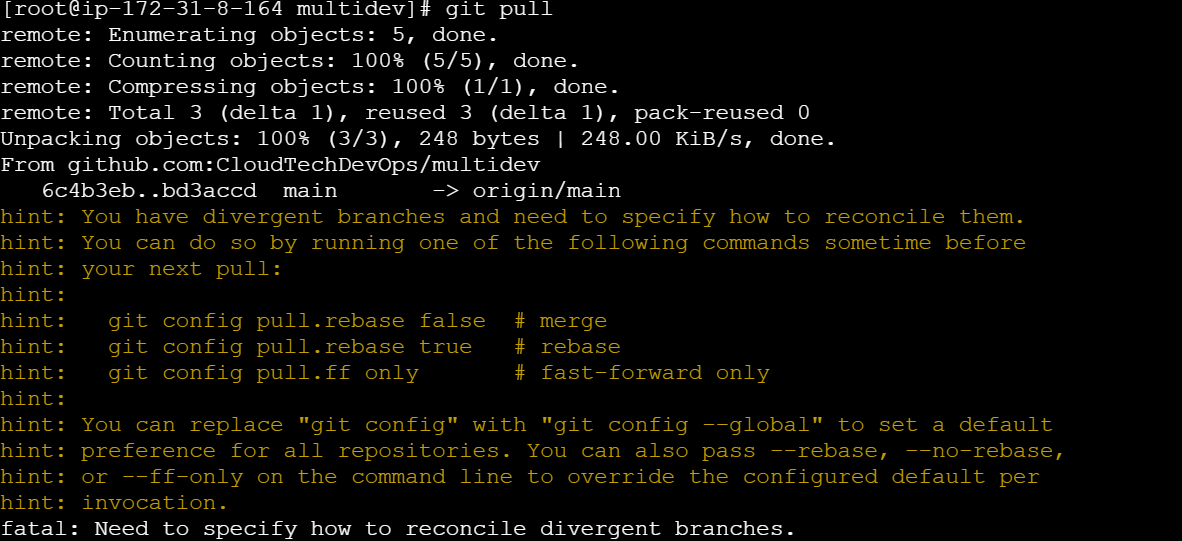
**Summary:**

* Use **git checkout** or **git restore** for reverting specific files to a previous state.
* Use **git revert** to create a new commit that undoes a previous commit.
* Use **git reset –hard**, **git reset --soft** or **git reset --mixed** for less destructive resets that preserve your working directory's changes.

* 1. Ho

**Git Diff Challenges:**

if your local branch has diverged from the remote branch, meaning that both your local branch and the remote branch have unique commits that aren't in the other.



To reconcile this, you need to choose how you want to integrate these changes. You have three main options:

**1. Merge (Default)**

* Merges the remote changes into your local branch, creating a merge commit.
* Command:

**git config pull.rebase false**

**git pull**

**2. Rebase**

* Rebases your local changes on top of the remote changes. This avoids a merge commit but can lead to conflicts that you'll need to resolve.
* Command:

**git config pull.rebase true**

**git pull**

**3. Fast-Forward Only**

* Only pulls if your branch can be fast-forwarded. This means it will only work if your local branch is behind the remote branch and has no new commits. If there are local changes, it will fail.
* Command:

**git config pull.ff only**

**git pull**

You can also specify these options directly in the pull command to avoid changing the configuration:

* For **merge**:

**git pull --no-rebase**

* For **rebase**:

**git pull --rebase**

* For **fast-forward only**:

**git pull --ff-only**

Choose the method that best suits your workflow and the current state of your repository. If you're unsure, the merge option (--no-rebase) is generally a safe default.

**Conditions:**

Git pull" is a common Git command that fetches changes from a remote repository and merges them into your current branch. However, it can present some challenges, especially in collaborative environments. Here are some common issues and how to address them:

**1. Merge Conflicts**

* **Problem**: When changes in the remote branch conflict with your local changes, Git can't automatically merge them.
* **Solution**: Git will mark the conflicting files. You must manually resolve the conflicts by editing the files and then commit the resolved changes.

**2. Fast-Forward vs. Merge Commits**

* **Problem**: By default, git pull tries to fast-forward the current branch, which can lead to a linear history. If you want to maintain a history of merge commits, this might not be desirable.
* **Solution**: Use git pull --no-ff to force a merge commit, or configure your branch to always create merge commits with git config branch.<branchname>.mergeOptions --no-ff.

**3. Overwriting Local Changes**

* **Problem**: If you have uncommitted changes, git pull might fail, or it could overwrite your work.
* **Solution**: Commit or stash your changes before running git pull. If you want to apply the remote changes over your local modifications, you can use git stash before the pull, then git stash pop afterward.

**4. Diverged Branches**

* **Problem**: When your local branch has changes that are not on the remote branch, git pull might result in a "diverged branches" situation.
* **Solution**: Use git pull --rebase to reapply your local commits on top of the pulled changes. This keeps a cleaner history by avoiding unnecessary merge commits.

**5. Authentication Issues**

* **Problem**: You might encounter authentication failures, especially when using HTTPS with credentials that are no longer valid.
* **Solution**: Ensure that your credentials (username and password or token) are up to date. For SSH, check that your SSH key is correctly configured and loaded.

**1. List Your Stashes**

First, confirm that your stash was created and note its identifier:

**git stash list**

You should see a list of stashes, such as:

**stash@{0}: WIP on main: 65484b4 Merge branch 'main' of github.com:CloudTechDevOps/multidev**

**2. View the Stash Contents**

To see what was included in the stash, especially to check for file200k, use:

**git stash show -p stash@{0}**

Replace stash@{0} with the appropriate stash reference if it's different. This command shows a detailed diff of the stashed changes

**3. Apply the Stash**

To recover the stashed file and return it to your working directory, you can apply the stash:

**git stash apply stash@{0}**

**or**

**git stash pop stash@{0}**

The apply command restores the changes without removing the stash, while pop applies the changes and removes the stash.

**Drop Stash**

* To remove a specific stash without applying it:

**git stash drop stash@{n}**

* **To remove all stashes:**

**git stash clear**

Revert

**Option 1: Create a New Commit to Revert the Changes**

If you want to keep the commit history clean without rewriting it, you can create a new commit that undoes the changes introduced by the previous commit.

1. **Find the Commit Hash**:

git log

Identify the commit hash of the commit you want to revert.

1. **Revert the Commit**:

git revert <commit-hash>

This will create a new commit that undoes the changes made by the specified commit.

1. **Push the Revert Commit**:

git push origin <branch-name>

**Option 2: Reset the Branch to a Previous Commit (Rewriting History)**

This option is more aggressive and rewrites the commit history. Be careful with this approach if others have already pulled the commit.

1. **Find the Commit Hash**:

git log

Identify the commit hash you want to revert to.

1. **Reset the Branch**:

git reset --hard <commit-hash>

This will reset your branch to the specified commit, discarding any commits after it.

1. **Force Push the Changes**:

git push --force origin <branch-name>

Force pushing will overwrite the remote branch with your local branch.

**git reset --soft <commit-hash>**

This moves the HEAD to the specified commit but keeps the changes in the working directory and staging area. It’s useful if you want to keep your changes but remove the commits.

**Example:**

git reset --soft 03eb7f6d

* This resets to 03eb7f6d but leaves all changes made after this commit in the staging area.

**5. git reset --mixed <commit-hash>**

This is the default behavior of git reset. It resets the HEAD to the specified commit and keeps the changes in the working directory but removes them from the staging area.

**Example:**

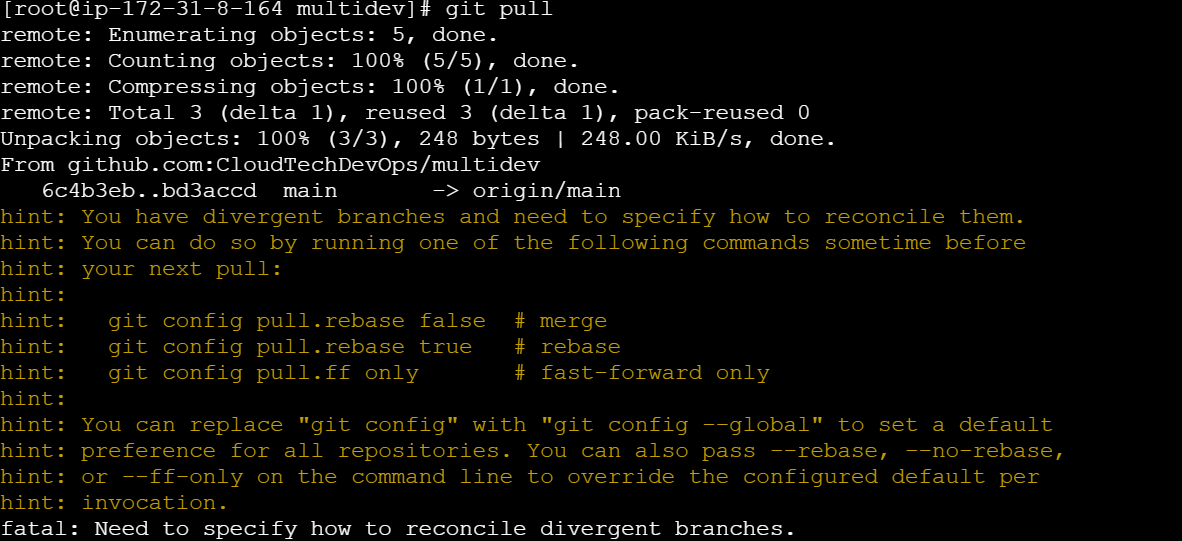
git reset --mixed 03eb7f6d

* This resets to 03eb7f6d, unstages the changes made after this commit, but keeps the changes in the working directory.

**Summary:**

* Use **git checkout** or **git restore** for reverting specific files to a previous state.
* Use **git revert** to create a new commit that undoes a previous commit.
* Use **git reset –hard**, **git reset --soft** or **git reset --mixed** for less destructive resets that preserve your working directory's changes.

**Challenges:** if your local branch has diverged from the remote branch, meaning that both your local branch and the remote branch have unique commits that aren't in the other.



To reconcile this, you need to choose how you want to integrate these changes. You have three main options:

**1. Merge (Default)**

* Merges the remote changes into your local branch, creating a merge commit.
* Command:

**git pull --no-rebase**

**2. Rebase**

* Rebases your local changes on top of the remote changes. This avoids a merge commit but can lead to conflicts that you'll need to resolve Command

**git pull --rebase**

**3. Fast-Forward Only**

* Only pulls if your branch can be fast-forwarded. This means it will only work if your local branch is behind the remote branch and has no new commits. If there are local changes, it will fail Command:

You can also specify these options directly in the pull command to avoid changing the configuration:

**git pull --ff-only**

Choose the method that best suits your workflow and the current state of your repository. If you're unsure, the merge option (--no-rebase) is generally a safe default.

**Conditions:**

Git pull" is a common Git command that fetches changes from a remote repository and merges them into your current branch. However, it can present some challenges, especially in collaborative environments. Here are some common issues and how to address them:

**1. Merge Conflicts**

* **Problem**: When changes in the remote branch conflict with your local changes, Git can't automatically merge them.
* **Solution**: Git will mark the conflicting files. You must manually resolve the conflicts by editing the files and then commit the resolved changes.

**2. Fast-Forward vs. Merge Commits**

* **Problem**: By default, git pull tries to fast-forward the current branch, which can lead to a linear history. If you want to maintain a history of merge commits, this might not be desirable.
* **Solution**: Use git pull --no-ff to force a merge commit, or configure your branch to always create merge commits with git config branch.<branchname>.mergeOptions --no-ff.

**3. Overwriting Local Changes**

* **Problem**: If you have uncommitted changes, git pull might fail, or it could overwrite your work.
* **Solution**: Commit or stash your changes before running git pull. If you want to apply the remote changes over your local modifications, you can use git stash before the pull, then git stash pop afterward.

**4. Diverged Branches**

* **Problem**: When your local branch has changes that are not on the remote branch, git pull might result in a "diverged branches" situation.
* **Solution**: Use git pull --rebase to reapply your local commits on top of the pulled changes. This keeps a cleaner history by avoiding unnecessary merge commits.

**5. Authentication Issues**

* **Problem**: You might encounter authentication failures, especially when using HTTPS with credentials that are no longer valid.
* **Solution**: Ensure that your credentials (username and password or token) are up to date. For SSH, check that your SSH key is correctly configured and loaded.

**1. List Your Stashes**

First, confirm that your stash was created and note its identifier:

Git stash

**git stash list**

You should see a list of stashes, such as:

**stash@{0}: WIP on main: 65484b4 Merge branch 'main' of github.com:CloudTechDevOps/multidev**

**2. View the Stash Contents**

To see what was included in the stash, especially to check for file200k, use:

**git stash show -p stash@{0}**

Replace stash@{0} with the appropriate stash reference if it's different. This command shows a detailed diff of the stashed changes

**3. Apply the Stash**

To recover the stashed file and return it to your working directory, you can apply the stash:

The apply command restores the changes without removing the stash

**git stash apply stash@{0}**

**or**

**git stash pop stash@{0}**

, while pop applies the changes and removes the stash.

**Drop Stash**

* To remove a specific stash without applying it:

**git stash drop stash@{n}**

* **To remove all stashes:**

**git stash clear**

Revert

**Option 1: Create a New Commit to Revert the Changes**

If you want to keep the commit history clean without rewriting it, you can create a new commit that undoes the changes introduced by the previous commit.

1. **Find the Commit Hash**:

git log

Identify the commit hash of the commit you want to revert.

1. **Revert the Commit**:

git revert <commit-hash>

This will create a new commit that undoes the changes made by the specified commit.

1. **Push the Revert Commit**:

git push origin <branch-name>

**Option 2: Reset the Branch to a Previous Commit (Rewriting History)**

This option is more aggressive and rewrites the commit history. Be careful with this approach if others have already pulled the commit.

1. **Find the Commit Hash**:

git log

Identify the commit hash you want to revert to.

1. **Reset the Branch**:

git reset --hard <commit-hash>

This will reset your branch to the specified commit, discarding any commits after it.

1. **Force Push the Changes**:

git push --force origin <branch-name>

Force pushing will overwrite the remote branch with your local branch.

**git reset --soft <commit-hash>**

This moves the HEAD to the specified commit but keeps the changes in the working directory and staging area. It’s useful if you want to keep your changes but remove the commits.

**Example:**

git reset --soft 03eb7f6d

* This resets to 03eb7f6d but leaves all changes made after this commit in the staging area.

**5. git reset --mixed <commit-hash>**

This is the default behavior of git reset. It resets the HEAD to the specified commit and keeps the changes in the working directory but removes them from the staging area.

**Example:**

git reset --mixed 03eb7f6d

* This resets to 03eb7f6d, unstages the changes made after this commit, but keeps the changes in the working directory.

**Summary:**

* Use **git checkout** or **git restore** for reverting specific files to a previous state.
* Use **git revert** to create a new commit that undoes a previous commit.
* Use **git reset –hard**, **git reset --soft** or **git reset --mixed** for less destructive resets that preserve your working directory's changes.
  1. between git pull vs git pull --rebase ## ---
  2. git pull fetches the latest changes of the curret branch from a remote and applies those changes to your local copy of the branch. Generally this is done by merging, i.e. the local changes are merged into the remote changes. So git pull is similar to git fetch & git merge.
  3. git pull --rebase :
  4. You can pull using rebase instead of merge. The local changes you made will be rebased on top of the remote changes, instead of being merged with the remote cha.
  5. o here local changes will be added on top of remote changes without merging extra commit look linear changes
  6. particular commit if you want push
  7. git log
  8. take git commit want to push
  9. gitpush origin b753622b12789c:refs/heads/main
  10. --------------- # git cherry pick #------------------------
  11. if we want merge specific commit into upstream branch (like main ) cherry pick will help us
  12. git checkout to required branch where you want add specific commit
  13. git cherry-pic <committed>
  14. git push origin <branch name>
  15. w To Resolve Merge Conflicts? ### ---(-conflicts raises due to in ""same branch while push time"" and ""different branches while merging time"" )
  16. Trying to update different data int o same file git conflicts occurred
  17. While merging the two branches if changes are made to two different branches then git will not merge automatically it prompts the user to resolve the merge conflicts manually. Below are the steps to resolve the merge conflicts in git:
  18. Step 1: Identify the conflict files.
  19. Git will automatically display a message by indicating the file to be resolved from merge conflicts. You have to resolve the coicts manually.
  20. Step 2: Open the conflict files by using vi
  21. Open the merge conflict files by using editors whatever you are convenient with like (IDE). After opening we can conflict markers as shown below it will indicate where ed.
  22. arkersSta------------------ # Issues in git # ------------------------------
  23. # unable to push ?
  24. ----if two developers are working with same repository in same branch
  25. This error occurs when you attempt to push your local changes to the