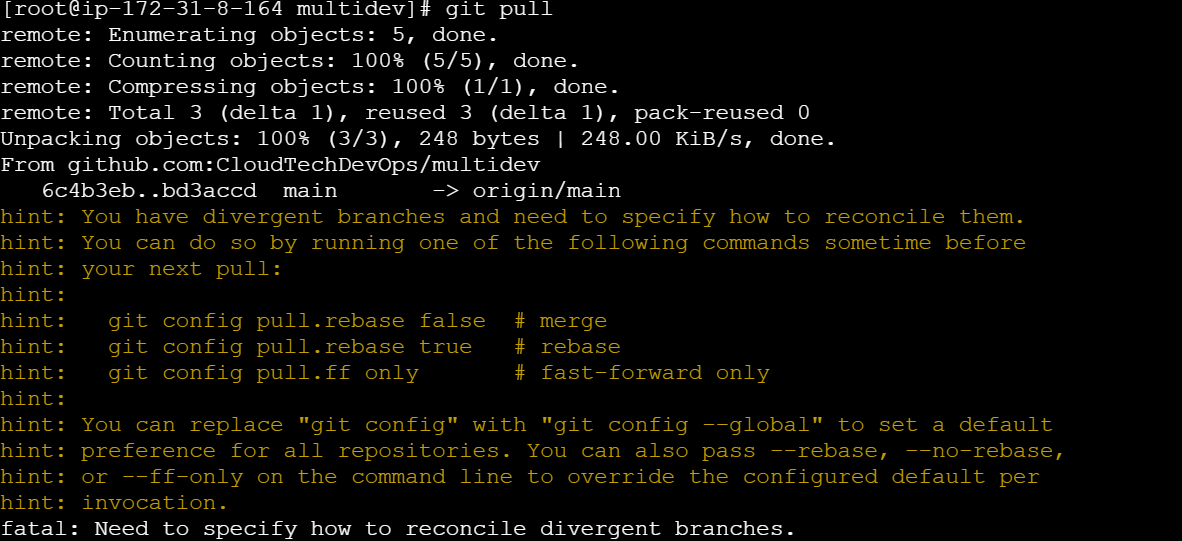
**Challenges:** if your local branch has diverged from the remote branch, meaning that both your local branch and the remote branch have unique commits that aren't in the other.



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.