Synchronizing with Remote branches

Add a remote repo

$ git remote add <alias> <url>

View all remote connections. Add -v flag to view urls.

$ git remote

Remove a connection

$ git remote remove <alias>

Rename a connection

$ git remote rename <old> <new>

Fetch all branches from remote repo (no merge)

$ git fetch <alias>

Fetch a specific branch

$ git fetch <alias> <branch>

Fetch the remote repo's copy of the current branch, then merge

$ git pull

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

$ git pull --rebase <alias>

Upload local content to remote repo

$ git push <alias>

Upload to a branch (can then pull request)

$ git push <alias> <branch>

Let's walk through the **git revert** command step by step with examples. This command is used to **undo changes** made by a specific commit **without rewriting history**, which makes it safer for collaborative work.

What is git revert?

git revert creates a **new commit** that undoes the changes made by a previous commit. It’s different from git reset, which can alter commit history.

Step 1: View Commit History

git log --oneline

Example: This shows a list of commits.

```
a1b2c3d Fix typo in README
e4f5g6h Add login feature
i7j8k9l Initial commit
```

Step 2: Choose the Commit to Revert

Let’s say you want to revert the commit e4f5g6h (Add login feature).

Step 3: Run git revert

git revert e4f5g6h

This will open your default editor to write a commit message for the revert. You can save and close it to proceed.

Step 4: Confirm the Revert

After saving, Git creates a new commit that undoes the changes from e4f5g6h.

git log --oneline

```
x1y2z3w Revert "Add login feature"
a1b2c3d Fix typo in README
e4f5g6h Add login feature
```

Optional Flags

- --no-edit: Skip the editor and use the default commit message.
    - git revert e4f5g6h --no-edit
- --mainline: Used when reverting a **merge commit**.
    - git revert -m 1 <merge_commit_hash>

- When to Use git revert
- ✅ Safe for shared branches (like main or master)
- ✅ Keeps history intact
- ❌ Not ideal for cleaning up local commits (use git reset instead)

🔧 What is git reset?

git reset is used to **undo commits** or **unstage files**, and it can also **remove changes from your working directory** depending on the mode you use.

Mode of GIT: There are three main modes:

Mode	Affects Commit History	Affects Staging Area	Affects Working Directory
`--soft`	✅ Yes	✅ Yes	❌ No
`--mixed`	✅ Yes	✅ Yes	❌ No
`--hard`	✅ Yes	✅ Yes	✅ Yes

Example Scenario

Let's say your commit history looks like this:

git log --oneline

```
c3d4e5f Add login feature
b2c3d4e Update README
a1b2c3d Initial commit
```

Step-by-Step Examples

1. git reset --soft HEAD~1

Use case: Undo the last commit but keep changes staged.

git reset --soft HEAD~1

- Removes the last commit.
- Keeps the changes in the **staging area**.
- You can recommit with a new message.

2. git reset --mixed HEAD~1 (default)

Use case: Undo the last commit and unstage the changes.

git reset HEAD~1

- Removes the last commit.
- Moves changes to the **working directory**.
- You can edit and stage them again.

3. git reset --hard HEAD~1

Use case: Completely undo the last commit and discard changes.

git reset --hard HEAD~1

- Removes the last commit.
- Deletes changes from both staging and working directory.
- ⚠️ **Irreversible**

View Changes After Reset

git status

This shows the current state of your working directory and staging area.

Safety Tip

If you're working on a shared branch (like main), avoid using --hard or --mixed unless you're sure. Use git revert instead to keep history clean.

Resetting to a Middle Commit

Let’s say your commit history looks like this: git log --oneline

```
f7g8h9i Add logout feature
e6f7g8h Add login feature
d5e6f7g Update README
c4d5e6f Initial commit
```

Now, you want to reset to d5e6f7g (Update README), which is **not the latest commit**.

Step-by-Step

- **Step 1: Choose the Commit**

You can reset to any commit using its hash:

git reset --soft d5e6f7g

This will:

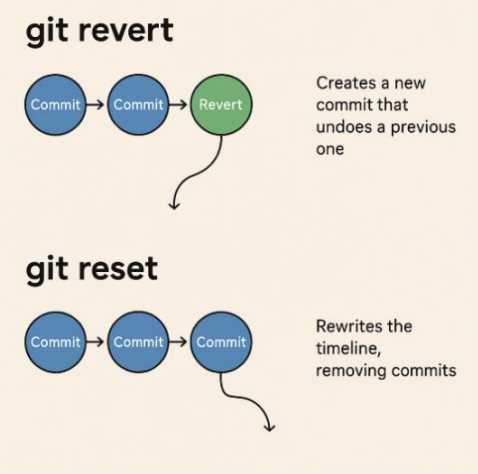
- Remove all commits **after** d5e6f7g.
- Keep changes from those commits in the **staging area**.

Step 2: Check the Result

git log --oneline

Now your history will show:

```
d5e6f7g Update README
c4d5e6f Initial commit
```



Stashing

What is git stash?

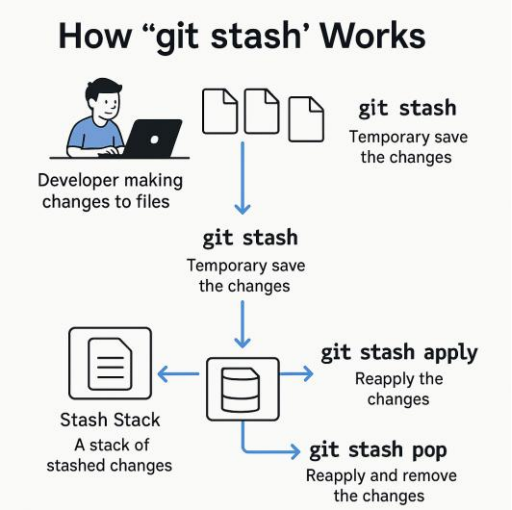
git stash is a Git command that allows you to temporarily save your uncommitted changes (both staged and unstaged) so you can work on something else—like switching branches—without losing your progress.

Why Use It?

Imagine you're working on a feature, but suddenly need to fix a bug on another branch. You don’t want to commit your half-done work, so you:

1. Stash your changes.
2. Switch branches and fix the bug.
3. Come back and reapply your changes.

Diagram to understand git stash works, including how changes are saved, reapplied, and managed. This will help you understand the flow clearly.



$ git stash

As above, but add a comment.

$ git stash save "comment"

List all stashes

$ git stash list

Re-apply the stash without deleting it

$ git stash apply

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

$ git stash pop stash@{2}

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

$ git stash show stash@{1}

Delete stash at index 1. Omit stash@{n} to delete last stash made

$ git stash drop stash@{1}

Delete all stashes

$ git stash clear