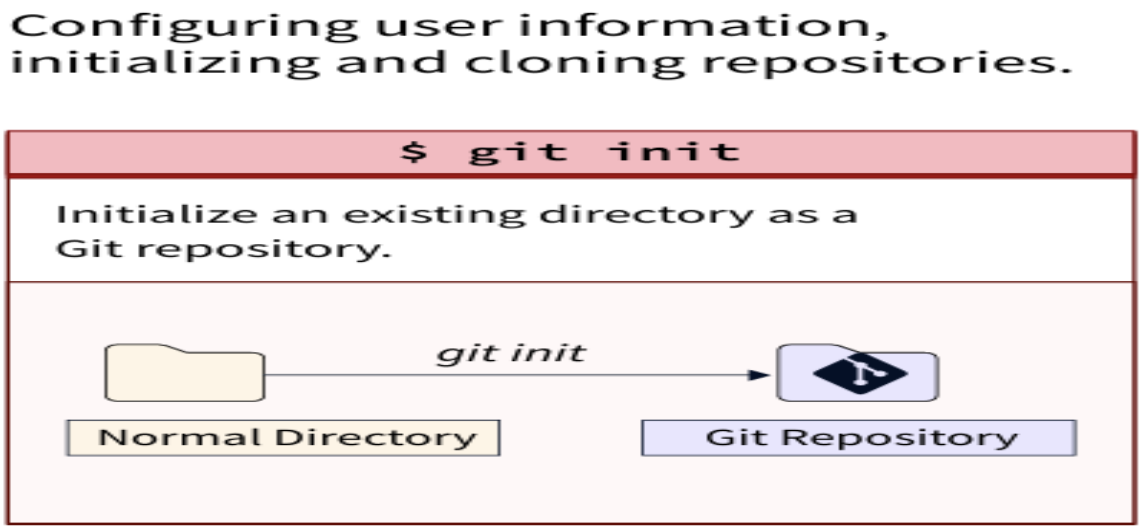
Git Commands

A **remote repository** (remote repo) and a **local repository** (local repo) are two terms commonly used in Git to describe different locations where your code is stored and managed.

Local Repository (Local Repo):

This is the version of the repository that is stored on your local machine. It contains your working directory, staging area, and a .git directory where all the configuration, logs, and history of commits are stored.



Remote Repository (Remote Repo):

A **remote repository** is a version of the repository hosted on a server. It could be a platform like GitHub, GitLab, Bitbucket, or a private server. This is where your code is shared with others. You push your changes to a remote repository, and you can pull others' changes from it.

In Git, **origin** is the default name for the remote repository. When you clone a repository, Git automatically assigns the name **origin** to the remote repository that you cloned from.

However, you can use any name for a remote repository. For example, you could add a remote with a different name like **origin** or **upstream** or **backup** , and then push to it using the command:

We add remote repo to local repo be like

git remote add <remote-name> <remote-repo-URL>

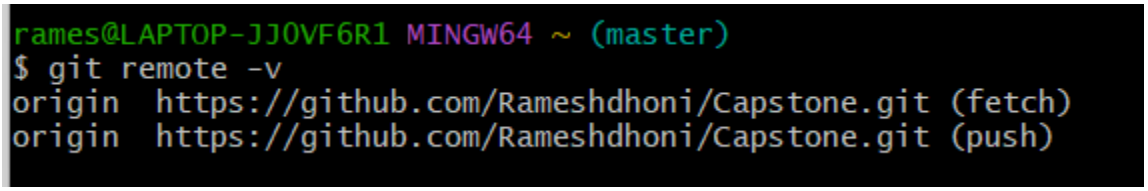
git remote add origin https://github.com/another-user/another-repo.git

git remote add backup https://github.com/another-user/another-repo.git

git remote add upstream https://github.com/another-user/another-repo.git

Check your remotes: To see the list of remotes in your repository:

git remote -v



Ex: using **origin** **remote repo** is the default

Summary of Commands

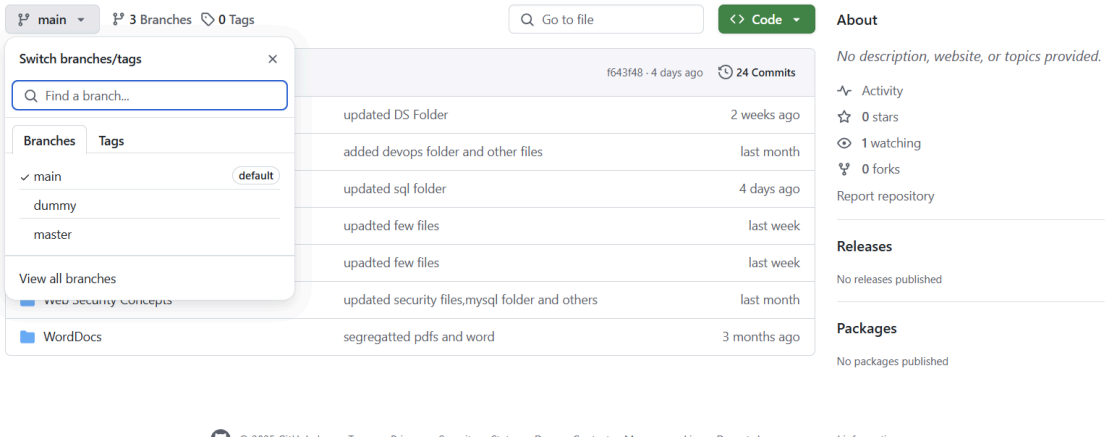
bash Copy Edit

```
cd path/to/GoogleAirQualityNow
git init
git remote add origin https://github.com/voltmx-marketplace/google-non-ai-services.git
git checkout -b feature-google-air-quality-now
git add .
git commit -m "Adding Google Air Quality Now project"
git push origin feature-google-air-quality-now
```

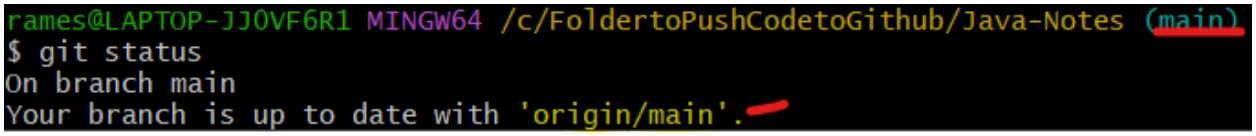
Or if we clone repo directly.

Clone the repo	`git clone https://github.com/voltmx-marketplace/google-non-ai-services.git`
Go inside	`cd google-non-ai-services`
Create branch	`git checkout -b feature-google-air-quality-now`
Add your folder manually	
Add & commit	`git add .` → `git commit -m "Added Google Air Quality Now project"`
Push	`git push origin feature-google-air-quality-now`

Branches



Remote names like origin, upstream are different from branch names like main, master, phx-dev.



See here "**main**" is branch name and "**origin**" is remote repo name.

1) git clone url

2) code. (If you open it from a cloned folder, it will directly redirect to VS code by importing code by default.)

3) git status

The git status command is one of the most frequently used commands in Git. It provides a snapshot of the current state of your working directory and staging area, helping you understand what changes have been made, which changes are staged for the next commit, and which files are untracked or ignored.

like if we create a new file which wasn't there previous and then if we run git status will get it as untracked.

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git status
On branch phx-dev
Your branch is up to date with 'origin/phx-dev'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   server/jenkins-build.sh

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        New Text Document.txt

no changes added to commit (use "git add" and/or "git commit -a")

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$
```

4) **git checkout:**
Using -b, you can create a new branch and immediately switch to it:

git checkout -b new-branch-name

You can use git checkout to switch to an existing branch.

git checkout branch-name

or

git branch newbranchname : it will create new branch.

git checkout branchname: it will switch to respective branch

git branch -d branchname :it will delete the respective branch

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/Test_10906/middleware/server (phx-dev)
$ git checkout -b MXOP-10906
Switched to a new branch 'MXOP-10906'
```

5) if we use **git add .**

It will add all the changes to the staging area. use if you want to add all files to the staging area.

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git add .

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git status
On branch phx-dev
Your branch is up to date with 'origin/phx-dev'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   New Text Document.txt
        modified:   server/jenkins-build.sh


jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$
```

6) like if we have multiple files but you complete one file then you can use **git add "filename" or git add "filename1" "filename2"**

then it will add only selected file.

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git add "New Text Document.txt"

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git status
On branch phx-dev
Your branch is up to date with 'origin/phx-dev'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   New Text Document.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   server/jenkins-build.sh


jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$
```

7) If we want to reset or revert all changes from staging area use **git reset**

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git reset
Unstaged changes after reset:
M       server/jenkins-build.sh

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git status
On branch phx-dev
Your branch is up to date with 'origin/phx-dev'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   server/jenkins-build.sh

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        New Text Document.txt

no changes added to commit (use "git add" and/or "git commit -a")

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$
```

8) If we want to reset or revert specific changes from staging area use **git reset "filename"**

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git reset "New Text Document.txt"

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git status
On branch phx-dev
Your branch is up to date with 'origin/phx-dev'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   server/jenkins-build.sh

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        New Text Document.txt


jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$
```

9) **git commit -m "message"**
The -m stands for "message."

The git commit -m "message related to change for better understanding" command is used in Git to make a commit with a specific message provided in the quotes after -m. The message should

describe the changes made, which helps in tracking history and understanding what each commit does.

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git commit -m "new file added"
[phx-dev 5e0722344e] new file added
 1 file changed, 1 insertion(+), 1 deletion(-)

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git ststus
git: 'ststus' is not a git command. See 'git --help'.

The most similar command is
        status

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$ git status
On branch phx-dev
Your branch is ahead of 'origin/phx-dev' by 1 commit.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        New Text Document.txt

nothing added to commit but untracked files present (use "git add" to track)

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/MiddlewareFolder/middleware (phx-dev)
$
```

10) **git pull**

The git pull command is used to update your local repository with changes from a remote repository.

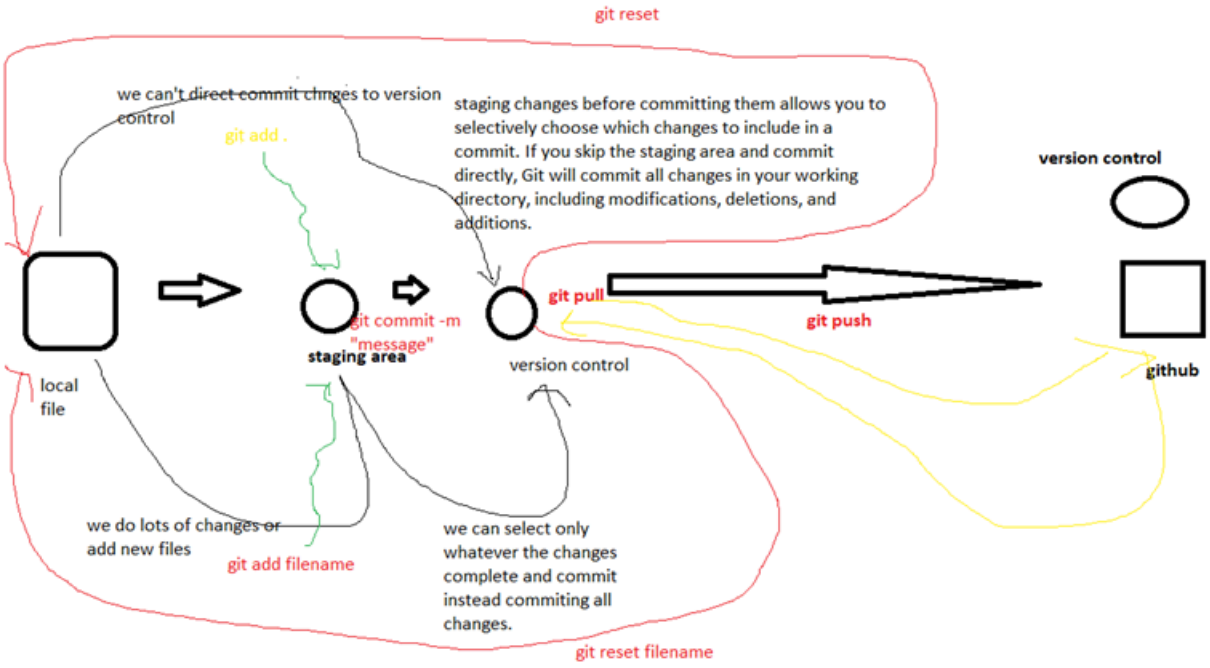
It combines two commands: git fetch (to download new changes) and git merge (to integrate those changes into your current branch).

```
jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/Middleware_E2E_Cloud/middleware (phx-d
ev)
$ git pull
Already up to date.

jakkula.ramesh@LP1-AP-52129375 MINGW64 /c/Middleware_E2E_Cloud/middleware (phx-d
ev)
$
```

11) **git push**

```
rames@LAPTOP-JJOVF6R1 MINGW64 /c/FoldertoPushCodetoGithub/Java-Notes (main)
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Java Collections.pdf

nothing added to commit but untracked files present (use "git add" to track)

rames@LAPTOP-JJOVF6R1 MINGW64 /c/FoldertoPushCodetoGithub/Java-Notes (main)
$ git add .

rames@LAPTOP-JJOVF6R1 MINGW64 /c/FoldertoPushCodetoGithub/Java-Notes (main)
$ git commit -m "java collection notes"
[main (root-commit) cc489a3] java collection notes
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Java Collections.pdf

rames@LAPTOP-JJOVF6R1 MINGW64 /c/FoldertoPushCodetoGithub/Java-Notes (main)
$ git push
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 1.23 MiB | 288.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:Rameshdhoni/Java-Notes.git
 * [new branch]      main -> main

rames@LAPTOP-JJOVF6R1 MINGW64 /c/FoldertoPushCodetoGithub/Java-Notes (main)
$
```

Overview



Cherry-picking in Git

It is a way to apply a specific commit from one branch into another branch — rather than merging the whole branch.

Imagine you're working on a feature branch and there's one specific commit you want to apply to the main branch without merging the entire feature branch.

✅ Command Syntax

git cherry-pick <commit-hash>

📌 Steps to Cherry-Pick a Commit

Get the commit hash:

git log

Switch to the target branch:
git checkout main

Cherry-pick the commit:
git cherry-pick a1b2c3d4

Merge in Git

In Git, **git merge** is a command used to **combine changes** from one branch into another.

Merging is the process of combining the histories of two branches.

It **preserves the complete history** of both branches and creates a **new merge commit** on the target branch that ties together the changes from both branches.

📌 **Real-World Analogy:**

Think of two parallel roads (branches) that eventually join into one — but you still see where each road came from. That's merging.

Example:

Suppose you are on the main branch and want to merge changes from feature-1:

git checkout main

git merge feature-1

Now:

- Git tries to **automatically combine** the code from feature-1 into main.
- If there are no conflicts, a **merge commit** is created.
- If there are conflicts, you need to resolve them manually (as explained earlier).

Rebase

simplified explanation of rebasing. When you rebase a branch, you're essentially moving the entire branch to the tip (latest commit) of another branch, like "replaying" all the commits from

your branch on top of the other branch. This can help keep a cleaner history and avoid merge commits.

To illustrate:

You have branchA with commits A1, A2, A3.
You have branchB with commits B1, B2.

If you rebase branchA into branchB, you end up with branchA having its commits A1, A2, A3 applied after branchB's commits B1, B2. The result would look like:

Before rebase:

```mathematica
main:    A --- B --- C
                  \
feature:           D --- E --- F
```

You run:

```bash
git checkout feature
git rebase main
```

After rebase:

```sql
main:    A --- B --- C
                        \
feature (new):           D' --- E' --- F'
```

> Commits `D`, `E`, and `F` are **replayed on top of** `C`. They're now called `D'`, `E'`, `F'` because they're **new copies** (Git gives them new commit IDs).

B1 -> B2 -> A1 -> A2 -> A3

What is a Merge Conflict in Git?

A **merge conflict** happens when Git cannot automatically resolve differences between two branches that you're trying to merge. This typically occurs when:

- Two branches have **changes to the same line(s)** of a file.
- One branch **modifies** a file that the other **deletes**.
- Changes occur in **overlapping areas** of a file.

Example:

You're working on feature-branch and another developer is working on main. Both of you edit the same line in app.java. When you try to merge main into your branch or vice versa, Git can't decide which change to keep — this causes a **merge conflict**.

How to Handle Merge Conflicts in Git

✅ **Step-by-step process:**
Run the merge command:

```
git merge branch-name
```

If conflicts exist, you'll get a message like:

```
CONFLICT (content): Merge conflict in app.java
Automatic merge failed; fix conflicts and then commit the result.
```

Open the conflicted file (e.g., app.java), and you'll see something like:

```
<<<<<<< HEAD
System.out.println("Hello from feature-branch");
=======
System.out.println("Hello from main");
>>>>>>> main
```

- HEAD: your current branch (e.g., feature-branch)
- The other branch is shown after =======

Manually resolve the conflict:
Choose the correct version, or merge both if necessary:
System.out.println("Hello from both branches");
Mark the file as resolved:
git add app.java
Commit the merge:
git commit

Tips for Avoiding merge Conflicts in git

Avoiding merge conflicts in Git requires a mix of good habits, collaboration, and understanding how Git works. Here are **practical tips** to help you minimize merge conflicts:

✅ **1. Pull Frequently**

- **Always pull** (git pull) the latest changes from the remote repository before starting new work or before pushing.
- This keeps your local branch up to date and minimizes divergence.

✅ **2. Use Feature Branches**

- Create a **separate branch** for each feature or bug fix (git checkout -b feature/your-task).
- Avoid working directly on the main or develop branch.

✅ **3. Commit Often & in Logical Units**

- Make small, focused commits with **clear purpose**.

- This makes resolving conflicts easier if they arise.

✅ 4. Avoid Long-Lived Branches

- Don’t let a branch drift too far from the main codebase.
- **Merge or rebase frequently** to keep it updated with changes from the base branch.

✅ 5. Communicate with Team Members

- Coordinate with your team if you're working on **the same files or features**.
- Avoid stepping on each other's changes.

✅ 6. Rebase Instead of Merge (When Safe)

Use git rebase to **apply your changes on top of the latest main branch**. This creates a linear history and reduces conflicts.

```
git fetch origin
git rebase origin/main
```

> ⚠️ Only rebase local branches that haven't been pushed/shared yet.

Understand Conflict Markers: If conflicts occur, understanding Git's conflict markers (<<<<<<<, =======, >>>>>>>) will help you resolve them quickly.