Git Notes (DevOps Learning)

---

1. Introduction to Git

- **Git =** Distributed Version Control System (DVCS).
- Tracks changes, allows rollback, supports collaboration.
- Key areas:
    - **Working Directory** → Your files.
    - **Staging Area (Index)** → Files prepared for commit.
    - **Local Repository** → Your commits/history.
    - **Remote Repository** → GitHub/GitLab, for collaboration.

---

2. Initial Setup

Create a Repository

git init

- Initializes Git in the current folder.
- Creates a hidden `.git` directory.

3. Git Metadata

Use the username and email to set them globally to see who made the commit

Global (applies everywhere)

git config --global user.name "Moulali"

git config --global user.email "moulali@example.com"

Local (applies only in current repo)

git config user.name "Learner"

git config user.email "learner@example.com"

Create/Edit a File

touch filename # Creates file
vi filename # Opens file in Vim editor

Vim Basics:

- `i` → insert mode
- `Esc` → exit insert mode
- `:w` → save
- `:q` → quit
- `:wq` → save + quit
- `:q!` → quit without saving

---

4. Tracking and Committing Changes

Stage Files

```
git add filename      # Add a specific file
git add .             # Add all files
```

⚠️ Warning (Windows): You might see
`LF will be replaced by CRLF` → line ending difference, safe to ignore.

Check Status

```
git status
```

- Shows untracked, modified, and staged files.

Commit Changes

```
git commit -m "message"
```

- Saves snapshot of staged changes.
- Example:

```
git commit -m "names.txt file modified"
```

Unstage a File

```
git restore --staged filename
```

- Removes a file from staging (not from disk).

View Commit History

```
git log
git log --oneline     # compact view
```

Example:

```
commit a689daf (HEAD -> master)
Author: MOULALIMS <donmoulali786@gmail.com>
Date:   Mon Sep 22 11:55:45 2025
    names.txt file modified
```

---

5. Deleting and Restoring Files

Delete File

```
rm -rf filename
```

Accidentally Deleted? Restore

- Find the last good commit:

```
git log
```

- Reset to it:

```
git reset <commit_id>
```

Example:

```
git reset a689daf7~   # go one commit before
```

---

6. Stashing Work

Stash = emergency drawer. Save work temporarily without committing.

```
git stash           # Save staged + tracked changes
git stash -u        # Save including untracked files
git stash list      # Show saved stashes
git stash show -p stash@{0}   # Show stash diff
git stash pop       # Apply and remove last stash
git stash drop stash@{0}   # Delete one stash
git stash clear     # Delete all stashes
```

Example:

```
touch new1.txt
git stash -u
git stash list
git stash show -p stash@{0}
```

---

7. Remote Repositories

Add Remote

git remote add origin <url>

View Remotes

git remote -v

Best Practice

🚫 Never push directly to `main` branch.
✅ Use feature branches → create PR → merge.

---

8. Working with Forks and Upstream

Fork Workflow

- You cannot commit directly to someone else's repo.
- Fork → clone → work → push → pull request.

Add Upstream Remote

```
git remote add upstream <original_repo_url>
```

Sync with Upstream

```
git fetch --all --prune
git reset --hard upstream/main
git pull upstream main
```

Or use GitHub's **Sync Fork** button.

---

9. Branches and Pull Requests

Branch Commands

```
git branch <branch_name>      # create new branch
git checkout <branch_name>    # switch branch
git checkout -b <branch_name> # create + switch
```

Pull Request Rules

- Each branch = **1 PR**.
- Any new commit on that branch → added to same PR.
- Use multiple branches for multiple features.

---

10. Summary Flow (Local to Remote)

1. `git init` → Start repo
2. `git add .` → Stage changes
3. `git commit -m "message"` → Commit changes
4. `git branch feature` → Create branch
5. `git push origin feature` → Push branch to remote
6. Open PR on GitHub (feature → main).

✅ With this structure, you now have a **clear progression**:

- Setup → Tracking → Commit → Restore → Stash → Remote → Branch → PRs.

📌 Git Notes (Extended + Continuous)

1. What is Git?

- A **Distributed Version Control System (DVCS)**.
- Every developer has a **full copy of the repo** (not just snapshots).
- Git tracks changes in **commits** → grouped changes with unique SHA IDs.
- Used in DevOps for **collaboration, CI/CD pipelines, code backup, branching strategies** (GitFlow, trunk-based dev).

---

2. Why Git in DevOps?

- Collaboration across distributed teams.
- Rollbacks (if a deployment breaks, revert to stable commit).
- Automation (CI/CD jobs trigger on commits/pushes).
- Branching supports parallel feature development.
- Integrates with GitHub/GitLab/Bitbucket → DevOps backbone.

---

3. Core Git Commands (Basics Recap)

- `git init` → Initialize repo.
- `git clone <url>` → Copy remote repo locally.
- `git add <file>` → Stage file for commit.
- `git commit -m "message"` → Save snapshot.
- `git status` → Check current state.
- `git log` → View commit history.
- `git diff` → See unstaged/staged changes.
- `git branch` → List/create branches.
- `git checkout <branch>` → Switch branch.
- `git merge <branch>` → Merge into current branch.
- `git pull` → Fetch + merge from remote.
- `git push` → Push local commits to remote.

---

4. Intermediate Git Commands

- `git fetch` → Download remote commits **without merging** (safe check).
- `git remote -v` → Show connected remote URLs.
- `git remote add origin <url>` → Link repo to remote.
- `git reset <file>` → Unstage file from staging area.
- `git reset --hard <commit>` → Roll back to specific commit (dangerous, wipes changes).
- `git rm <file>` → Remove file from repo.
- `git mv old new` → Rename/move file.

---

5. Branching & Collaboration

- `git checkout -b feature-xyz` → Create + switch to new branch.
- `git branch -d feature-xyz` → Delete branch (after merge).
- `git push -u origin branch-name` → Push branch to remote.
- `git pull origin branch-name` → Pull changes from remote branch.
- **DevOps Tip:** Branching strategy (GitFlow, trunk-based) is critical in CI/CD.

---

6. Undoing Changes (Life Saver Commands)

- `git restore <file>` → Discard local changes in file.
- `git restore --staged <file>` → Unstage staged file.
- `git revert <commit>` → Safely undo a commit (creates new commit instead of deleting).
- `git stash` → Save changes temporarily without committing.
- `git stash pop` → Reapply stashed changes.

---

7. Advanced Git (DevOps Ready)

- `git cherry-pick <commit>` → Apply commit from another branch.
- `git rebase <branch>` → Re-apply commits on top of another branch (clean history).
- `git reflog` → Show **all HEAD history** (even lost commits).
- `git tag <name>` → Mark releases (important for deployments).
- `git describe` → Show tag info closest to current commit.

---

8. Git Config & Multiple Accounts

- `git config --global user.name "Your Name"`
- `git config --global user.email "your@email.com"`
- For multiple accounts:

- Use **SSH keys** (`ssh-keygen`) for each GitHub/GitLab account.
- Add keys to `~/.ssh/config` with separate Host aliases.

Example:

```
Host github-personal
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_rsa_personal

Host github-work
  HostName github.com
  User git
  IdentityFile ~/.ssh/id_rsa_work
```

-
- Then clone like: `git clone git@github-personal:username/repo.git`

---

9. Git in DevOps CI/CD

- Webhooks: triggers pipelines on `git push`.
- Tags: mark release versions (e.g., `v1.0.0`).
- Branch protections: enforce code review before merge.
- GitOps: managing infra using Git as single source of truth.

---

10. Daily Workflow (DevOps Style)

1. `git pull origin main` → Update local branch.
2. `git checkout -b featureX` → New feature branch.
3. `git add . && git commit -m "feat: add featureX"`
4. `git push -u origin featureX` → Push to remote.
5. Open Pull Request (PR) → CI pipeline runs.
6. Merge after review → Deployed to staging/prod.

🔹 Git Tags (Versioning & Releases)

What are Tags?

- A **tag** is a label pointing to a specific commit.
- Unlike branches (which move with new commits), tags are **permanent snapshots**.
- Commonly used for **releases, versioning, and deployment** in DevOps pipelines.

Types of Tags

Lightweight Tag (just a pointer to a commit)

```
git tag v1.0.0
```

✅ Quick label for a commit.

Annotated Tag (preferred for releases, includes metadata like message, author, date)

```
git tag -a v1.0.0 -m "Release version 1.0.0 with bug fixes"
```

Viewing Tags

```
git tag           # list all tags
git show v1.0.0    # details of tag + commit info
```

Pushing Tags to Remote

By default, tags are not pushed with commits.

```
git push origin v1.0.0     # push a single tag
git push origin --tags     # push all local tags
```

Deleting Tags

Locally:

```
git tag -d v1.0.0
```

Remote:

```
git push origin --delete v1.0.0
```

Checkout / Work on a Tag

To move into the state of a tag:

```
git checkout v1.0.0
```

⚠️ This puts you in **detached HEAD** state (not on a branch).

To make changes based on a tag, create a new branch:

```
git checkout -b hotfix-v1.0.0 v1.0.0
```

DevOps Use Case

- Tags act as **release markers** in CI/CD pipelines.
- Example: pushing a new tag like `v2.0.0` can automatically trigger a pipeline to build Docker images and deploy code.

```
git tag -a v2.0.0 -m "Major release with new API"
git push origin v2.0.0
```
