Module: Git & GitHub for DevOps

1. Core Concepts

What is Git?

Git is a **Version Control System (VCS)** that tracks changes in files.

- **Why DevOps needs it:** It allows teams to collaborate on code, roll back to previous versions if a deployment fails, and maintain a history of infrastructure changes.

Types of VCS

- **Centralized (CVCS):** One server holds the history (e.g., SVN). If the server goes down, nobody can work.
- **Distributed (DVCS): Git is Distributed.** Every developer has a full copy of the entire history on their local machine.

Git vs. GitHub

- **Git:** The local tool (the engine) that tracks changes.
- **GitHub:** The remote platform (the cloud) that hosts your Git repositories and provides collaboration features like Pull Requests and Issues.

2. Setting Up & Configuring

Before you commit, Git needs to know who you are. This information is attached to every change you make.

Configuring Git:

Bash
```
git config --global user.name "Your Name"
git config --global user.email "your@email.com"
git config --list  # Verify settings
```

3. The Local Workflow (The 3 Areas)

Understanding the "Three Trees" of Git is the most important part for students.

1. **Working Directory:** Where you edit files.
2. **Staging Area (Index):** The "waiting room" where you pick which changes to include in the next snapshot.
3. **Local Repository:** Where Git permanently saves the snapshots (commits).

Commands:

Bash
```
git init                # Create a new local repository
git status              # See which files are in which area
git add <filename>      # Move file from Working -> Staging
git commit -m "Message"  # Move from Staging -> Local Repo
```

4. Connecting to GitHub

To share your work, you must link your local folder to a remote GitHub repository.

Method A: Link Existing Local to Remote

Bash
```
git remote add origin https://github.com/user/repo.git
git branch -M main      # Rename default branch to main
git push -u origin main  # Push for the first time
```

Method B: Cloning (Starting from Remote)

If the project already exists on GitHub:

Bash
```
git clone https://github.com/user/repo.git
```

Method C: Pulling (Getting Updates)

To get the latest changes from your teammates:

Bash
```
git pull origin main
```

5. Branching & Collaboration

In DevOps, we never work directly on the `main` branch. We use **Feature Branches**.

Branching Strategy

1. **`main`**: Production-ready code only.
2. **`develop`**: Where features are integrated.
3. **`feature/xyz`**: Where individual tasks are done.

Commands:

Bash
git branch feature-login # Create branch
git checkout feature-login # Switch to it
OR
git checkout -b feature-login # Create and switch in one step

6. Advanced Operations (The "DevOps Pro" Tools)

1. The Concept: Two Ways to Combine

Imagine two developers started from the same point. Developer A stayed on `main`, and Developer B went to a `feature` branch.

A. Git Merge (The Honest Historian)

Merging takes the contents of your feature branch and integrates them into the main branch in one "Merge Commit."

- **What it looks like:** A "diamond" shape in the logs. It shows exactly when the branches split and when they came back together.
- **Why use it:** It preserves the **complete history**. You can see exactly what happened and when.
- **Downside:** If you have many developers, the history graph becomes a "spaghetti" mess of lines and merge commits.

B. Git Rebase (The Clean Storyteller)

Rebasing "moves" your entire feature branch so it starts from the very latest commit on the main branch. It rewrites history.

- **What it looks like:** A single straight line. It looks like you developed the feature perfectly on top of the latest code.

- **Why use it:** It creates a **clean, linear history**. It's much easier to read and follow.
- **Downside:** It **rewrites history**. You should *never* rebase a branch that has already been pushed to a public repository (The Golden Rule of Rebasing).

Feature	Git Merge	Git Rebase
History	Preserves original history	Rewrites history to be linear
Commit Logs	Adds an extra "Merge commit"	No extra merge commits
Complexity	Simple and safe	Can be tricky (Conflict resolution)
Traceability	Easy to see where features started	Harder to see when a branch was created

3. Hands-on Lab: "The Conflict Race"

We need to see what happens when we try to change the same file.

Step 1: The Setup

1. Create a folder and a file `script.py`. Add one line: `print("Start")`.
2. Commit it to `main`.
3. Create a branch: `git checkout -b feature-a`.

Step 2: The Divergence

1. **On `feature-a`**: Change the line to `print("Feature A version")`. Commit it.
2. **Switch to `main`**: `git checkout main`.
3. **On `main`**: Change the same line to `print("Main branch version")`. Commit it.
   Now the two branches have "diverged" at the same line.

Step 3: The Merge Challenge

1. While on `main`, run: `git merge feature-a`.
2. **Result:** A **Merge Conflict**!
3. Open the file, delete the `<<<< HEAD` markers, pick the version they want, and commit.

Step 4: The Rebase Challenge

(Undo the merge or start a new branch `feature-b` to try this)

1. `git checkout feature-b`.
2. Run: `git rebase main`.
3. **Observation:** You will notice they have to fix the conflict **commit by commit** rather than all at once. Once finished, run `git log --oneline --graph` to see the straight line.

Special Tools

- **Squash:** Combines 10 small "fix typo" commits into 1 clean "Feature Complete" commit.
- **Cherry Pick:** Grab a single specific commit from another branch and apply it to yours. `git cherry-pick <commit-hash>`
- **Tagging:** Marking a specific point in history as a release (e.g., `v1.0`). `git tag -a v1.0 -m "Release version 1.0"`