# GIT

**Version Control Made Simple, Collaboration Made Powerful**

Krishnan

**Introduction to Version Control and GIT Basics**

## 1. Version Control Systems

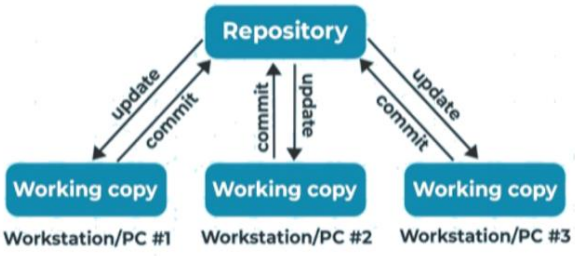
Version Control Systems (VCS) are essential tools in software development and project management, allowing multiple contributors to work together on a project while keeping track of changes, avoiding conflicts, and maintaining a detailed history of modifications.

### Version Control

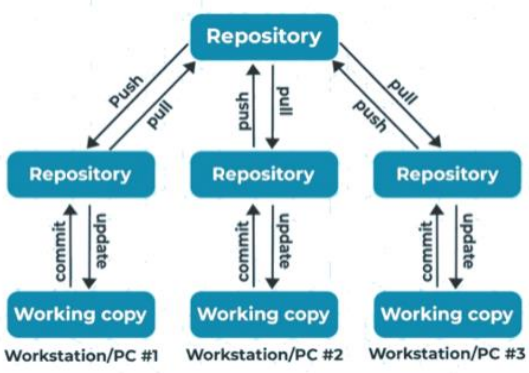
Version control is the practice of tracking changes to files (code, documents, etc.) in a systematic way. It helps in managing multiple versions of a file, keeping a detailed history of who made changes, what changes were made, and when.

### Types of Version Control Systems

1. **Centralized Version Control (CVCS)**:
   - A central server holds the code, and all changes must be committed to this server (e.g., SVN).
   - **Advantages**: Simple and straightforward for small teams.
   - **Disadvantages**: If the server fails, access to the project is lost, and offline work is limited.




2. **Distributed Version Control (DVCS)**:
   - Every developer has a full copy of the repository also in the cloud repository (e.g., Git).
   - **Advantages**: Offline work possible, no central point of failure, easier collaboration.
   - **Disadvantages**: Slightly more complex setup, larger initial clone due to full history.



**Benefits of Version Control**

- **Collaboration:** Developers can work independently on different parts of the project and merge changes without overwriting each other's work.
- **Backup and Recovery:** Since every change is saved, it's easy to recover previous versions or undo mistakes.
- **Tracking Project History:** Detailed logs of changes (commits) are stored, providing insights into what changed, why, and by whom, making it easier to track the evolution of a project.
- **Code Conflict Resolution:** When multiple people work on the same files, version control helps merge their changes, minimizing conflicts and reducing the risk of lost work.
- **Branching and Experimentation:** Developers can create isolated branches for new features or experiments, keeping the main project stable until changes are ready to be merged.

**Popular Version Control Systems**

**Git:** A distributed version control system, Git is the most widely used VCS due to its flexibility, speed, and support for branching and merging. It allows each developer to have a local repository with the full project history.

**SVN (Subversion):** A centralized version control system, SVN stores all changes on a central server. It's still used in some organizations but lacks the flexibility and offline capabilities of Git.

**Mercurial:** Similar to Git in its distributed model, Mercurial is known for being simple and user-friendly but is less widely adopted.

## 2. Getting Started with Git

**Installing Git**

- **Windows**:
    1. Visit the *official Git website*. Download the Windows installer.
    2. Run the installer and follow the prompts, accepting the default settings for most users. You'll also be prompted to install Git Bash, which provides a command-line interface for executing Git commands.
- **macOS**:

  Make sure you have homebrew installed in the Mac Device. To install Git using Homebrew by run the following command in the Terminal:

  ```
  brew install git
  ```

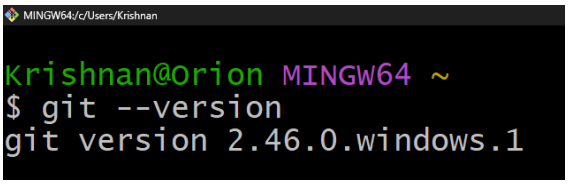
- **Linux**:

  Most Linux distributions come with Git available in their package managers. Use the following command based on your distribution:

    - For Ubuntu/Debian: `sudo apt install git`
    - For Fedora: `sudo dnf install git`
    - For Arch Linux: `sudo pacman -S git`

After installing Git, to verify whether GIT installed or not execute the below command in Terminal or bash window. This will display the version of git we have installed

```
git --version
```




It's essential to configure it with your user information. To do that execute commands in the terminal:

```
git config --global user.name "YourName"
git config --global user.email "youremail@example.com"
```

```
C:\Users\Krishnan>git config --global user.name "dreadwing148"

C:\Users\Krishnan>git config --global user.email "krishnan.tlab.py@gmail.com"
```

to verify configurations, we can run the below commands:

```
git config --list
```

This command displays your current configuration settings.

### 3. Basic Git Commands

#### Initializing Repositories

Git manages project versions through repositories. To create a new repository, follow these steps:

1. **Navigate to Your Project Directory**: Open your terminal and change to your project folder using the cd command. For example:

   ```
   cd path/to/your/project
   ```

2. **Initialize the Repository**: Run the command:

   ```
   git init
   ```

   This command creates a hidden **.git** directory within your project folder. This directory contains all the metadata and history for your project.

3. **Checking the Repository Status**: After initializing, you can check the status of your repository using:

   ```
   git status
   ```

   This command shows the current state of the repository, indicating any untracked or modified files. If you see "No commits yet," it means you haven't added any files to your repository yet.

4. **Understanding the .git Directory**: The .git directory is crucial for Git's functionality. It contains:
   - **Objects**: Store the actual data (blobs for file content, trees for directories).
   - **Refs**: Keep track of branches and tags.
   - **HEAD**: Points to the current branch or commit.

## 4. Basic Git Operations

### Cloning Repositories

Cloning is the process of creating a local copy of a remote repository. This is useful when you want to contribute to an existing project or access its files.

1. **Find the Repository URL**: Go to the repository page on GitHub and locate the HTTPS or SSH URL.
2. **Clone the Repository**:

   ```
   git clone https://github.com/username/repository.git
   ```

   This command creates a directory named after the repository and copies all files and the entire history into it.
3. **Navigate into the Cloned Directory**: Once cloned, change into the newly created directory:

   ```
   cd repository
   ```
4. **Explore the Cloned Repository**:

   ```
   git log
   ```
   or
   ```
   git log --oneline
   ```
   or
   ```
   git log --graph
   ```

   This command displays the commit history, including commit hashes, authors, dates, and messages. So whenever it is needed to see the history of the repository use any of the above commands.

### Making Changes and Adding Files(Staging)

After modifying files or creating new ones, you need to stage them before committing. Staging files prepares them for a commit.

1. **Stage Individual Files**: To add a specific file to the staging area, use:

   ```
   git add filename
   ```
2. **Stage All Changes**: To stage all modified and new files, use:

   ```
   git add .
   ```
   or:
   ```
   git add -A
   ```

   This stages all changes, including deletions.
3. **Check Staged Files**: This command what is the current status of repository you cloned to local directory(folder).

   ```
   git status
   ```

### Committing Changes

Committing saves your changes to the local repository with a descriptive message.

1. **Create a Commit**: After staging your files, create a commit with a message describing the changes:

   ```
   git commit -m "Your commit message here"
   ```

   It's good practice to write concise, informative commit messages.
2. **Amending a Commit**: If you need to modify the last commit (e.g., to add missed changes), use:

   ```
   git commit --amend
   ```

   This opens an editor to modify the commit message or include additional changes staged before running the command.

## 5. Git Help and Documentation

### Using Git Help

Getting Help: Git includes a robust help system. You can access help for any command with:

```
git help <command>
```

For example, to get help on the *commit* command:

```
git help commit
```

### Understanding Config Files

Git stores configuration settings at three different levels, allowing developers to customize Git's behaviour on a global, system, or repository-specific level. Understanding and managing these configuration files is key to optimizing your workflow.

Basically, all the user credentials are stored within the ***.gitconfig*** file in the user folder.

- **Git Configuration Levels:**
  - **System-Level Configuration** (/etc/gitconfig): Settings that affect all users on a machine. Requires superuser privileges to modify.
  - **Global-Level Configuration** (~/.gitconfig or ~/.config/git/config): Settings that apply to the user across all repositories. This is the most commonly modified config file.
  - **Local-Level Configuration** (<repo>/.git/config): Settings that only apply to a specific Git repository. Local settings override global ones.
- **Editing Git Config Files:**
  - You can view or edit these configuration files manually or through the Git command-line interface. To view or modify your global configuration, you can run:

    ```
    git config --global –edit
    ```

  - This opens the global configuration file in your default text editor, where you can add or change settings. Once if changes made close the file everything will be saved.

## GIT Advanced Operations and Collaboration

### 1. Branching and Merging

Git's branching and merging allow developers to work independently on features, bug fixes, or experiments without affecting the main project. This ensures changes can be developed, tested, and reviewed before being merged into the main codebase.

**Managing Branches**

A branch in Git represents an independent line of development. The default branch created when you initialize a repository is typically called main (or master in older Git versions). Branches allow you to isolate work, making it easier to test and develop new features or fixes without impacting the main code.

**Key Concepts:**

**Creating a Branch**: To create a new branch, you use git branch <branch-name>. For example, to create a branch called feature-xyz, you'd run:

```
git branch feature-xyz
```

To immediately switch to the new branch after creating it, use:

```
git checkout -b feature-xyz        or        git switch -c feature-xyz
```

**Listing and Viewing Branches**: To see a list of all branches in your repository, use:

```
git branch
```

- The current branch you are working on will have an asterisk (*) next to it.
- You can also list remote branches with:

```
git branch -a
```

**Switching Between Branches**: To move between branches, use the git checkout or git switch commands:

```
git checkout feature-xyz        or        git switch feature-xyz
```

**Deleting Branches**: Once you are done with all the work , you can delete it:

```
git branch -d feature-xyz
```

This deletes the branch **locally** (if it's fully merged with another branch). To force-delete an unmerged branch, use

```
git branch -D feature-xyz
```

**Renaming Branches**: To rename the current branch, use:

```
git branch -m new-branch-name
```

**Renaming Other Branch**: To rename the other branch, use:

```
git branch -m oldBranchName newBranchName
```

## 2. Managing Repositories

### Managing Remote Repositories

A **remote repository** is hosted on platforms like GitHub, GitLab, or Bitbucket. It allows multiple developers to collaborate by pushing and pulling changes.

You can add a remote repository to your local repo by using:

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

Example:

```
git remote add origin https://github.com/username/repo.git
```

You can view the list of remote repositories associated with your project using:

```
git remote -v
```

This will display the fetch and push URLs for each remote.

### Collaborating via repositories (push, pull, fetch)

Collaboration in Git revolves around syncing changes between local and remote repositories. The most common Git commands used for collaboration are push, pull, and fetch.

**git push (Uploading Changes to the Remote Repository) :** After making changes locally, you can upload your changes to the remote repository using git push.

```
git push <remote-name> <branch-name>
```

Example:

```
git push origin main
```

This pushes your commits in the main branch to the origin remote (e.g., GitHub).

**git pull (Downloading and Merging Changes from the Remote):** To download changes from the remote repository and merge them into your local branch, use git pull:

```
git pull <remote-name> <branch-name>
```

Example:

```
git pull origin main
```

This fetches the latest changes from the remote origin and merges them into your local main branch.

**git fetch (Downloading Changes without Merging):** git fetch only downloads changes from the remote repository but does not automatically merge them into your working branch. You can inspect the changes first and merge them manually if needed.

```
git fetch <remote-name>
```

Example:

```
git fetch origin
```

After fetching, you can view the new changes (commits) and decide whether to merge them.

### 3. Merging Branches(Combining Branches)

Once if all the work is done in the branches we created, we can combine it to the main or master branch.

In order merge one branch to another we can use the below command,

```
git merge feature-xyz
```

Based on the current status of repository git will perform Fast-Forward Merges or Three-Way Merge.

**Types of Branching:**

**Fast-Forward Merges**:

- This is the simplest type of merge.
- It happens when your current branch hasn't changed since you created your new branch.
- In this case, Git just moves the branch pointer forward, like flipping a page in a book. No new merge commit is created because the changes are directly added to the current branch.
- In this case, no extra commits are created because the branch history is linear.

**Example**:

- You create a new branch, make some changes, and then merge it back to the main branch.
- If no one else has changed the main branch, Git can "fast-forward" the main branch to include your changes without creating extra commits.

**Three-Way Merges**:

- This type of merge happens when both your branch and the main branch have new changes since you branched off.
- Git compares three versions of the code: your branch, the main branch, and the point where both branches started (the common ancestor).
- It combines the changes from both branches into one new commit, called a **merge commit**

**Example:**

- You and your friend both make changes in different branches.
- When you try to merge your branch back into the main branch, Git creates a new merge commit to combine the changes from both of you.

**Merge Conflicts**:

- Conflicts occur when Git cannot automatically combine changes from two branches because the same file or line was changed in incompatible ways in both branches.
- Git will mark the conflict and stop the merge process. You'll need to resolve the conflicts manually by editing the files and then committing the resolution.

## 4. Git Workflows

### 1. Centralized Workflow

In the Centralized Workflow, a single central repository serves as the authority where all developers commit their changes. It mimics the workflow used by older version control systems like SVN, where there is only one main branch, often called the main or master branch, and all developers work directly on this branch.

**How it Works**:

- Developers clone the central repository.
- They make changes in their local environment.
- Once changes are ready, they push them directly to the central repository.
- There's no branching involved unless absolutely necessary, and all changes are committed directly to the main branch.

**Advantages**:

- Simple and easy to understand.
- Ideal for small teams or single-developer projects.

**Disadvantages**:

- Hard to scale for large teams. Lacks flexibility for feature isolation
- Higher risk of conflicts since all changes are directly on the same branch.

### 2. Feature Branch Workflow

In this workflow, each new feature or bug fix is developed in its own isolated branch. Once the feature is complete, the branch is merged back into the main branch (often the main or develop branch).

**How it Works**:

- Developers create a new branch for each feature (git checkout -b feature-branch).
- They work on the feature in isolation.
- Once the feature is complete, it is merged into the main branch using a pull request or a direct merge.
- The feature branch is then deleted.

**Advantages**:

- Isolates work, reducing the risk of conflicts.
- Encourages clean and organized commit histories.
- Suitable for teams working on multiple features simultaneously.

**Disadvantages**:

- Requires more branch management.
- Continuous testing and integration are needed to ensure compatibility of different branches when merged.

### 3. Gitflow Workflow

Gitflow is a structured branching model introduced by Vincent Driessen. It is designed for projects with scheduled releases and a defined structure for how work progresses through different stages.

**How it Works**:

- The repository has two long-lived branches: main (or master) for production-ready code and develop for ongoing development.
- Developers create feature branches from develop for new work.
- Once features are complete, they are merged back into develop.
- When preparing for a release, a release branch is created from develop to prepare the code for production.
- Once the release is stable, it is merged into both main and develop.
- For urgent fixes, a hotfix branch is created from main and merged back into both main and develop after the fix.

**Advantages**:

- Provides a structured and clear release process.
- Isolates different stages of development, making it easier to manage production releases and ongoing development.

**Disadvantages**:

- Can be complex for small teams or projects with continuous delivery.
- Requires careful branch management to avoid confusion.

### 4. Forking Workflow

Popular in open-source projects, where external contributors do not have direct access to the main repository. Instead, they fork the repository, make changes in their copy, and submit a pull request to merge their changes.

**How it Works**:

- A developer forks the original repository, creating a personal copy under their account.
- They make changes in their fork, typically using feature branches.
- Once changes are complete, they push them to their fork and create a pull request to the original repository.
- Project maintainers review and merge pull requests into the main project.

**Advantages**:

- Maintains strict control over the main repository.
- Ideal for large-scale, open-source projects where multiple contributors are involved.

**Disadvantages**:

- Higher overhead for maintainers to review and merge pull requests.
- Contributors need to keep their forks in sync with the original repository.