Overview

- Version control systems are essential tools for managing changes to code, documents, and other files.
- Git is one of the most popular version control systems, known for its speed, efficiency, and distributed nature.
- It allows multiple developers to collaborate on projects seamlessly.

Installing Git Client (CLI and GUI) in Linux Environment:

CLI Installation:

1. Open a terminal window.

2. Run the following command to install Git:

```
sudo apt update
sudo apt install git
```

3. Once installed, verify the installation by typing:

```
git --version
```

GUI Installation:

1. There are various Git GUI clients available, such as GitKraken, Sourcetree, and GitHub Desktop.

2. Download and install your preferred Git GUI client following the instructions on their respective websites.

Initializing a Git Repository and Exploring Git

Initializing a Repository:

1. Navigate to the directory where you want to initialize the Git repository.

2. Run the following command to initialize a Git repository:

```
git init
```

Exploring Git Help

1. To access Git's help documentation, use the following command:

```
git --help
```

2. This command provides a comprehensive list of Git commands along with their descriptions and usage examples.

Exploring GitHub and Creating a Public Repository

Creating a Repository on GitHub:

1. Sign in to your GitHub account.
2. Click on the "+" icon in the top-right corner and select "New repository."
3. Fill in the repository name, description, and choose whether it should be public or private.
4. Click on "Create repository."

Understanding Controls on the GitHub Panel:

1. The repository homepage displays various tabs such as Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, Settings, etc.
2. Each tab provides specific functionalities for managing the repository, tracking issues, reviewing pull requests.
3. Running automated workflows, managing project boards, documenting with wikis, monitoring security vulnerabilities.
4. Analysing repository insights, and configuring repository settings.

Working on GitHub Alone

1. As a solo developer, you can utilize GitHub for version control, issue tracking, project management, and documentation.
2. You can push your local Git repository to GitHub using the `git push` command.

3. You can create branches, commit changes, and merge them using GitHub's web interface or Git commands.

Significance of Git Client for GitHub Utilization:

1. Git clients provide a user-friendly interface for interacting with Git repositories, making it easier to visualize changes, manage branches, and collaborate with others.

2. Git GUI clients offer features such as visual commit history, branch management, conflict resolution, and integration with issue trackers.

3. They streamline the Git workflow, especially for those who are less comfortable with command-line interfaces.

Conclusion:

Version control is a crucial aspect of software development, and Git, along with platforms like GitHub, provides powerful tools for managing project collaboration and codebase maintenance.

By understanding how to set up Git clients, initialize repositories, explore Git commands, create repositories on GitHub, and utilize Git GUI clients effectively, developers can streamline their workflow and enhance productivity.

Working With Git

1. **Initiating a Repository**

To initialize a new Git repository in the current directory, use:

```
git init
```

2. **Managing Repositories**

- To clone an existing repository from a remote location, use:

```
git clone <repository_url>
```

- To add files to the staging area, use:

```
git add <file_name>
```

- To commit changes to the repository, use:

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

- To view the status of files in the repository, use:

```
git status
```

Life Cycle of a File in Git Managed Repositories:

1. **Untracked :**
   Files that are not yet tracked by Git.
2. **Staged :**
   Files that are added to the staging area using `git add`.
3. **Committed :**
   Files that are permanently stored in the repository after being committed using `git commit`.

Git Branches and HEAD

Branches:

Independent lines of development within a Git repository.

HEAD:

A reference to the latest commit in the currently checked-out branch.

Branch Management

- To list all branches in the repository, use:

```
git branch
```

- To create a new branch, use:

```
git branch <branch_name>
```

- To switch to a different branch, use:

```
git checkout <branch_name>
```

Commit Changes in a New Branch

1. Create a new branch:

```
git branch <new_branch>
```

2. Switch to the new branch:

```
git checkout <new_branch>
```

3. Make changes to files and commit:

```
git add <file_name>
git commit -m "Commit message"
```

Explore Commit in the New Branch

- To view commit history in the current branch, use:

```
git log
```

- This displays a list of commits with their hashes, authors, dates, and commit messages.

Git Cloning and Exploring Public Repositories

Cloning a Repository

- To clone a public repository from GitHub, use:

```
git clone <repository_url>
```

Exploring Contents of Cloned Repository:

Navigate to the cloned repository directory.

Use commands such as `ls` to list files and directories, and `cat` or `less` to view file contents.

Unpacking Git Objects

Git objects such as commits, trees, and blobs are stored in the `.git` directory.

These objects can be explored further using Git commands and by examining the contents of the `.git` directory.

Exploring Cloned Repository in GitHub Desktop

- Open GitHub Desktop.

- Click on "File" -> "Clone repository" and select the cloned repository from the list.

- Explore the repository's files, commit history, and branches using the GitHub Desktop interface.

Commit Changes in the Cloned Repository

- Make changes to files in the cloned repository.

- Stage the changes using `git add`.

- Commit the changes using `git commit`.

Git Configuration Files

Creating Personalized Configurations

Git configuration settings are stored in two main files:

system-wide (`/etc/gitconfig`) and user-specific (`~/.gitconfig`).

- To create personalized configurations, use:

```
git config --global <key> <value>
```

- Example:

```
git config --global user.name "Your Name"
```

Conclusion

Understanding Git commands for repository management, file lifecycle, branch management, cloning, and configuration is essential for efficient collaboration and version control in software development projects.

With these skills, developers can effectively initiate repositories, manage branches, explore and modify public repositories, and personalize their Git configurations according to their needs.

Git Attributes

They are used to control how Git handles files on a repository-wide or per-file basis.

They can be used to set attributes such as text/binary, line-ending preferences, and custom merge strategies.

Attributes are defined in a `.gitattributes` file in the root directory of the repository.

Managing, Filtering, and Masking

Managing:

Git attributes can manage how files are treated by Git during operations like merging and diffing.

Filtering:

Git attributes can be used to filter content during operations like smudging (on checkout) and cleaning (on commit).

Masking:

Attributes can also be used to mask sensitive information such as passwords or API keys.

Gitignore

`.gitignore` files specify intentionally untracked files to ignore.

They can be used to exclude files like build artifacts, logs, and temporary files from version control.

Syntax:

Patterns in `.gitignore` follow a specific syntax.

Wildcards (`*` for any number of characters, `?` for a single character) can be used.

Prefixing a pattern with a `/` specifies a directory pattern.

A leading `!` negates the pattern.

Staging Files

Staging files prepares them to be included in the next commit.

Use `git add <file>` to stage specific files or `git add .` to stage all modified files.

The staging area is where changes are prepared before committing to the repository.

Working With Git History

Forensics on Git Logs:

Git logs provide a detailed history of commits.

Use `git log` to view the commit history, including commit messages, authors, dates, and hashes.

Additional options can be used to filter and format the log output.

Graphical History:

Tools like `gitk` or graphical interfaces like GitHub's commit graph provide a visual representation of the commit history.

This allows for a clearer understanding of branch histories and relationships between commits.

Undo Changes in History

Amend :

Use `git commit --amend` to modify the last commit, combining staged changes with the previous commit.

Reset :

Use `git reset HEAD~1` to undo the last commit, keeping changes in the working directory.

Revert:

Use `git revert <commit>` to create a new commit that undoes the changes introduced by a specific commit.

Merge Resolution in Git

Branching and Tagging

Branching allows for parallel lines of development within a repository.

Tags provide named snapshots of specific commits for easy reference.

Scenario Creation for Conflict Creation

Single User :

Create conflicting changes on different branches and attempt to merge them.

Multiple Users : Collaborate with others, each making conflicting changes on separate branches, then merge them.

Conflict Resolution

Git provides tools to resolve merge conflicts, such as:

Manual editing of conflicted files.

`git mergetool` for using a visual merge tool.

`git checkout --ours` or `git checkout --theirs` to choose one side's changes.

Conclusion

Understanding Git attributes, Gitignore, staging files, working with Git history, and merge resolution is essential for effective version control and collaboration.

With these skills, developers can manage file attributes, ignore untracked files, stage changes, analyze commit history, undo changes, handle merge conflicts, and ensure smooth branching and tagging workflows in their Git repositories.

Git Branch

Basics of Branching:

Branching allows for parallel development paths within a Git repository.

- To create a new branch, use:

```
git branch <branch_name>
```

- To switch to a different branch, use:

```
git checkout <branch_name>
```

Basic Conflict and Merge Resolution Workflow:

1. **Create Changes on Different Branches :**

   Make changes on different branches, for example, `feature` and `master`.

2. **Merge Branches :**

   Attempt to merge the `feature` branch into `master`.

3. **Conflict Occurs :**

   If changes on `feature` and `master` conflict, Git marks the files as conflicted.

4. **Resolve Conflict:**

   Manually resolve conflicts by editing the conflicted files.

5. **Commit Changes:**

After resolving conflicts, commit the changes to finalize the merge.

GitHub and Remote Repositories

- To clone a remote repository from GitHub, use:

```
git clone <repository_url>
```

Remote Repository

A remote repository is a version of a repository stored on a server, typically accessible over the internet.

It allows multiple developers to collaborate on a project by pushing and pulling changes.

Git Push, Fetch, and Pull Operations

- To push local commits to the remote repository, use:

```
git push origin <branch_name>
```

FETCH_HEAD

After fetching changes from the remote repository, Git stores information about the fetched branch in `FETCH_HEAD`.

Use `git merge FETCH_HEAD` to merge the fetched changes into the current branch.

Performing a Git Pull

- To fetch and merge changes from the remote repository, use:

```
git pull origin <branch_name>
```

Git Pull with Fast Forward Merge

If the remote branch has advanced since the last local pull, Git performs a fast-forward merge.

This updates the local branch to match the remote branch's state without creating a merge commit.

Resolving Conflicts During Git Pull

If changes on the remote branch conflict with local changes, Git marks the files as conflicted during the pull operation.

Resolve conflicts by editing the conflicted files, then commit the changes to complete the pull.

Conclusion:

Understanding Git branching, conflict resolution workflows, remote repositories, and push/pull operations is crucial for efficient collaboration and version control in software development projects.

With these skills, developers can effectively manage branches, resolve conflicts, clone and interact with remote repositories, and synchronize changes between local and remote repositories using Git.