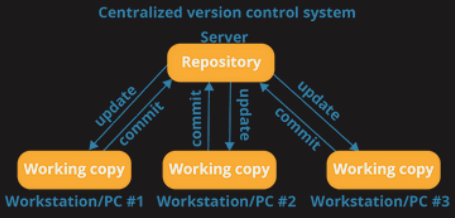
Git & Github Short Notes

Before getting dive into Git and GitHub we've to know some basic terminologies

There is a term Source Code Management and it has two types:-

- CVCS—Centralized Version Control System
- DVCS—Distributed Version Control System

CVCS—Centralized Version Control System

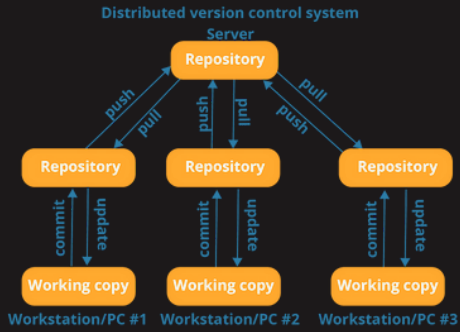


source: edureka

Note

- It is not locally available, meaning we've always needed to be connected to a network to perform any action.
- Since everything is centralized, if the central server gets failed, you will lose the entire data

DVCS — Distributed Version Control System



source: edureka

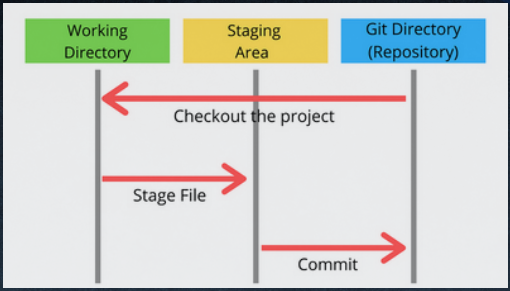
Note

- In DVCS, every contributor has a local copy or 'clone' of the main repository i.e- everyone maintains a local repository of their own, which contains all the files & metadata present in the main repository.

Why do we need Source Code Management as a DevOps Engineer?

- To use the CICD pipeline in DevOps, you must have the most recent project updates on hand. Because DevOps monitors the most recent code and creates definitions that execute a variety of tasks by user needs, the release definitions that assist in deploying the most recent binaries on your primary environment also use these definitions. Any end server where the finished product is made ready for usage might be your client computer, the production environment, or both.

Git three-stage architecture



Important Terms

Repository

- A repository is a place where you have all your codes or kind of folder on the server.
- It is a kind of folder related to one product.
- Changes are personal to that particular repository.

Server

- It stores all repository
- It contains metadata also

Working directory

- Where you see files physically and do the modification.
- At a time, you can work on a particular branch.

Commit

- Store changes in the repository. You will get one Commit-Id.
- It is 40 Alpha-Numeric characters.
- It uses the SHA1 checksum concept.
- Even if you change one dot, Commit-Id will change.
- Commit is also named the SHA-1 hash

Commit Id/Version-Id/Version

- Reference to identify each change.
- To identify who changed the file.

Tags

- Tags assign a meaningful name with a specific version in the repository. Once a tag is created for a particular save, even if you create a new commit, it will not be updated.

Snapshots

- Represents some date of a particular time.
- It is always incremental i.e- It stores the change (append date) only. Not the entire copy.

Push

- Push operations copy changes from a local repository server to a remote or central repository. This is used to store the changes permanently in the git repository.

Pull

- Pull operation copies the changes from a remote repository to a local machine. The pull operation is used for synchronization between the repository.

All about Git Branch

- The product is the same, so one repository but a different task.
- Each task has one separate branch.
- Finally merges(code) all Branches.
- Changes are present in that particular branch.
- The default branch is ‘Master’.
- File created in the workspace will be visible in any of the branch workspaces until you commit, once you commit then that file belongs to that particular branch.
- After done with the code, merge other branches with ‘Master’.
- This concept is useful for parallel development.
- You can create any number of branches.
- When a new branch is created, data from the existing branch is copied to the new branch.

Commands for Branch

To show all branches

```
git branch
```

Create a new branch

```
git branch <branch_name>
```

For going to a specific branch/Change branch

```
git checkout <branch_name>
```

Delete a branch

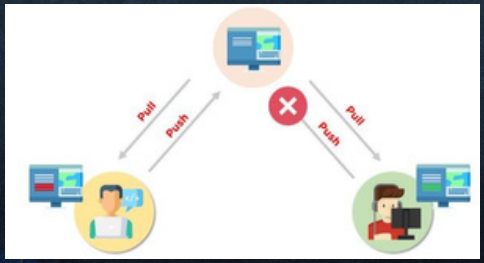
```
git branch -d <branch_name>
```

Commands for Branch merge

- We can't Merge branches of different Repositories.
- We use the pulling mechanism to merge Branches.

```
git merge <branch_name>
```

Conflicts in Git and how to resolve



source: simplilearn

- When the same file has different content in different branches, if you do merge, conflict occurs (Resolve conflict then add and commit)

How Do You Fix Conflicts When Merging in Git?

The procedures required to resolve merge conflicts in Git might be shortened by taking a few specific actions.

- Opening the conflicting file and making the appropriate adjustments is the simplest approach to remedy the issue.
- After making changes to the file, we may stage the newly merged material using the git add command.
- The git commit command is used to generate a new commit as the last step.
- To complete the merging, Git will generate a new merge commit.

Git commands to resolve conflicts

The 'git log --merge' command helps to produce the list of commits that are causing the conflict

```
</>
git log merge
```

The 'git diff' command helps to identify the differences between the state's repositories or files

```
</>
git diff
```

The 'git checkout' command is used to undo the changes made to the file, or for changing branches

```
git checkout
```

The 'git reset --mixed' command is used to undo changes to the working directory and staging area

```
git reset mixed
```

The git merge --abort command helps in exiting the merge process and returning back to the state before the merging began

```
git merge abort
```

The git reset command is used at the time of merge conflict to reset the conflicted files to their original state

```
git reset
```

Basic Git commands

Set global username and email for Git (Locally).

```
git config --global user.name "<your username>"
git config --global user.email "<your email>"
```

Initialise an empty Git Repository

```
git init
```

Clone an existing Git Repository

```
git clone <repository_url>
```

Add file/stage to git

```
git add <filename>
```

Add all the current directory files to git

```
git add .
```

Commit all the staged files to git

```
git commit -m "<your_commit_message>"
```

Restore the file from being modified to Tracked

```
--> git restore <filename>
--> git checkout <filename>
```

Show the status of your Git repository

```
git status
```

Show the branches of your git repository

```
git branch
```

Checkout to a new branch

```
git checkout -b <branch_name>
```

Checkout to an existing branch

```
git checkout <branch_name>
```

Remove a branch from Git

```
git branch -d <branch_name>
```

Show remote origin URL

```
git remote -v
```

Add remote origin URL

```
git remote add origin <your_remote_git_url>
```

Remove remote origin URL

```
git remote remove origin
```

Fetch all the remote branches

```
git fetch
```

Push your local changes to the remote branch

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

Pull your remote changes to the local branch

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

Check your git commits and logs

```
git log
```

📌 You can also refer to my git gist note for these basic commands

```
gist.github.com/LondheShubham153/0b367734a02e8cf77e6ea1dc90ee0f38
```

What is Cherry Picking in Git

Cherry picking is the act of picking a commit from a branch and applying it to another. git cherry-pick can be useful for undoing changes. For example, say a commit is accidentally made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong.

Git Stash and pop

Git Stashing

- Generally, the stash means “store something safely in a hidden place.” Suppose you’re implementing a new feature for your product. Your choice is in progress and suddenly a customer escalation comes because of this, you have to keep aside your new feature work for a few hours. You cannot commit your partial code and also cannot throw away your changes. so you need some temporary storage, where you can store your partial changes and later on commit them.

Command for Stashing

To stash an item

```
git stash
```

To see stashed items list

```
git stash list
```

To apply stashed items

```
git stash apply stash@{<list_number>}
```

To clear the stash items

```
git stash clear
```

Git Stash Pop (Reapplying Stashed Changes)

- Git allows the user to re-apply the previous commits by using the git stash pop command. The popping option removes the changes from the stash and applies them to your working file.

Poping an item from stash

```
git stash pop
```

Git Stash Drop (Unstash)

- The git stash drop command is used to delete a stash from the queue. Generally, it deletes the most recent stash.

```
git stash drop
```

What is git rebase?

- Rebasing is the process of moving or combining a sequence of commits to a new base commit. Rebasing is most useful and easily visualized in the context of a feature branching workflow. The primary reason for rebasing is to maintain a linear project history.

Git Stash Drop (Unstash)

```
git rebase <base>
```

What is git squash?

- To “squash” in Git means to combine multiple commits into one. You can do this at any point in time (by using Git’s “Interactive Rebase” feature), though it is most often done when merging branches.

How to Squash Your Commits

- There are different ways and tools when it comes to squashing commits. In this post, we’ll talk about Interactive Rebase and Merge as the two main ways to squash commits.
- Step1: Check the commit history

To check the commit history, run the below command:

```
git log --oneline
```

- Step 2: Choose the commits to squash.
- Suppose we want to squash the last commits. To squash commits, run the below command:

```
git rebase -i HEAD ~3
```

- The above command will open your default text editor and will squash the last commits.
- Step 3: update the commits
- On pressing enter key, a new window of the text editor will be opened to confirm the commit. We can edit the commit message on this screen.