-: DevOps - Source Code Management – GIT :-

Git basics

Git is a free and open source version control system, originally created by Linus Torvalds in 2005.

Git is a Distributed Version Control system (DVCS). It can track changes to a file and allows you to revert back to any particular change. A distributed version control system (DVCS) is a type of version control where the complete codebase — including its full version history — is mirrored on every developer's computer.

Git has excellent support for branching, merging, and rewriting repository history, which has lead to many innovative and powerful workflows and tools. Pull requests are one such popular tool that allow teams to collaborate on Git branches and efficiently review each others code. Git is the most widely used version control system in the world today.

Here is a basic overview of how Git works:

1. Create a "repository" (project) with a git hosting tool
2. Copy (or clone) the repository to your local machine
3. Add a file to your local repo and "commit" (save) the changes
4. "Push" your changes to your main branch
5. Make a change to your file with a git hosting tool and commit
6. "Pull" the changes to your local machine
7. Create a "branch" (version), make a change, commit the change
8. Open a "pull request" (propose changes to the main branch)
9. "Merge" your branch to the main branch

Git Init

The git init command is usually the first command you'd run in any new project that is not already a Git repository (also commonly called repo).

It can be used to convert an existing, un-versioned folder into a Git repository. Also, you can use it to initialize an empty repo.

Using Git Init Command

cd into the directory you want to initialize.

Then, run this command.

$ git init

This will transform the current directory into a Git repository. A **.git** sub-directory will be added. This will allow you to start recording multiple versions of your project.

Note: Running **git init** *on an already initialized directory will not override any of your settings.*

Let's create our first repository via GIT UI : (Lab Part-1)

1. *Login to* **www.github.com**
2. *Click on User settings*
3. *Click on Your Repositories*
4. *Click on New to create a new repository*
5. *Enter Repository name - <your_name>_project_repo*
6. *Add Description – "This is my project repo for source code"*
7. *Select Public option*
8. *Check Add a README file*
9. *Click on Create Repository*

Git Clone

The git clone command is used to download the source code from a remote repository (like GitHub, Bitbucket, or GitLab).

Using Git Clone Command

$ git clone <https://url-of-the-repository>

When you clone a repo, the code is automatically downloaded to your local machine.

Let's clone your repository in to your laptop: (Lab Part-2)

1. *Go to Repositories page*
2. *Click on your repository - <your_name>_project_repo*
3. *Click on 'Code'*
4. *Copy HTTPS URL - https://github.com/<yourGitid>>/<yourname>_project_repo.git*
5. *Clone repository*
   $ git clone https://github.com/<yourGitid>>/<yourname>_project_repo.git
6. *Switch to project directory*
   $ cd <yourname>_project_repo
7. *List files*
   $ ls
   README.md
   $ git branch
   ** main*

Git Branch

This command allows teams to work on the same code base in parallel.

Say, you're working on "*Feature A*". And your teammate is working on "*Feature B*". By creating a separate branch for each feature, both of you can work on the same code base in parallel without having to worry about conflicts (at least while you're writing the code).

We can use this command to create a new branch, view existing branches, or delete a branch.

Using Git Branch to create a new branch

$ git branch <branch-name>

This command will create a new branch only in your local system. If you want this to be visible to all the members in the repo, you'll still have to push the branch.

Command to view all branches

$ git branch
$ git branch --list

Git command to delete a branch

$ git branch -d <branch-name>

Let's create few branches in our repository: (Lab Part-3)

1. Create **release_2021_1.0** branch under main
   $ git branch release_2021_1.0
2. Checkout to that branch
   $ git checkout release_2021_1.0
   Switched to branch 'release_2021_1.2'
3. Verify if your local active branch is now release_2021_1.0
   $ git branch
   main
   * release_2021_1.0
4. Push this new branch to github
   $ git push --set-upstream origin release_2021_1.2

Git Checkout

After we create a new branch , we need to switch to the new branch to start working on it. We use 'checkout' command to switch between branches.

Using Git Checkout Command

$ git checkout <branch-name>

This will automatically switch you to the branch name you mentioned in the command.

However, when switching from one branch to another, you need to keep two things in mind:

- If you made some changes in the previous branch, you will have to first commit and push them to your remote repo. Or stash them.
- The branch you want to switch must be present in your local system. If not, you can pull them.

Git Add

Every time you create a new file, delete it, or make a change, you'll have to tell Git to track it and add it to the staging area. Otherwise, push does not pick your changes.

Using Git Add Command

$ git add <file-name>

This command will add only a single file to your next commit. If you want to add all the files to which changes were made, you can use

$ git add --all

Let's add few folders and files: (Lab Part – 4)

```
$ git checkout release_2021_1.0
$ mkdir 01_shell_scripts
$ cd 01_shell_scripts
$ cat > first_file.txt
This is my first file in GIT.
I have created this on this auspicious day.
<CTRL-D>
$ cd ..
$ cat > second_file.txt
This is second dummy file
<CTRL-D>
$ git status
  On branch release_2021_1.0
  Your branch is up to date with 'origin/release_2021_1.0'.

  Untracked files:
   (use "git add <file>..." to include in what will be committed)
  01_shell_scripts/
  second_file.txt

  nothing added to commit but untracked files present (use "git add"
  to track)
$ git add second_file.txt
$ git add –all
$ git status
  On branch release_2021_1.0
  Your branch is up to date with 'origin/release_2021_1.0'.
```

```
    Changes to be committed:
      (use "git restore --staged <file>..." to unstage)
            new file:   01_shell_scripts/first_file.txt
            new file:   second_file.txt
$ git commit -m "Added first, second files to new branch"
  [release_2021_1.0 0069a02] Added first, second files to new branch
  2 files changed, 2 insertions(+)
  create mode 100644 01_shell_scripts/first_file.txt
   create mode 100644 second_file.txt
$ git status
  On branch release_2021_1.0
  Your branch is ahead of 'origin/release_2021_1.0' by 1 commit.
  (use "git push" to publish your local commits)
$ git push
  Enumerating objects: 6, done.
  Counting objects: 100% (6/6), done.
  Delta compression using up to 12 threads
  Compressing objects: 100% (3/3), done.
  Writing objects: 100% (5/5), 514 bytes | 514.00 KiB/s, done.
  Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
  To https://github.com/vbhaskarbe/development_repo.git
     db9b64d..0069a02  release_2021_1.0 -> release_2021_1.0
           nothing to commit, working tree clean
$ git status
  On branch release_2021_1.0
  Your branch is up to date with 'origin/release_2021_1.0'.

    nothing to commit, working tree clean
$ git log
     commit 0069a02b2fb387dc5590750d03ac056d110f8aef (HEAD -
  release_2021_1.0, origin/release_2021_1.0)
  Author: bhasvara <bhasvara@cisco.com>
  Date:   Mon Sep 13 14:17:52 2021 +0530
```

Added first, second files to new branch

Git commit

It's commonly used to save your changes.

Every time you commit your code changes, you'll also include a message to briefly describe the changes you made. This helps other team members quickly understand what was added, changed, or removed.

Using Git Commit Command

$ git commit -a

This will commit all the changes in the directory you're working in. Once this command is run, you'll be prompted to enter a commit message.

Alternatively, you can enter the commit message in the command itself and skip the additional step where you'll be prompted to enter the commit message.

$ git commit -am "<commit-message>"

Note: The git commit command saves the changes only in your local repository. Must push changes for them to be available to others..

Amend the most recent commit message with
$ git commit --amend -m "New amended commit message"

Git push

To make all your committed changes available to your teammates, you'll have to push them to the remote origin.

Using Git Push Command

$ git push <remote> <branch-name>

It's important to remember that git push command will upload only the changes you've committed.

Git Status

The Git Status command can tell you all the information about current work you have been fdoing.

Using Git Status Command

$ git status

This can give you information such as:

- Your current branch
- Whether your current branch is up to date
- If there's anything in the branch that needs to be committed, pushed, or pulled.
- If you have any files that are either staged or not staged.
- And if you have any files that are created, modified, or deleted.

Git Log

The Git Log command gives you the information about the commit history for the repository.

Using Git Log Command

$ git log

This displays the entire commit history. If your commit history is large, it'll show only a portion of it and you can hit *[space]* to scroll or type *q* to quit.

- If you want to view only the last 3 commit history, you can use the following command: **git log -n 3**.
- To condense the commit history into a single line and view them, run **git log --oneline**. This is the easiest way to get a high level overview of all the commit history. It might still be a bit too much if you've got a lot of commits.
- If you want to view the commit history by a specific author, run **git log --author"<author-username>"**.

Git Pull

The git pull command allows you to fetch all the changes that your teammates pushed and automatically merge them into your local repo.

Using Git Pull Command

$ git pull <remote>

In many cases, you will run into conflict because you had changed a line in a file that another teammate added. In such cases, you need to resolve the conflicts manually.

Git Diff

The Git Diff command is used to quickly see the difference between your current branch and another branch (usually the branch You are merging into).

Using Git Diff Command

$ git diff

This will show you any uncommitted changes in your local repo.

To compare two branches

$ git diff branch1..branch2

This will show all the file differences between the two branches.

To compare a file from two branches

$ git diff branch1 branch2 ./path/to/file.txt

This command will show a comparison of the changes made to file file.txt across the branches branch1 and branch2.

Git Stash

Git Stash temporarily shelves your work, so you can switch to another branch, work on something else, and then come back to this at a later time.

It's perfect if you need to work on something else and you're midway through a code change, but aren't ready to commit the code.

Using Git Stash Save Command

$ git stash save "<stash-message>"

This will stash your changes with the message you entered. This can be helpful when you want to come back and restore your stash, especially when you have several stashes.

However, this will only stash your tracked files that you added using git add. If you want to include the untracked files as well, run

$ git stash save -u

Using Git Stash List Command

When you want to view all the stashed code, you can view them using this command. Once you stash your code, git will assign a stash id, so you can restore a specific stashed code later.

For example:

$ git stash list

This might show the following

stash@{0}: On master: Stashed with message1
stash@{1}: On master: Stashed with message2

Using Git Stash Apply Command

This will automatically restore and apply the topmost stash in the stack.

$ git stash apply

If you want to restore a specific stash that you want to apply, using the above example, you can simply run **git stash apply stash@{1}**.

Note: When you use **git stash apply***, the stashed version will be applied to your current working branch. However, it will not delete the stash from the stack.*

Using Git Stash Pop Command

To automatically also delete the stash from the stack, the git stash pop command is used.

If you want to do it for a specific stash in the stack, run **git stash pop stash@{0}**.

Git Merge

Once you're done with development inside your feature branch and tested your code, you can merge your branch with the parent branch. This could be either a develop branch or a master branch depending on the git workflow you follow.

When running a git merge command, you first need to be on the specific branch that you want to merge with your feature branch.

Imagine you're currently in your feature branch called feature1 and you're ready to merge it to the develop branch.

We must first switch to the develop branch using the checkout command.

$ git checkout develop

Before merging, you must make sure that you update your local develop branch. This is important because your teammates might've merged into the develop branch while you were working on your feature. We do this by running the pull command.

$ git pull

If there are no conflicts while pulling the updates, you can finally merge your **feature1** branch into the **develop** branch.

We do this by using the git merge command followed by the branch name that we want to merge into our current branch.

$ git merge feature1

Git Fetch

git fetch is a primary command used to download contents from a remote repository.

git fetch is used in conjunction with git remote, git branch, git checkout, and git reset to update a local repository to the state of a remote.

git fetch has similar behaviour to git pull, however, git fetch can be considered a safer, non-destructive version.

$ git fetch origin
$ git fetch -al