☺ Short Notes .... MIX Git and GitHub :-

Thursday, 20 June, 2024 04:32 PM

❖ Git :-

Git is a Distributed Version Control System (VCS).
It is a tools that helps to Track Changes in the code. Git is a tools that helps multiple people work on the same code project or Project or documents by Tracking and managing changes to the files.

What is VCS :-

Every time you make a change , whether it's adding a sentence to a document or altering a line of code, the VCS Records and saves the outcome.

Features of Git - VCS / why USE :-

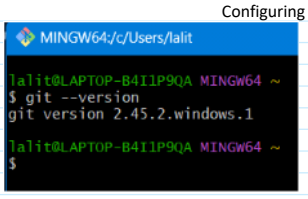
- **Backup and Restore :-** Files are safe against accidental Losses or mistakes.
- **Collaboration :-** Multiple people can work on the same project Simultaneously.
- **Branching and Merging :-** Users can Diverge from the main base of code , experiment , and then bring changes back in line without work.
- **Tracking Changes :-** You can see Specific changes made and by whom.

It is :-

- Popular
- Free & Open Source
- Fast & Scalable

Configuring Git / SetUp :-

Configuring user information used across all local repositories

- 

- Inside " ~ " known as ***Root Directory***(***System ka Main Folder***).

1. git config --global user.name “[firstname lastname]” :- set a name that is identifiable for credit when review version history
2. git config --global user.email “[valid-email]” :- set an email address that will be associated with each history marker
3. git config --list :- what config in GitHub.
4. git config --global color.ui auto :- set automatic command line coloring for Git for easy reviewing

Stage & Snapshot :-

◇ Working with snapshots and the Git staging area.

1. git status :-

show modified files in working directory, staged for your next commit.

```
PS D:\GitHub code\DSA-with-JAVA-Program-> git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- Some git status :-
  - i) UnTracked :- new files that git doesn't yet Track.
  - ii) Modified :- any Track file Changed.
  - iii) Staged :- add file and it is File Ready to be Committed.
  - iv) Unmodified :- All file add after Committed Know as Unchanged.

2. See previous version code :-

- Way 1 :- small Code view.

Syntax :-

git ls-tree <-hashcode->
git show <-Hash-Code->

git show Hash-Code: file_Name.

```
PS D:\GitHub code\LocalRepo> git ls-tree 56fc6d2a74804f1d4b853a466d78473f58713a31
>>
100644 blob 0fe7277dad2549569817b49577bd95d77bbcd75d    Simple.java
100644 blob b289a298d91ed19f8b6b30e3bfa36b7a1cdcb5f5    a.java
100644 blob f5961a01108e6e3ed8bf1b9da8fc47c288e71b7e    b.java
PS D:\GitHub code\LocalRepo> git show 0fe7277dad2549569817b49577bd95d77bbcd75d
//It is a simple java simple code.
// Two no. Add code.
PS D:\GitHub code\LocalRepo>
```

- Way 2 :- large code view .inside on the spot changes in file.

Syntax :-

git checkout Hash-Code -- <-file name -> ya *

  - Visit in latest version/ Code file :- Syntax :- git checkout master -- *

3. git add <- File Name -> :-

Adds new or Change files in your working Directory to the Git staging Area.

```
PS D:\GitHub code\DSA-with-JAVA-Program-> git add README.md
```

All files Add using This Syntax :-

```
PS D:\GitHub code\DSA-with-JAVA-Program-> git add .
```

4. git reset [file] :-

unstage a file while retaining the changes in working directory

5. git diff :-

diff of what is changed but not staged

6. git diff --staged :-

diff of what is staged but not yet commited

7. git commit -m <- “descriptive message” -> :-

commit your staged content as a new commit snapshot.

```
PS D:\GitHub code\DSA-with-JAVA-Program-> git commit -m "First Time Update "
[master 68e5ff9] First Time Update
 1 file changed, 2 insertions(+), 1 deletion(-)
PS D:\GitHub code\DSA-with-JAVA-Program->
```

SetUp & init :-

Configuring user information, initializing and cloning repositories

1. ***Init Commands :-***

init Used to Create a new git repo. And Upload in GitHub.

- Upload a Local Repo. in GitHub :-
  - Step 1 :- Create a folder in VS :-

```
PS D:\GitHub code\DSA-with-JAVA-Program-> cd ..
PS D:\GitHub code> mkdir LocalRepo


    Directory: D:\GitHub code


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        23-06-2024  04:39 PM                LocalRepo
```

  - Step 2 :-

git init :- Create a new git repo. .

```
PS D:\GitHub code> git init
Initialized empty Git repository in D:/GitHub code/.git/
```

  - Step 3 :-

Create a file :-

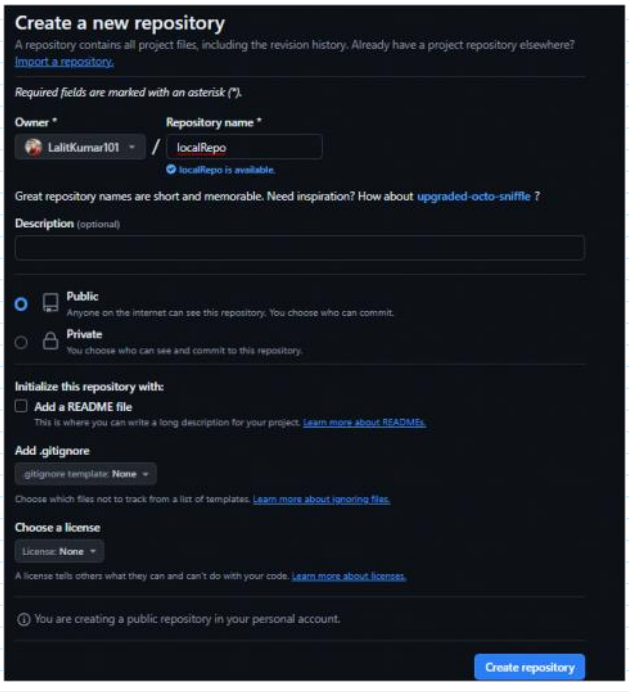
  - Step 4 :-

Add and Commit file :-

```
PS D:\GitHub code\LocalRepo> git add .
PS D:\GitHub code\LocalRepo> git commit -m "Create a New Java File"
[main b6ee5a5] Create a New Java File
 1 file changed, 1 insertion(+)
 create mode 100644 LocalRepo/Simple.java
PS D:\GitHub code\LocalRepo> git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
PS D:\GitHub code\LocalRepo>
```

  - Step 5 :-

Create a Repo in GitHub :-



  - Step 6 :- Copy HTTPS URL

- Step 7 :- Paste URL inside Syntax :- git remote add origin <-URL->

```
PS D:\GitHub code> cd LocalRepo
PS D:\GitHub code\LocalRepo> git remote add origin https://github.com/LalitKumar101/LocalRepo.git
PS D:\GitHub code\LocalRepo>
```

- Step 8 :- To verify Remote.
  Syntax :- git remote -v

```
PS D:\GitHub code\LocalRepo> git remote -v
origin  https://github.com/LalitKumar101/LocalRepo.git (fetch)
origin  https://github.com/LalitKumar101/LocalRepo.git (push)
```

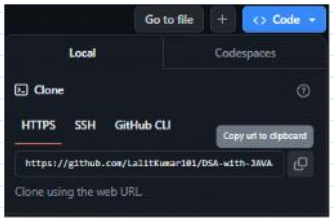
- Step 9 :- To Check Branch.
  Syntax :- git branch

```
PS D:\GitHub code\LocalRepo> git branch
  JDBC
  main
* master
```

- Step 10 :- To Rename Branch. git branch -M <- new Branch Name ->
  Syntax :- git branch -M main

- Last Step 11 :- push all Repo :-
  Syntax :- git push -u origin main :- inside ***-u*** *(Get upstream) again push repo so can't write full syntax only write* ***git push.***

2. git clone [url] :-
   retrieve an entire repository from a hosted location via URL
   And other words Cloning a repository from a Remote Machine (GitHub) on our Local Machine (PC, Laptop).
   - ***Step By Stape Cloning , For Ex. :-***
     - ***Step 1 :-*** *Open a Empty folder in VS code.*
     - ***Step 2 :-*** *Copy HTTPS like from GitHub..*



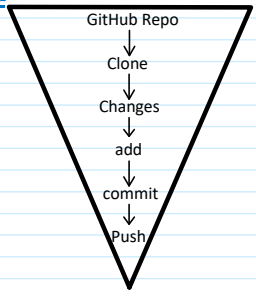
     - ***Step 3 :-*** *Open VS code Terminal and Write …*

```
PS D:\GitHub code> git clone https://github.com/LalitKumar101/DSA-with-JAVA-Program-.git
```

3. ls :- see all files .

WorkFlow :-

Local Git :-



Branch & Merge :-

Isolating work in branches, changing context, and integrating changes.

- ***Branch :-***
  1) git branch :- Check Branch.
     list your branches. a * will appear next to the currently active branch
  2) git branch [branch-name] :-
     create a new branch at the current commit
  3) git branch -M main :-
     To rename Branch.
  4) git checkout -b <- New Branch Name -> :-
     create a new Branch
  5) git checkout <- branch Name-> :-
     To Navigate.
     switch to another branch and check it out into your working directory.
  6) git branch -d <-Branch Name -> :-
     To Delete Branch.
- ***Merge :-***
  1) git diff <- Branch Name -> :-
     To compare commits, Branches, Files And more.
  2) git merge [branch] :- To merge 2 Branches.
     merge the specified branch's history into the current one
  3) git log :-
     show all commits in the current branch's history
  - ***Merging Code :-***
    - ***Way 1 :-*** *git merge <- branch Name ->*
    - ***Way 2 :-*** *Create a PR (Pull Request)*

Pull Request :-

It lets you tell others about changes you've Pushed to a branch in a Repository on GitHub.

Inspect & Compare :-

Examining logs, diffs and object information

1. git log :-
   show the commit history for the currently active branch

2. git log branchB..branchA :-
   show the commits on branchA that are not on branchB
3. git log --follow [file] :-
   show the commits that changed file, even across renames
4. git diff --cached.
5. git diff branchB...branchA :-
   show the diff of what is in branchA that is not in branchB
6. git show [SHA] :-
   show any object in Git in human-readable format

- ***Tracking Path Changes :-***
  Versioning file removes and path changes

1. git rm [file] :-
   Delete the file from project and stage the removal for commit
2. git mv [existing-path] [new-path] :-
   change an existing file path and stage the move
3. git log --stat -M :-
   show all commit logs with indication of any paths that moved

- ***Share & Update :-***
  Retrieving updates from another repository and updating local repos

1. git remote add [alias] [url] :-
   add a git URL as an alias
2. git fetch [alias] :-
   fetch down all the branches from that Git remote
3. git merge [alias]/[branch] :-
   merge a remote branch into your current branch to bring it up to date
4. git push [alias] [branch] /
   git push origin master :- Inside ***origin*** is a default GitHub repo. name and code push in main Branch.
   Transmit local branch commits to the remote repository branch
   Upload local repo(VS code) content to Remote repo (GitHub) using Git command .

```
PS D:\GitHub code\DSA-with-JAVA-Program-> git push origin master
info: please complete authentication in your browser...
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Writing objects: 100% (3/3), 281 bytes | 281.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/LalitKumar101/DSA-with-JAVA-Program-.git
   3ce2b4b..68e5ff9  master -> master
PS D:\GitHub code\DSA-with-JAVA-Program-> git push origin master
Everything up-to-date
PS D:\GitHub code\DSA-with-JAVA-Program->
```

- ***Pull Command :-***
  It used to fetch and Download content from a remote repo. And merge any commits from the tracking remote branch.
  - ***Syntax :-*** git pull origin main

> Other Word Pull command use to fetch some code from GitHub. Syntax :- ***git pull*** --> in vs code
Ex. :-

- ***Resolving Merge Conflicts :-***
  An event that track place when Git is unable to Automatically Resolve Differences in Code Between Two Commits.

- ***Working with Git Restore / Undoing Changes :-***

  - Case 1 :- staged Changes (add but not commit).
    Syntax :- git reset <-file Name->
    git reset

  - Case 2 :- Any changes inside file can't commands. (not add , commit).
    Syntax :- git restore . or <-File_Name-> (. 'Dot' means all Files)

  - Case 3 :- If we added the changes using "git add" then..
    Syntax :- git restore --staged . Or <-File_path-> # To unstage
    git restore <-File_Path-> # To Discard changes in the working Directory.

  - Case 4 :- Added changes to Staging area (Didn't commit) After this added more changes to file.
    Syntax :- // To get the staged changes
    git restore --worktree . Or <-File Path/Name->

  - Case 5 :- How about if we did commit also wrong files.
    Syntax :- git reset --soft HEAD^ (Uncommit and keep the Changes)

git reset --hard HEAD^ (Uncommit and Discard the changes)

- Case 6 :- Committed changes (For one commit)
  Syntax :- git reset HEAD~1

- Case 7 :- Committed changes (For many Commits)
  Syntax :- git reset <-commit hash_Code->
  git reset --hard <-commit hash_Code->
  git

Useful Tips / Different of Git Log :-

- ***Useful Log Options :-***

1) ***git log -p -2 :-*** *Last two commit with diff. .*
2) ***git log --stat :-*** *Summary of Changes.*

```
PS D:\GitHub code> git log --stat
commit 05bb93a6065313afb1a6292ff57f7baa3fc302eb (HEAD -> main, origin/main)
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:58:08 2024 +0530

    Create a class inside a.java file

 LocalRepo/a.java | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)

commit e1034225dacd6632bf1865572dfd8ca7af781632
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:55:42 2024 +0530

    Commit a.java file

 a.java | 1 +
 1 file changed, 1 insertion(+)

commit 1efe3edc7c151e4adcc34a83c28daa3daaa1e9a7
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 08:11:10 2024 +0530
```

1) ***git log --pretty=oneline :-*** *all Hash-code in oneline .*

```
PS D:\GitHub code> git log --pretty=oneline
05bb93a6065313afb1a6292ff57f7baa3fc302eb (HEAD -> main, origin/main) Create a class inside a.java file
e1034225dacd6632bf1865572dfd8ca7af781632 Commit a.java file
1efe3edc7c151e4adcc34a83c28daa3daaa1e9a7 Java
55eeb043579eff0c585216784c1b0729fb730ecb add content inside simple file
56fc6d2a74804f1d4b853a466d78473f58713a31 add content inside simple file
b6ee5a566862e0929a86bdd796d9bb4a5a34022c Create a New Java File
510d190824b78661a5c369c20e0f10abc30003b8 b.java file add
f4b7ca5c2bf5ad24b6d0e461ec18bf25fe377ded (JDBC) add java File
PS D:\GitHub code>
```

2) ***git log --pretty=format:"%h - %an, &ar : %s" :-*** *all log info. In a format/Sequence .*

```
PS D:\GitHub code> git log --pretty=format:"%h - %an, &ar : %s"
05bb93a - Lalit Kumar, &ar : Create a class inside a.java file
e103422 - Lalit Kumar, &ar : Commit a.java file
1efe3ed - Lalit Kumar, &ar : Java
55eeb04 - Lalit Kumar, &ar : add content inside simple file
56fc6d2 - Lalit Kumar, &ar : add content inside simple file
b6ee5a5 - Lalit Kumar, &ar : Create a New Java File
510d190 - Lalit Kumar, &ar : b.java file add
f4b7ca5 - Lalit Kumar, &ar : add java File
PS D:\GitHub code>
```

1) ***git log -S "Function_Name" :-***

```
PS D:\GitHub code> git log -S "import java.io.*;"
commit e1034225dacd6632bf1865572dfd8ca7af781632
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:55:42 2024 +0530

    Commit a.java file

commit f4b7ca5c2bf5ad24b6d0e461ec18bf25fe377ded (JDBC)
Author: Lalit Kumar <lalitranwa2216>
Date:   Mon Jun 24 11:29:17 2024 +0530

    add java File
PS D:\GitHub code>
```

2) ***git log --grep="fix bug" :-*** *Search commit msg.*

```
PS D:\GitHub code> git log --grep="a.java"
commit 05bb93a6065313afb1a6292ff57f7baa3fc302eb (HEAD -> main, origin/main)
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:58:08 2024 +0530

    Create a class inside a.java file

commit e1034225dacd6632bf1865572dfd8ca7af781632
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:55:42 2024 +0530

    Commit a.java file
PS D:\GitHub code>
```

1) ***git log --since="2024-06-24" :-***

2) ***git log --until="2024-06-24" :-***

```
PS D:\GitHub code> git log --until="2024-06-24"
commit b6ee5a566862e0929a86bdd796d9bb4a5a34022c
Author: Lalit Kumar <lalitranwa2216>
Date:   Mon Jun 24 13:07:31 2024 +0530

    Create a New Java File

commit 510d190824b78661a5c369c20e0f10abc30003b8
Author: Lalit Kumar <lalitranwa2216>
Date:   Mon Jun 24 11:39:12 2024 +0530

    b.java file add

commit f4b7ca5c2bf5ad24b6d0e461ec18bf25fe377ded (JDBC)
Author: Lalit Kumar <lalitranwa2216>
Date:   Mon Jun 24 11:29:17 2024 +0530

    add java File
PS D:\GitHub code>
```

1) ***git log --author="Name" :-*** *Find a particular author.*

```
PS D:\GitHub code> git log --author="lalitranwa"
commit 05bb93a6065313afb1a6292ff57f7baa3fc302eb (HEAD -> main, origin/main)
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:58:08 2024 +0530

    Create a class inside a.java file

commit e1034225dacd6632bf1865572dfd8ca7af781632
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 12:55:42 2024 +0530

    Commit a.java file
```

2) ***git log --no-merges.***

- ***Working With Remote Repo. :-***

- ***Fork :-*** Fork is a Rough copy.
  A fork is a new Repository that shares code and visibility settings with the original "UpStream" Repository.

- ***Rewrite History :-***
  Rewriting branches, updating commits and clearing history
  1. git rebase [branch] :-
     Apply any commits of current branch ahead of specified one
  2. git reset --hard [commit] :-
     clear staging area, rewrite working tree from specified commit

- ***Temporary Commits :-***
  Temporarily store modified, tracked files in order to change branches
  1. git stash :-
     Save modified and staged changes
  2. git stash list :-
     list stack-order of stashed file changes
  3. git stash pop :-
     write working from top of stash stack
  4. git stash drop :-
     discard the changes from top of stash stack

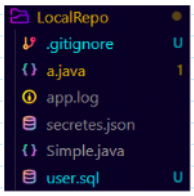
- ***Ignoring Patterns :-***
  Preventing unintentional staging or commiting of files

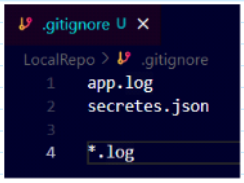
```
logs/
*.notes
pattern*/
```

  - Save a file with desired paterns as .gitignore with either direct string matches or wildcard globs.

  1. git config --global core.excludesfile [file] :-
     system wide ignore patern for all local repositories
  2. .gitignore :- It is a type of file inside any file can't commit.

     Ex. :-

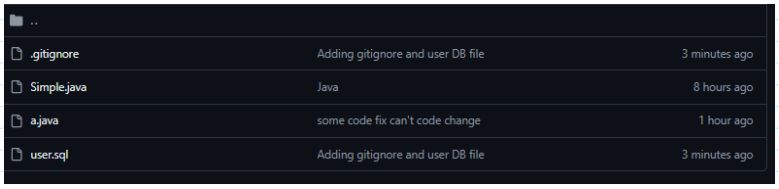





```
PS D:\GitHub code\LocalRepo> git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        user.sql

nothing added to commit but untracked files present (use "git add" to track)
PS D:\GitHub code\LocalRepo>
```

```
PS D:\GitHub code\LocalRepo> git add .
PS D:\GitHub code\LocalRepo> git commit -m "Adding gitignore and user DB file
[main f00329d] Adding gitignore and user DB file
 4 files changed, 4 insertions(+), 3 deletions(-)
 create mode 100644 LocalRepo/.gitignore
 delete mode 100644 LocalRepo/a.class
 delete mode 100644 LocalRepo/b.java
 create mode 100644 LocalRepo/user.sql
PS D:\GitHub code\LocalRepo> git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (5/5), 517 bytes | 517.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/LalitKumar101/LocalRepo.git
   833a923..f00329d  main -> main
```

```
PS D:\GitHub code\LocalRepo> git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
PS D:\GitHub code\LocalRepo>
```



- ***What is Git clean :-***

Clean all Untracked files.

```
PS D:\GitHub code\LocalRepo> git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        Normal_Work.txt
        temporary_file.txt

nothing added to commit but untracked files present (use "git add" to track)
PS D:\GitHub code\LocalRepo>
```

Syntax :-

git clean -n --> it using check to Untrack file.

```
PS D:\GitHub code\LocalRepo> git clean -n
Would remove Normal_Work.txt
Would remove temporary_file.txt
PS D:\GitHub code\LocalRepo>
```

git clean -f --> it using remove untrack file.

```
PS D:\GitHub code\LocalRepo> git clean -f
Removing Normal_Work.txt
Removing temporary_file.txt
PS D:\GitHub code\LocalRepo>
```

```
PS D:\GitHub code\LocalRepo> git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
PS D:\GitHub code\LocalRepo>
```

- **What are Git Tags :-**
  - > *Inside Hash-code with a Identifier tag.*

Syntax :- git tag -a <-Tag_Name-> -m "Message" (To create Annotated tags)
git tag (See all Tags).
git log (See log with Tag).

```
PS D:\GitHub code\LocalRepo> git tag -a version1 -m "This a local Version1"
PS D:\GitHub code\LocalRepo> git tag
version1
commit f00329d1257fbb19d25ba8e6565ad3e4eb71d1ab (HEAD -> main, tag: version1, origin/main)
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 16:35:39 2024 +0530

    Adding gitignore and user DB file
```

- *More tags :-*

a) *git tag -a v1.0 -m "My version 1.0"*

b) *git show v1.0 (To show Tags Data)*

```
PS D:\GitHub code> git show version1
tag version1
Tagger: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 17:56:50 2024 +0530

This a local Version1
```

c) *git tag -a v1.2 <-Hash_Code-> -m "Message" (To tag old commit in case you Forgot)*

```
PS D:\GitHub code> git tag -a codefix 833a9231e848d5baa712055689db1d15709eb661 -m "add Tag in some code fix"
PS D:\GitHub code> git show codefix
tag codefix
Tagger: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 18:16:53 2024 +0530

add Tag in some code fix

commit 833a9231e848d5baa712055689db1d15709eb661 (tag: codefix)
Author: Lalit Kumar <lalitranwa2216>
Date:   Tue Jun 25 15:15:32 2024 +0530

    some code fix can't code change

diff --git a/LocalRepo/a.class b/LocalRepo/a.class
```

d) *git tag -d <-Tag_Hash_Code-> (To Delete a tag)*

e) *git push origin version1.2 (Tages Created remain local, to move it to Remote)*

```
PS D:\GitHub code> git push origin --tags
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 358 bytes | 358.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/LalitKumar101/LocalRepo.git
 * [new tag]         codefix -> codefix
 * [new tag]         gitignore -> gitignore
 * [new tag]         version1 -> version1
PS D:\GitHub code>
```

f) *git push origin --tags (For all tags move in Remote)*

g) *git checkout -b version2 v2.0.0 (It will Create new branch also)*

- *Host Resume , Project , Social Media Links ,Website Using GitHub…….*

❖ **GitHub :-**

It is a Website that allows Developers to Store AND Manage their code using **Git .**

- **https://github.com**
- Inside all code in the form of a *Folder* Known as a *Repository*.

- **What use :-**

☺ Full Explain -> Page No. Git & GitHub :-

Sunday, 2 June, 2024 09:30 PM

GitHub CLI | Take GitHub to the command line

✵ ## Basic Git :-

1. **Introduction to Git**
   - What is Git?
   - Version Control Systems
   - History of Git
   - Git vs Other VCS
   - Basic Concepts

2. **Installation and Configuration**
   - Installing Git on Windows, macOS, and Linux
   - Initial Configuration (git config)

3. **Getting Started with Git**
   - Basic Git Commands
   - Initializing a Repository (git init)
   - Cloning a Repository (git clone)
   - Understanding the Working Directory, Staging Area, and Repository

4. **Basic Operations**
   - Checking Repository Status (git status)
   - Adding Files (git add)
   - Committing Changes (git commit)
   - Viewing Commit History (git log)

5. **Branching and Merging**
   - What is a Branch?
   - Creating and Switching Branches (git branch, git checkout)
   - Merging Branches (git merge)
   - Resolving Merge Conflicts

6. **Remote Repositories**
   - Adding a Remote (git remote add)
   - Fetching Changes (git fetch)
   - Pushing Changes (git push)
   - Pulling Changes (git pull)

7. **Undoing Changes**
   - Stashing Changes (git stash)
   - Reverting Commits (git revert)
   - Resetting Commits (git reset)
   - Removing Files (git rm)

✵ Intermediate Git :-

1. **Working with Branches**
   - Branching Strategies (e.g., Git Flow)
   - Tracking Branches
   - Deleting Branches (git branch -d)

2. **Advanced Merging**
    - Fast-Forward Merge
    - 3-Way Merge
    - Merge Tools and Strategies

3. **Tagging**
    - Creating Tags (git tag)
    - Annotated vs Lightweight Tags
    - Listing and Deleting Tags

4. **Rebasing**
    - Introduction to Rebasing (git rebase)
    - Interactive Rebasing
    - Rebasing vs Merging

5. **Cherry-Picking**
    - Cherry-Picking Commits (git cherry-pick)

6. **Working with Remote Repositories**
    - Renaming Remotes (git remote rename)
    - Removing Remotes (git remote remove)
    - Working with Multiple Remotes

7. **Submodules**
    - Adding Submodules (git submodule add)
    - Updating Submodules (git submodule update)
    - Removing Submodules

Advanced Git :-

1. **Advanced Log and Diff**
    - Customizing git log Output
    - Comparing Commits (git diff)
    - Understanding Patch Files

2. **Git Internals**
    - Understanding Git Objects (Blobs, Trees, Commits, Tags)
    - Git Object Model
    - Git References

3. **Rewriting History**
    - Rewriting Commits (git commit --amend)
    - Using git rebase for History Rewriting
    - Filter-Branch (git filter-branch)
    - git reflog for Recovery

4. **Advanced Configuration**
    - Git Aliases
    - Git Hooks
    - Customizing Git with .gitconfig

5. **Performance Tuning**
    - Optimizing Git Performance
    - Large Repository Management
    - Using git gc and git fsck

6. **Security**
    - Signing Commits (git commit -S)
    - Verifying Signed Commits
    - Secure Transport (SSH, HTTPS)

❖ *Basic GitHub :-*

1. **Introduction to GitHub**
    - What is GitHub?
    - Creating a GitHub Account

2. **Repositories on GitHub**
    - Creating a Repository
    - Cloning a Repository
    - Forking a Repository

3. **GitHub Flow**
    - Issues and Bug Tracking
    - Pull Requests
    - Reviewing Pull Requests
    - Merging Pull Requests

4. **Collaboration on GitHub**
    - Managing Collaborators
    - Using GitHub Issues
    - Labels, Milestones, and Projects

5. **GitHub Pages**
    - Creating a GitHub Page
    - Customizing GitHub Pages

❖ *Intermediate GitHub :-*

1. **Advanced Repository Management**
    - Branch Protection Rules
    - Enabling Continuous Integration (CI)
    - Using GitHub Actions
2. **GitHub Packages**
    - Introduction to GitHub Packages
    - Publishing and Consuming Packages
3. **GitHub REST API**
    - Introduction to GitHub API
    - Authenticating API Requests
    - Using API to Automate Tasks
4. **Managing Organizations**
    - Creating and Managing Organizations
    - Teams and Permissions
    - Billing and Payments

❖ *Advanced GitHub :-*

1. **Security on GitHub**
    - Securing Repositories
    - Security Alerts and Vulnerability Management
    - Dependency Graph
2. **GitHub Advanced Features**

- GitHub CLI
- GitHub Actions Advanced Workflows
- GitHub Apps and OAuth Apps

3. **Enterprise GitHub**
   - GitHub Enterprise Overview
   - Managing GitHub Enterprise
   - Best Practices for Enterprise
4. **Custom GitHub Workflows**
   - Creating Custom GitHub Actions
   - Managing Complex CI/CD Pipelines
   - Integrating Third-Party Tools

☺ ## Basic Git :-

Tuesday, 2 July, 2024 08:44 PM

❖ Basic Git Topics :-

1. Introduction to Git :-

- ***What is Git :-***
  - Git is a distributed version control system (DVCS) designed to handle everything from small to very large projects with speed and efficiency.
  - It allows multiple developers to work on a project simultaneously without overwriting each other's changes.
  - Git keeps track of changes made to files and allows you to revert to previous states, branch out to develop new features independently, and merge changes seamlessly.
-
- ***Version Control Systems :-***
  - Version Control Systems (VCS) are tools that help software teams manage changes to source code over time.
  - They track every modification.
  - If a mistake is made, developers can turn back the clock and fix the mistake while minimizing disruption to all team members.

  ➢ ***There are several types of VCS :-***

  1. **Local Version Control Systems** :-
     - This involves a simple database that keeps all changes to files under revision control.
     - The most prominent example is RCS (Revision Control System), which keeps patch sets (differences between files) on disk.
  2. **Centralized Version Control Systems (CVCS)** :-
     - These systems have a single server containing all the versioned files, and a number of clients that check out files from this central place.
     - Examples include CVS, Subversion (SVN), and Perforce.
     - The primary advantage of CVCS is that everyone knows to a certain degree what everyone else on the project is doing.
  3. **Distributed Version Control Systems (DVCS)** :-
     - In these systems, clients don't just check out the latest snapshot of the files; they fully mirror the repository. Thus, if any server dies, and these systems were collaborating via it, any of the client repositories can be copied back up to the server to restore it.
     - Every clone is really a full backup of all the data.
     - Examples include Git, Mercurial, and Bazaar.

- ***History of Git :-***
  - Git was created by Linus Torvalds in 2005 for the development of the Linux kernel.
  - The Linux kernel is a large, complex, and popular open-source software project that necessitated a robust and efficient VCS.
  - Torvalds had used a proprietary DVCS called BitKeeper for several years, but when the licensing terms of BitKeeper changed, the Linux development community needed a new system.

  - *Torvalds and others designed Git with the following goals :-*
    - **Speed** :- Handling very large projects efficiently.
    - **Simple Design** :- Making it easy to understand.
    - **Fully Distributed** :- Ensuring that every developer has a complete copy of the project history.
    - **Strong Support for Non-linear Development** :- Allowing thousands of parallel branches.
    - **Able to Handle Large Projects Like the Linux Kernel** :- Providing efficient handling of large projects.

- ***Git vs Other VCS :-***

  ➢ **Git vs. Centralized VCS / CVCS (e.g., Subversion) :-** Uses a single server to store all changes and versions.
  1. **Repository Model**:
     - **Git**: Distributed model; every developer has a full copy of the entire repository.
     - **CVCS**: Centralized model; the repository is stored on a central server.
  2. **Branching and Merging**:
     - **Git**: Lightweight, easy to create, and merge branches.
     - **CVCS**: Branching and merging can be resource-intensive and cumbersome.
  3. **Speed**:
     - **Git**: Local operations are extremely fast because they do not need network access.
     - **CVCS**: Operations often require network access, which can be slower.
  4. **Availability**:
     - **Git**: Every copy of the repository is a full backup; highly resilient.
     - **CVCS**: If the central server goes down, no one can commit changes.

  ➢ **Git vs. Other Distributed VCS / DVCS (e.g., Mercurial) :-** Every user has a full copy of the repository, including the history.
  1. **Popularity**:
     - **Git**: More popular with a larger community and ecosystem.
     - **Mercurial**: Less popular, smaller community.
  2. **Complexity**:
     - **Git**: Has a steeper learning curve, but offers more powerful features.
     - **Mercurial**: Easier to learn, more user-friendly for beginners.
  3. **Performance**:
     - **Git**: Generally faster, especially for large projects.
     - **Mercurial**: Performance is also good, but can be slower for very large projects.
  4. **Platform Support**:
     - **Git**: Excellent cross-platform support, widely used in open-source and commercial projects.
     - **Mercurial**: Also good cross-platform support, but less adoption in the industry.

- ***Basic Concepts :-***
  1. ***Repository*** :- A directory where Git is initialized, containing all project files and their history.
  2. ***Commit*** :- A snapshot of your project at a point in time. Each commit has a unique ID.
  3. ***Branch*** :- A pointer to a specific commit. The default branch in Git is called main or master.
  4. ***Merge***:- Combining changes from different branches.
  5. ***Clone*** :- A copy of a repository.
  6. ***Remote*** :- A version of your repository hosted on the internet or network, enabling collaboration. Example: GitHub, GitLab.

2. Installation and Configuration :-

- ***Installing Git on Windows, macOS, and Linux :-*** Go to the Git official website and download the latest version.

- ***Initial Configuration (git config) :-*** *Setting up your identity (user.name, user.email)*

  - After installing Git, you need to set up your identity and configure some settings.

  ➢ **Setting Up Your Identity :-**
  1. **Open your Terminal or Command Prompt**.
  2. **Configure User Name / Set your name :-**
     Code :- git config --global user.name "Your Name"
  3. **Configure User Email** :-
     Code :- git config --global user.email "you@example.com"

  ➢ **Other Useful Configuration Settings :-**
  1. ***Set Default Text Editor*** :-
     - You can set your default text editor (e.g., VS Code, Nano, Vim).

- Code :-

```
git config --global core.editor "code --wait"    # For VS Code
git config --global core.editor "nano"           # For Nano
git config --global core.editor "vim"            # For Vim
```

2. ***Enable Colored Output*** :-
   - Code :- git config --global color.ui auto

3. ***Configure Line Ending Handling*** :-
   - If you are working on a project that involves different operating systems, this setting can help manage line endings.
   - Code :-
```
git config --global core.autocrlf true    # For Windows
git config --global core.autocrlf input   # For macOS/Linux
```

4. ***Viewing Configuration*** :-
   - To see your current configuration.
   - code :- git config --list

5. ***Edit Configuration File Directly*** :-
   - You can also edit the .gitconfig file directly located in your home directory.
   - Code :- nano ~/.gitconfig # or use your preferred text editor

6. ***Change configuration for a specific repository :-***
   - Navigate to the repository directory and use:
   - Code :-
```
git config user.name "Your Name"
git config user.email "your.email@example.com"
```

3. Getting Started with Git :-

- ***Basic Git Commands :-***
  - *Here are some of the most commonly used Git commands that form the foundation of working with Git :-*

1. **git init**: Initialize a new Git repository.
2. **git clone**: Clone an existing repository.
3. **git add**: Add files to the staging area.
4. **git commit**: Commit changes to the repository.
5. **git status**: Show the status of the working directory and staging area.
6. **git log**: View the commit history.
7. **git branch**: List, create, or delete branches.
8. **git checkout**: Switch branches or restore working tree files.
9. **git merge**: Merge branches.
10. **git pull**: Fetch and integrate changes from a remote repository.
11. **git push**: Update remote references along with associated objects.

- ***Initializing a Repository (git init) :-***
  - *To start a new project with Git, you need to initialize a new repository :-*

1. ***Navigate to Your Project Directory :-***
   - Open your terminal and navigate to the directory where you want to create your Git repository.
   - Code :- cd /path/to/your/project

2. ***Initialize the Repository :-***
   - Use the git init command to initialize a new Git repository.
   - Code :- git init
   - This command creates a hidden .git directory in your project folder, which contains all the metadata and object database for your repository.

- ***Cloning a Repository (git clone) :-***
  - If you want to work on an existing project, you can clone a repository.
  - Use the git clone command followed by the repository URL:
    - Code :- git clone https://github.com/username/repository.git

- ***Understanding the Working Directory, Staging Area, and Repository :-***
  - ***To effectively use Git, it's important to understand the three main areas where files are managed :-***

1. ***Working Directory :-***
   - This is the directory on your filesystem where you work on your project.
   - It contains the actual files and directories you are editing.
2. ***Staging Area (Index) :-***
   - The staging area is like a preparation area where you can format and review changes before committing them to the repository.
   - Files are added to the staging area using the git add command.
     Code :- git add filename
3. ***Repository :-***
   - The repository is where Git stores the history of your project.
   - It includes all commits, branches, and tags.
   - When you commit changes, they move from the staging area to the repository.
   - Code :- git commit -m "Commit message"

  - ***Example Workflow :-***

1. **Make changes in the working directory**:
   - Edit files as needed.
   - **Create a New Directory and Initialize Git** :-
```
Code :- mkdir my_project
        cd my_project
        git init
```
   - **Create a New File and Add Some Content :-**
```
Code :- echo "Hello, Git!" > hello.txt
```
2. **Check the Status** :- Code :- git status
3. **Stage changes**:
   - Add the changes to the staging area :- Code :- git add hello.txt
4. **Commit changes**:
   - Move the changes from the staging area to the repository :- Code :-
```
git commit -m "Add hello.txt"
git commit -m "Commit message"
```
5. **Push changes** (if working with a remote repository):
   - Upload changes to the remote repository :- Code :- git push
6. **View the Commit History** :- Code :- git log

4. Basic Operations :-

- ***Checking Repository Status (git status) :-***
  - The *"git status"* command is used to display the state of the working directory and the staging area.
  - It shows which changes have been staged, which haven't, and which files aren't being tracked by Git.
  - **Usage :-**
    Code :- git status
  - **Output :-**
    - ***Untracked files :-*** Files in your working directory that are not yet tracked by Git.

- ***Changes not staged for commit :-*** Modified files that haven't been added to the staging area.
- ***Changes to be committed :-*** Files that have been added to the staging area and are ready to be committed.

- ***Adding Files (git add) :-***
  - The *"git add"* command adds changes in the working directory to the staging area.
  - This command does not affect the repository until you commit the changes.
  - ***Usage :-***
    Code :- **git add <file>**
    Or to add all changes :-
    - Code :- **git add .**
  - ***Examples :-***
    - *Add a single file :-* **code ->** git add filename.txt
    - *Add all modified and new files :-* **code ->** git add .

- ***Committing Changes (git commit) :-***
  - The git commit command captures a snapshot of the project's currently staged changes.
  - Committed snapshots can be thought of as "safe" versions of a project, since Git won't change them unless you explicitly ask it to.
  - ***Usage :-***
    Code :- git commit -m "Your commit message"
  - ***Best Practices :-***
    - Write meaningful commit messages that describe the changes you made.
    - Make small, frequent commits with focused changes.
  - ***Examples :-***
    - Commit staged changes with a message:
    - ***Code :-*** git commit -m "Add new feature"

- ***Viewing Commit History (git log) :-***
  - The git log command shows the commit history for the repository.
  - It displays a list of commits with details such as the commit ID, author, date, and commit message.
  - ***Usage :-***
    Code :- git log
  - ***Output :-***
    - **Commit hash**: A unique identifier for each commit.
    - **Author**: The name and email of the person who made the commit.
    - **Date**: The date and time when the commit was made.
    - **Commit message**: A short message describing the commit.
  - ***Examples :-***
    - *View the commit history :-* **code ->** git log
    - *View a simplified commit history with one-line messages :-* **code ->** git log --oneline
    - *View a specific number of commits :-* **code ->** git log -n 5

5. Branching and Merging :-

- ***What is a Branch :-***
  - A branch in Git is a lightweight movable pointer to a commit.
  - It allows you to diverge from the main line of development and continue to work without affecting that main line.
  - You can think of branches as a way to develop features, fix bugs, or experiment in isolation.

  > **Default Branch**: The default branch when you create a repository is called main (formerly master).
  > **Feature Branch**: Used to develop new features or changes.
  > **Hotfix Branch**: Used to quickly patch production code.

- ***Creating and Switching Branches (git branch, git checkout) :-***

1. ***Creating a Branch :-*** To create a new branch, use the git branch command.
   - ***Code :-*** git branch <branch-name>
   - ***Example :-*** git branch feature-xyz
2. ***Switching Branches :-*** To switch to an existing branch, use the git checkout command.
   - ***Code :-*** git checkout <branch-name>
   - ***Switch Back to main Branch :-*** git checkout main
   - ***Example :-*** git checkout feature-xyz
3. ***Creating and Switching in One Step :-*** You can create and switch to a new branch in a single command using the -b option.
   - ***Code :-*** git checkout -b <branch-name>
   - ***Example :-*** git checkout -b feature-xyz
4. ***Listing Branches :-*** To list all branches in your repository.
   - ***Code :-*** git branch

- ***Advanced Branching :-***
  - git branch -m <new-branch-name> ***(rename current branch)***
  - git branch -r ***(list remote branches)***
  - git branch -a ***(list all branches)***

- ***Merging Branches (git merge) :-***
  - Merging combines changes from one branch into another.
  - Typically, you merge a feature branch into the main branch after development is complete.

1. ***Switch to the Target Branch :-***
   - First, switch to the branch you want to merge into (e.g., main).
   - ***Code :-*** git checkout main
2. ***Merge the Branch :-***
   - Use the git merge command followed by the branch name you want to merge.
   - ***Code :-*** git merge <branch-name>
   - ***Example :-*** git merge feature-xyz

- ***Resolving Merge Conflicts :-***
  - Sometimes, merging branches can result in conflicts.
  - This happens when changes in different branches conflict with each other.
  - Git will mark these conflicts in the affected files and require you to resolve them manually.

1. ***Identify Conflicted Files :-***
   - After attempting a merge, Git will indicate which files have conflicts.
   - ***Code :-*** git status

2. ***Open the Conflicted Files :-***
    - Conflicted files will contain conflict markers ("<<<<<<<", "=======", ">>>>>>>") to indicate the differing changes.

3. ***Resolve the Conflicts :-***
    - Edit the files to resolve the conflicts, then remove the conflict markers.
    - Choose the changes you want to keep or combine them as necessary.

4. ***Add the Resolved Files :-***
    - After resolving conflicts, add the resolved files to the staging area.
    - ***Code :-*** git add <file>

5. ***Commit the Merge :-***
    - Complete the merge by committing the resolved changes.
    - ***Code :-*** git commit

6. Remote Repositories :-

- ***Working with Remote Repositories :-***
    - Remote repositories are versions of your project that are hosted on the internet or a network, enabling collaboration with others.
    - Common platforms for hosting Git repositories include GitHub, GitLab, and Bitbucket.

- ***Adding a Remote (git remote add) :-***
    - To connect your local repository to a remote server, you need to add a remote repository.
    - This typically involves providing a URL to the remote repository.
    - ***Usage :-***
        - Code -> git remote add <remote-name> <remote-url>
    - ***Example :-***
        - Code -> git remote add origin https://github.com/username/repository.git
            - **"origin"**: The default name for a remote repository.
            - **"https://github.com/username/repository.git"**: The URL of the remote repository.
    - *You can verify the remotes configured for your repository using :-* ***Code ->*** git remote -v

- ***Fetching Changes (git fetch) :-***
    - The git fetch command downloads commits, files, and refs from a remote repository into your local repository, but it doesn't merge the changes into your working directory.
    - This allows you to review changes before integrating them.
    - ***Usage :-***
        - Code -> git fetch <remote-name>
    - ***Example :-***
        - Code -> git fetch origin
        - After fetching, you can view the changes with / **Review Fetched Changes** :-
        - Code -> git log origin/main

- ***Pushing Changes (git push) :-***
    - The git push command uploads your local commits to a remote repository.
    - It updates the remote branch with the commits from your local branch.
    - ***Usage :-***
        - Code -> git push <remote-name> <branch-name>
    - ***Example :-***
        - Code -> git push origin main
        - *If it's your first time pushing a branch to a remote repository, you might need to set the upstream branch :-*
        - Code -> git push --set-upstream origin main

- ***Pulling Changes (git pull) :-***
    - The git pull command is a combination of git fetch and git merge.
    - It downloads changes from a remote repository and merges them into your current branch.
    - ***Usage :-***
        - Code -> git pull <remote-name> <branch-name>
    - ***Example :-***
        - Code -> git pull origin main

7. Undoing Changes :-

- Undoing changes in Git is a crucial part of managing your version control workflow.
- There are several commands to handle different scenarios, including stashing changes, reverting commits, resetting commits, a nd removing files.

- ***Stashing Changes (git stash) :-***
    - When you need to temporarily save changes that are not yet ready to be committed, you can stash them.
    - This allows you to clean your working directory without committing the changes.
    - ***Usage :-***
        - Code -> git stash
    - ***Example :-***
        1. ***Make changes to a file :-***
            - ***Code :-*** echo "Some changes" >> file.txt
        2. ***Stash the changes :-***
            - ***Code :-*** git stash
        3. ***Apply the stashed changes later :-***
            - ***Code :-*** git stash apply
        4. ***List stashes :-***
            - ***Code :-*** git stash list
        5. ***Apply a specific stash :-***
            - ***Code :-*** git stash apply stash@{1}
        6. ***Drop a specific stash :-***
            - ***Code :-*** git stash drop stash@{1}

- ***Reverting Commits (git revert) :-***
    - Reverting a commit creates a new commit that undoes the changes made by a previous commit.
    - This is useful for undoing changes in a way that maintains the commit history.
    - ***Usage :-***
        - Code -> git revert <commit-hash>
    - ***Example :-***
        1. ***View commit history to find the commit hash :-***
            - ***code :- git log***
        2. ***Revert a specific commit :-***
            - ***Code :-*** git revert 1a2b3c4d(Hash-Code)

- ***Resetting Commits (git reset) :-***
    - Resetting changes the state of your repository to a previous commit.
    - It can be used to unstage changes, move the HEAD pointer, or both.
    - There are three modes of git reset: --soft, --mixed, and --hard.
    - ***Usage :-***
        - *Code ->* git reset [<mode>] <commit-hash>
    - **--soft :-** Moves the HEAD pointer to the specified commit and leaves changes in the staging area.
    - **--mixed (default) :-** Moves the HEAD pointer to the specified commit and unstages changes, leaving them in the working directory.
    - **--hard :-** Moves the HEAD pointer to the specified commit and discards all changes in the working directory and staging area.

- **Examples :-**
  1. *Reset to a previous commit and leave changes staged :-*
     - **Code :-** git reset --soft 1a2b3c4d
  2. *Reset to a previous commit and unstage changes :-*
     - **Code :-** git reset --mixed 1a2b3c4d
  3. *Reset to a previous commit and discard all changes :-*
     - **Code :-** git reset --hard 1a2b3c4d

- **Removing Files (git rm) :-**
  - The git rm command removes files from the working directory and stages the removal for the next commit.
  - **Usage :-**
    - Code -> git rm <file>
  - **Example :-**
    1. *Remove a specific file :-*
       - **Code :-** git rm filename.txt
    2. *Remove a file but keep it in the working directory :-*
       - **Code :-** git rm --cached filename.txt

☺ Intermediate Git :-

Tuesday, 2 July, 2024 08:43 PM

❖ Intermediate Git Topics :-

1. Working with Branches in Git :-

- ✧ Branching is a powerful feature in Git that allows you to create, manage, and merge branches for different tasks.
- ✧ Here, we'll explore branching strategies, how to track branches, and how to delete branches.

✧ Branching Strategies (e.g., Git Flow) :-

- ○ Branching strategies help organize the workflow and manage the development process.
- ○ Different strategies suit different project needs.

➢ *Below are some common branching strategies :-*

1. Git Flow :-

- Git Flow is a popular branching model that defines a strict branching strategy for managing features, releases, and hotfixes.
- **Main Branches :-**
  - *main :-* Contains the production code.
  - *Develop :-* Contains the latest development changes.
- **Supporting Branches :-**
  - *Feature branches :-* Used for developing new features.
    - Code -> git checkout -b feature/<feature-name> develop
    - Ex. :- git checkout -b feature-xyz develop
  - *Release branches :-* Prepares for a new production release.
    - Code -> git checkout -b release/<version> develop
  - *Hotfix branches :-* Quick fixes to the production code.
    - Code -> git checkout -b hotfix/<version> main
- **Merging :-**
  - Feature branches are merged into develop.
  - Release branches are merged into main and develop.
  - Hotfix branches are merged into main and develop.

2. GitHub Flow :-

- GitHub Flow is a simpler workflow, ideal for continuous deployment.
- **Main Branch :-**
  - *main :-* Always deployable.
- **Feature Branches :-**
  - Each feature or bug fix is developed in a new branch.
    - **Code :-** git checkout -b <feature-name>
- **Merging :-**
  - Feature branches are merged into main via pull requests.

3. Trunk-Based Development :-

- In Trunk-Based Development, developers work on short-lived branches and merge them frequently into the main branch.
- **Main Branch :-**
  - *main :-* The only long-lived branch.
- **Feature Branches :-**
  - Short-lived branches for features or fixes.
    - **Code :-** git checkout -b <short-lived-branch>
- **Merging :-**
  - Feature branches are merged into main multiple times a day.

✧ Tracking Branches :-

- ○ Tracking branches are local branches that have a direct relationship to a remote branch.
- ○ They help synchronize changes between your local and remote repositories.

➢ **Creating a Tracking Branch :-**

□ **Create and Track a New Branch :-**

- **Code :-** *git checkout -b <branch-name> --track <remote>/<branch-name>*
- **Example :-** *git checkout -b feature-xyz --track origin/feature-xyz*

□ **Set Up Tracking for an Existing Branch :-**

- **Code :-** *git branch --set-upstream-to=<remote>/<branch> <local-branch>*
- **Example :-** *git branch --set-upstream-to=origin/main main*

□ **Push a Branch and Set Up Tracking :-**

- **Code :-** git push -u <remote> <branch>
- **Example :-** git push -u origin feature-xyz

□ **View Tracking Branch Information :-**

- **Code :-** git branch -vv

✧ Deleting Branches (git branch -d) :-

- ○ Deleting branches helps keep your repository clean and manageable.

➢ **Deleting a Local Branch :-**

□ **Delete a Local Branch**:

- **Code :-** git branch -d <branch-name>

► *Use -D to force delete a branch that hasn't been fully merged.*

- **Code :-** git branch -D <branch-name>

➢ **Deleting a Remote Branch :-**

□ **Delete a Remote Branch :-**

- **Code :-** git push <remote> --delete <branch-name>
- **Example :-** git push origin --delete feature-xyz

2. Advanced Merging :-

- ○ Merging in Git is the process of combining changes from different branches.
- ○ Understanding advanced merging techniques is crucial for effectively managing and resolving conflicts in a collaborative envi ronment.
- ○ In this section, we will cover fast-forward merges, 3-way merges, and explore merge tools and strategies.

✧ ***Fast-Forward Merge :-***

- A fast-forward merge occurs when the branch you are merging into has not diverged from the branch you are merging.
- In this scenario, Git simply moves the branch pointer forward to the latest commit.

➢ ***Usage :-***

□ *Code :-* git checkout <target-branch>
git merge <source-branch>

➢ ***Example :-***

□ *Code :-* git checkout main
git merge feature-branch

➢ In this example, if main has not diverged from feature-branch, Git will move the main pointer to the latest commit in feature-branch.

✧ ***3-Way Merge :-***

- A 3-way merge is used when the branches have diverged.
- Git uses the common ancestor of the branches and the two sets of changes to create a new merge commit.

➢ ***Usage :-***

□ *Code :-* git checkout <target-branch>
git merge <source-branch>

➢ ***Example :-***

□ *Code :-* git checkout main
git merge feature-branch

➢ In this case, if both main and feature-branch have diverged, Git will create a new merge commit that incorporates changes from both branches.

○ ***Conflict Resolution :-***

□ **If there are conflicts, Git will pause the merge and mark the conflicting files.**
□ **You need to resolve the conflicts manually.**

1. ***Identify Conflicted Files :-***
   □ *Code :-* **git status**

2. ***Edit Conflicted Files to Resolve Conflicts :-***
   - Conflicted sections are marked with <<<<<<, ======, and >>>>>>.

3. ***Add Resolved Files :-***
   □ *Code :-* git add <resolved-file>

4. ***Complete the Merge :-***
   □ *Code :-* git commit

✧ ***Merge Tools and Strategies :-***

○ ***Merge Tools :-***

- Using external merge tools can help resolve conflicts more efficiently.
- Git can integrate with various merge tools such as kdiff3, meld, vimdiff, tortoisegit, etc.

□ ***Configure a Merge Tool :-***

1. *Set the Merge Tool :-*
   - ***Code :-*** git config --global merge.tool <tool-name>

2. *Set the Path to the Tool (if needed) :-*
   - ***Code :-*** git config --global mergetool.<tool-name>.path <path-to-tool>

3. *Invoke the Merge Tool :-*
   - ***Code :-*** git mergetool

□ ***Example :-***

Code :- *git config --global merge.tool meld*
git mergetool

○ ***Merge Strategies :-***

- Different merge strategies can be applied depending on the situation.
- The most common strategies are recursive, ours, and octopus.

1. *Recursive Strategy :-*
   - This is the default merge strategy used for two branches.
   - It performs a 3-way merge and can handle most typical cases.

   > **Code :-** git merge -s recursive <branch>

2. *Ours Strategy :-*
   - This strategy effectively discards changes from the other branch and keeps the changes from the current branch.

   > **Code :-** git merge -s ours <branch>

3. *Octopus Strategy :-*
   - This strategy is used for merging more than two branches.

   > ***Code :-*** git merge -s octopus <branch1> <branch2> <branch3>

3. *Tagging :-*

□ Tags in Git are used to mark specific points in a repository's history, such as releases.
□ There are two types of tags: annotated and lightweight.
□ This guide will cover how to create tags, the differences between annotated and lightweight tags, and how to list and delete tags.

✧ ***Creating Tags (git tag) :-***

- Creating tags in Git is straightforward.
- Tags can be created at any point in your repository's history, typically on a commit.

✧ ***Lightweight VS Annotated Tags :-***

○ ***Lightweight Tags :-***

- Lightweight tags are simple and contain just the commit hash.
- They are essentially bookmarks to a commit.

□ *Create a Lightweight Tag :-*
- ***Code :-*** *git tag <tag-name>*
- **Example :-** *git tag v1.0*

○ ***Annotated Tags :-***

- Annotated tags are stored as full objects in the Git database.
- They contain additional metadata such as the tagger name, email, date, and a message.

□ *Create an Annotated Tag :-*
- ***Code :-*** *git tag -a <tag-name> -m "Tag message"*
- **Example :-** *git tag -a v1.0 -m "Release version 1.0"*

✧ ***Listing and Deleting Tags :-***

○ ***Listing Tags :-***

- You can list all tags in your repository using the git tag command.

- *List All Tags :-*
  - ***Code :-*** git tag

- *List Tags with Descriptions :-*
  - ***Code :-*** git tag -n
  - ***Example :-*** git tag
    v1.0
    v1.1
    v2.0
    git tag -n
    v1.0 Release version 1.0
    v1.1 Release version 1.1
    v2.0 Release version 2.0

- ***Viewing Tag Details :-***
  - To see details about a specific tag, use the git show command.

  - *View Tag Details :-*
    - ***Code :-*** *git show <tag-name>*
    - ***Example :-*** *git show v1.0*

- ***Deleting Tags :-***
  - Tags can be deleted both locally and remotely.

  - *Delete a Local Tag :-*
    - ***Code :-*** *git tag -d <tag-name>*
    - ***Example :-*** *git tag -d v1.0*

  - *Deleting a Remote Tag :-*
    - To delete a tag from a remote repository, you need to push a delete operation.

    - ***Code :-*** *git push <remote> --delete <tag-name>*
    - ***Example :-*** *git push origin --delete v1.0*

- ***Pushing Tags :-***
  - By default, tags are not pushed to remote repositories.
  - You need to explicitly push them.

  - *Push a Single Tag :-*
    - ***Code :-*** *git push <remote> <tag-name>*
    - ***Example :-*** *git push origin v1.0*
  - *Push All Tags :-*
    - ***Code :-*** *git push <remote> --tags*
    - ***Example :-*** *git push origin --tags*

4. Rebasing :-

- ***Rebasing*** is a powerful Git operation used to integrate changes from one branch into another by moving or combining a sequence of commits onto a different base commit.
- This guide will cover the basics of rebasing, interactive rebasing, and compare rebasing with merging.

Introduction to Rebasing (git rebase) : -

- ***Rebasing allows you to :-***
  - *Integrate Changes :-* Move commits from one branch to another.
    Maintain a Clean History :- Create a linear history by avoiding unnecessary merge commits.
- ***Basic Rebasing :-***
  - *To rebase a branch onto another branch :-*

    1. *Switch to the target branch (the branch you want to rebase onto) :-*
       - ***Code :-*** *git checkout <target-branch>*
    2. *Start the rebase :-*
       - ***Code :-*** *git rebase <source-branch>*
    - ***Example :-***
      git checkout main
      git rebase feature-branch
    - In this example, commits from feature-branch are reapplied on top of main, moving the branch pointer to the tip of main.

- ***Rebasing Interactive :-***
  - Interactive rebasing allows you to edit, combine, or remove commits during the rebase process.
  - This is useful for cleaning up commit history before merging into a main branch or squashing commits into more meaningful units.

  - *Start an Interactive Rebase :-*
    - ***Code :-*** *git rebase -i <base-branch>*
    - ***Example :-*** *git rebase -i main*
    - This command opens an editor with a list of commits.
    - You can choose actions for each commit like pick, reword, edit, squash, fixup, or drop.

Example Workflow :-

1. ***Rebase a Feature Branch onto Main :-***
   - ***Code :-*** *git checkout main*
     git pull origin main
     git checkout feature-branch
     git rebase main

2. ***Resolve Conflicts (if any) :-***
   - If there are conflicts during rebase, Git will pause the process.
   - Resolve conflicts in conflicted files, stage them, and continue the rebase.

   - ***Code :-*** *git add <resolved-file>*
     git rebase --continue
3. ***Complete the Rebase :-***
   - Once all conflicts are resolved, complete the rebase process.

   - ***Code :-*** *git rebase --continue*

4. **Push the Rebased Branch** :-
   - ***Code :-*** *git push origin feature-branch --force-with-lease*

Interactive Rebasing : -

- ***Interactive rebasing allows you to :-***

  - *Reword Commits :-* Change commit messages.
  - *Squash Commits :-* Combine multiple commits into one.
  - *Edit Commits :-* Amend commit contents or split changes.

- ***Start an Interactive Rebase :-***
  - ***Code :-*** *git rebase -i <base-branch>*
  - ***Example :-*** *git rebase -i main*

- This command opens an editor with a list of commits.
- You can choose actions for each commit like pick, reword, edit, squash, fixup, or drop.

✧ ***Rebasing vs Merging :-***

○ ***Rebasing :-***

- *Advantages :-*
  - Creates a linear history.
  - Avoids unnecessary merge commits.
  - Easier to understand project history.
- *Disadvantages :-*
  - Rewriting history can cause conflicts.
  - Requires careful handling to avoid losing commits.

○ ***Merging :-***

- *Advantages :-*
  - Preserves the history of each branch.
  - Maintains chronological order of commits.
- *Disadvantages :-*
  - Can clutter the history with merge commits.
  - History may become less readable over time.

5. Cherry-Picking :-

✧ Cherry-picking in Git is the process of selecting specific commits from one branch and applying them onto another branch.
✧ It's useful when you want to apply a single or a few selected commits from one branch to another without merging the entire b ranch.

✧ ***Cherry-Picking Commits (git cherry-pick) :-***

○ ***To cherry-pick a commit, follow these steps :-***

1. ***Identify the Commit to Cherry-Pick :-***
   - First, find the commit hash of the commit you want to apply to another branch.
   - You can find this hash using git log or any other Git history visualization tool.
   - ***Code :-*** *git log*
   - ***Example output :-***

     SQL Code:-

     ```
     commit abc1234def5678...
     Author: John Doe <johndoe@example.com>
     Date:   Tue Jan 1 12:00:00 2024 +0100
     Implement feature X
     ```

2. ***Switch to the Target Branch :-***
   - Checkout the branch where you want to apply the commit.
   - ***Code :-*** *git checkout <target-branch>*

3. ***Cherry-Pick the Commit :-***
   - Use the git cherry-pick command followed by the commit hash you identified in step 1.
   - ***Code :-*** git cherry-pick <commit-hash>
   - ***Example :-*** git cherry-pick abc1234def5678...

4. ***Resolve Conflicts (if any) :-***
   - If Git encounters conflicts during the cherry-pick operation, it will pause and mark the conflicted files.
   - You need to resolve these conflicts manually.
   - ***Code :-***

     ```
     git status  # View conflicted files
     # Resolve conflicts in conflicted files
     git add <resolved-file>
     git cherry-pick --continue
     ```

5. ***Complete the Cherry-Pick :-***
   - After resolving conflicts, complete the cherry-pick operation.
   - ***Code :-*** git cherry-pick --continue

✧ ***Notes :-***

□ Cherry-picking applies the changes introduced by the selected commit onto the current branch.
□ It does not retain the original commit message timestamp and author; it creates a new commit with new metadata.
○ Use cherry-picking cautiously, especially in shared repositories, as it can lead to duplicated commits and potential confusion if not managed carefully.
◆ Cherry-picking is useful for selectively applying specific changes without merging entire branches, which is beneficial in situations like hotfixes or backporting changes to stable branches.

6. Working with Remote Repositories :-

✧ ***Working*** with remote repositories in Git involves managing connections to repositories hosted on remote servers.
✧ This guide covers renaming and removing remotes, as well as working with multiple remotes.

✧ ***Renaming Remotes (git remote rename) :-***

○ Renaming a remote allows you to change the name of an existing remote repository URL.
○ This can be useful when you want to update the name of a remote repository due to organizational changes or to align with nam ing conventions.

➢ ***Syntax :-*** *git remote rename <old-name> <new-name>*
➢ ***Example :-***
   - *If you have a remote named origin and you want to rename it to upstream :-*
   - ***Code :-*** *git remote rename origin upstream*

✧ ***Removing Remotes (git remote remove) :-***

○ Removing a remote disconnects your local repository from the remote repository.
○ This is useful when you no longer need to fetch or push changes to a particular remote repository.

➢ ***Syntax :-*** *git remote remove <remote-name>*
➢ **Example :-**
   - *To remove a remote named upstream:*
   - ***Code :-*** *git remote remove upstream*

✧ ***Working with Multiple Remotes :-***

○ Git supports working with multiple remote repositories, allowing you to collaborate with different teams or manage different versions of your project.

□ ***Adding a New Remote :-***
   - *To add a new remote repository :-*
     - **Syntax :-** git remote add <remote-name> <remote-url>
     - ***Example :-*** git remote add upstream https://github.com/organization/project.git

□ ***Listing Remote Repositories :-***
   - *To list all remote repositories associated with your local repository :-*
     - ***Syntax :-*** git remote -v
       - ***This command lists the names and URLs of all remote repositories.***

□ ***Fetching from Multiple Remotes :-***
   - *To fetch changes from a specific remote repository :-*
     - ***Syntax :-*** git fetch <remote-name>

- **Example :-** git fetch upstream
  - ***This command fetches changes from the upstream remote repository.***

- ***Pushing Changes to Multiple Remotes :-***
  - To push changes to a specific remote repository :-
    - ***Syntax :-*** git push <remote-name> <branch-name>
    - ***Example :-*** git push origin main
      - ***This command pushes changes from the main branch to the origin remote repository.***

- ***Difference between Fetch and Pull :-***
  - git fetch only downloads commits and does not merge them.
  - git pull downloads and merges changes into your current branch.

7. Working with Git Submodules :-

- ***Git*** submodules allow you to include and manage external repositories within your own repository as a subdirectory.
- This guide covers adding submodules, updating them, and removing them from your Git project.

✧ *Adding Submodules (git submodule add) :-*

- Adding a submodule connects an external repository to your current repository, making it possible to include and track external codebases within your own project.
- ***Code :-***
  - *git submodule update*
  - *git submodule init*
  - *git submodule foreach <command>*
  - *git submodule sync*

- ***Syntax :-*** ***git submodule add <repository-url> <path>***
  - *<repository-url> :-* The URL of the repository you want to add as a submodule.
  - *<path> :-* The path where you want the submodule to be located within your project.

- ***Example :-*** git submodule add https://github.com/username/submodule-repo.git path/to/submodule

- This command clones the external repository specified by <repository-url> into the <path> directory within your project and stages it for commit.

✧ *Updating Submodules (git submodule update) :-*

- After adding a submodule, you need to initialize and update it to fetch the submodule contents into your project.

- ***Initialization and Updating :-***

  - ***Code :- git submodule update --init --recursive***

  - This command initializes and updates all submodules within your repository, including nested submodules (--recursive option).

✧ *Removing Submodules :-*

- Removing a submodule involves a few manual steps since Git does not provide a direct command to remove a submodule.

1. ***Delete Submodule Entry :-***

   - Remove the submodule entry from the .gitmodules file.
   - ***Code :-*** git submodule deinit -f -- <path/to/submodule>

2. ***Remove Submodule Directory :-***

   - Delete the submodule directory and stage the changes.
   - ***Code :-*** rm -rf <path/to/submodule>
     git rm -f <path/to/submodule>

3. ***Remove Submodule Configuration :-***

   - Remove submodule configuration from the Git repository.
   - ***Code :-*** git config -f .git/config --remove-section submodule.<path>
     git config -f .gitmodules --remove-section submodule.<path>

4. ***Commit the Changes :-***

   - Commit the removal of the submodule from your repository.
   - ***Code :-*** git commit -m "Removed submodule <path>"

8. Git Hooks :-

- Git hooks are scripts that run automatically on certain events in the Git lifecycle.

✧ *Types of Hooks :-*

- ***Client-side hooks :-*** Operate on the local repository (e.g., pre-commit, post-commit).
- ***Server-side hooks :-*** Operate on the remote repository (e.g., pre-receive, post-receive).

✧ *Setting Up Hooks :-*

1. Navigate to the .git/hooks directory.
2. Create or edit a hook script (e.g., pre-commit).
3. Make the script executable :- ***Code ->*** chmod +x .git/hooks/pre-commit

✧ *Example Pre-commit Hook :-*

- ***Code :-***

```
#!/bin/sh
# Check for trailing whitespace
if grep -q '[[:blank:]]$' "$@"; then
  echo "Error: Trailing whitespace found."
  exit 1
fi
```

☺ ADVANCED GIT :-

Tuesday, 2 July, 2024 08:47 PM

❖ Advanced Git Topics :-

✵ Advanced Concepts :-

1. ***Rebasing :-***
   - Reapplying commits on top of another base tip.
   - ***Code :-*** git rebase branch-name
2. ***Stashing :-***
   - Temporarily saving changes not ready to commit.
   - ***Code :-*** git stash
     git stash apply
3. ***Cherry-Picking :-***
   - Applying a specific commit from one branch to another.
   - ***Code :-*** git cherry-pick commit-id
4. ***Interactive Rebase :-***
   - Allows rewriting commit history.
   - ***Code :-*** git rebase -i commit-id
5. ***Tags :-***
   - Marking specific points in history as important.
   - ***Code :-*** git tag tag-name
6. ***Hooks :-***
   - Scripts that run at specific points in the Git workflow.
   - ***Example :-*** pre-commit, pre-push.
7. ***Submodules :-***
   - Including other repositories within a repository.
   - ***Code :-*** git submodule add https://github.com/username/repository.git
8. ***Bisect :-***
   - Finding the commit that introduced a bug using binary search.
   - ***Code :-*** git bisect start
     git bisect bad
     git bisect good commit-id

✵ Best Practices :-

1. ***Commit Often :-***
   - Make frequent commits with meaningful messages.
2. ***Branch Strategy :-***
   - Use branches for new features, bug fixes, and experiments.
3. ***Code Review :-***
   - Use pull requests (PRs) for code review and collaboration
4. ***Avoid Committing Large Files :-***
   - Use .gitignore to exclude unnecessary files.
5. ***Backup Regularly :-***
   - Push to remote repositories frequently.

1. Advanced Log and Diff :-

- Advanced log and diff functionalities in Git provide deeper insights into project history and changes between commits.
- This guide covers customizing git log output, comparing commits with git diff, and understanding patch files.

Customizing "git log" Output :-

- git log displays a chronological list of commits in your repository.
- Customizing its output helps to filter and format the commit history according to specific needs.

Basic git log Command :-

- ***Code :-*** *git log*

Customizing Output :-

- ***Code :-***

```
git log --oneline          # Compact single line per commit
git log --graph --decorate    # Graphical representation with branch and tag names
git log --since="2 weeks ago"  # Commits since a specific date or time
```

Filtering by Author :-

- **Code :-** *git log --author="John Doe"*

Limiting Output :-

- ***Code :-*** git log -n 5 **# Show only the latest 5 commits**

Formatting Output :-

- ***Code :-*** *git log --pretty=format:"%h - %an, %ar : %s"*

Comparing Commits ("git diff") :-

- git diff shows changes between commits, branches, or between the working directory and the index.

Comparing Branches :-

- ***Code :-*** *git diff branch1..branch2*

Comparing Files :-

- ***Code :-*** git diff file1.txt file2.txt

Comparing Working Directory with Index :-

- ***Code :-*** *git diff*

Comparing with Previous Commit :-

- ***Code :-*** *git diff HEAD^ HEAD*

Compare Commits with git diff :-

- ***Code :-*** *git diff HEAD~2..HEAD* ***# Compare last two commits***

Generating Patch Files :-

- ***Code :-***

```
git diff > my-changes.patch  # Create a patch file
git apply my-changes.patch  # Apply the patch to another branch or repository
```

Understanding Patch Files :-

- Patch files (*.patch or *.diff) are textual representations of differences between two commits or sets of files.
- They are useful for sharing changes between repositories, applying changes, or reviewing proposed modifications.

Patch File Format :-

diff

Code :-

```
diff --git a/file1.txt b/file1.txt
index abc123..def456 100644
--- a/file1.txt
+++ b/file1.txt
```

```
@@ -1,2 +1,2 @@
Line 1 - Original content
-Line 2 - Original content
+Line 2 - Modified content
```

- ***Applying Patch Files :-***
  - ***Code :-*** *git apply my-changes.patch* ***# Apply patch file to a branch or repository***

2. Git Internals :-

- Understanding Git internals provides insights into how Git stores and manages versioned data.
- This guide covers Git objects (Blobs, Trees, Commits, Tags), the Git object model, and Git references.

- ***Understanding Git Objects (Blobs, Trees, Commits, Tags) : -***
  - ***Git uses four main types of objects to store and manage repository data :-***
    - ***Related Internals Code :-***
      - *git cat-file -t <object>*
      - *git hash-object <file>*
      - *git update-index --assume-unchanged <file>*
      - *git update-index --no-assume-unchanged <file>*
      - *git ls-tree <tree-ish>*
      - *git rev-parse <revision>*
    - ***Blobs (git cat-file -p) :-***
      - Stores file data (content).
      - ***Example :- Php Code ->*** echo "Hello, Git!" | git hash-object -w --stdin
    - ***Viewing Git Objects :-***
      - ***Code :-*** *git cat-file -p <object-hash>*
    - ***Trees (git ls-tree -r HEAD) :-***
      - Represents the directory structure at a specific state, referencing blobs or other trees.
      - ***Example :-*** git ls-tree HEAD
    - ***Commits (git show) :-***
      - Points to a snapshot of your repository at a given time, including metadata (author, timestamp) and references to parent commits.
      - ***Example :- SQL Code ->*** git show HEAD
    - ***Tags (git show-ref --tags) :-***
      - A named pointer to a specific commit, used to label significant points in history (releases, milestones).
      - ***Example :-*** git tag v1.0.0
- ***Git Object Model :-***
  - ***The Git object model represents the structure and relationships between different Git objects :-***
    - ***Commit Object :-***
      - Points to a tree object representing the state of the repository at the time of the commit, with metadata such as author, committer, and commit message.
    - ***Tree Object :-***
      - Represents a directory and its contents at a specific state, storing references to blobs (file contents) or other trees (subdirectories).
    - ***Blob Object :-***
      - Stores file contents, uniquely identified by a SHA-1 hash of its content.
    - ***Tag Object :-***
      - Points to a commit and adds extra metadata (annotated tags) or is a lightweight reference to a commit (lightweight tags).
- ***Git References :-***
  - ***Git references (refs) are pointers to Git objects (usually commits) that can be updated to reflect changes in the repository's history :-***
    - ***Branches (git branch) :-***
      - A movable pointer to a commit, typically used to represent a line of development.
      - ***Example :-***
        Arduino Code -> git branch new-feature
    - ***HEAD (git symbolic-ref) :-***
      - A symbolic reference to the current branch (or commit if in detached HEAD state), indicating the current position in the repo sitory's history.
    - ***Tags (git tag) :-***
      - A named reference to a specific commit, often used to mark releases or significant points in history.
    - ***Remote Branches (git fetch origin) :-***
      - References to branches on remote repositories, used for syncing changes between local and remote repositories.
  - ***Inspecting References :-***
    - ***Code :-*** git show-ref --tags
      git branch -a

3. Rewriting History :-

- Rewriting history in Git involves altering commit history, which can be useful for cleaning up commits, combining commits, orreordering changes.
- This guide covers rewriting commits using git commit --amend, git rebase, git filter-branch, and using git reflog for recovery.

- ***Rewriting Commits (git commit --amend) :-***
  - The git commit --amend command allows you to modify the most recent commit.
  - It's useful for making small changes to the commit message or adding files you forgot to include.
  - ***Usage :-***
    - ***Code :-*** *git add forgotten-file*
      git commit --amend
  - ***Example Workflow :-***

```
# Edit the file(s)
git add modified-file
git commit --amend
```

- ***Using git rebase for History Rewriting :-***
  - git rebase rewrites commit history by moving, combining, or squashing commits onto a new base commit.
  - It's useful for cleaning up history before merging branches.
  - ***Interactive Rebase** (git rebase -i) :-*
    - ***Code :-*** *git rebase -i <base-branch>*
  - ***Squashing Commits :-***
    - ***Code :-*** *git rebase -i HEAD~3* ***# Squash last 3 commits into one***
  - ***Reordering Commits :-***
    - ***Code :-*** *git rebase -i HEAD~5* ***# Reorder last 5 commits***

- ***Filter-Branch (**git filter-branch**) for Filtering History:-***
  - git filter-branch is a powerful but potentially dangerous command that can rewrite history by applying custom filters to commits, such as

removing sensitive data or splitting repositories.

- **Example (Removing a File from History) :-**
  - **Code :-** git filter-branch --tree-filter 'rm -f passwords.txt' HEAD
- **Use with Caution :-**
  - This command alters commit history permanently and affects all downstream users.

- **"git reflog" for Recovery :-**
  - git reflog records updates to the Git references (e.g., branches, HEAD), even after they've been modified or deleted.
  - It's useful for recovering lost commits or branches.

    - **Viewing git reflog :-**
      - **Code :-** *git reflog*
    - **Recovering Lost Commits :-**
      - **Code :-** *git checkout HEAD@{1}* **# Checkout a previous commit from reflog**
    - **Example Workflow :-**
      - **Code :-**
```
# Accidentally reset branch, recover previous state
git reflog
git checkout HEAD@{1}
```

- **Best Practices :-**
  - **Communicate Changes :-**
    - Rewriting history can impact collaborators, so communicate changes before rewriting shared branches.
  - **Backup :-**
    - Before using commands like git filter-branch, ensure you have backups or clones of important repositories.
  - **Use Interactive Rebase Wisely :-**
    - Interactive rebase (git rebase -i) offers precise control over history but requires caution to avoid losing valuable changes.

4. Advanced Configuration :-

- Advanced configuration in Git goes beyond basic settings and involves setting up aliases for commands, using hooks for automation, and customizing Git behavior through .gitconfig files.
- **Here's a detailed overview of these advanced configuration options :-**

- **Git Aliases :-**
  - Git aliases allow you to create shortcuts or custom commands for commonly used Git operations.
  - They can simplify your workflow and make Git commands more intuitive.

    - **Setting Aliases :-**
      - Aliases are defined in your .gitconfig file or set via the command line using git config.

      - **Code :-** git config --global alias.co checkout
      - **Example Aliases** :-
```
git config --global alias.st status
git config --global alias.ci commit
git config --global alias.br branch
```
    - **Complex Aliases :-**
      - Aliases can execute multiple Git commands or include parameters.

      - **Code :-** *git config --global alias.hist 'log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short'*
- **Git Hooks :-**
  - Git hooks are scripts that Git executes before or after certain actions (e.g., commit, push). They enable automation and enfo rce policies within your Git workflow.
  - **Types of Git Hooks :-**
    - Common hooks include pre-commit, post-commit, pre-push, and more.
    - These scripts are stored in the .git/hooks directory of your repository.
    - **Types of Git Hooks :-**
      1. **Pre-commit** :- Run before committing changes.
      2. **Pre-receive :-** Run on the server before accepting a push.
      3. **Post-receive :-** Run on the server after receiving a push.
      4. **Pre-push** :- Run before pushing changes to a remote repository.

  - **Example Use Cases :-**
    - **Pre-commit :-** Enforce code style guidelines or run automated tests before committing changes.
    - **Post-commit :-** Trigger notifications or update issue trackers after commits.
    - **Pre-push :-** Validate changes before pushing to a remote repository.
  - **Creating a Hook :-**
    - *Create an executable script in .git/hooks/ with the hook name (e.g., pre-commit).*

- **Customizing Git with .gitconfig :-**
  - The .gitconfig file is where Git stores configuration settings for each user.
  - It can be customized to adjust Git's behavior, define aliases, set user information, and more.
  - **Location :-**
    - *Global :-* ~/.gitconfig (user-level configuration)
    - *Local :-* <repository>/.git/config (repository-specific configuration)

  - **Editing .gitconfig :-**
    - You can edit .gitconfig directly or use git config commands to add or modify settings.
  - **Example Configuration :-**
    - **Code :-**
```
[user]
   name = Your Name
   email = your.email@example.com
[alias]
   st = status
   ci = commit
```

- **Best Practices :-**
  - **Use Meaningful Aliases** :-
    - Choose aliases that make Git commands more intuitive or faster to type.
  - **Share Hooks** :-
    - Share common hooks across your team to enforce consistent practices.

  - **Backup and Version Control** :-
    - Keep backups of important .gitconfig files and document customizations.

5. Performance Tuning :-

- Optimizing Git performance involves enhancing speed, managing large repositories efficiently, and maintaining repository integrity.
- This guide covers techniques for optimizing Git performance, managing large repositories, and using git gc and git fsck for maintenance.

- ***Optimizing Git Performance :-***
  - ***Improving Git performance involves several strategies to enhance speed and efficiency :-***
    - ***Commit Frequency :-***
      - Make smaller, focused commits rather than large, infrequent ones.
    - ***Use of Git Commands :-***
      - Utilize commands like git fetch and git pull --rebase to efficiently update local branches.
    - ***Reduce History :-***
      - Limit the depth of commit history when cloning repositories (--depth option).
    - ***Packing Commits :-***
      - Use git gc to pack loose objects and optimize repository storage.
    - ***Configuring Git :-***
      - Adjust configuration settings (core.*) to optimize for your workflow.
- ***Large Repository Management :-***
  - ***Managing large repositories requires special considerations to maintain performance and usability :-***
    - ***Sparse Checkout :-***
      - Use git sparse-checkout to work with a subset of files, reducing checkout time and disk space usage.
    - ***Shallow Clone :-***
      - Clone only the latest commit history (--depth) for faster initial setup.
    - ***Git LFS :-***
      - Use Git Large File Storage (git lfs) for managing large files separately from the Git repository.
    - ***Splitting Repositories :-***
      - Consider splitting large repositories into smaller, more manageable ones based on logical project boundaries.
- ***Using git gc and git fsck :-***
  - ***git gc (garbage collection) and git fsck (file system check) are commands for maintaining repository integrity and optimizing performance :-***
    - ***git gc :-***
      - ***Purpose :-*** Consolidates and optimizes Git repository storage by packing loose objects.
      - ***Usage :-*** Run periodically to reclaim space and improve performance.
      - ***Code :-*** git gc
    - ***git fsck :-***
      - ***Purpose :-*** Checks the integrity of the Git repository and its objects.
      - ***Usage :-*** Identifies and corrects errors in the repository structure.
      - ***Code :-*** git fsck

- ***Example Workflow :-***
  1. ***Optimizing Performance :-***
     - ***Code :-*** *git fetch origin*
       git pull --rebase
  2. ***Managing Large Repositories :-***
     - ***Code :-*** *git clone --depth 1 <repository-url>* ***# Shallow clone***
       git sparse-checkout init --cone ***# Sparse checkout***
  3. ***Using Maintenance Commands :-***
     - ***Code :-*** *git gc* ***# Optimize repository storage***
       git fsck ***# Check repository integrity***

- ***Best Practices :-***
  - Schedule git gc and git fsck maintenance tasks during off-peak times to minimize disruption.
  - Monitor repository size and performance regularly, especially in large or active repositories.

6. Security :-

- Ensuring Git repository security involves implementing measures to sign commits, verify signatures, and secure transport prot ocols like SSH and HTTPS.

- ***Here's a detailed overview of these security practices :-***
  - ***Signing Commits (git commit -S) :-***
    - Signing commits allows you to verify the authenticity and integrity of commits, ensuring they have not been tampered with.
    - **Signing a Commit :-**
      - ***Code :-*** git commit -S -m "Commit message"
    - ***Generating a GPG Key :-***
      - Before signing commits, you need to generate a GPG key and add it to your Git configuration.
      - ***Code :-***

```
# Generate a new GPG key
gpg --gen-key

# Add GPG key to Git
git config --global user.signingkey <gpg-key-id>
```

- ***Verifying Signed Commits :-***
  - Verifying signed commits ensures their authenticity and integrity.
  - ***Verifying a Commit :-***
    - ***Code :-*** *git log --show-signature*
  - **Inspecting Commit Signature :-**
    - ***Code :-*** *git show <commit-hash>*
- ***Secure Transport (SSH, HTTPS) :-***
  - Securing Git transport protocols ensures that data transmitted between repositories and servers remains confidential and tamp er-proof.
  - ***SSH :-***
    - Uses public-key cryptography for secure authentication.
    - Configure SSH keys and use them for remote repository access.
    - ***Example clone using SSH :-***
      - ***Code :-*** *git clone git@github.com:user/repo.git*
  - **HTTPS :-**
    - Uses SSL/TLS encryption for secure data transmission.
    - Authenticate using username/password or personal access tokens.

- ***Example clone using HTTPS :-***
  - ***Code :-*** *git clone https://github.com/user/repo.git*

- **Best Practices :-**
  - ***Use Strong Authentication :-***
    - Utilize SSH keys or HTTPS with secure authentication methods (e.g., personal access tokens).
  - ***Regularly Verify Commits :-***
    - Ensure commits are signed and verify signatures to detect unauthorized changes.
  - ***Configure Git Settings :-***
    - Set up GPG keys for commit signing and configure Git to use them globally or per repository.

7. Git Worktree :-

- Git worktree allows you to maintain multiple working directories (worktrees) from a single repository.
- Each worktree has its own branch and can be used for simultaneous development tasks.
- **Usage :-**
  - ***Code :-***

```
git worktree add <path> <branch-name>
git worktree list
```

8. Git Rerere :-

- Git rerere (reuse recorded resolution) remembers how you've resolved conflicts in the past and automatically applies them when similar conflicts occur again.
- ***Usage :-***
  - Enable rerere globally :-

    Code :- git config --global rerere.enabled true

9. Git Bisect :-

- Git bisect helps find the commit that introduced a bug by performing a binary search through the commit history.
- ***Usage :-***
  - ***Code :-***

```
git bisect start
git bisect bad                  # Mark current commit as bad
git bisect good <commit>        # Mark known good commit
git bisect reset                # End bisect session
```

☺ ## Basic GitHub :-

Tuesday, 2 July, 2024 08:53 PM

Keyboard shortcuts - GitHub Docs

❖ Basic GitHub Topics :-

1. Introduction to GitHub :-

- **What is GitHub :-**
  - GitHub is a web-based platform built around Git, a distributed version control system.
  - It allows developers to collaborate on projects, track changes to code, and manage software development workflows efficiently.

- **Creating a GitHub Account :-**
  - To get started with GitHub, you need to create an account at github.com. Signing up is free and requires a username, email address, and password.

- **GitHub UI Overview :-**
  - **Dashboard :-**
    - Shows your repositories, recent activity, and projects you're involved in.
  - **Repositories :-**
    - Your projects and their versions.
  - **Issues :-**
    - Used to track bugs and feature requests.
  - **Pull Requests :-**
    - Proposals to make changes to the codebase.
  - **Projects :-**
    - Organize work with Kanban-style boards.
  - **Gists :-**
    - Share code snippets and notes.

2. Repositories on GitHub :-

- A repository, or repo, is a collection of files and folders that are tracked by Git. It exists as a directory on your local machine associated with a remote repository on GitHub or another Git host.

- **Creating a Repository :-**
  - **Follow this Steps :-**
    - Click on the "+" icon in the top-right corner and select "New repository".
    - Enter a repository name.
    - Optionally, add a description.
    - Choose visibility (public or private).
    - Initialize with a README file (optional but recommended).
    - Click "Create repository".

- **Cloning a Repository :-**
  - Cloning creates a local copy of a repository on your machine.
  - **Get the Repository URL**: Go to the repository, click on the "Code" button, and copy the URL.
  - **Clone using Git**: Open Git Bash or Terminal, use git clone <repository_url>.

- **Forking a Repository :-**
  - Forking creates a copy of someone else's repository under your GitHub account.
  - It allows you to freely experiment with changes without affecting the original project.
  - **Fork Button**: Click on the "Fork" button on the repository page.
  - **Work on Forked Repository**: You can make changes, commit them, and even create pull requests to contribute back to the original repository.

3. GitHub Flow :-

- **Issues and Bug Tracking :-**

  Issues are used to track tasks, enhancements, and bugs for a repository.
  - **Creating Issues**: Click on "Issues" tab in a repository, then "New issue". Provide a title and description.
  - **Assignees and Labels**: Assign issues to collaborators and use labels for categorization.

- **Pull Requests :-**

  Pull Requests (PRs) propose changes to a repository and facilitate code review.
  - **Creating a PR**: Click on "Pull requests" tab, then "New pull request". Compare changes and submit.
  - **Requesting Reviews**: Assign reviewers to assess the changes.

- **Reviewing Pull Requests :-**

  Reviewers examine proposed changes, comment on lines of code, approve or request modifications.
  - **Reviewing Changes**: View file changes, add comments, and approve or request changes.

- **Merging Pull Requests :-**

- **Purpose**: Integrate approved changes from a feature branch into the main branch, ensuring the main branch remains stable and up-to-date.
- **Steps**:
  - **Merge Button**: Once all reviews are complete and any required checks (e.g., CI passing) are successful, merge the pull request using the "Merge pull request" button.
  - **Squash and Merge**: Combine all commits into a single commit when merging, maintaining a clean and concise commit history.
  - **Rebase and Merge**: Rebase the feature branch onto the base branch (main branch) before merging, incorporating any changes from the base branch.

4. Collaboration on GitHub :-

□ *Managing Collaborators :-*

Control repository access by adding collaborators.

- **Settings**: Navigate to repository settings, then "Manage access". Invite collaborators via their GitHub usernames or email addresses.

□ *Using GitHub Issues :-*

- Issues can be assigned to collaborators.
- Use comments to discuss issues.

□ *Labels, Milestones, and Projects :-*

- **Labels**: Categorize issues and pull requests.
  1. Go to the "Issues" tab.
  2. Click "Labels".
  3. Create or edit labels.
- **Milestones**: Group issues and pull requests.
  1. Go to the "Issues" tab.
  2. Click "Milestones".
  3. Create a new milestone.
- **Projects**: Create Kanban boards to organize tasks.
  1. Go to the "Projects" tab.
  2. Click "New project".
  3. Set up columns and add cards.

5. GitHub Pages :-

□ *Creating a GitHub Page :-*

Host a static website directly from a GitHub repository.

- **Repository Settings**: Scroll down to "GitHub Pages" section in repository settings.
- **Choose Source**: Select a branch (typically main or master) and a folder containing your HTML, CSS, and JavaScript files.

□ *Customizing GitHub Pages :-*

Customize your GitHub Pages site with a custom domain, themes, and additional settings.

- You can use Jekyll themes or create custom HTML/CSS.
- Modify the _config.yml file for Jekyll settings.
- Add custom HTML/CSS files to the repository.

- **Custom Domain**: Link a domain name to your GitHub Pages site.
- **Themes**: Choose from available themes or create a custom theme using Jekyll.

6. GitHub Tips and Tricks :-

□ Useful Git Aliases

- **Shortcuts**: Define aliases for commonly used Git commands to save time. Add these to your .gitconfig file:

```ini
[alias]
  co = checkout
  br = branch
  ci = commit
  st = status
  lg = log --oneline --graph --decorate --all
```

□ Commit Signatures

- **GPG Signatures**: Sign commits with GPG to verify the authenticity of your commits. Configure Git with your GPG key:

```sh
git config --global user.signingkey YOUR_GPG_KEY
git config --global commit.gpgsign true
```

```sh
git config --global user.signingkey YOUR_GPG_KEY
git config --global commit.gpgsign true
```

Git Hooks for Automation

- **Hooks**: Use Git hooks to automate tasks during the Git workflow. For example, use a pre-commit hook to run tests before allowing a commit:

```sh
# .git/hooks/pre-commit
#!/bin/sh
npm test
```

7. Best Practices:

1. **Commit Guidelines**:
   - Write clear, concise commit messages that describe the purpose of each change.
2. **Branching Strategy**:
   - Adopt a branching strategy like Gitflow to manage feature development, releases, and hotfixes effectively.
3. **Code Reviews**:
   - Use code reviews to maintain code quality, provide feedback, and ensure that changes align with project standards.
4. **Documentation**:
   - Maintain comprehensive documentation within the repository, including README files and inline comments.

8. Basic GitHub Commands

1. **Repositories on GitHub**
   - gh repo create <repo-name>
   - gh repo clone <repository-url>
   - gh repo fork <repository-url>
   - gh repo list
2. **GitHub Flow**
   - gh issue create
   - gh issue list
   - gh issue status
   - gh pr create
   - gh pr list
   - gh pr status
   - gh pr merge <pr-number>
3. **Collaboration on GitHub**
   - gh collaborator add <username>
   - gh collaborator remove <username>
   - gh project create
   - gh project list
4. **GitHub Pages**
   - gh repo view <repo-name> --web

Intermediate GitHub Commands

1. **Advanced Repository Management**
   - gh repo edit
   - gh repo delete
   - gh actions list
   - gh workflow view <workflow-name>
2. **GitHub Packages**
   - gh package list
   - gh package view <package-name>
   - gh package delete <package-name>
3. **GitHub REST API**
   - gh api <endpoint>
4. **Managing Organizations**
   - gh org create <org-name>
   - gh org list
   - gh team add <team-name> <username>
   - gh team remove <team-name> <username>

Advanced GitHub Commands

1. **Security on GitHub**
   - gh secret set <secret-name> -b"<secret-value>"
   - gh secret list
   - gh secret remove <secret-name>
2. **GitHub Advanced Features**
   - gh auth login
   - gh auth logout

- gh alias set <alias> <command>
- gh alias list
- gh alias delete <alias>

3. **Enterprise GitHub**
   - Enterprise-level management usually involves web-based configurations and may include custom CLI commands depending on the specific setup.
4. **Custom GitHub Workflows**
   - Custom commands and scripts can be created as part of GitHub Actions using .github/workflows/<workflow>.yml.

☺ INTERMEDIATE GITHUB :-

Tuesday, 2 July, 2024 08:53 PM

❖ Intermediate GitHub Topics :-

1. Advanced Repository Management :-

- **Branch Protection Rules :-**
  - **Purpose**: Prevent direct commits to critical branches (e.g., main or master) to maintain code quality and stability.
  - **Enforcement**: Restrict merging without code reviews or required status checks (e.g., CI passing).
  - **Setup**: Navigate to repository settings, "Branches", and configure rules for specific branches.

- **Enabling Continuous Integration (CI) :-**
  - **Purpose**: Automate testing and build processes upon code changes to ensure code quality.
  - **Integration**: Use CI services like GitHub Actions, Travis CI, CircleCI, etc., configured via YAML files in the repository.

- **Using GitHub Actions :-**
  - **Purpose**: Automate workflows such as testing, building, and deployment directly within GitHub.
  - **Configuration**: Create workflows using YAML files (workflow.yml) in .github/workflows/ directory.
  - **Examples**: Run tests on each push, deploy to staging/production environments on merge to main.

2. GitHub Packages :-

- **Introduction to GitHub Packages :-**
  - **Purpose**: Host and manage software packages (e.g., npm packages, Docker containers) within GitHub.
  - **Usage**: Store and share packages privately within organizations or publicly for open-source projects.

- **Publishing and Consuming Packages :-**
  - **Publishing**: Use tools like npm publish for npm packages or Docker CLI for containers to publish packages to GitHub Packages.
  - **Consuming**: Access packages in other projects by configuring package registries (npm login, Docker login) and adding package dependencies.

3. GitHub REST API :-

- **Introduction to GitHub API :-**
  - **Purpose**: Access GitHub's features programmatically to automate tasks and integrate with external tools.
  - **Endpoints**: Interact with repositories, issues, pull requests, users, organizations, etc., using HTTP requests.

- **Authenticating API Requests :-**
  - **Authentication**: Generate personal access tokens or use OAuth tokens for authentication.
  - **Scopes**: Define scopes (permissions) to limit API access based on the tasks to be performed.

- **Using API to Automate Tasks :-**
  - **Examples**: Create issues, manage pull requests, update repository settings, fetch user information, automate administrative tasks.
  - **Libraries**: Utilize client libraries (e.g., Octokit for JavaScript) for easier API interaction.

4. Managing Organizations :-

- **Creating and Managing Organizations :-**
  - **Purpose**: Manage teams, repositories, and permissions centrally for groups of GitHub users.
  - **Creation**: Create an organization from your GitHub account settings.
  - **Administration**: Set member permissions, manage billing, and configure security settings.

- **Teams and Permissions :-**
  - **Teams**: Group members with specific permissions and access levels (e.g., admin, write, read).
  - **Permissions**: Control access to repositories and organization settings based on team memberships.

☺ ADVANCED GITHUB :-

Tuesday, 2 July, 2024 08:53 PM

❖ Advanced GitHub Topics :-

1. Security on GitHub :-

□ Securing Repositories :-

- **Access Controls**: Use repository settings to manage access permissions for collaborators.
- **Branch Protection**: Enforce rules (e.g., required reviews, status checks) to prevent unauthorized changes.
- **Secrets Management**: Use GitHub Secrets to securely store and use sensitive information in workflows.

⊙ **Enable Two-Factor Authentication (2FA)**:
- Go to your GitHub account settings.
- Click on "Security" and enable 2FA.

⊙ **Set Branch Protection Rules**:
- Go to your repository settings.
- Under "Branches", add branch protection rules to prevent unauthorized changes.

⊙ **Manage Access**:
- Limit collaborator access and use teams to manage permissions.
- Use least privilege principle to assign the minimum necessary permissions.

⊙ **Enable Secret Scanning**:
- Go to the repository settings.
- Under "Security & analysis", enable secret scanning to detect leaked secrets.

□ Security Alerts and Vulnerability Management :-

- **Automated Alerts**: GitHub scans repositories for known security vulnerabilities in dependencies and notifies repository maintainers.
- **Dependency Insights**: Use Dependency Graph to visualize and manage dependencies, including security vulnerabilities.

⊙ **Dependabot Alerts**:
- Dependabot automatically alerts you to vulnerabilities in your dependencies.
- Enable it in your repository settings under "Security & analysis".

⊙ **Viewing and Managing Alerts**:
- Go to the "Security" tab of your repository.
- Review and address alerts by updating dependencies or applying suggested fixes.

□ Dependency Graph :-

- **Purpose**: Visualize the dependency tree of a project, including direct and transitive dependencies.
- **Usage**: Identify outdated or vulnerable dependencies, take actions to update or mitigate risks.

⊙ **Enabling Dependency Graph**:
- Go to the repository settings.
- Under "Security & analysis", enable the dependency graph.

⊙ **Using Dependency Graph**:
- Navigate to the "Insights" tab of your repository.
- Click on "Dependency graph" to view and analyze your project's dependencies.

2. GitHub Advanced Features :-

□ GitHub CLI :-

⊙ **Purpose**: Command-line interface for GitHub operations, allowing automation and scripting.

⊙ **Usage**: Perform repository, issue, pull request, and workflow operations directly from the command line.

GitHub CLI provides a command-line interface for interacting with GitHub.

1. **Installing GitHub CLI**:
   - Follow the installation instructions on the GitHub CLI documentation.
2. **Common GitHub CLI Commands**:

bash
Copy code
```
# Authenticate GitHub CLI
gh auth login

# Create a new repository
gh repo create my-repo

# Clone a repository
gh repo clone owner/repo

# Create a pull request
```

```
gh pr create --title "New Feature" --body "Description of the feature"

# View notifications
gh issue list
```

- ***GitHub Actions Advanced Workflows :-***
  - **Advanced Configurations**: Use matrix builds, caching, and environment variables for flexible and efficient workflows.
  - **Conditional Workflows**: Trigger actions based on specific conditions (e.g., branch names, event types).

- ***GitHub Apps and OAuth Apps :-***
  - **Purpose**: Extend GitHub functionalities and integrate with external services.
  - **GitHub Apps**: Provide granular permissions and access to repositories and organizations.
  - **OAuth Apps**: Authenticate users and access GitHub resources on their behalf.
  - **Creating a GitHub App**:
    - Go to GitHub settings -> Developer settings -> GitHub Apps.
    - Click "New GitHub App" and configure its settings.
    - Install the app on repositories or organizations.
  - **Creating an OAuth App**:
    - Go to GitHub settings -> Developer settings -> OAuth Apps.
    - Click "New OAuth App" and configure its settings.
    - Use the client ID and client secret to authenticate users via OAuth.

3. Enterprise GitHub :-

- ***GitHub Enterprise Overview :-***
  - **Purpose**: Self-hosted GitHub instance for organizations requiring additional security, compliance, and administrative controls.
  - **Features**: Includes GitHub Enterprise Server (on-premises) and GitHub Enterprise Cloud (hosted by GitHub).

- ***Managing GitHub Enterprise :-***
  - **Setting Up GitHub Enterprise**:
    - Deploy GitHub Enterprise Server or use GitHub Enterprise Cloud.
    - Configure network settings, SSL, and authentication methods.
  - **User Management**:
    - Use SAML or LDAP for single sign-on (SSO).
    - Manage users and teams centrally.
  - **Installation**: Set up and configure GitHub Enterprise Server, manage user authentication, and integrate with existing infrastructure.
  - **Administration**: Control access, permissions, and security settings at the organization level.

- ***Best Practices for Enterprise :-***
  - **Security**: Implement secure access controls, monitor and manage security alerts.
  - **Scalability**: Plan for growth, optimize repository and workflow performance.
  - **Compliance**: Ensure adherence to regulatory requirements and industry standards.

4. Custom GitHub Workflows :-

- ***Creating Custom GitHub Actions :-***
  - **Purpose**: Extend GitHub Actions by creating custom actions tailored to specific project requirements.
  - **Development**: Develop actions using JavaScript, Docker, or any executable binary, and publish them on the GitHub Marketplace.

- ***Managing Complex CI/CD Pipelines :-***
  - **Pipeline Orchestration**: Integrate multiple stages, tests, and deployment steps into streamlined CI/CD workflows.
  - **Automation**: Use GitHub Actions or third-party CI/CD tools (e.g., Jenkins, CircleCI) for automated builds, tests, and deployments.

- ***Integrating Third-Party Tools :-***
  - **Integrate CI/CD Tools**:
    - Use GitHub Actions or third-party CI/CD services like Jenkins, CircleCI, or Travis CI.
  - **Integrate Project Management Tools**:
    - Use tools like Jira, Trello, or Asana to manage tasks and track progress.
  - **Security Tools**:
    - Integrate security tools like Snyk, WhiteSource, or CodeQL to scan code for vulnerabilities.
  - **API Integrations**: Use GitHub API to automate interactions with external tools and services.

- **Webhooks**: Receive real-time notifications and trigger actions in response to GitHub events.
- **Extensions**: Explore GitHub Apps and integrations available on the GitHub Marketplace for additional functionalities.