VERSION CONTROL

Git & Github

GIT DIFF

The git diff command is a useful tool in Git to view the differences between various states of your repository.

Let's Setup a new Repository:

Create Directory: mkdir demo

Move to the directory: cd demo

Initialize a Git repository: git init

GIT DIFF

- Edit the file: echo "Added new line" >> file.txt
- See the difference: git diff
- It shows changes in the working directory compared to the last committed version.
- Added lines are prefixed with +
- removed lines are prefixed with –

STAGED CHANGES

Stage the changes: git add file.txt

View differences between the staged version and the last commit:

git diff --cached

Displays changes between the staging area (index) and the last commit.

COMPARE DIFFERENT COMMITS

Commit the staged changes

```
git commit -m "Added a new line"
```

Make further changes

```
echo "Another change" >> file.txt
```

```
git add file.txt
```

```
git commit -m "Added another change"
```

View differences between two specific commits

```
git diff <commit1> <commit2>
```

GIT DIFF FLAGS

Show Differences for a Specific File:

- git diff file.txt

You can also compare 2 different branches:

- git diff branch1 branch2

RENAME

Rename:

- git mv <old-filename> <new-filename>

If you renamed a file accidentally, you can revert before committing:

- git restore --staged old_filename new_filename

After committing, you can view renames in the diff:

- git diff --find-renames

UNDO CHANGES

Undoing changes to staged files:

- git reset HEAD <filename>

Undoing a commit: If you have already committed changes and want to undo the last commit

- git reset --soft HEAD^

This will undo the last commit but leave your changes staged.

Completely discarding all local changes:

- git reset --hard HEAD

REVERT

If you have already pushed changes to a remote repository and want to revert a commit, you can use:

- git revert <commit-hash>

This will create a new commit that undoes the changes introduced by the specified commit.

SHOW COMMIT HISTORY

Git log: show all commits

To exit from many commit you can press q

git log -n 5: This will show the last 5 commits in your repository's history.

Search within git log:

- git log --grep="search-term"

Git log in one line: git log –oneline

Git show: showing details of commit

BRANCHING

- Create one folder for understanding the branch concepts
  - mkdir git-branching-demo
  - cd git-branching-demo
  - git init
- Create a file
  - echo "This is the main branch." > app.txt
- Stage and commit the file
  - git add app.txt
  - git commit -m "Initial commit: Add main branch content“
- git branch add-login
  - git checkout –b add-login

BRANCHING

- Verify Branch
  - git branch
  - The active branch will be highlighted with *.
- Make Some chnages into new Branch:
  - echo "Feature: Add user login functionality." >> app.txt
- Stage and commit the changes
  - git add app.txt
  - git commit -m "Add login feature“
- Switch back to main Branch:
  - git checkout main
- Verify the file:
  - cat app.txt

MERGING

- Switch to the main branch
  - git checkout main
- Merge the feature branch
  - git merge add-login
- Verify the file:
  - cat app.txt
- Delete the Branch:
  - git branch -d add-login

SOME MORE

- Create and switch to a bug-fix branch
  - git checkout -b fix-login-bug
- Make changes
  - echo "Fix: Correct login functionality." >> app.txt
- Stage and commit the fix
  - git add app.txt
  - git commit -m "Fix login bug"
- Merge back into main
  - git checkout main
  - git merge fix-login-bug
- Delete the bug-fix branch
  - git branch -d fix-login-bug

EXAMPLE

- Create a new branch named 'feature/new-feature' and switch to it
  - git checkout -b feature/new-feature
- Make changes to your code and commit them on the 'feature/new-feature' branch
  - git add .
- git commit -m "Added new feature"
- Switch back to the main branch
  - git checkout main
- Compare tips of two branches (main and feature/new-feature)
  - git diff main feature/new-feature
- View all branches (local and remote)
  - git branch -a
- View HEAD pointers of all branches
  - git show-ref
- Merge 'feature/new-feature' branch into 'main' (assuming no conflicts)
  - git merge feature/new-feature
- Delete the 'feature/new-feature' branch after merge
  - git branch -d feature/new-feature

RESOLVING CONFLICT MANUALLY:

- Create and switch to a new branch (feature/branch-c):
  - git checkout -b feature/branch-c
- Make changes to file.txt and commit them
  - echo "Feature branch content" > file.txt
  - git add file.txt
  - git commit -m "Made changes in feature branch"
- Switch back to main and make conflicting changes:
  - git checkout main
- Make conflicting changes to file.txt and commit them
  - echo "Main branch content" > file.txt
  - git add file.txt
  - git commit -m "Made changes in main branch"

- Merge feature/branch-c into main, causing conflicts:
  - git merge feature/branch-c
- Git will detect that file.txt has conflicting changes in both branches and will output a message indicating a merge conflict.
- Decide which changes to keep or modify file.txt accordingly.
  - Feature branch content and Main branch content combined (modify it manually)
- Stage the resolved file and commit the merge:
  - git add file.txt
  - git commit

TAGGING

- tags are used to mark specific points in the repository's history, such as releases or significant commits.
- There are two types of tags:
  - lightweight tags
  - annotated tags.

LIGHTWEIGHT TAGS

- A lightweight tag is simply a pointer to a specific commit. It's similar to a branch that doesn't change — it's just a reference to a commit.
- git tag <tag-name>
- E.g. git tag v1.0.0

ANNOTATED TAGS:

- An annotated tag, on the other hand, is stored as a full object in the Git database.
- It includes a tagger name, email, date, and a tagging message.
- Annotated tags are recommended for most use cases as they provide more information and context about the tag.
- git tag -a <tag-name> -m "Tagging message"
- E.g. git tag -a v1.0.0 -m "Initial release version 1.0.0"

VIEWING TAGS:

- To view all tags in the repository:
  - Git tag
- View details of specific tag:
  - git show <tag-name>

PUSHING TAGS:

- Tags created locally are not automatically pushed to remote repositories.
- To push tags to a remote repository:
  - git push origin <tag-name>
- To push all tags to the remote repository:
  - git push origin --tags

DELETING TAGS:

- To delete a tag locally:
  - git tag -d <tag-name>

- To delete a tag from the remote repository (after it has been pushed):
  - git push origin --delete <tag-name>

TAGS WITH REAL SCENARIO

- You are working on a project and want to tag significant milestones, such as "v1.0" for the first release and "v1.1" for an update.
  - Create folder, move to it and initialize git repo.
  - Create a file, stage and commit.
  - Create light weight tag: git tag v1.0
- Verify tag: git tag
- You can also create annotated tags because they include metadata such as the author, date, and a message.
  - git tag -a v1.0 -m "First release version 1.0“
- Verify: git show v1.0
- Push Tags: git push origin v1.0
- Make changes to the existing file, stage and commit.

TAGS WITH REAL SCENARIO

- Create Tag version: git tag -a v1.1 -m "Second release version 1.1“
- See all tags: git tag
- If you want to tag some older commit then:
  - git tag -a v0.9 hash-code -m "Tag for pre-release version 0.9“
- Delete locally
  - git tag -d v0.9
- Delete from remote (if already pushed)
  - git push origin --delete v0.9

KEY TAKEAWAYS

A tag always points to a specific commit.

If you commit first and then add a tag, the tag will be associated with the commit you just made.

Tags are useful for marking specific points in your repository's history, like releases or milestones.

FORKING A REPOSITORY

- Forking a repository in GitHub creates a personal copy of someone else's repository under your account.
- Benefits:
- Make changes to a project without affecting the original repository.
- Submit improvements or bug fixes by creating pull requests from your forked repository.
- Start your own project based on the original repository.

LET'S FORK ONE REPO

- Go to any repository which you want to fork and click on fork button to fork the repository.
- When you fork the repository its copy is getting created in your account.
- To get that code in you local system you can clone that copy repo from your account and set upstream to existing repo.
- git remote add upstream https://github.com/original-author/project-repo.git
- To make some changes you can create branch
- Add and commit the changes and push changes to your branch.
- git push origin feature-branch
- You can submit pull request

ACTIVITY