Git & GitHub Theory

https://www.atlassian.com/git

30 May 2024 00:17

Git
Free and open source distributed vcs (version control system)

- Useful to track code changes
- Gives ease of collaboration (without over-writing)
- Great for taking Backup
- Allows user to free experiment since separate branches can be created without affecting master or main branch
- Industry standard (fast & scalable)
- Open-source Projects contributions can be done

Git Repository Structure
4 parts:

1. **Working directory:** This is our local directory where we make the project (write code) and make changes to it.
2. **Staging Area** (cache area or index): An area where we first need to put our project before committing. This is used for code review by other team members.
3. **Local Repository**: Our local repository where we commit changes to the project before pushing them to the central repository on GitHub. This is what is provided by the distributed version control system. This corresponds to the .git folder in our directory.
4. **Central Repository**: This is the main project on the central server (viz. GitHub main repo), a copy of which is with every team member as a local repository.

Git Workflow
init/clone --> add to staged area by **add** command --> commit the file to complete the changes --> push the changes to remote repo
(OR create a pull request in remote repo)

*Note: From long time, default branch name was '**master**' which has now been proposed to change as '**main**' in near future (once they're (Git founders) comfortable that existing customers 'master' named branch is not impacted)*

Merging in Git
2 ways:

i) Creating a pull request from GitHub UI, the target branch owner will review & approve the merge.
After that we can use **git pull** to get changes to our local git further.
ii) Using **git merge <branch name to merge>** : Creates a new commit in current branch by combining the code received from **<branch name to merge>** branch

.gitignore files
.gitignore files are used to tell the git tool to intentionally ignore some files in a given Git repository. For example, this can be useful for configuration files or metadata files that a user may not want to check into the master branch.

Note: When we rename file in local, Git considers it like we've deleted the file and add a new file

Fork a Repo
Add a copy of someone else Repo into our GitHub using Fork button on GitHub

GitHub
Used to hosts Git repositories online. Helps users share their git repository online with other users, or access it remotely.
We can host a public repository for free on GitHub.

User share their repository online for various reasons like project deployment, project sharing, open source contribution, helping out the community etc

Various remote repository hosting sites: GitHub, BitBucket, Gitlab

Setting up Git bash

30 May 2024 00:40

//To see whether Git is properly installed
//Use double-dash before version
git --version

//Set global profile (can be specific for diff user too)
git config --global user.name myName
git config --global user.email myEMail@com
//set a name that is identifiable for credit when review version history
//set an email address that will be associated with each history marker

//set automatic command line colouring for Git for easy reviewing
git config --global color.ui auto

//To view global configuration
git config --global --list

//Since GitHub has moved to default branch as '**main**' instead of earlier '**master**',
// we can configure same on our Git as well
git config --global init.defaultbranch main

//To unset/remove any config (key mean the config name to unset)
git config --global --unset key

//To edit config file
git config --global --edit

Note:
As long as the bash current directory is not a git enabled (git init or clone has run) directory, the config can be only applied for global.
To see current working directory: pwd

Eclipse & Git

30 May 2024 01:31

Open Help --> 'Eclipse Marketplace' in Eclipse
Search for **EGit**
Choose the **EGit - Git Integration ….** & install

Goto Window --> Show View --> Other
Choose Git --> Git Repositories

Click on 'Clone a Git Repository' & choose 'Clone URI'
Provide GitHub Repo link (Open repo on GitHub, click Code & copy the link) & necessary details

Right click on existing project (if any) --> Team --> Share Project
Choose your Git from the dropdown and finish

However, do note that the terminal is not available in Eclipse by EGit Integration.
We either go for other plugins OR better use Git bash itself by navigating the directory where eclipse git repo in present in local

Git Commands

30 May 2024 01:58

git clone <--clone-link-->
To initialize our local workspace as a Git enabled directory (workspace/repository) and retrieve an entire repository from a hosted location via URL (create a copy of an existing remote (GitHub) repository in our local directory/workspace)

git status
Shows status of our codebase
(whether any untracked, modified files in working directory, tracked file staged for commit OR unchanged)

4 status is possible
untracked: new file's that isn't yet tracked by Git (need to use **add** command to add them to staging (index) area for git to consider the file for tracking further)
modified: any changes to a git tracked file (whose new changes has not been added yet)
staged: files is ready to commit [includes files which were earlier untracked & then added using add and files which were tracked earlier also but has been modified & then add command is run]
unmodified: Meaning no changes have done so far since the previous commit

git add *<filename>*
Add our changes/untracked file to the staging area of Git in order to commit further. *Use '.' dot for all files.*

git restore <file-path with file-name>
Restores the tracked files as per latest commit

git reset *<filename>*
Reset our staging area changes by unstaging for the changes
[File changes will be visible in our workspace however not in the staging area; i.e. file is in modified/untracked status]

git clean
Removes untracked files from the working directory. This is similar to git reset, which (typically) only operates on tracked files.

git commit -a
Automatically adds all modified files in the staging area, they're ready to be committed.
Note that new (untracked) files are not staged; only existing files which are modified are staged.

git commit -m "[descriptive message]"
Commit our changes in staged area as a new commit snapshot. This symbolises a record of change

git commit --allow-empty -m "[descriptive message]"
Allows to commit even if we don't have any changes after the latest commit. Used to mark some event such as start of new feature OR new team OR new sprint etc

git log
To see all commits details starting from latest. [Latest commit is called head]

git push origin
To push our local commits to remote repo of GitHub into all the branches [not recommended]

git push origin main
To push our local commits to remote repo of GitHub into it's **main** branch only
[here by default 'origin' is the remote repo name generally used in Git]

For ease of use, remote repo name 'origin' & branch 'main' can be set as upstream;
git push -u origin main
*Now, only **git push** is sufficient to push our local commits to GitHub*

git pull origin <remote-branch-name>
To fetch the latest code from remote branch and add them into our current local branch
https://www.atlassian.com/git/tutorials/syncing/git-pull
All new commits from point where we create our local current branch will be imported in local branch

//The above was feasible when we've data in remote repo (GitHub)
//However, to start a new project from local, use **init** command to make local directory git enabled
//Next, create empty repo on GitHub and copy it's link from 'Code' button
//Now, we've to link our Git to point the **origin** repo to our correct GitHub repo

git init
initialize an existing local directory as a Git repository

git remote add origin <--repo_link-->
To point the origin (Git's default name for remote repo) to our correct GitHub repo

git remote -v
To ensure our Git points to correct link of the GitHub remote repo

Quick Summary: https://www.geeksforgeeks.org/essential-git-commands

git rm: To remove (delete) file **git mv**: To move a file
git remote: Lists all remote repos

vi editor on Git bash

To move left, press h	To move right, press l
To move down, press j	To move up, press k

Branch
Branch is a new/separate version of the main repository, so that collaboration is easy & efficient. For each task/user, a branch can be created & changes pushed into that which has to be later merged into the main.
Whenever we checkout to a branch using Git, the data of that respective branch only is visible in our local computer Git enabled folder.

//Do Note that in below, simply 'branch' refers to branch in our local repo

git branch
To view list of branches (in local Git)

git branch -r
To view list of branches on remote repo (GitHub)

git branch -a
To view list of branches on local Git as well as remote repo (GitHub)

git branch -M main
To rename the current branch as **main**

git branch -d <branch-name>
To delete a branch [error comes if we delete current branch; first switch (checkout) to another branch & then delete]. Also it will work if latest commits/changes are merged to the current branch fully. Still if deletion is required use, ***git branch -D <branch-name>***

git branch -d <remote-branch-name>
It will only delete the remote branch from Git local view, however will remain on GitHub
To remove from GitHub too, run **git push origin -d <remote-branch-name>**

git push origin -d <remote-branch-name>
To delete a remote branch

git checkout <branch-name>
To switch to different branch

git branch <branch-name>
To create a new branch at the current commit (all history of commits starting from initial commit of the current branch is preserved & present); ***we'll be still at current branch***
[Only local branch can be created; remote branch can't be created directly, we need to push our commits to remote by specifying a new-branch-name for new branch creation

git checkout -b <branch-name>
To create a new branch at the current commit and ***switch to the newly created branch***

git checkout --orphan <branch-name>
To create a new branch at the current commit (with all commit history from initial commit of the current branch) and switch to the newly created branch

git push -u origin newbranch
Pushes local commits to remote GitHub newbranch branch (creates a new branch 'newbranch', if it doesn't exists)

git pull origin <remote-branch-name>
To fetch the latest code from remote branch and add them into our current local branch
[git fetch does same but doesn't automatically add changes to our local]

git diff <branch-name>
To compare difference of any branch-name vs current branch (latest commit)

git diff -u <branch-name>
Used to compare branch difference, line by line, and have the differing lines compared side-by-side in the same output with latest commit of our current branch

git diff --staged
Show all staged files compared to the current branch (latest commit)

git diff -u <branch-name-1> <branch-name-2>
Shows difference b/w the two branches, line by line

git merge <branch-name>
We can combine <branch-name> into our current branch and a new commit is added in current branch

Undoing Changes

31 May 2024 00:38

Case 0: non-staged changes
git restore <file-path with file-name> [only tracked files will be restored as per latest commit]

Case 1 : staged changes
git reset <- file name ->
git reset

Case 2 : committed changes (for one commit)
git reset HEAD~1 OR git reset --soft HEAD~1 [both commands will be doing same]

Case 3 : committed changes (for many commits)
git reset <- commit hash -> [local files not affected, will be visible as modified unstaged files in local]
git reset --hard <- commit hash -> [will reset files in local directories too]

NOTE: Using **git reset** All commits done after the mentioned commit-id will be removed

git revert HEAD~1 [HEAD~1 means we want to refer commit 1 before the current HEAD (latest commit)]
git revert <- commit hash ->
Revert will create a new commit by reversing the changes from latest commit back to mentioned commit-id (like a roll-back of changes).
Will not remove any commit history.

git commit --amend
Useful to update our commit message
This command will create a new commit with our updated commit message & all if any more ready to be commited files in staged area along with the changes of our latest commit and then it deletes the latest commit.
So, we get only one commit overall.

Now, to update the same on GitHub (Remote Repo)
git push --force origin <branch-name>

Recovering Lost Commits

git reflog
Shows record of when the tips of branches & other references were updated in local repo
It will display all the checkouts, commits, pull from remote repo, merges that have been done on the HEAD for all the branches. We can get our commit hash from here

git cherry-pick <commit-hash>
This will create a new commit by merging the commit specified by commit-hash

Other few commands:
https://www.geeksforgeeks.org/difference-between-add-a-add-u-add-and-add
https://www.geeksforgeeks.org/git-working-with-stash [used for temporary saving local vidout commit]
https://www.geeksforgeeks.org/git-squash [used for cleaner commit history]
https://www.geeksforgeeks.org/how-to-use-git-log-to-format-the-commit-history

Merge

30 May 2024 23:21

MERGE Conflict
Often, multiple people can change a same file (generally when a same line is impacted) from their respective branches (or even a single user does changes from multiple branch) and hence, Git is unable to resolve whose changes to keep.

When Git is unable to resolve difference between current vs another branch file, it's called MERGE CONFLICT.
We need to manually resolve this. [without resolving, we won't be able to further push our local commits].

Scenario:
We commited something in local main.
Next we commited something in same file in local secondaryBranch

Now, when we checkout to main branch and do git merge secondaryBranch, it will have a conflict as Git is not able to resolve which among the main branch or secondaryBranch commited changes to be given priority OR both is needed.

Generally, difference in different lines gets resolved by Git however in same line it will show conflict message and ask us to resolve.

Then, manually go to that file and perform changes.

Git will also add the changes marker like below; HEAD point to current branch, 'daily' is name of branch which we're merging to our current branch:

<<<<<<< HEAD
Create empty Git repo in the specific directory in local repo
=======
Create empty Git repo in the specific directory in local
>>>>>>> daily

Remove the markers and non-needed lines and do git add and git commit to complete the Merge successfully.

Fast-Forward Merge
Take two branches, say 'main' and 'feature'
Now, assume 'feature' & 'main' is currently having same data in latest commit.

Next 'feature' branch committed 1 or more times.
However, 'main' branch has no further commit since then.

Now, if we merge 'feature' into 'main' branch; since linear changes are present, it is fast-forward merging and won't create separate commit-id in 'main' and use same commit of feature branch in both branches

git merge --abort
To abort a merge operation

git-blame <file-name>
git-blame is a Git command that displays each line of a file with information about the commit that last modified it. This includes the author of the change, the commit hash, and the timestamp of the change. This information can help you understand the context of changes and identify who to ask about specific lines of code.

Merge vs Rebase

01 June 2024 12:50

Merging
In Git Merge, a new commit is created and the commit history from the point where branch was initially separated till latest commit of both branch are preserved in the reverse chronological order of time.

Example:
Let's assume a 'main' branch having commits C1, C2, C3.
Create a 'temp' branch from 'main'.
'temp' branch will have commit log history as:
C3
C2
C1

Now, checkout to 'main' branch and do a commit, say MC1.
Checkout to 'temp' and do commit, say TC1.
Again, checkout to 'main' branch & do commit MC2.
Checkout back to 'temp' and do commit, say TC2.
Now merge 'main' into 'temp' branch.

'temp' branch will have updated commit log history as below:
TC2
MC2
TC1
MC1
C3
C2
C1

Hence, 'temp' branch commit log history will store all the commits of both 'temp' and 'main' together as they were done in time.

Rebasing
It is used to change the point of separation.

Example:
Let's assume a 'main' branch having commits C1, C2, C3.
Create a 'temp' branch from 'main'. [Basically, branch creation point is C3]
'temp' branch will have commit log history as:
C3
C2
C1

Now, checkout to 'main' branch and do a commit, say MC1.
Checkout to 'temp' and do commit, say TC1.
Again, checkout to 'main' branch & do commit MC2.
Checkout back to 'temp' and do commit, say TC2.
Now run*, git rebase main* to rebase 'main' into 'temp' branch.

'temp' branch will have updated commit log history as below:
TC2
TC1
MC2
MC1
C3
C2
C1

Hence, commits of 'main' will be first stored and then the commits of 'temp'

Reason: Separation point which was initially C3, is now moved to MC2 and then the further commits of 'temp' will be applied on top of the same.

It's like, pull all commits of other branch onto current branch till latest commit of other branch and then apply all current branch commits on top of them.

Git Cheat-Sheet

02 June 2024 00:34

BASICS

1. **git init** <**directory**>
Create empty Git repo in the specific directory

2. **git clone** <**repo**>
Clone repo located at the <repo> defined. Can be either local or removed via HTTP or SSH

3. **git config user.name** <**name**>
Author name that would be used for all commits in the current repo

4. **git add** <**directory**>/<**file**>
Adds the directory/file on your next commit

5. **git commit -m** "<**message**>"
Commits your staged content as the new commit snapshot with a message

6. **git status**
Shows modified files in the working directory

7. **git log**
Shows all commits in the current branch's history

8. **git diff**
Shows the difference of what's changed but not staged

UNDOING CHANGES

1. **git revert** <**commit**>
Undoes all the changes made in <commit> then apply it to the the current branch

2. **git reset** <**file**>
Removes the file from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes

3. **git clean**
Used to remove the unwanted files from your working directory. Many options to remove the unwanted files i.e., -f, -n, -d, etc.

REWRITING GIT HISTORY

1. **git commit --amend**
Replace the last commit with the staged changes and the last commit combined. Use with nothing staged to edit the last commit's message

2. **git rebase** <**base**>
Rebase the current branch onto <base>. <base> can be a commit ID, branch name, a tag, or a relative reference to HEAD

3. **git reflog**
Show a log of changes to the local repository's HEAD

GIT BRANCHES

1. **git branch**

Lists all of the branches in your repo

2. **git branch <branch-name>**

Create a new branch of the current commit

3. **git checkout**

Switch to another branch and check it out into your working directory

4. **git merge <branch>**

Merge the specified branch's history into the current one

REMOTE REPOSITORIES

1. **git remote add <name> <url>**

Create a new connection to a remote repo. After adding a remote, you can use <name> as a shortcut for <url> in other commands

2. **git fetch <remote> <branch>**

Fetches a specific <branch>, from the repo. Leave off <branch> to fetch all remote refs

3. **git pull <remote>**

Fetch the specified remote's copy of the current branch and immediately merge it into the local copy

4. **git push <remote> <branch>**

Push the branch to <remote>, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist