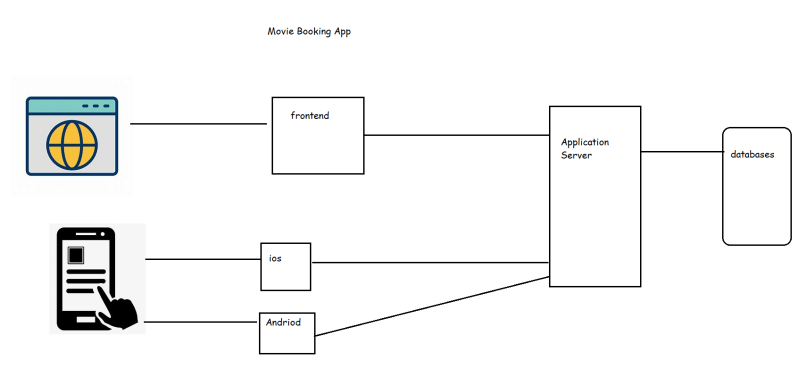
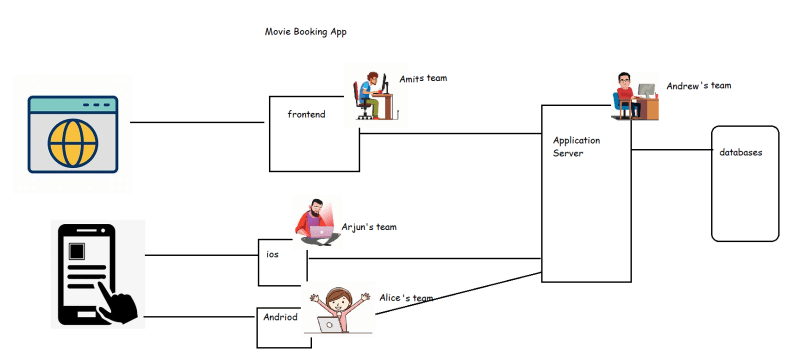
CI/CD-Git-1 29/June/2021

Need for Version Control System and Continuous Integration

* Multiple Developers working on to build an application.
* Each Developer works on his local system, How can we integrate the code from multiple developers
* Big Bang Integration:
  + Consider the movie booking app



* + Each team is working on their own 
  + Just before 15 days to release all the teams try to make their applications interact with each other by integrating the code.
  + It was observed in many cases that the team will observe lot of failures and to meet the release date, they make temporary fixes which will ruin the application quality
* To make the application development process simpler continuous integration was introduced where all the code by different teams will be integrated every time (hourly, daily) then create an application package
* Options for sharing code
  + email
  + shared folder
* Merging the code has to be done by all the developers manually.
* We need the following options
  + versions
  + merge the changes
  + revert the changes
* Version Control is a system that records changes to a file or group of files and directories over time and it allows multiple developers to integrate their work.
* Some popular Version Control Systems
  + Subversion(SVN)
  + IBM Clear Case
  + Team Foundation Version Control
  + Perforce
  + Git
* Version Control Architecture Evolution.
  + Client Server to Distributed Systems

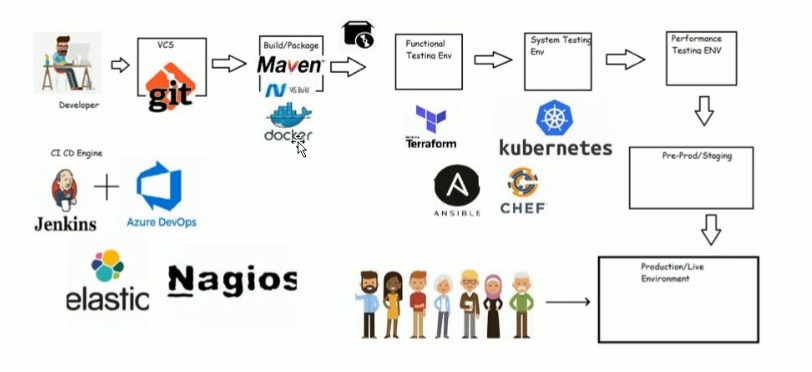
Relevance of Version Control for DevOps Engineer

* As a DevOps Engineer, we will work to develop scripts or configurations (Terraform, Chef, Ansible, Docker, k8s) and we need to integrate this activity with the team, so we need to have a good working knowledge on Version Control System.
* In some organizations, DevOps Engineers are expected to ensure all the team members are using version control system efficiently.
* The CI/CD Pipeline starts from VCS, so we need to understand the VCS to trigger the builds & continue with pipeline.

References

* Installing necessary software’s [Refer Here](https://www.youtube.com/watch?v=mRILfUNbsIo&list=PLuVH8Jaq3mLud3sVDvJ-gJ__0zd15wGDd&index=14)

<https://www.youtube.com/watch?v=mRILfUNbsIo&list=PLuVH8Jaq3mLud3sVDvJ-gJ__0zd15wGDd>

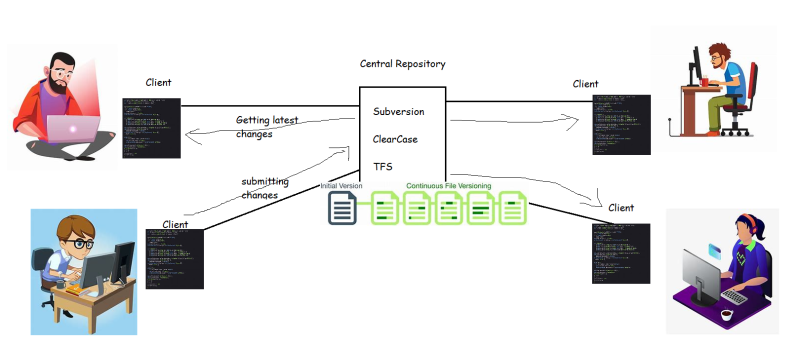


===========================================================================

CI/CD-Git-2 30 /June/2021

**Centralized Version Control System**

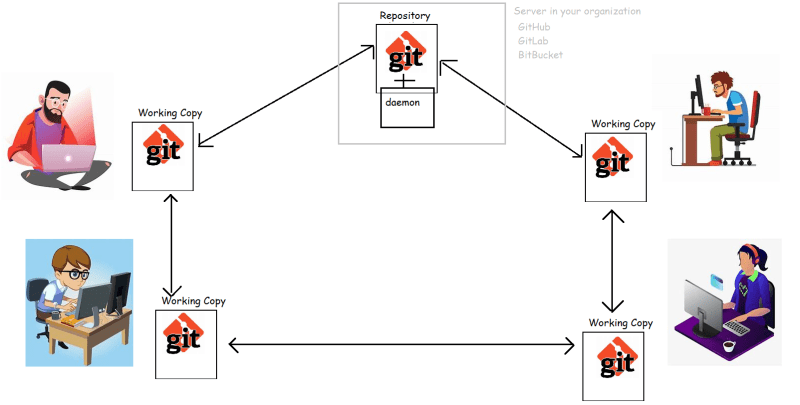
* Version control (Revision Control) is a system that records changes to a file or groups of files and directories over time, so that we can review or go back to specific versions. In addition to this it will allow multi user mode
* In Centralized System, anyone who makes changes needs to give access to the central location (Central Repository)

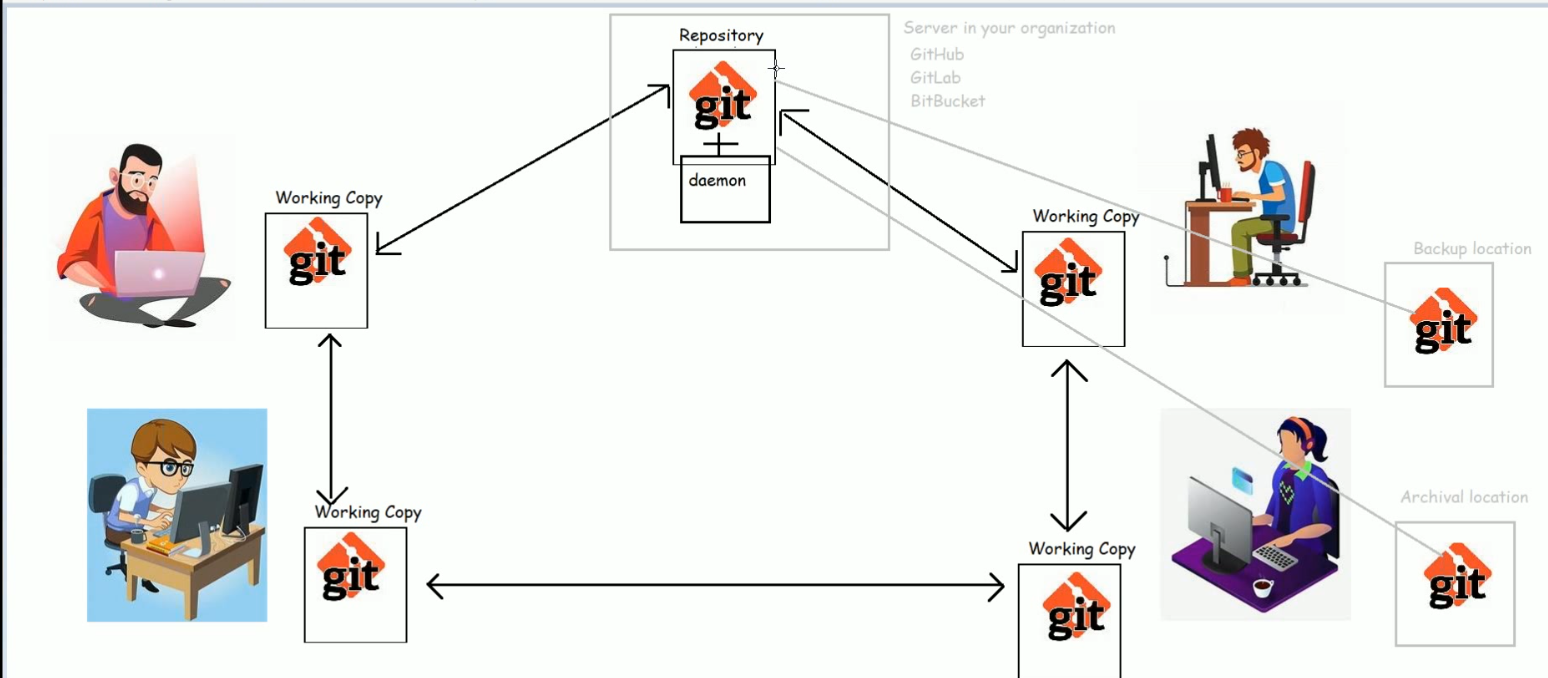


* Once they have the access and the client software, the users can get the working copy, make changes, get latest changes etc…..
* For any reason if the central repository is down, all the developers will not be able to
  + make changes in connected centralized version control system
  + update/get changes in the disconnected centralized version control system
* To resolve this, we have version control administrators who configure
  + Primary and Standby Servers
* For any version control system, backups and archivals are important, Administrators configure this (Subversion Administrator, Clear case Administrators)

## Distributed Version Control System

* Git was created by Linus Torvalds (who is also known for the Linux kernel).
* The primary objective behind Git was to implement and design a version control system that is distributed, reliable and fast.





Archival is for disaster recovery.

## Working with Git

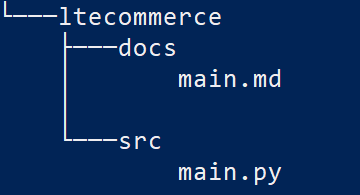
* Installing Git:
  + Windows: [Refer Here](https://git-scm.com/download/win) (https://git-scm.com/download/win)
  + Mac: [Refer Here](https://git-scm.com/download/mac)
  + (https://git-scm.com/download/mac) follow homebrew approach
  + Linux:
    - RedHat: sudo yum install git -y
    - Ubuntu: sudo apt install git -y
* Learning Git Commands (git cheat sheet in google)

==========================================================================

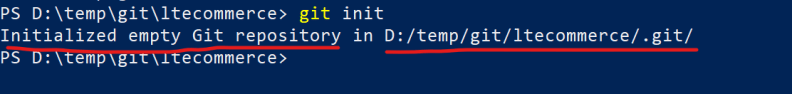
CI/CD-Git-3 01/July/2021

Working with working tree and staging area:

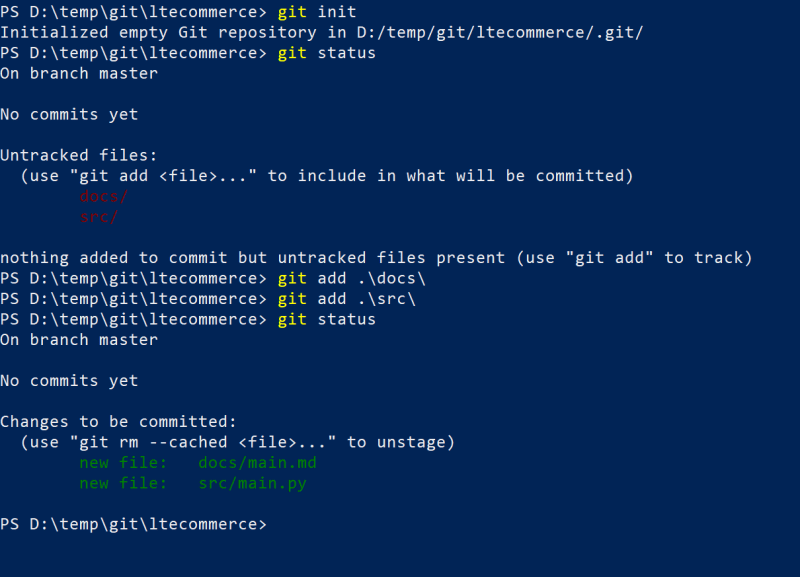
* Let’s understand the basic process of Version Control with git.
* Create a directory and some files and directories into it



* We need to add version control capabilities to this directory. In Git if we need a version control, we need to have a repository.
* So to make this directory structure a repository (local repository) we need to initialize using git init



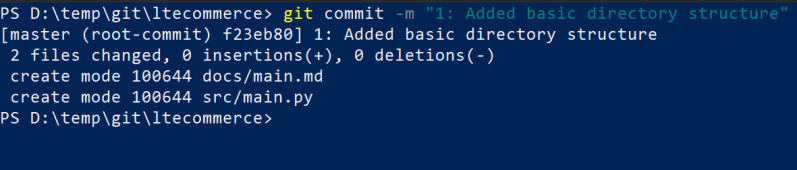
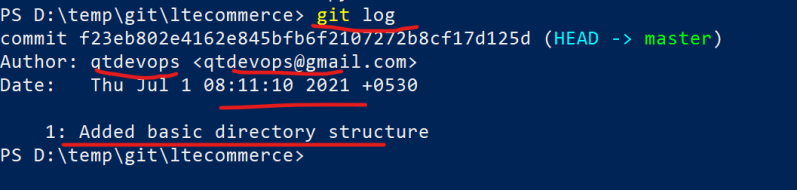
* Let’s add the changes from working tree to staging area by using add command

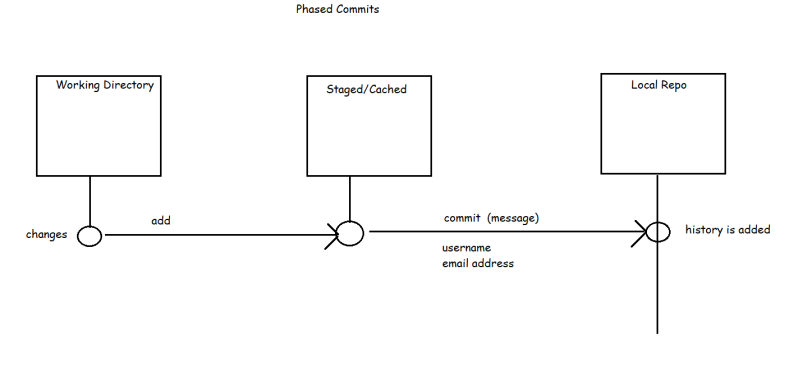


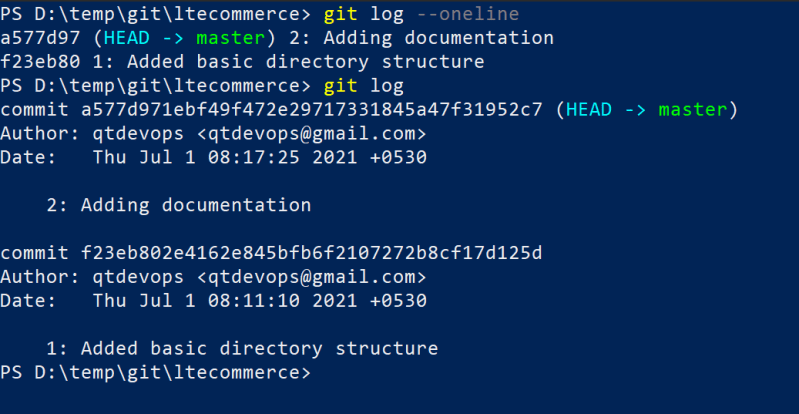
* Since commit requires username and email let’s try to configure the username and email (This is one time activity)

git config --global user.name 'qtdevops'

git config --global user.email 'qtdevops@gmail.com'

* Now let’s commit the changes from staged area to local repository
*  
* The above process can be summarized as shown below



* Let’s add some more changes as second commit
* 
* **Quick commands to explore**
  + git init
  + git status
  + git add
  + git commit
  + git log

===============================================

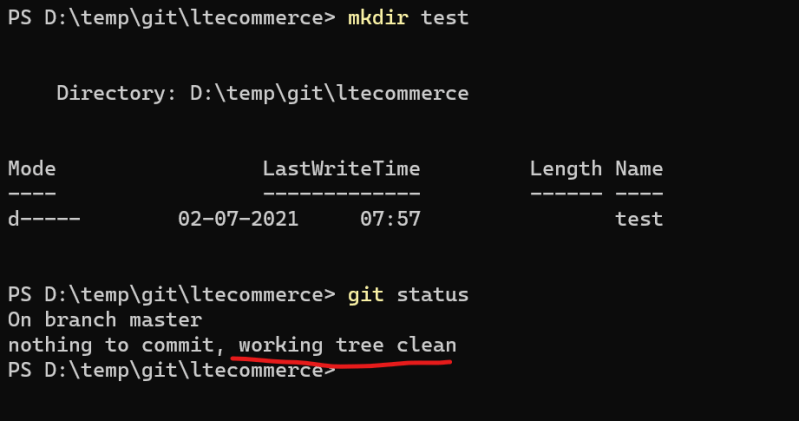
CI/CD-Git-4 **02/July/2021**

## Adding new files to working tree:

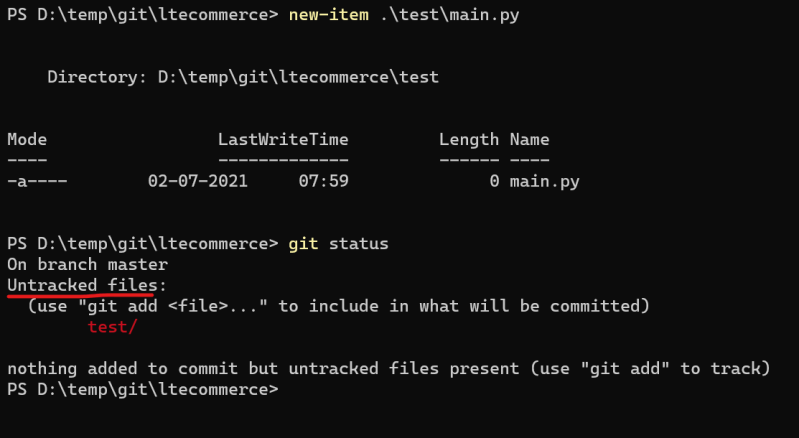
## Let’s create a new folder in the working tree/directory

**Note**: To install and configure Windows Terminal [Refer Here](https://www.youtube.com/watch?v=qLVn2EvPsYc&list=PLuVH8Jaq3mLud3sVDvJ-gJ__0zd15wGDd&index=11) <https://www.youtube.com/watch?v=qLVn2EvPsYc&list=PLuVH8Jaq3mLud3sVDvJ-gJ__0zd15wGDd&index=12>

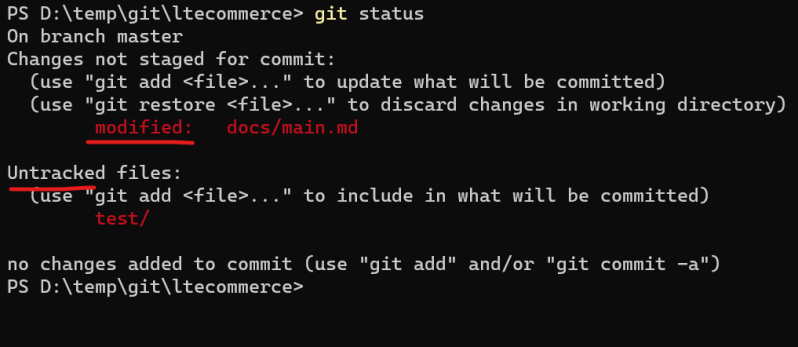
* Git will not detect(GURTINCHADAM) empty folders as changes because git works with files and paths of files



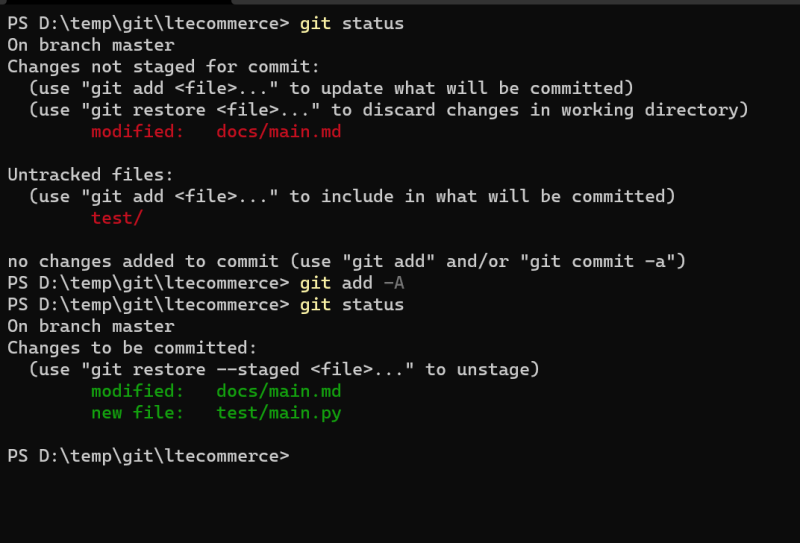
* Let’s add some file in the empty directory

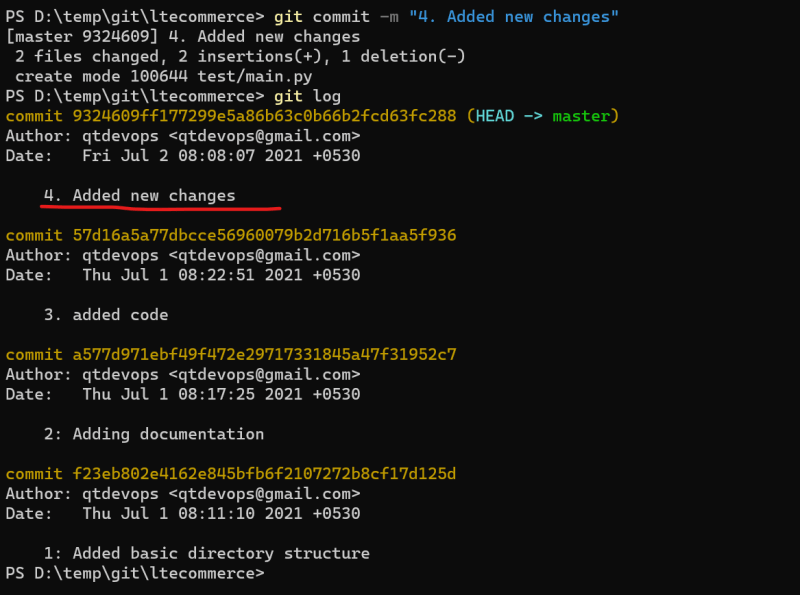


* Now let’s make some changes in existing files

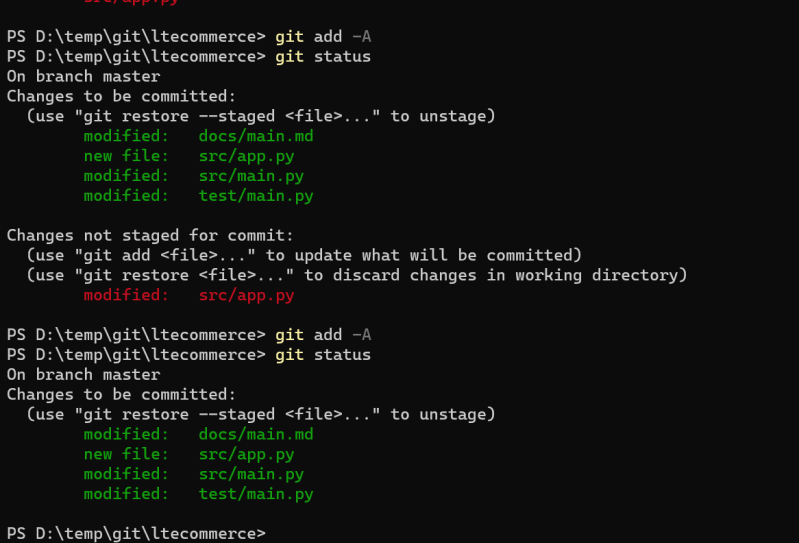


* **Untracked** files are the newly added files which are not part of git local repository and modified are the changes done the existing files which are part of git local repository
* Now let’s add all the changes to the staging area git add -A

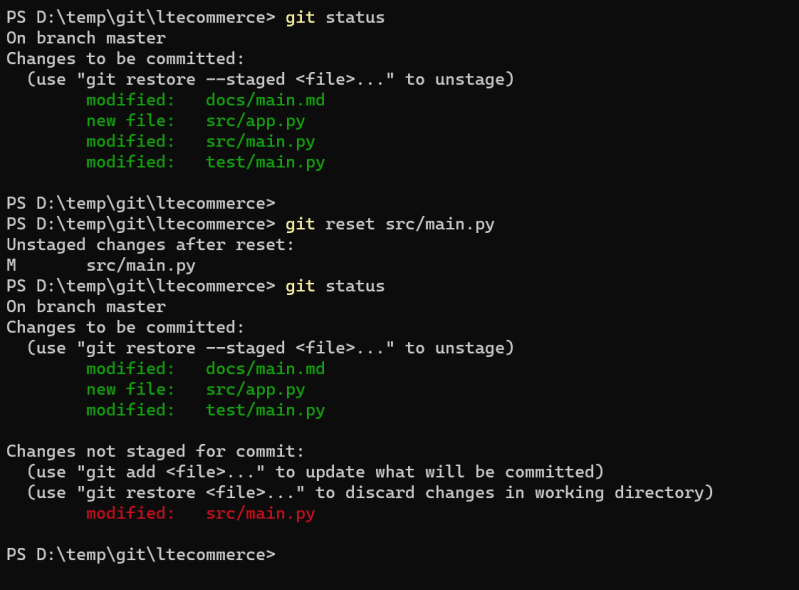




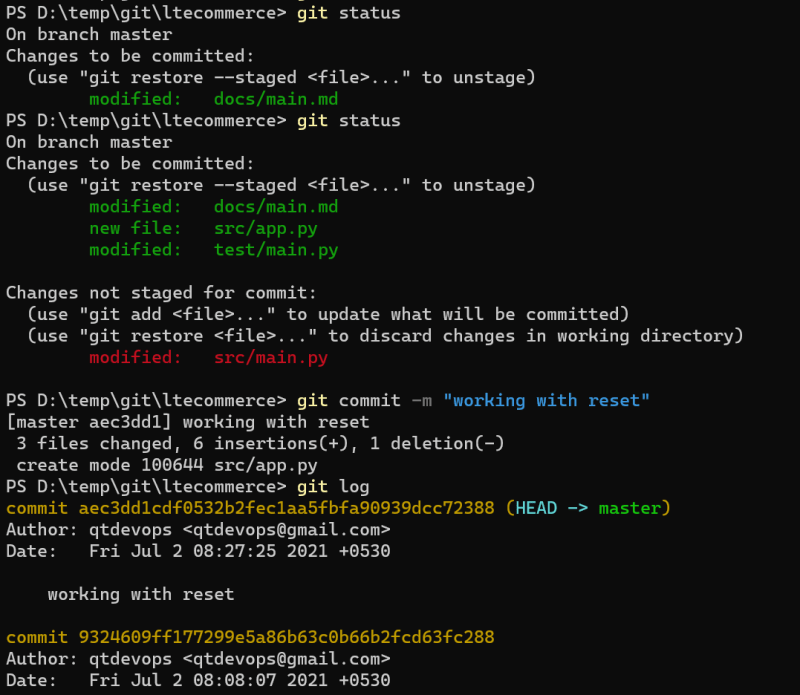
* Adding changes to staging area



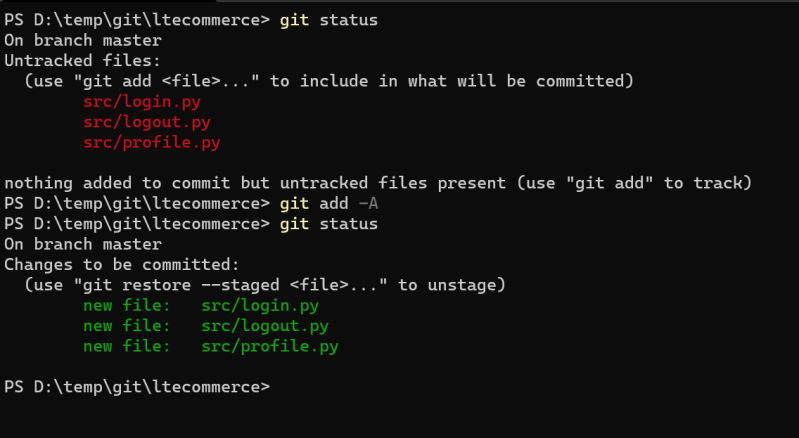
* Changes can removed from staging area by using reset command. Let’s remove the changes from some file in the staging area



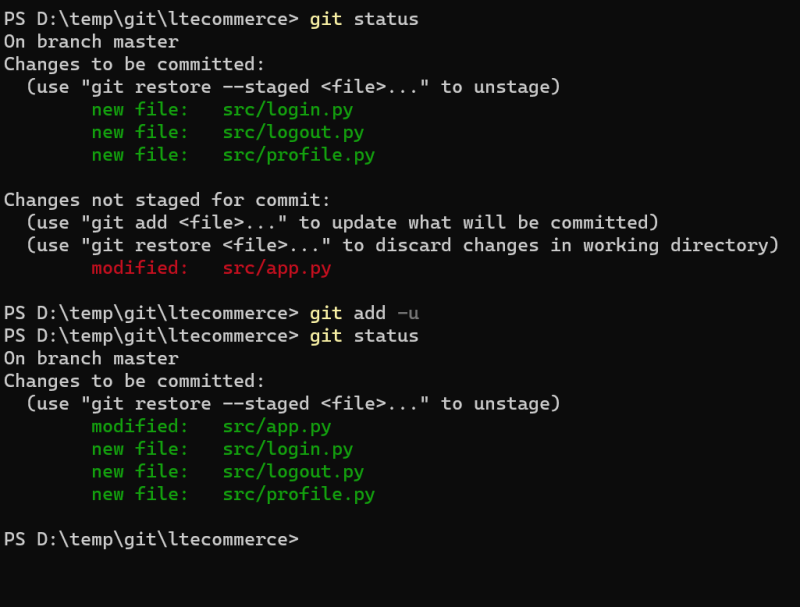
* Now commit the remaining changes in the staging area



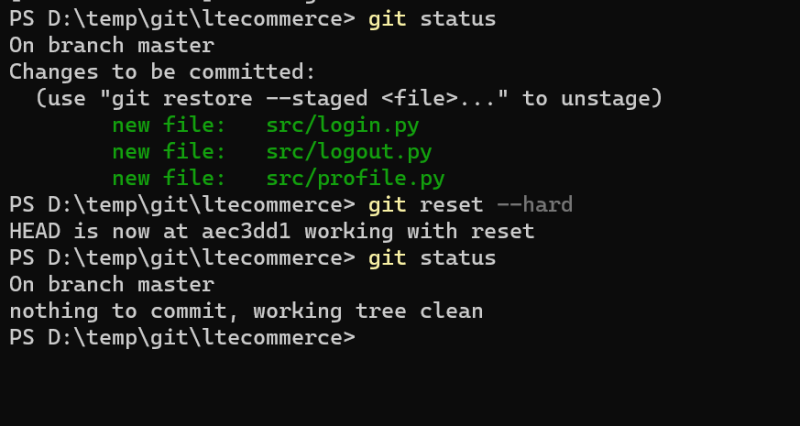
* git restore can be used to remove the modified changes from the working directory
* Let’s add 3 new files and add them to staging area.



Now add one change in existing file and add it to staging area



* “git reset –hard” can be used to move the changes from staging area to working tree and clean the working tree as if no changes have been done.

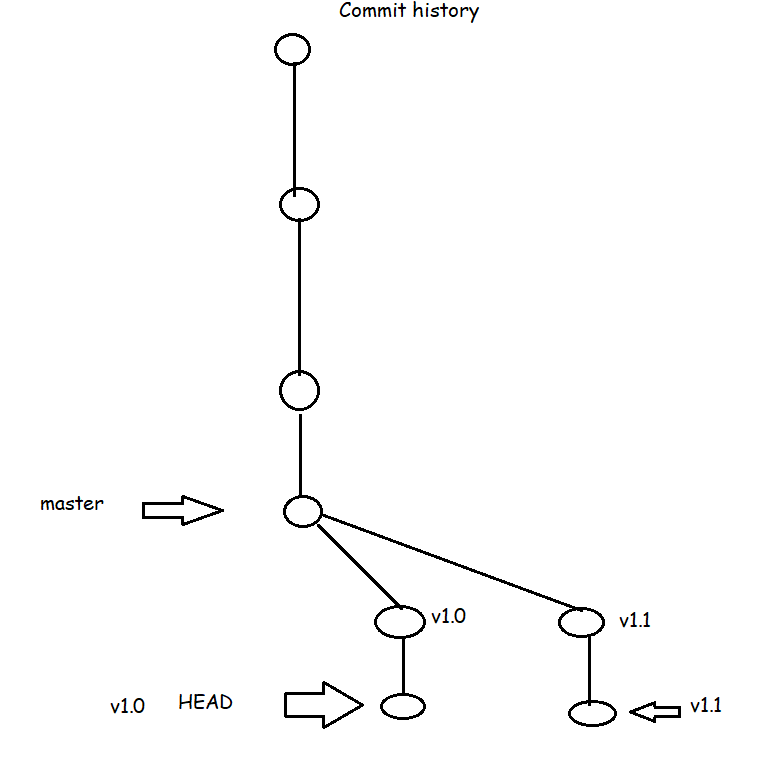


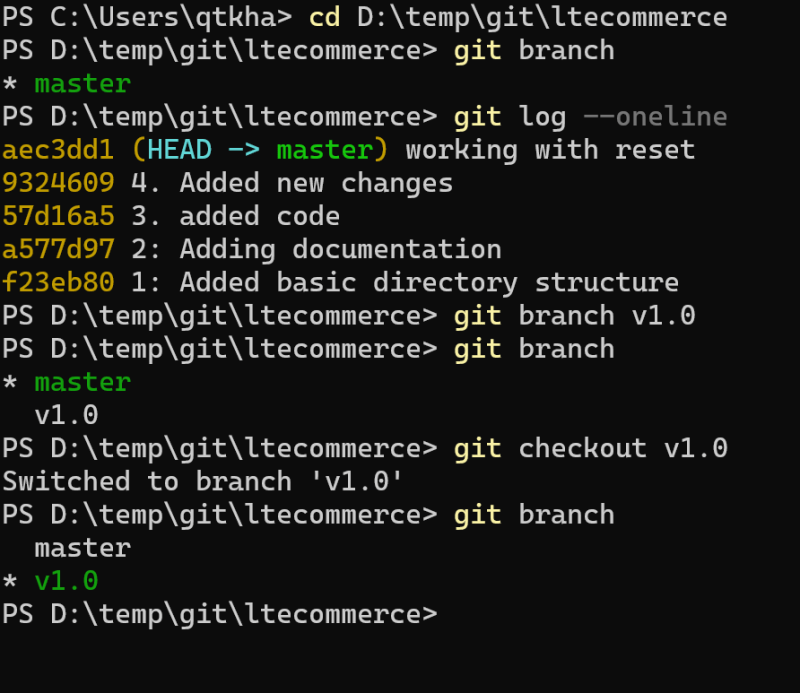
* In git there is one more command called git reset --soft find out what is does?
* To reset back the file and directories from local repository to staging area

=========================================

CI/CD-Git-5 **03/Jul/2021**

## Creating Branches and Traverse HEAD across Branches:

* We have to make two different releases of the application
  + v1.0 => New Customers
  + v1.1 => Existing Customer (Tesla) has some requirements
* Git supports branching. In git by default we have a branch called as master.
* On beside of branch name if we have “\*”, it means HEAD and cursor position
* Git branches are effectively a pointer to the snapshot of chain
* In Development of applications, we generally come across situations which need parallel development of releases/features, we use git branching for that



* Branch creation commands

git branch <branch-name>

git checkout -b <branch-name>

* To move from one branch to other

git checkout

head is always look for latest working branch or commit

git checkout cid(to change head from nranch to commit)

Inside of .git folder it contains HEAD folder. HEAD is a pointer towards what your current directory showing.

Open HEAD folder (it says in which branch we are in) ex: ref: refs/Head/v2.0

Now go to ref folder and open it. In refs we can see heads folders. In that folder we can see total branches in our system.

Commit ID’s are provided by following checksum technology. If there is any minor change also it generate new commit ID. Otherwise if the data is same the commit won’t change. This will be absorb while we use git status command.

To practice checksum (<https://tools4noobs.com>) online hash calculator.

* **Git objects:** 
  + Two major object types in git are
    - **Blob**: represents a file (giving information about files)
    - **Tree**: represents a directory/folder
  + We have the following object types
    - **Commit**: Commit is set of changes submitted along with date time, message, author name and email.

To check mode of file **$git cat-file -p (23d6214)cid**

To check type file **$git cat-file –t cid (**it tells file type like blob or tree**)**

====================================================

CI/CD-Git-6 04/Jul/2021

**How Git Works**

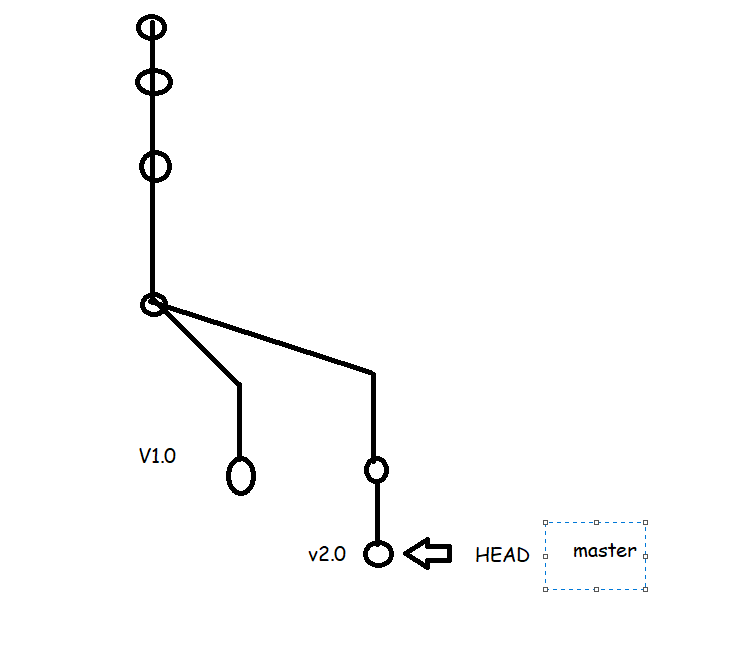
(How Git Works Internally and Understanding Merging FastForwarding Concepts):

* HEAD file in .git folder represents the branch. It should ideally point towards the branch. If HEAD is pointing to some commit rather than branch it is called as DETACHED HEAD state
* Use the class room recording to understand the git traversal across commits to populate working directory
* git cat-file -t cid (to see the type of file)
* git cat-file –p cid (to see the parent cid or contents of commit)
* git cat-file -t firstcommit cid (gives documents information)

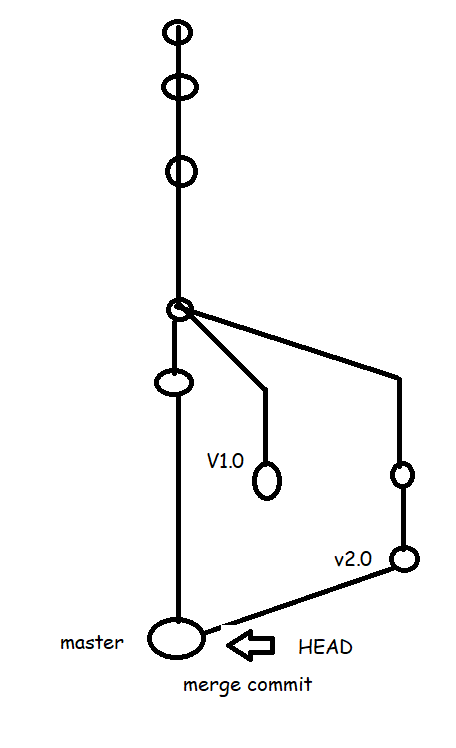
The only option to de-code the content is “cat-file –p” command

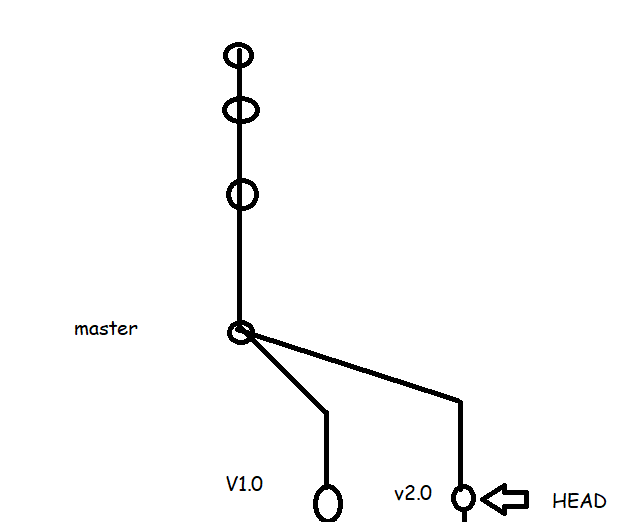
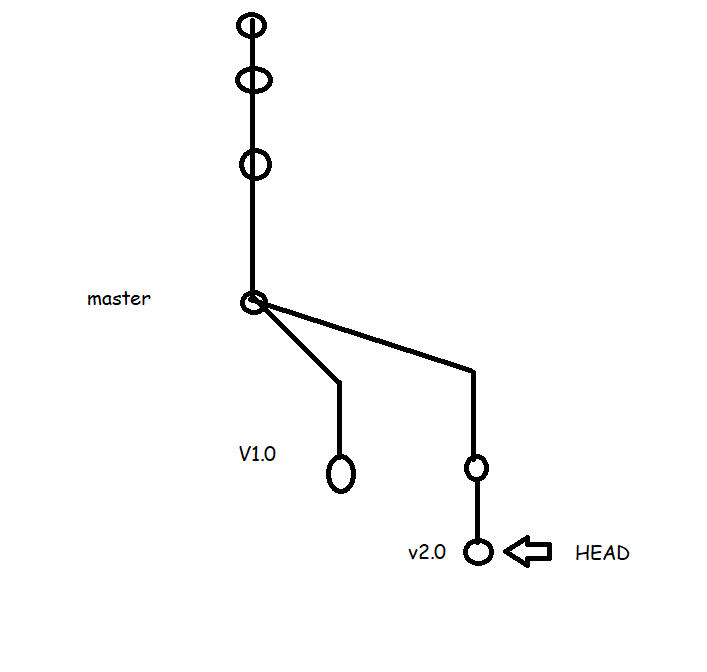
Merging Branches:

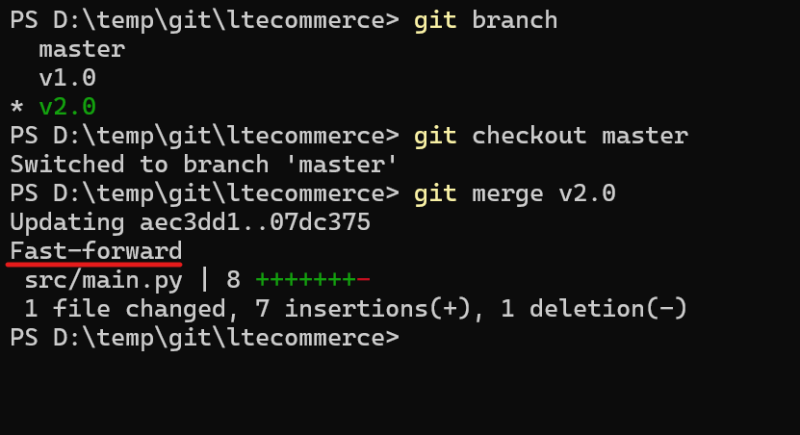
* Fast-Forward: When branch moves to the latest commit of branch which we are merging from i.e. considered as fast forward



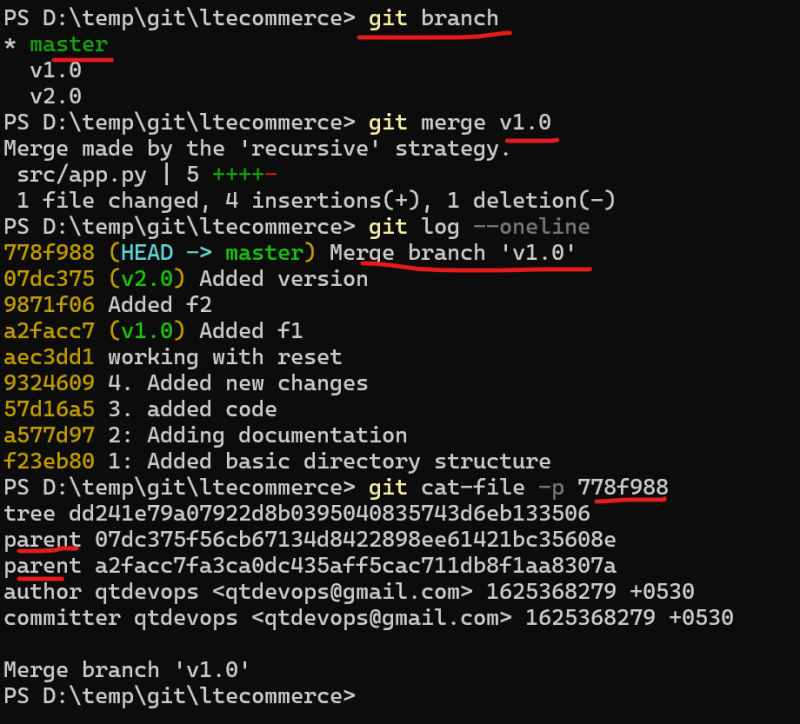
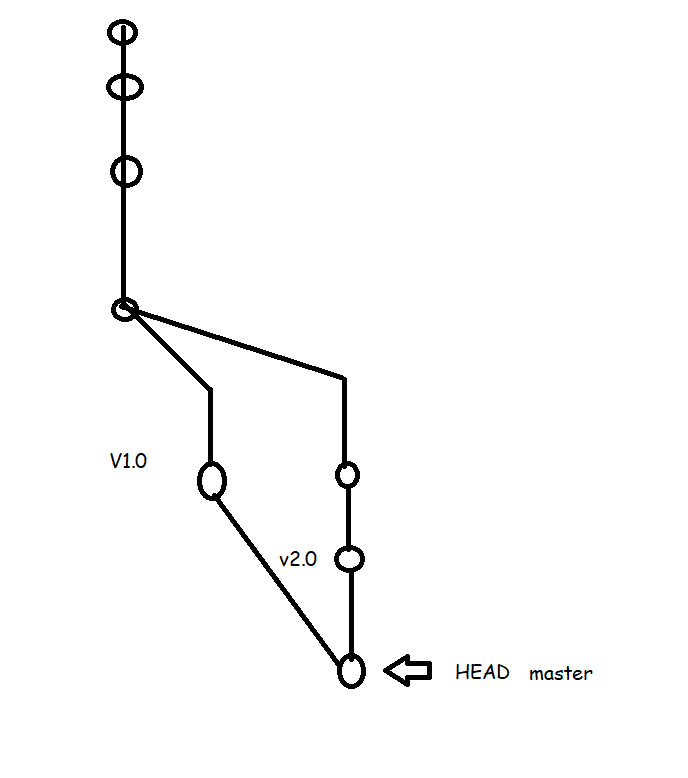
* Merge: This creates an extra commit on the destination branch which has merge commit (which has two parents)



* Let’s assume we have the branch which looks as shown below
* 
* Now let’s assume the user has created one more commit
* 
* Now we need to have whatever changes which we have done on v2.0 branch in master branch also
* Steps for merging:
  + Checkout to the branch where we need to merge  git checkout <dest-branch>
  + Now execute the command git merge <source-branch>



* Now Contents of V1.0 branch needs to be merge to master branch

* **Next Steps:**
  + Searching for history and differences in command line
  + Merge Conflicts
  + Cherry-pick
  + Re-base
  + Add the Remote Repository

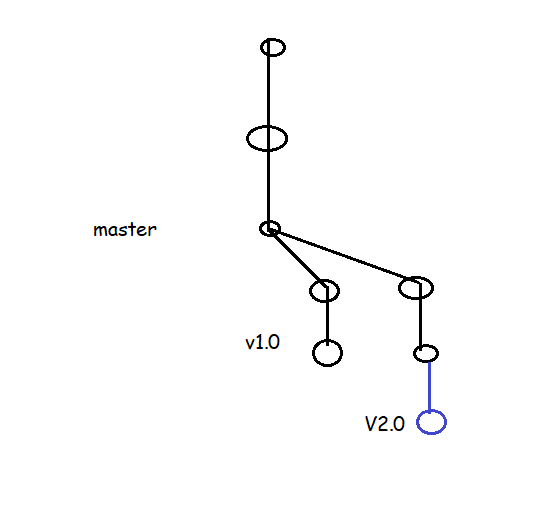
# ==================================================================

CI/CD-Git-7 06/Jul/2021

**Cherry-pick**

To open visual studio code in current directory= **code** .

* As shown in the below image we would like to have one commit (in blue) created in branch v2.0 to be added to v1.0 as it is important fix.



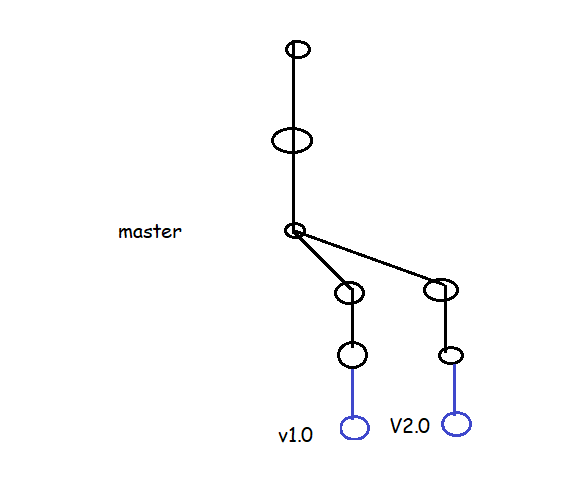
To move one individual commit from one branch to another branch we use cherry-pick command. Here we need to add last commit of v2.0 to v1.0. Then first we need to checkout to that v1.0 branch, later we need to cherry-pick that last commit of v2.0. Here we might be to get a chance of conflict problem, if we get conflict here then remove those arrow marked lines by vim editor.

Then $git status

$git add filename

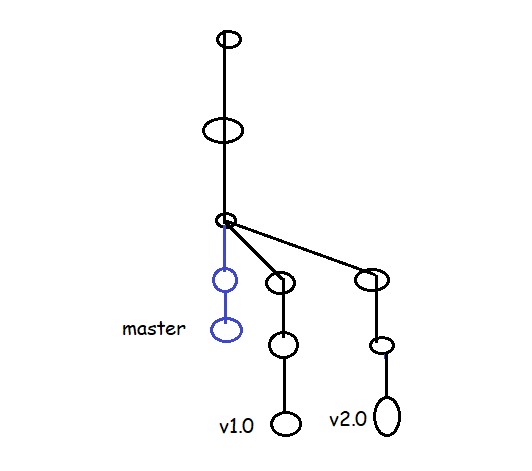
$git commit (no need to give -m and label here)

* **Post Cherry-pick**



## Rebase

* As shown in the below image we have made changes in the master branch and we want those changes in the v1.0 and v2.0 branches



What are the commits we are done in master branch, if we want those commits in v1.o,v2.0 or other branches , we can do in it by two ways

1. cherry-pick

2. rebase

1. By following cherry-pick we can bring only one commit at a time. It takes lot of time if we have to bring number of commits.

2. rebase: as shown in above image we can do by following rebase command.

Made changes in master branch, later

$git checkout v1.0 || v2.0

$git log --oneline || ‘

$git rebase master ||

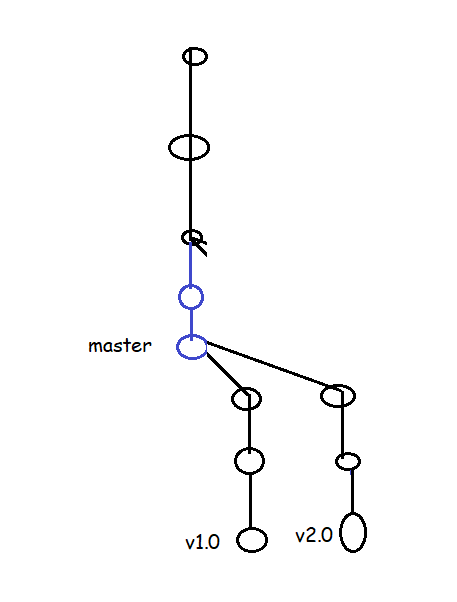
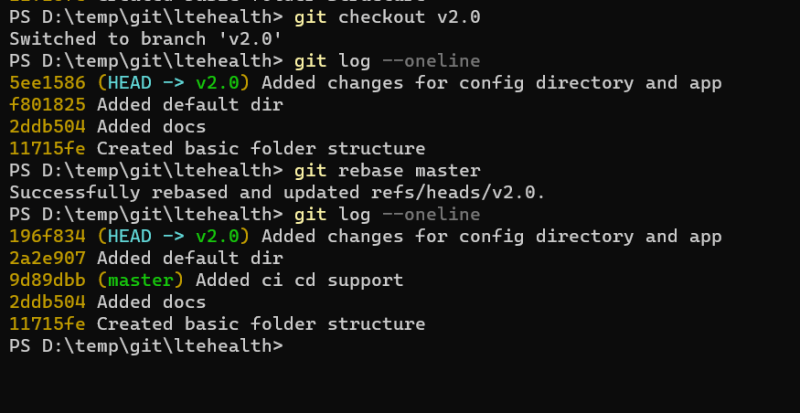
$git log --oneline ||

Meanwhile if we got conflict errors, edit files by vim

$git add .

$git rebase --continue

* **After rebase**

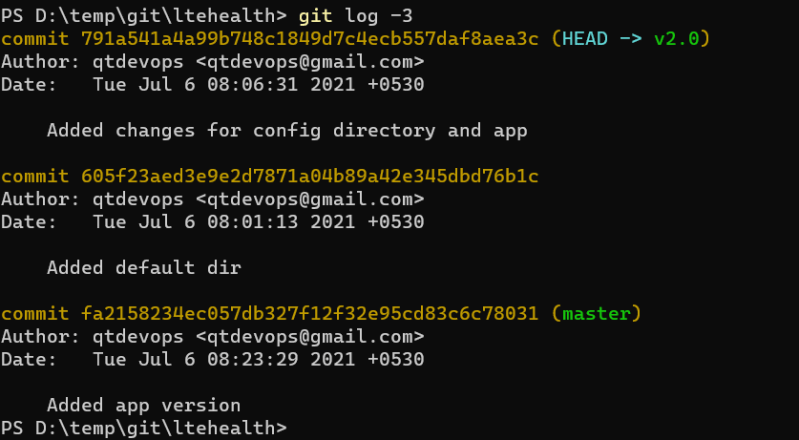
 

## Some Useful Commands

* View history of last n commits

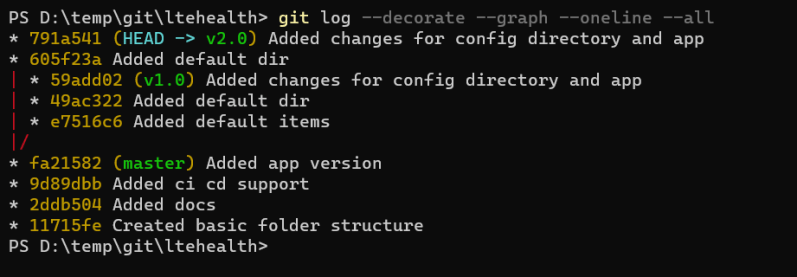
git log -n

git log -3



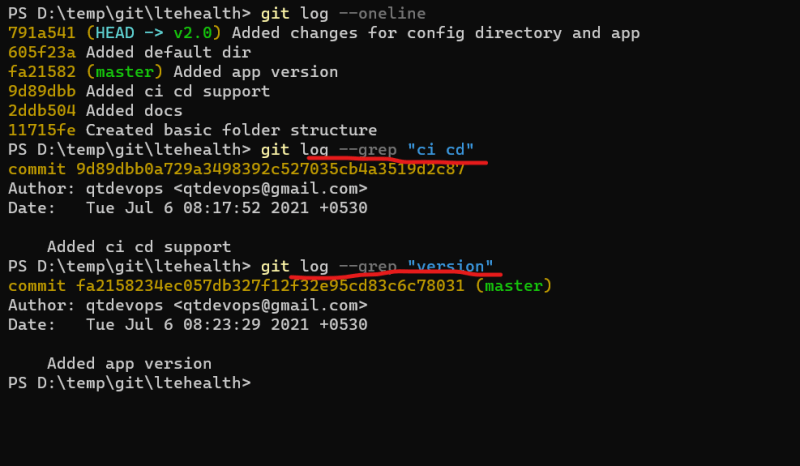
* View the graph

git log --decorate --graph --oneline --all



* Search commit by message

git log --grep "expression"



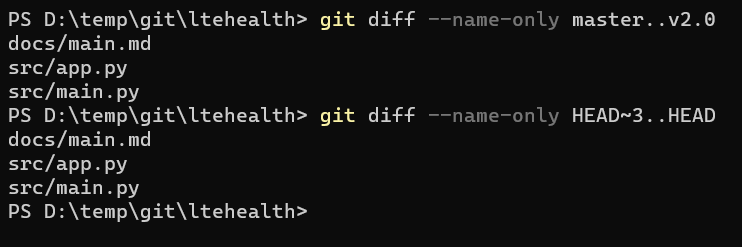
* Viewing the differences b/w any two commits

git diff <commitA>..<commitB>

git diff <branchA>..<branchB>

git diff --name-only <commitA>..<commitB>

git diff --name-only <branchA>..<branchB>

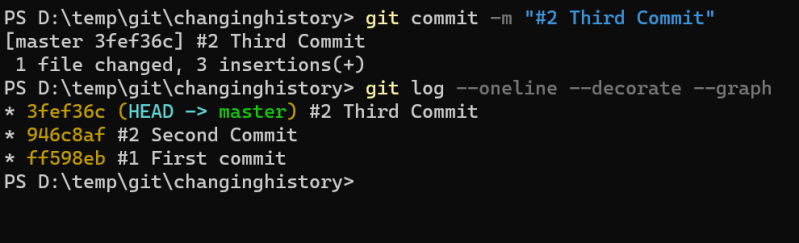


===============================================

CI/CD-Git-8 07/Jul/2021

## Making changes in commits

* Changing the commit message of the immediate commit which head is looking at

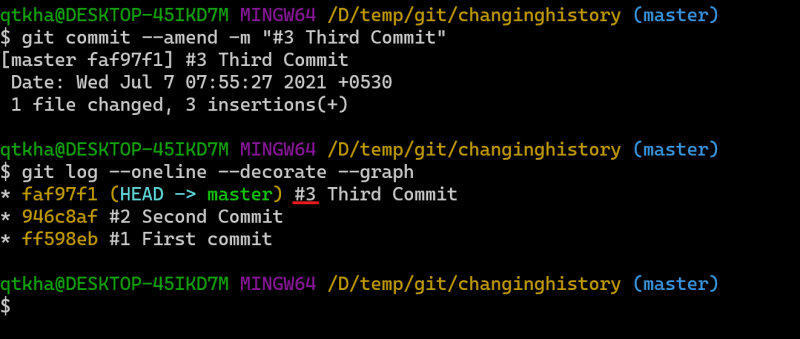


AMEND: To change commit message or label message we use amend but it was works for only latest labels only. –amend

$git commit --amend -m “label” (to change the label message or commit message)

$git commit --amend -m “label” (to commit new files into already existed commit id)

$git show cid (to see how many file in our cid)



\*The Git commit is used to commit the changes from staging area to local git repo

\*git commit --amend can be used to change the message of the immediate commit

\* We can also perform operations such as:

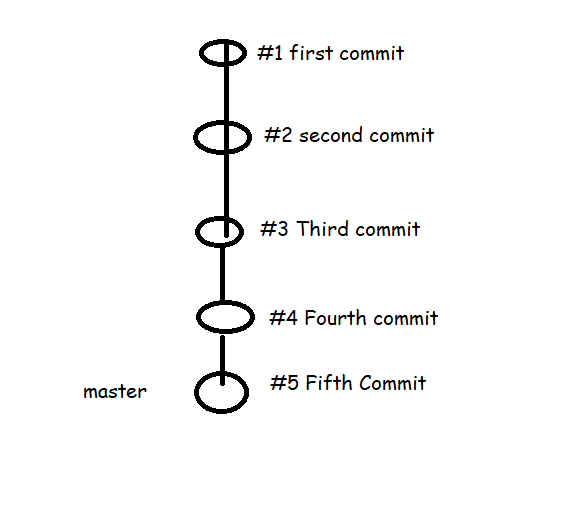
\*deleting commits

\*changing commit messages

\* combining commit messages

\*For the above mentioned changes we need to do interacting rebasing

* Change the commit messages way back in history



* Understanding the commits and changes;
  + Now let’s use the pattern #0001 from #1 i.e single digit to four digits
* git rebase -i --root
* git rebase -i HEAD~2
* Delete the commit in the history. Let’s delete the second commit

git rebase -i HEAD~4

drop commit 2

* Combining two commits into one commit
  + Here we use squash option

git rebase -i HEAD~2

pick third commit

squash fourth commit

pick fifth commit

## Working with Remote Repository

## Working with Remote git

## 

## To get clone the remote repo into local repo

$ git remote add origin https://github.com/naresh808/ltecommerce.git

$ git push -u origin master

## INIT: init means we don’t have repository and we need to create a repository.

## Clone: means we have repository and need to add that repository into central repository.

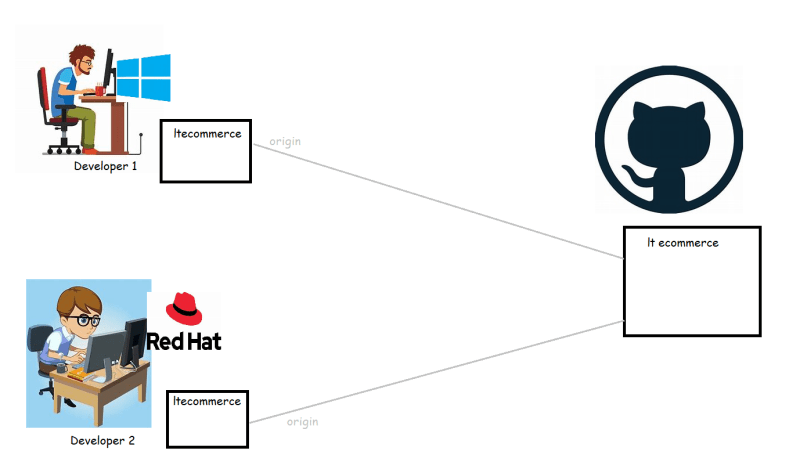
## 

============= ================================

CI/CD-Git-9 08/Jul/2021

## Configuring and Working with Remote Git Repository and Demonstrates How Two Developer’s of a Team Work Together with Git Simultaneously

* Scenario: We will be simulating two developers working on git

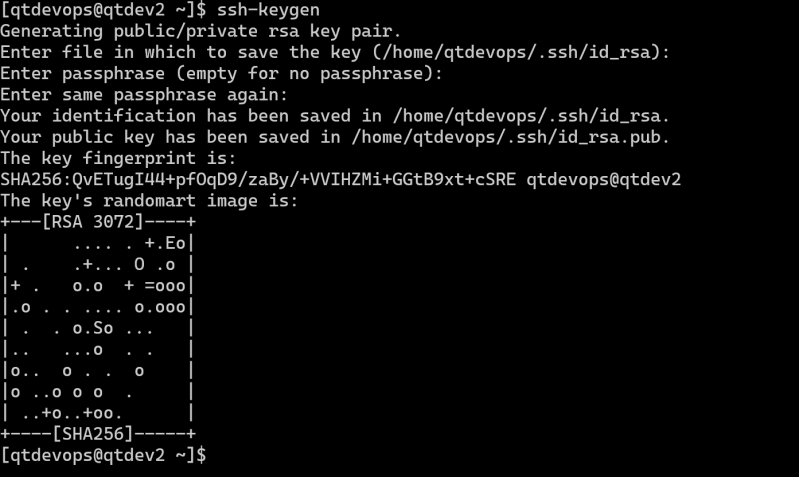


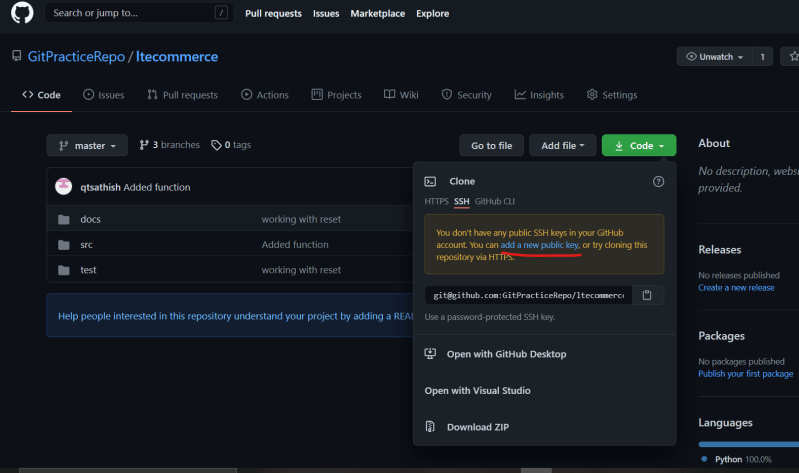
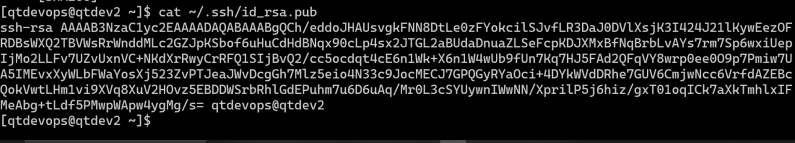
* Configure username and email for the git users in Linux and windows

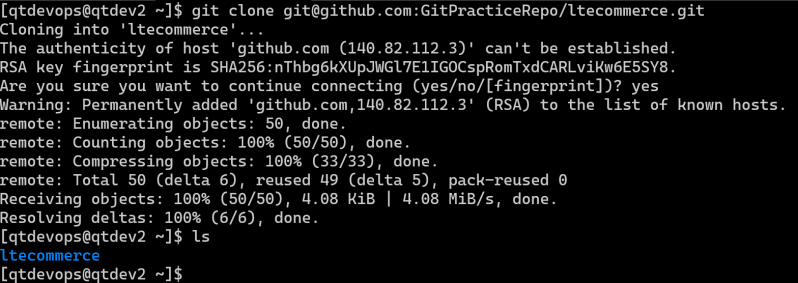
git config --global user.name "your-username"

git config --global user.email "your-email"

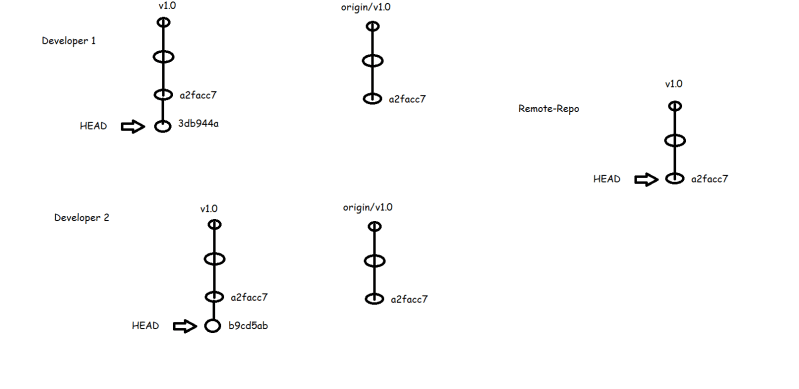
* On the Linux machine let’s use ssh to connect to git, for that we need to configure ssh keys in the remote repository
  + create ssh-key pair
* ssh-keygen

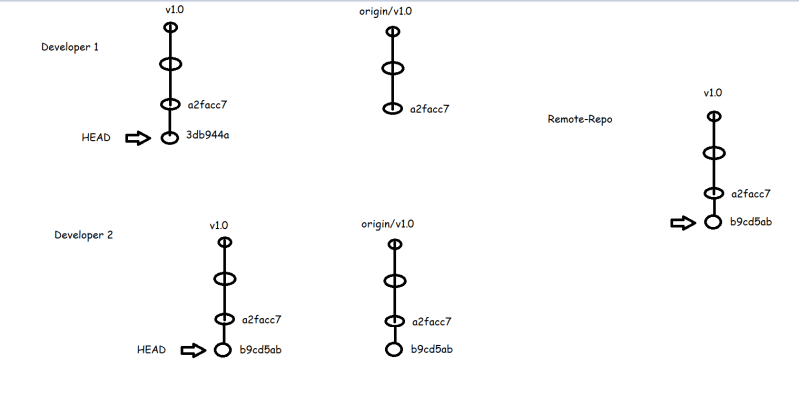


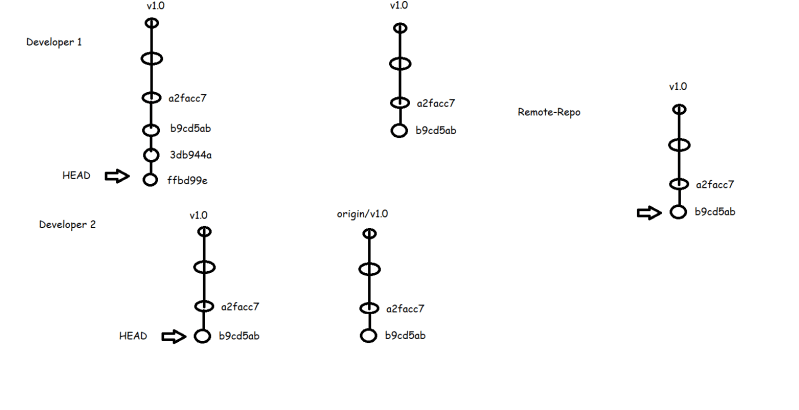
* + Copy the public ssh-key to github ssh keys 
  + clone the repo on the Linux machine for developer 2



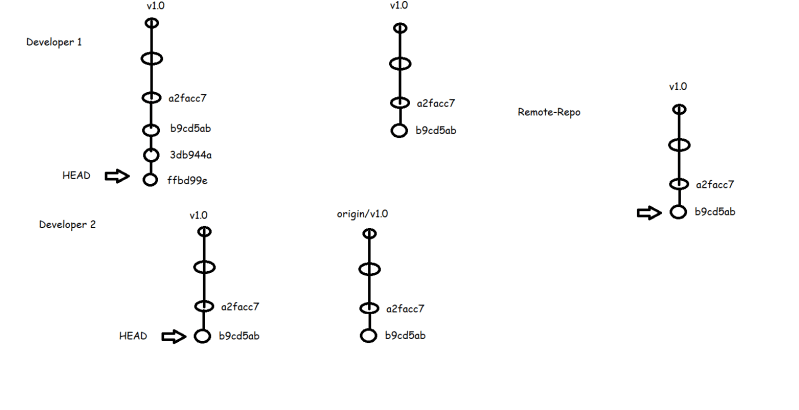
* Developer 1 starts working on a feature f1001 in the v1.0 branch
  + commit the changes of developer 1
* Developer 2 also starts working on feature f1002 in the v1.0 branch
  + commit the changes of developer 2
* As of now the state of branches after developers have done local changes



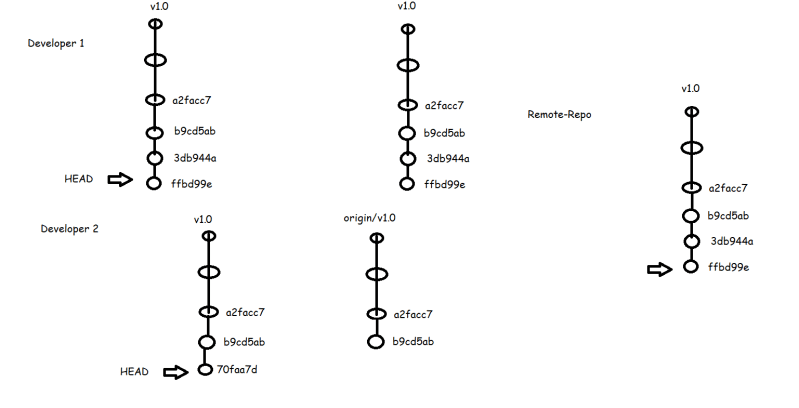
* Now Developer 2 has pushed the changes to the remote repository 
* In Git when we want to push the changes the remote-repository branch (which looks at latest commit ) the origin/branch should exactly match with the remote repository branch
* Now Developer 1 wants to push the changes, this leads to error as origin/v1.0 is not matching the remote branch
* 
* Now Developer 1 updates the local repo by using pull command, fixes the merge



* Now Developer 1 can push his changes

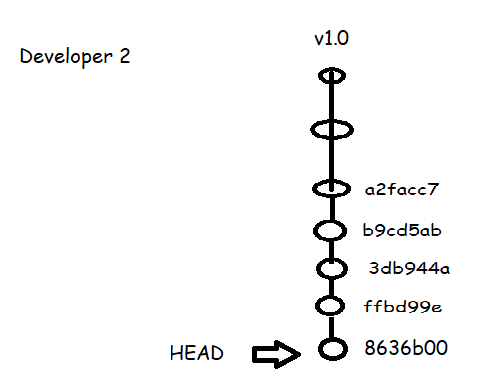


* Now Developer 2 starts working on feature f1003

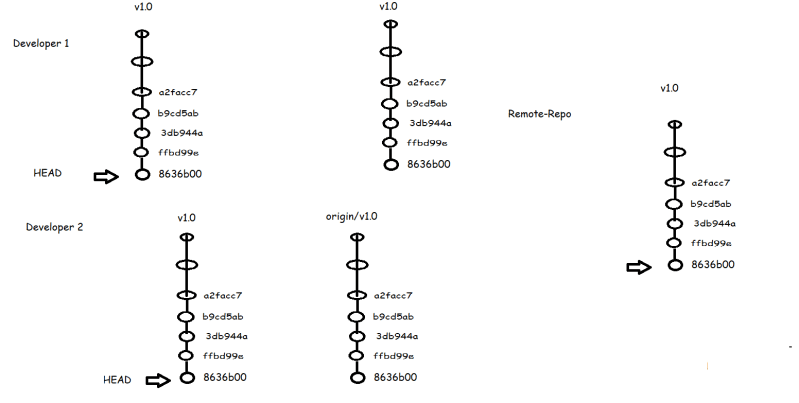


* When developer 2 pushes the changes he will get error as he needs to pull the changes. Developer 2 wants to pull the changes but he does not want a new merge commit to be created.

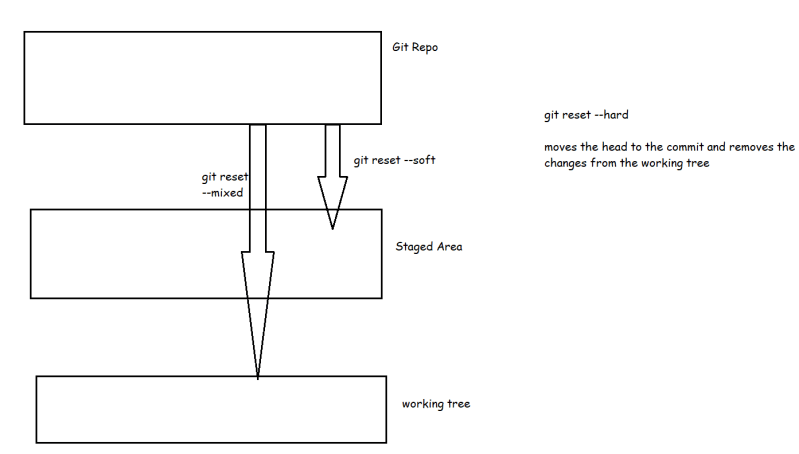
git pull --rebase



* Now both developer 1 can get the changes using git pull

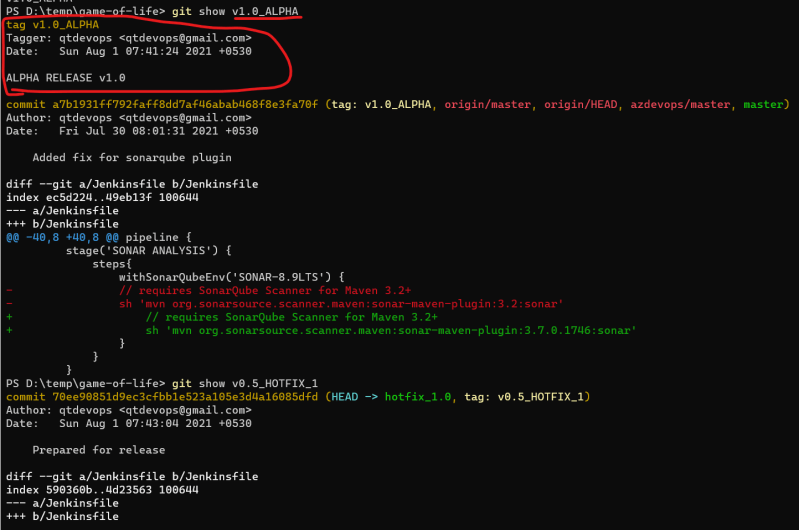
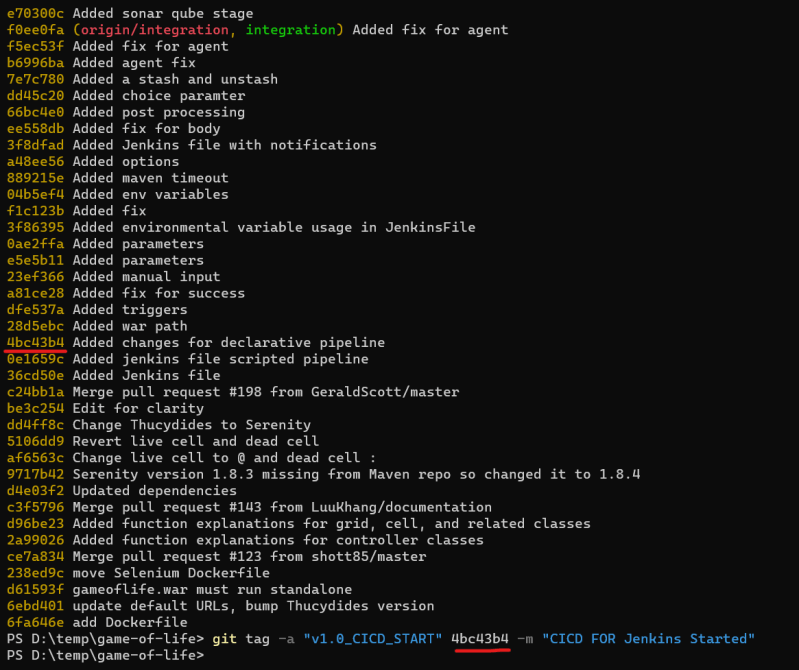
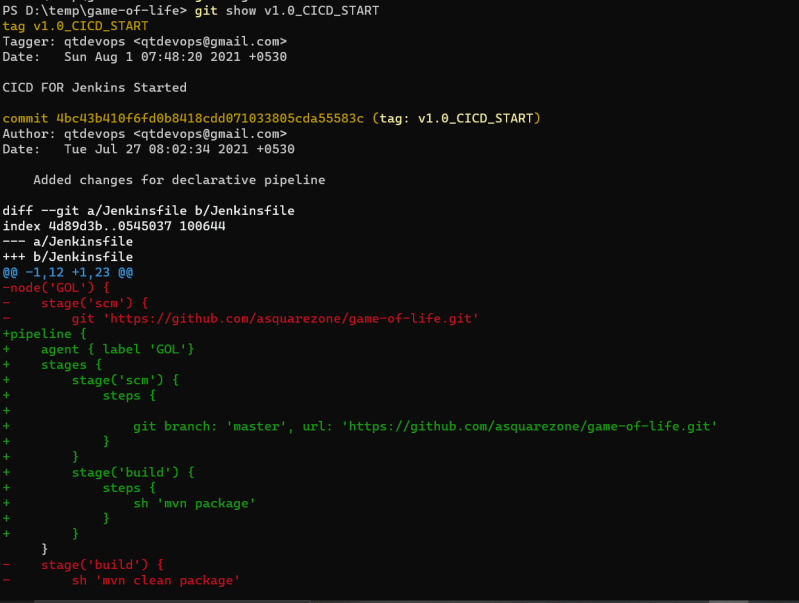
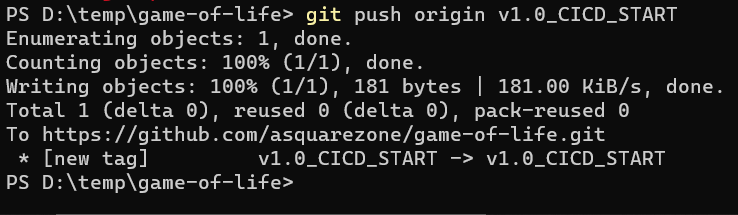
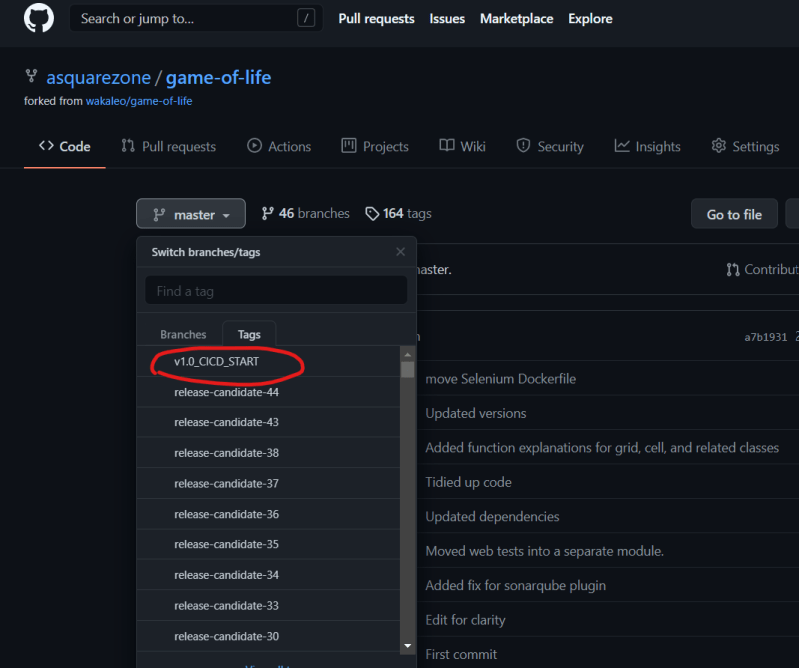


## Git Reset Hard vs mixed vs soft



CI/CD-Git-10 01/August/2021

## Git Tag

* Tags are ref’s that point to specific points (commits) in the Gi History.
* Tagging is generally used to capture a point in history i.e. used for marked version release.
* A tag is like a branch that doesn’t change, Unlike brnaches tags after being created have no further history of commits
* Creating a tag:
  + We can create two types of tags:
    - annotated: They are store as complete objects in Git database (local repo). They are checksum, require a message and store other important data such as name, email and date
  + git tag -a "<tag-name>" -m "<message>"
    - lightweight: They don’t require a message and they don’t store other data rather they are just a pointer to specific commit
  + git tag "<tag-name>"
  + To view Tags data use git show <tag-name>
  + 
  + Tag older commits
  +  
  + Pushing Tags git push <remote-name> <tag-name>
  + 
  + 
  + To push multiple tags at once use –tags
  + To delete the tag git tag --delete <tagname>

## Day Builds and Night Builds

* Day Builds refer to builds done when developers are working and day builds should have ci/cd pipeline which gives feedback to the developer whether their commit is working correctly or not
* Night Builds refer to combination of all the work done by your dev team and this build will be taken by QA to perform Testing. We have only one Night Build per day. On the Night build we tend to run all the tests (unit, integration, load) so QA team will have confidence to test further.

## Branching Strategy

* In this session we will be learning one braching strategy referred as GitFlow [Refer Here](https://directdevops.blog/2019/01/04/best-branching-strategy-git-flow/) (<https://directdevops.blog/2019/01/04/best-branching-strategy-git-flow/>)

## Git Stash

* Git stash temporarily shelves or stashes changes you have made in the working tree, so that you can work on some thing else
* To Stash your work: After you have made changes

git stash

* Re applying your stashed changes

git stash apply # apply the changes but the changes will still be present in stash list

git stash pop # apply the changes and removes the stashed changes from stash list

* Git stash will stash
  + Changes that have beend added to staged area, changes made to files that are currently tracked.
* Git stash will not stash
  + new files (untracked) in the working copy and files that have been ignored.
* When we have multiple stash items we can apply/pop any item from stash list

git stash pop --index n

* If you want to drop a stash git stash drop

## Git Bare Repo

* Generally the Remote Repositories are hosted on the servers where they donot need working tree. All the changes to the Remote Repostiories will be pushed
* Git has a way of cloning git repository where only .git will be available this way of cloning is referred as bare clone.

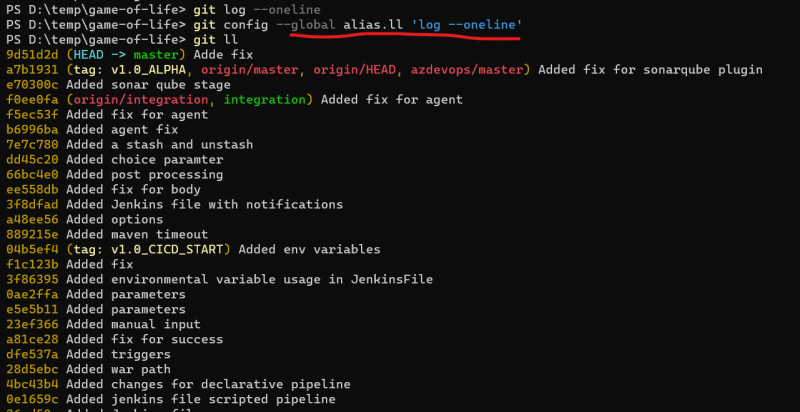
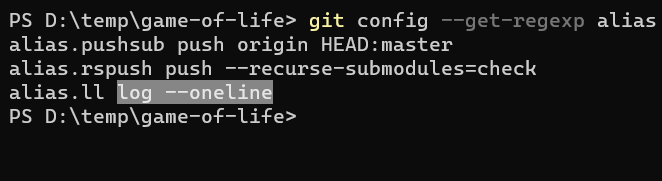
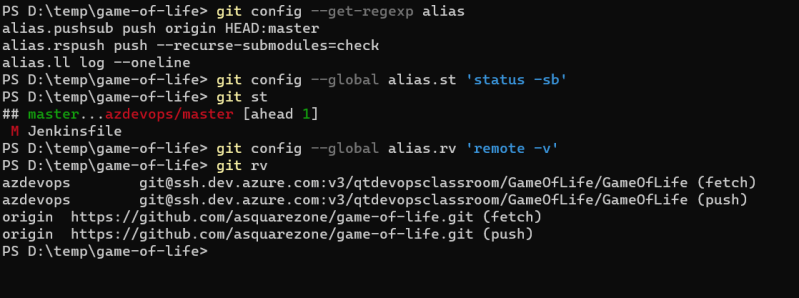
git clone --bare <git url>

## Git Mirror Repo

* When we want to have backup of the repository where all the extended references to remot repository are also made available

git clone --mirror <git url>

## Git Aliases

* Git gives us a capabiliyt to create alias commands so that we can save typing the whole command
* 
*  

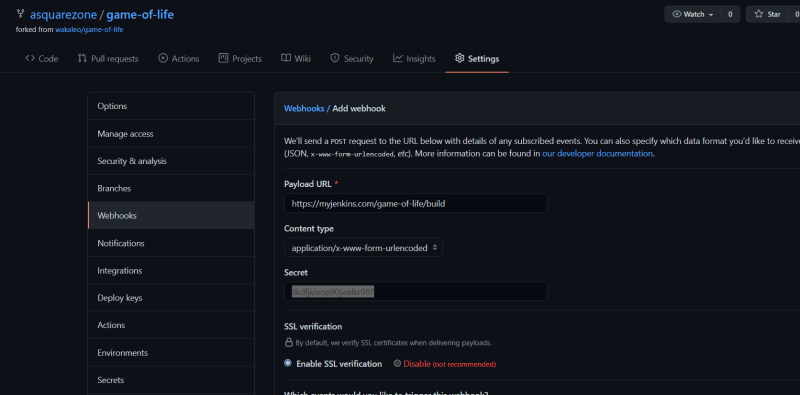
## Git Submodules

* If you want to have a git repository with in other git repository [Refer Here](https://github.blog/2016-02-01-working-with-submodules/) (<https://github.blog/2016-02-01-working-with-submodules/>)

## Git Hooks

* Git Hooks are scripts that run automatically every time a particular event occurs in git repository
* In .git folder we have hooks folder
* Local Hooks
  + pre-commit
  + prepare-commit-msg
  + commit-msg
  + post-commit
  + post-checkout
  + pre-rebase
* Server-Side Hooks
  + pre-recieve
  + update
  + post-recieve
* [Refer Here](https://www.atlassian.com/git/tutorials/git-hooks) (<https://www.atlassian.com/git/tutorials/git-hooks>)
* When we have our own git server and then we can use hooks to add some kind of behavior or scripts to certain

## Web Hooks

* For the cloud based git repositories like github/bitbucket etc we cannot control by writing Server Side hooks using .git\hooks folder so these cloud providers offer Web Hooks 
* [Refer Here](https://docs.github.com/en/developers/webhooks-and-events/webhooks/about-webhooks) (<https://docs.github.com/en/developers/webhooks-and-events/webhooks/about-webhooks>)

### CI/CD-Git-11

# 01/Aug/2021

## Pull Request

* [Refer Here](https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests) for understanding the pull requests(<https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests>)

## CICD Pipelines

* As discussed in the class create nodes with deployment tools installed and change the Jenkinsfile to have a stage for CD or create a new job and in build triggers configure the deployment job as downstream.
* From jenkins we just need to add an sh step as shown below. Refer stage(Ansible) from below

pipeline {

agent { label 'GOL'}

triggers {

cron('H \* \* \* \*')

pollSCM('\* \* \* \* \*')

}

parameters {

string(name: 'BRANCH', defaultValue: 'master', description: 'Branch to build' )

choice(name: 'GOAL', choices: ['package', 'clean package', 'install'], description: 'maven goals')

}

options {

timeout(time: 1, unit: 'HOURS')

retry(2)

}

environment {

CI\_ENV = 'DEV'

}

stages {

stage('scm') {

environment {

DUMMY = 'FUN'

}

steps {

mail subject: 'BUILD Started '+env.BUILD\_ID, to: 'devops@qt.com', from: 'jenkins@qt.com', body: 'EMPTY BODY'

git branch: "${params.BRANCH}", url: 'https://github.com/asquarezone/game-of-life.git'

//input message: 'Continue to next stage? ', submitter: 'qtaws,qtazure'

echo env.CI\_ENV

echo env.DUMMY

}

}

stage('build') {

steps {

echo env.GIT\_URL

timeout(time:10, unit: 'MINUTES') {

sh "mvn ${params.GOAL}"

}

}

}

stage('Ansible') {

agent { label 'ANSIBLE'}

steps{

// requires SonarQube Scanner for Maven 3.2+

sh 'cd deployment && ansible-playbook -i hosts deploy.yaml'

}

}

}

post {

success {

archive '\*\*/gameoflife.war'

junit '\*\*/TEST-\*.xml'

mail subject: 'BUILD Completed Successfully '+env.BUILD\_ID, to: 'devops@qt.com', from: 'jenkins@qt.com', body: 'EMPTY BODY'

}

failure {

mail subject: 'BUILD Failed '+env.BUILD\_ID+'URL is '+env.BUILD\_URL, to: 'devops@qt.com', from: 'jenkins@qt.com', body: 'EMPTY BODY'

}

always {

echo "Finished"

}

changed {

echo "Changed"

}

unstable {

mail subject: 'BUILD Unstable '+env.BUILD\_ID+'URL is '+env.BUILD\_URL, to: 'devops@qt.com', from: 'jenkins@qt.com', body: 'EMPTY BODY'

// for unstable

}

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*end\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*