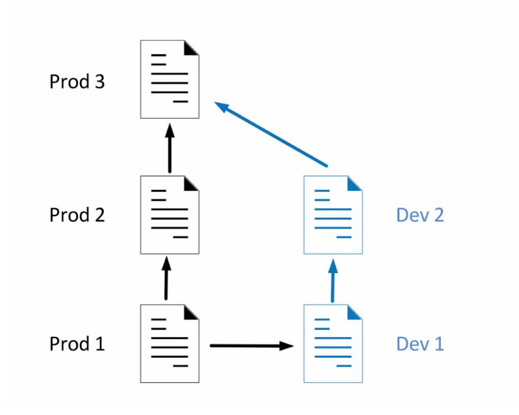
Introduction to Git Crore Topics:

Central Git Concepts:

A Git is a kind of VCS (version control system) that helps us document any modification to our files.Git works well specially with text files.

Tasks that can be done using Git:

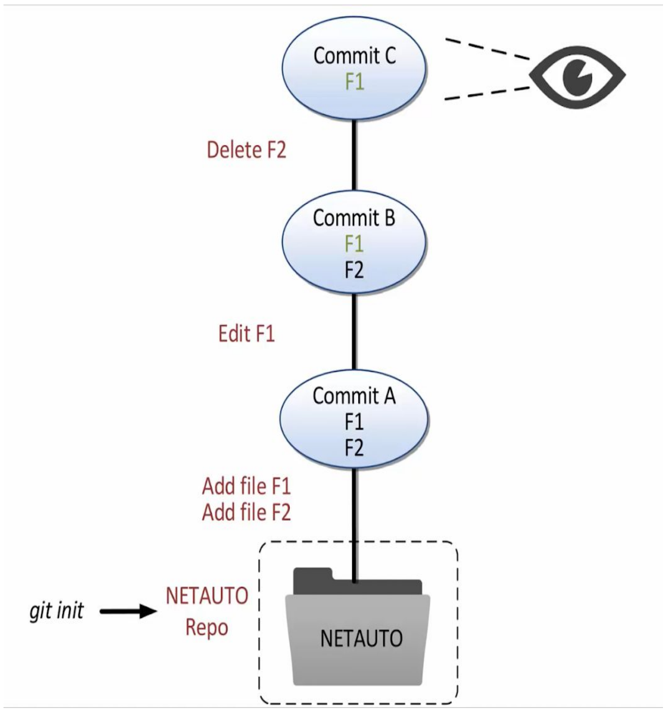
- Take shots of files
- Brings back previous versions of files we made earlier
- Work on multiple versions of files simultaneously for example working on a document and keeping a fresh copy aside and work on another of the same kind. Ultimately when we feel ready, we can merge them into the same file like in the image below.



Two basic diagrams:

- Git commit graph
  - Enables us to take snap shots of files at any time. These Snap-shots are called commits.
  - In a repository we start by using the git init command.
  - Let's say we added two files F1 and F2 to our directory and then we decided to take our first snapshot .
  - So this is a saved version of the file we are working on and we can always go back to it.
  - Suppose then we edit F1 and then we take another snapshot
  - For example then we decide to delete F2 and we do commit to update the version
  - In this way we can update our versions using comment and restore it back whenever we want. This is very useful for a large projects.

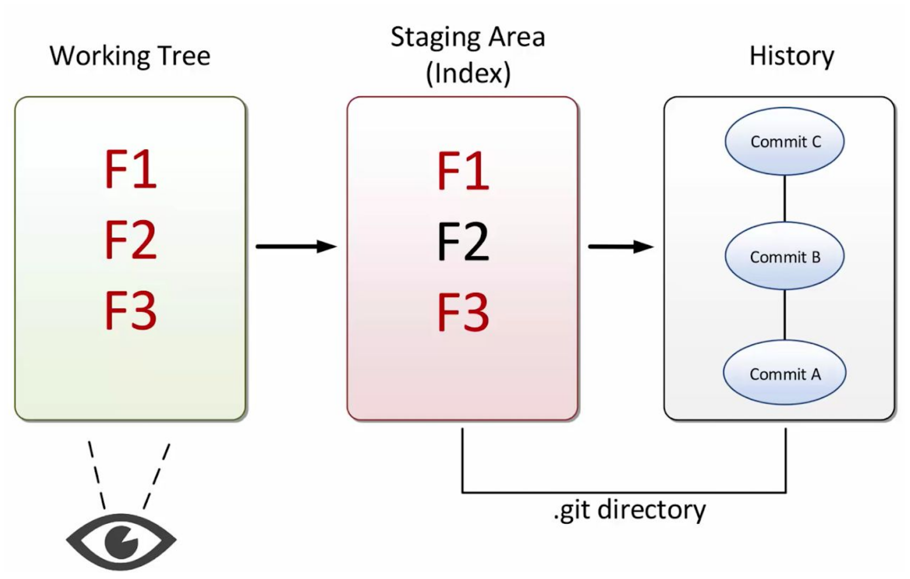
Commit Graph:



A commit saves the state of the file at a particular point in time.Usually we make a commit when a logical unit of work is done.

- Three conceptual areas for files:
  - There are three areas, The working tree, the staging area or index and the history.
  - When we add delete or edit files we do that in the working tree
  - They get history is similar to the commit graph we did above.This history is kept in a hidden directory .git. It contains our object database and metadata that make our repo. If we happen to send our .git directory to someone else then that person would have complete access to our full git project and its history including all versions of the file and commits.
  - As we are working on a project we make changes in the working tree. Git Gives us full control over what changes from our working tree we are putting into our next commit. For example if we edit three files F1 F2 and F3 files how is the ever we only want the new versions from those files to be taken snapshots of for our next commit.This control is through staging area
  - We could add the two files we want to the staging area and when it is right we make a commit. The file that was left out can be included in our next/future commits.

3 area diagram:



Hands-on - create a repo:

After successful installation of the gate documentation, we start with a standard directly on our file system
The first go to a new directory and add a file name ,say S1 . For this we can use any text editor like Emacs, vi etc.The command will be ***vi S1***. Then add some text data to indicate few variables for the network switch S1. Save it and exit. Then we have a single file S1 in our new directory. It can hold a git project with the ***git init*** command.

According to the diagram if we see in the working tree we have our first file S1. After running the git init command, It creates a .git sub directory in our main directory.

Before this it is important to configure our username and email whenever we make a commit includes name ,email and a timestamp with a commit. It is used to track if any changes are made to the project and Are made by which member of the project.

For this we do,
 — global user.name "My name"
— global user.email "My@some-email"
 git config —list
This ***git config —list*** shows us our name and email that we just set. As we used —global flag with the commands, our name and email will be used in future and we would not have to repeat it.

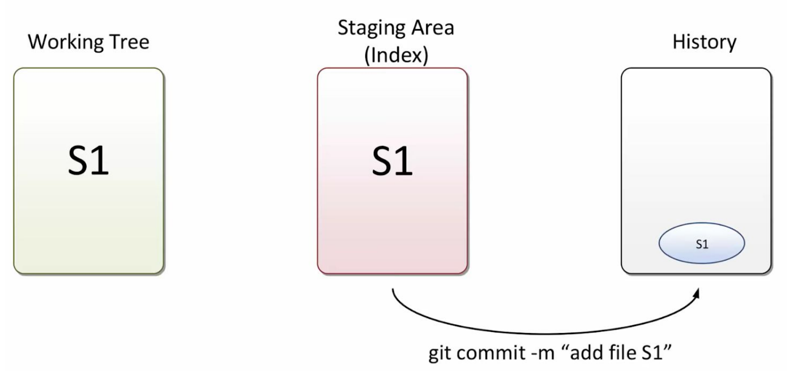
Coming back to the diagram,As we know that git tracks changes to files over time but for now it is not tracking S1. Soon after we add S1 to our staging area it will be tracked.

Use:
git status

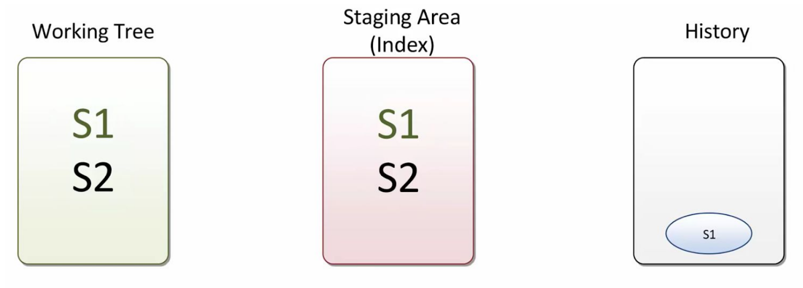
Which will Tell us how things stand in our working tree and in our staging area. We also notice that we are on branch master. It also guide us how to get S1 into the staging area. We stage S1 with a git add command

git add S1

After this command it moved as one into the staging area as desired.We can check it by running its we can check it by running ***git status*** command. The “changes to be committed” indicates that we are in the staging area.
Now git is tracking S1.
It’s time to make our first commit using:
git commit -m “add file S1”
This command creates a commit with whatever is in the staging area. The –m gives a short message as a description of what has been changed.
If we check again by ***git status*** command we can see that the working directory is clean which means there is nothing new in our main directory. Every commit has a unique SHA -1 hash.

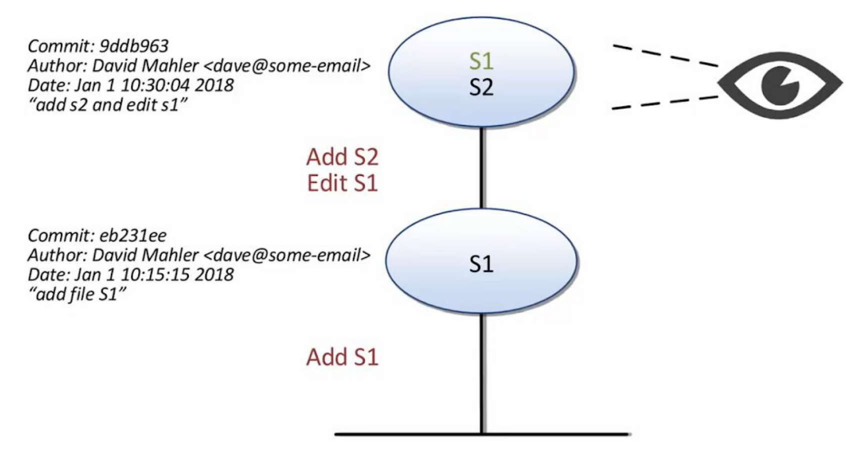


Similarly we work on a second file S2 using the same steps.After adding it in the working tree we will see that S2 is on track but then S1 is being tried since we previously committed it. We can confirm it by running get status command.
A git diff command shows the difference between tracked files in the working area and the Staging area
To add git files we could do
git add S1 S2
Alternatively we could also do
git add .
The doubts indicate adding all the new and modified files to our staging area.
To double check it we can always use ***git status*** command.



This looks good, so let us commit using

git commit -m "add file S2 and edit S1"
Now we notice that we got a unique hash for the 2nd commit as well


Commit: 9ddb963
Author: David Mahler <dave@some-email>
Date: Jan 1 10:30:04 2018
"add s2 and edit s1"
S1
S2
Add S2
Edit S1
Commit: eb231ee
Author: David Mahler <dave@some-email>
Date: Jan 1 10:15:15 2018
"add file S1"
S1
Add S1

We notice that our most recent comment is at the top and the previous one is below that.

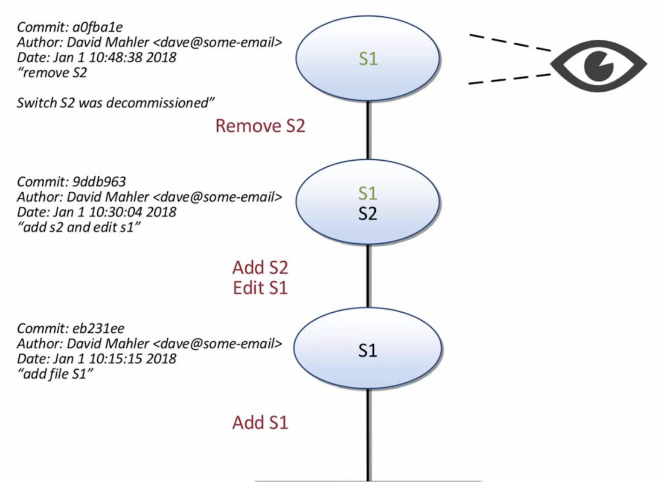
How to remove a file:

git rm command helps us remove.
We can remove S2 with ***git rm S2***

This command does two things at once:

- It removed S2 from our working tree
- And also staged this removal therefore S2 is removed from the staging area as well

We can then make a commit and check. We will notice that our last commit will have S2 removed from our directory in our commit graph.


Commit: a0fba1e
Author: David Mahler <dave@some-email>
Date: Jan 1 10:48:38 2018
"remove S2
Switch S2 was decommissioned"
S1
Remove S2
Commit: 9ddb963
Author: David Mahler <dave@some-email>
Date: Jan 1 10:30:04 2018
"add s2 and edit s1"
S1
S2
Add S2
Edit S1
Commit: eb231ee
Author: David Mahler <dave@some-email>
Date: Jan 1 10:15:15 2018
"add file S1"
S1
Add S1

git log can be used to see our recent commits.

How to Undo a working tree change:

- Lets make some changes to our S1

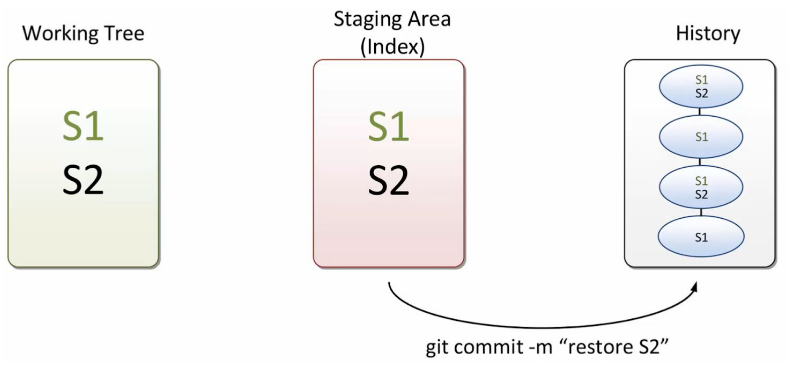
- Then when we look at our 3 area diagram, we notice that S1 has changed in the working tree but not yet updated in the staging area
- ***git checkout --S1***
- This will bring back S1 to how it was previously.
- We can double check using ***git diff*** command which will show that there are no modifications as our working tree is clean.

How to Undo staging of files:

- Edit S1 and stage the change
- ***git diff*** shows our changes
- Perform ***git add S1*** and ***git diff*** will not show anything which means our working tree and staging area match.
- Our last commit does not have the snapshot of our S1 so we can use ***git diff --stage*** that will show us the difference between the staging area and our latest commit.
- Then we can do ***git status*** which will indicate us the latest changes made which can be now committed.
- The ***git reset HEAD S1*** will help us unstage S1.
- We can restore the working tree by using ***git checkout --S1*** command.

How to restore from an earlier commit:

- As S2 is not in our working tree or staging area anymore, let us try getting S2 back from commit before it was deleted.
- We can notice commits affecting our file by using ***git log -- S2*** . We can notice from our commit messages of where we added S2.
- Use ***git checkout commit hash -- S2***
- We can check by doing ***ls***
- Now make a commit git ***commit -m "restore S2"***



.gitignore

- There are git files which we do not need.
- For say there are files like *logs/* and *myapp.pyc* which we want git to ignore

- Use ***.gitignore***
- Use ****.pyc***
- When you check by running ***git status*** , we will notice that it is no more *logs/* and *myapp.pyc* but there is a ***.gitignore file***
- Use ***git add.***
- And ***git commit -m "add .gitignore file"***

Introduction to Git - Branching and Merging

Branch

Branch will allow us to work in the different versions of a file parallely. Our work is independent and we can determine if we want to merge our work into other branches. Separate versions of the same file are functions of the branch. We can branch for different purposes.

Branch Implementation

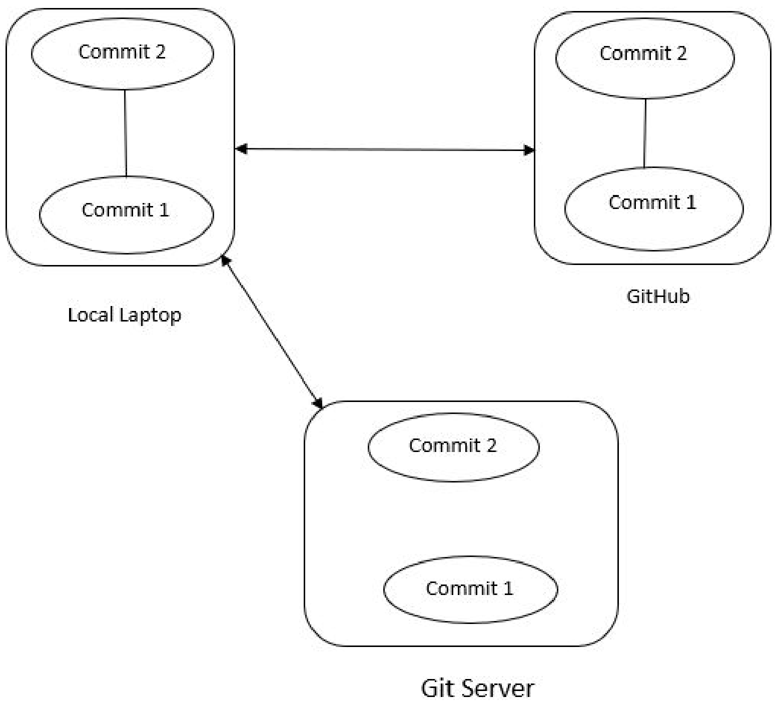
A commit has 40-hexadecimal sha-1 hash. Git will create a master branch by default. A branch is a pointer, and a branch will point to the sha-1 hash. When we are at our master branch every time we make a commit the branch will move up. Git will know what branch we are at by a pointer called head. Head points to a branch and it is called a symbolic pointer because it only points to a branch.

Creating Branch

We can make a new branch with the command "git branch <name of the branch>".

Git Remote

A remote is a repository in another location from where we are now working and has been connected to our local laptop. For example, we might be working on the project on our local laptop and the version of the project is on GitHub. In respect to our local laptop, GitHub is a remote repository. A remote can be in any location with Git repo under a different platform. Some of the git repo platforms are Git Server with a bar repository, copy of repo within GitHub, group project with many branches etc. We work interactively back and forth between our laptop and remote repository. When there is a change on the remote repository, we can download changes to get them locally. If we make changes to our local repository, we can upload those changes to the remote repository. The following is an example of interrelationship in Git repositories.



Create a repository in GitHub.

To create a GitHub repository, you need to create an account on GitHub. After that you can click start the project, and that will lead you to create a new repository window. On that window you can fill in the repository name, and description of the project. You need to make a repo public by checking the public button so that anyone can see it. Initialize the repository with a README and GitHub will create a file README.md. A README file can be used for the details of the repository for those who will read it. Then click create repository and it is created. GitHub made one commit automatically to add the REAME.md file to the repository.

Clone a GitHub Repository

To perform a clone, we need a proper URL. That can be found on the clone or download button. Click the drop-down button and you can see the clone with SSH and with Https options. When using the SSH option you need to put the SSH public key in GitHub. Copy the URL address to your laptop text editor or git Bash with git command. Git clone command downloaded an existing repository from a server to your local machine. For example,

```
phili@DESKTOP-GQPPAA8 MINGW64 ~
$ gitclone git@github.com:WillPeers/ENSE375-groupE.git
```

We cloned the above repository (ENSE375-groupE.git) from git@github.com on account of WillPeers. After the project has been cloned, we can change into our local directory, and start with the git config option command:

Git config –local user.name “phil”
Git config –local user .email paa180@uregina.ca

This command helps you to set up the username and email address which can be used with your commit file. When typing a graph on the command prompt it will show you the local master branch pointing to the single commit. In addition, there is a Head pointer point to master, and there is origin slash master and origin slash head. Origin slash master is a special branch that functions as a remote tracking branch. The local origin master branch tells us that our master branch and origin branch in GitHub are pointing to the same commit. The remote tracking branch on the other hand cannot be checked with git checkout origin/master. It will end up in a ‘detached HEAD’ state because it is not the same as the standard local branch. However, in normal conditions we do want to stay on standard local branches.

Git fetch and Git merge

Git fetch command is an important feature in GitHub. Its function is to extract information, or the latest update commits to the origin branch with its related objects. Developers used git fetch to see how the history has progressed. Git merge command is used to merge the file of a specific branch into the master branch. We can go back to our GitHub account and create a new file. Write in a file and commit it. The new file will be a second commit file in the repository. Then go back to the local machine and type in git status to see that the standard branch does not recognize it yet. When typing git fetch origin, the system reaches out to GitHub and finds the new commit and brings it back. After the repository has been updated, type in merge origin/master to merge commit reference by origin/master to local branch.

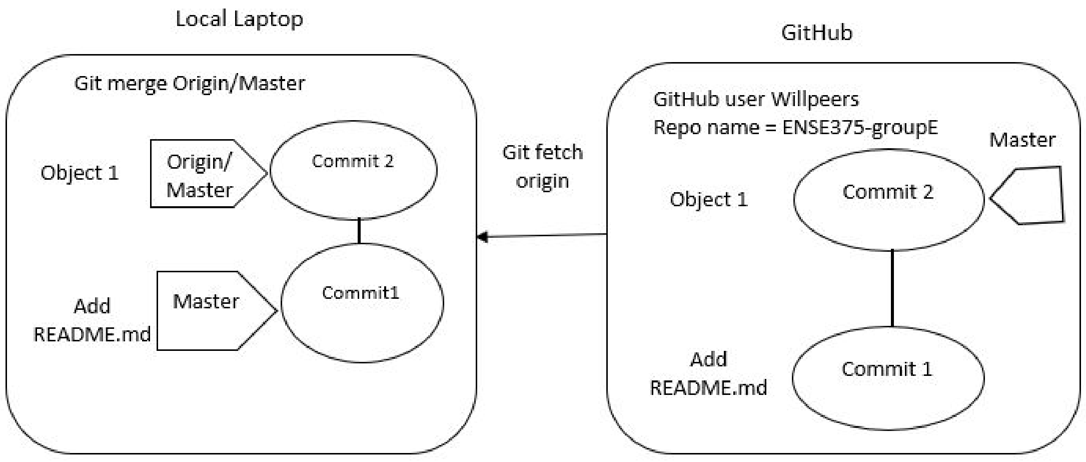


Fig. 2. Git fetch and Git merge

Git Push

Git push origin master command is used to push the contents of the local repository to the remote repository. This git push is relying on GitHub authentication which was setup on an SSH key. This happens when the file is committed and ready to be merged into the master branch. After the push command is done, the following changes will happen: commit will be updated, the latest message will show, and the latest commit hash will appear.

Create a GitHub Fork

A fork repository is a copy of a repository into another new master repo. On the same account for example, Willpeers account, you can click create fork and the window will open on your account. The name of the repository will be the same as ENSE375-groupE, but the account is different. The following flow diagram shows how fork repo is created.

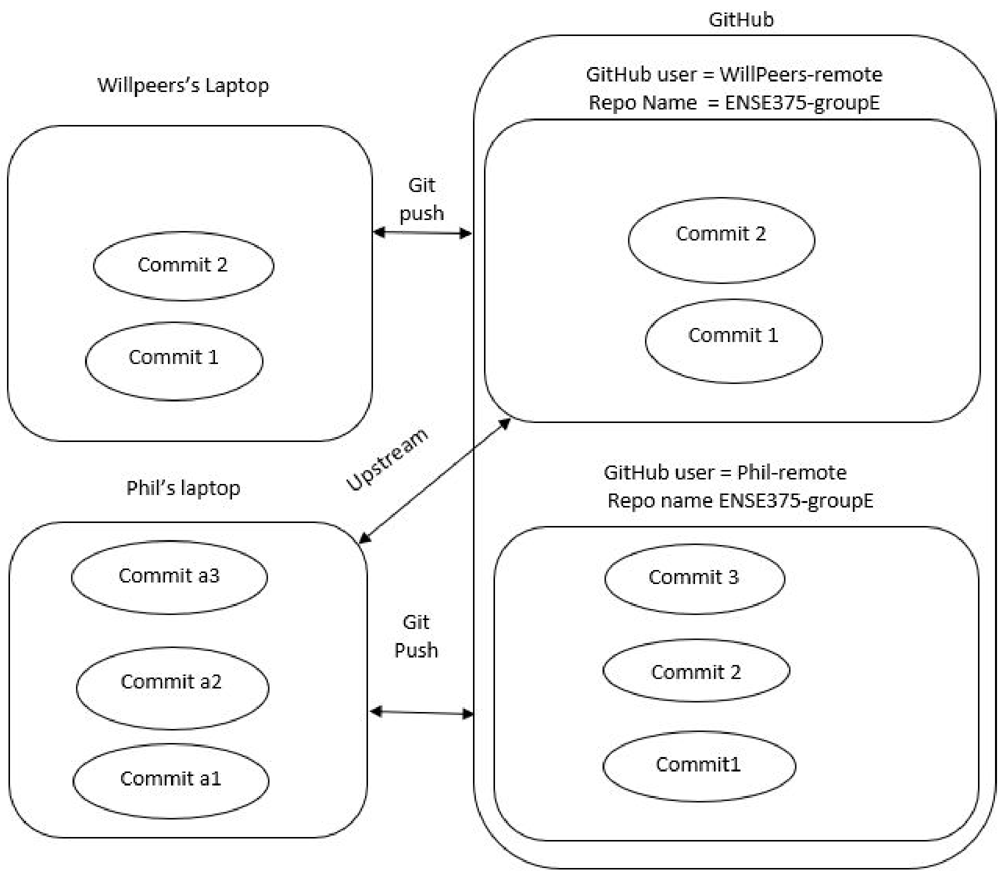


Fig. 3. GitHub Fork

6.5. The Repository, Index and Working Areas.

Working Area:

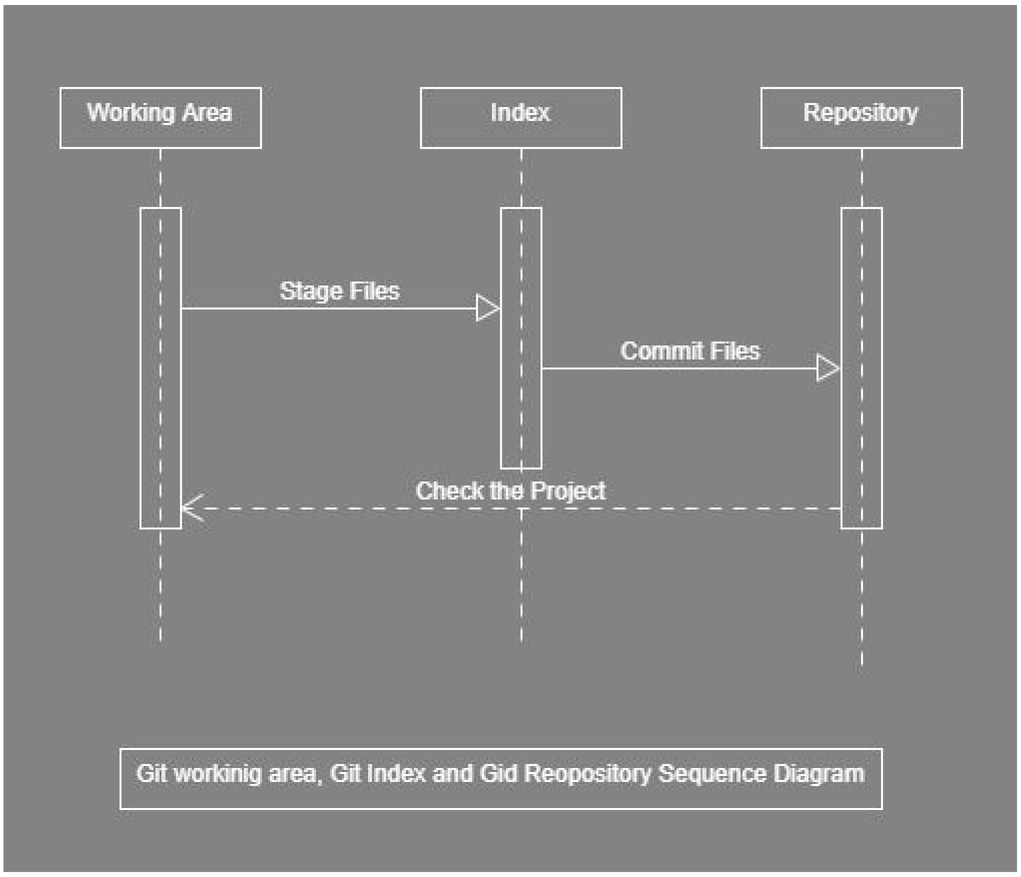
When starting the git repository, you can follow the procedures and type the commands to create a repository directory on the local machine. The following are some of the commands you need to create a git repository in the directory.
$mkdir testgit
$cd testgit
$git init

The git repository is created in the testigit directory, but it has nothing in it yet. At this point there are many options to take depend on what you want to do, for example, you can branch, tag, or do many other things using the git commands. To fill in git repository in testgit directory we need to create a .git subdirectory using git init. We could go further and create files and directories in .git subdirectory. The following are just the default subdirectories in .git.

- HEAD
- Config
- Description
- Hooks
- Reference

These subdirectories may have more extended subdirectories, and files in the local machine or working area. After work is done on the local machine, we can move to the staging area.

Index:
Git Index is another important file on this process. It is used to stage the work done on the local machine before committing it to the local repository. if there is specific change made on a file, it must be described in the commit message. You can use git command *git add -p* to specify a particular change on a file at staging area.
Repository:
We can move away from the staging area to the git repository by committing our files. The git checkout branch command executes branch check and moves the HEAD reference to branch reference. The git add files command executes the function of shaping processes and specified files committed from git index. The following sequence diagram summarized the transition in these three important steps.



6.6 The Object Database

Git object database or object directory is a container that stores all pointers to the local contents. The main four basic primitive objects in git are tree, blob, commit and tag. Each basic primitive object may have type, size, and content attributes. Every object in the git directory with no referral pointer can be collected by garbage collection, and the Git database is cleaned up.

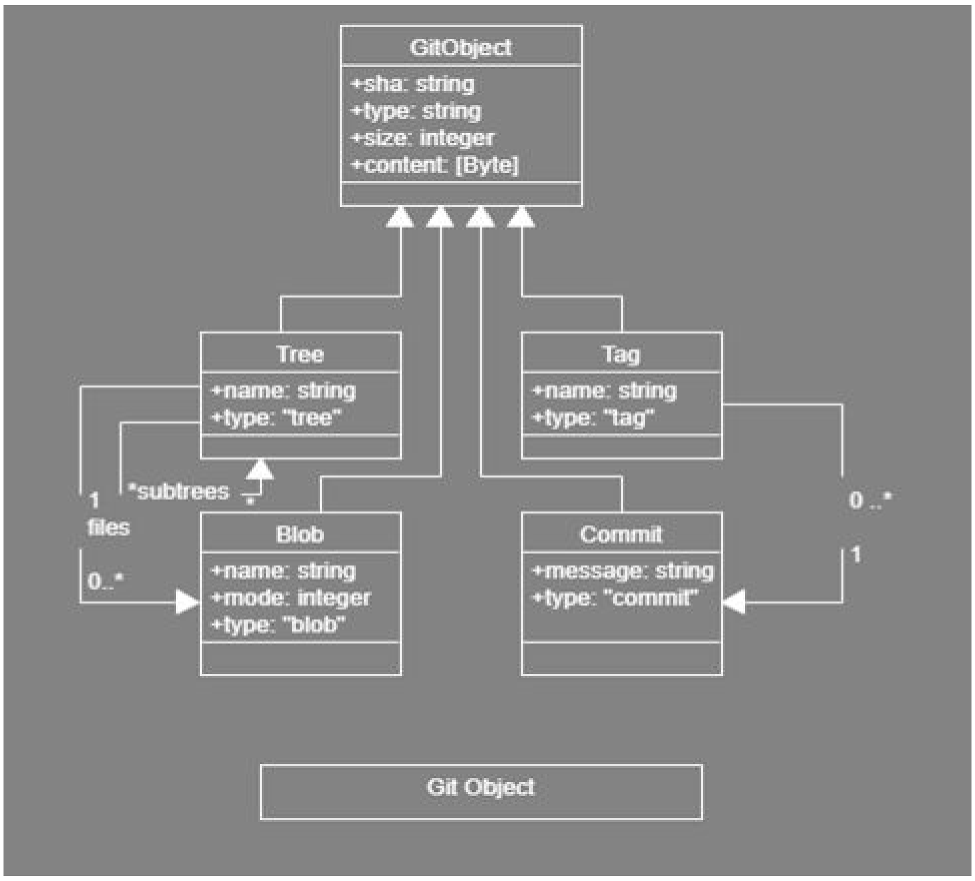
· Tree: It is an object type that is involved with the issue of storing the file name and allowing the group of files to be stored together. The store contained in git resembles that of a UNIX file system although it is not a complicated structure. Many contents are stored as tree or blob objects. A single tree may contain blobs or subtree and resemble another tree structure.

· Blob: Blob stands for "binary large object", and it is the first kind of object created by git when a file is committed. An example is a README.md file which is created and committed before any file in the repository. A blob always stores the information of the file. The directory creates the hash of that file name and stores it.

Commit: unlike tags that point to commit, commit point to the tree representing the higher-level directory. You can create a new commit to represent advance Head and the current index to point to a new branch. Commit object can be created by the git and point it to its tree object.

Tag: The tag object has some similarity to commit, except that it doesn't point to a tree but to commit. Tag objects contain tag name, object type, message, and tag pointer. The tag type contains the hash of the tag object. The two types of tags are annotated and lightweight. An annotated tag is a bit complicated then the light lightweight tag.

The SHA stands for "Secure Hash Algorithm". It is used to store the information required to represent repo history and is referenced by 40-digit object name. The importance of SHA is that git can determine the similarity of the two objects if they have the same SHA. Different SHAs mean not identical objects. These features are used in many ways, for example tracking of merging trees. Like other hashtags if the object has been changed the SHA can identify that changes. Gits uses this property to determine the change made to the file in the repository.



12.2.3 The Three Revlogs

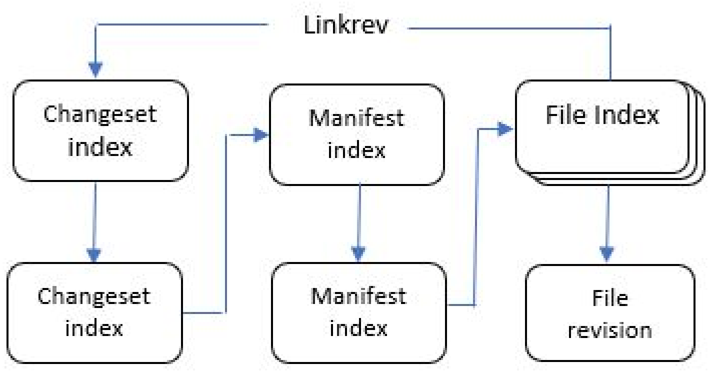
1. Changelog: A changelog is an updating of change in the record or the file of the project. An example of changelog is an update when a developer fixes a file or adds

a new feature to the program. Changelog in the Revlogs Structure contains metadata for each change in the file.

2. Manifests: It is a file that contains a list of filenames with the individual file's node id. It has a reference link to file revision in file log.

3. FileLogs: A FileLogs is a file that describes action occurring on the file. This file is contained in Mercurial's internal store directory.

For the Revlogs layers, and data structure examples, you can find them on the Architecture of Open-Source Application on "Mercurial(aosabook.org)". The Revlogs generic structure is as follow:



Log Structure

12. 2. 4 The Working Directory

Another important data structure is the working directory or dirstate. Dirstate is the reflection of what is contain in working directory. It keeps monitoring any change in the directory whether there should be comparisons from the status command or diff command. when the file is about to be merged, dirstat split into two parents, one set to merge and change it to another set. This is because the common operations in dirstat are status and diff commands.

Adding Collaborators

While working in group it is essential to have files that are in common to avoid file duplication and keeping track of work. Github Allows us to add collaborators in the repository such that everyone invited to the repository can manage the files and access the files as per their need. To add collaborator, **open the repository** you want to share >> go to **Settings** >> click on **Manage access** >> click on **invite a collaborator** >> **enter the username** you want to invite. That person has now been invited and the owner can also cancel the invite request.