Git Basics

Agenda:

1. To understand the important Git concepts and terminology.
2. To understand the changes in the status of the files present in the Git repository during the execution of the Git commands.
3. Hands-on:
    - initializing a git repository
    - git configurations
    - adding files to the repository
    - checking the status of the files
    - staging/unstaging of files
    - committing of files
    - removing of files
    - restoring of files
    - ignoring files.

Activity: Version control plays a very important role when developing any software (ML/AI) application. In this activity you will learn how to use Git to effectively manage your code and also to do version controlling.

Step 1: Checking the Git version:

git --version

Check the current version of Git by running the command in a terminal.

Step 2: Create a Git repository.

There are mainly two ways in which we can create a Git repository.

1. Initializing a Repository in an Existing Directory.
2. Cloning an Existing Repository.

In this activity we will create a git repository using the first option.

Initializing a Git repository:

If you have a project directory that is currently not under version control and you want to start controlling it with Git, you first need to go to that project's directory and type the git init command. This creates a new subdirectory named .git/ that contains all of your necessary repository files i.e. a Git repository skeleton.

- Navigate to the MLApp folder that has 'requirements.txt' and 'docker-compose.yml'

  cd MLApp

  git init

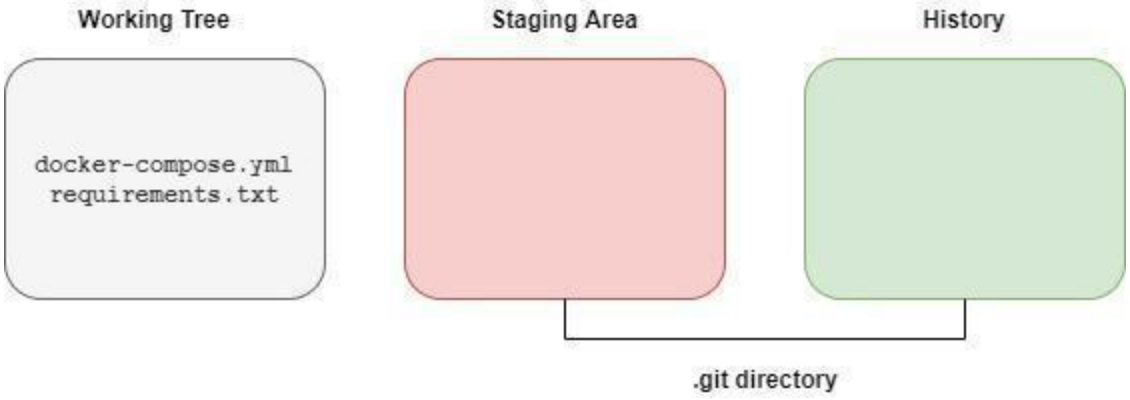

Figure 1

Step 3: Configuring Git username and email:

git config --global user.name <<User Name>>

git config --global user.email <<User Email ID>>

E.g.:

git config --global user.name "Jeevan Sreerama"

git config --global user.email "jeevan.sreerama@insofe.edu.in"

This command is used to set your name and email address. This is important because every Git commit uses this information, and it's immutably baked into all your commits. By passing the **--global** option the configuration will be affected for all of the repositories which you work with on your system.

If you need a different name and email for a particular repository, then use the **--local** option.

git config --local user.name <<User Name>>

git config --local user.email <<User Email ID>>

Step 4: Listing all the Git configurations:

git config --list

This command is used to view all of your Git configuration settings.

NOTE: If you need to remove any git configuration you can use the following command:

git config --unset <<key>>

To remove repository configuration.

git config --global --unset <<key>>

To remove global configuration.

Step 5: Checking the status of the file:

git status

The git status command is used to determine which files are in which state.

NOTE: The 'git status' command will show us that we are on the master branch. The default branch name in Git is master.

Step 6: To track new files/ To stage modified files:

git add <<File Name>>

E.g.:

git add requirements.txt

git add docker-compose.yml

Use git add command to begin tracking the requirements.txt and docker-compose.yml files. The new files will be tracked and staged to be committed.

NOTE: When you have more than two files, you can use asterisk (*) to represent all the files.

Eg:

git add *

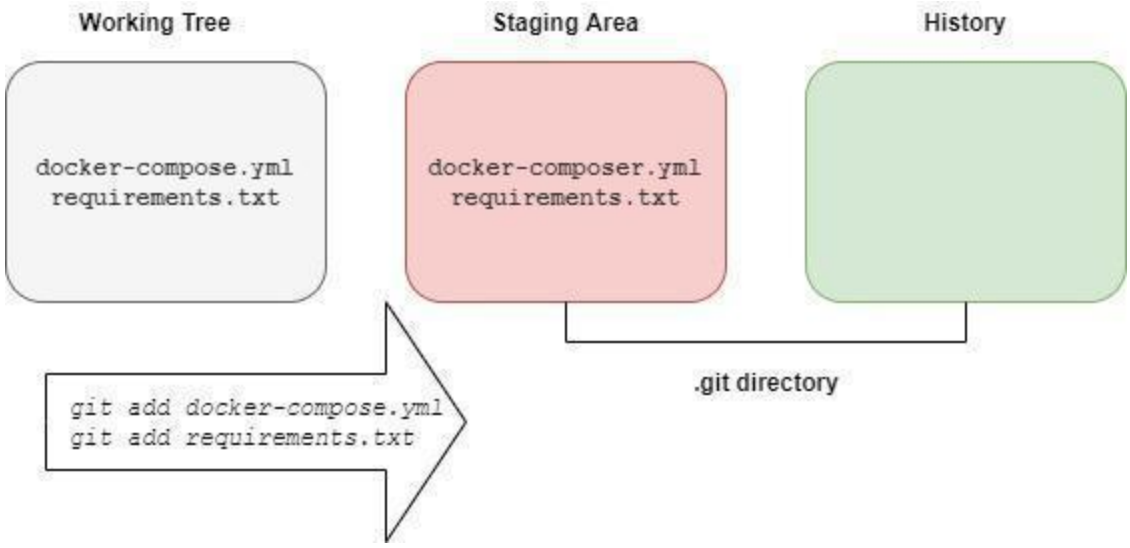

Figure 2

Step 7: Committing the changes to Git repository:

git commit -m <<Type your commit message here >>

E.g.:

git commit -m "Committing docker-compose.yml and requirements.txt files"

The commit records the snapshot of the files you set up in your staging area. Anything you didn't stage will still remain as modified. The commit gives you some output about itself: which branch you committed to, what SHA-1 checksum the commit has, how many files were changed, and statistics about lines added and removed in the commit.

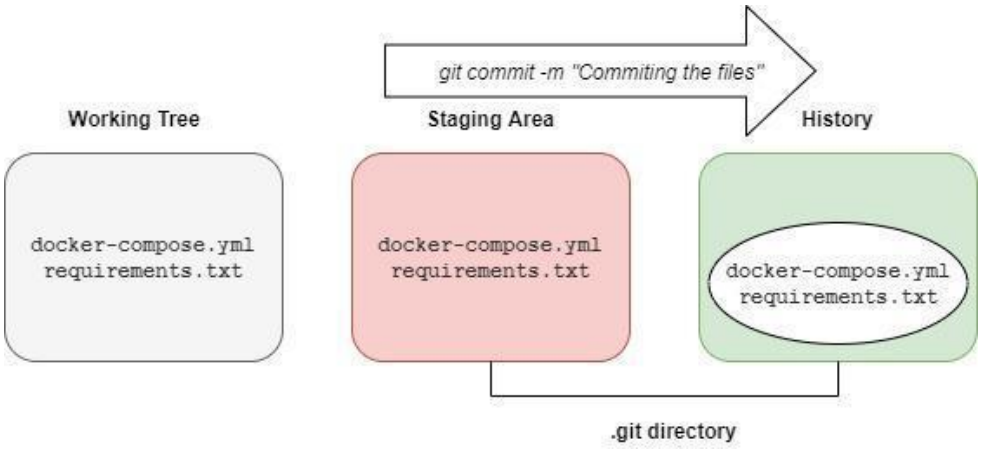

Figure 3

Adding the ***-a* option** to the git commit command makes Git automatically stage every file that is already tracked before doing the commit, letting you skip the 'git add' step.

git commit -a -m <<Type your commit message here >>

Step 8: Viewing the commit history:

git log

The command lists the commits made in that repository in reverse chronological order; that is, the most recent commits show up first. It lists each commit with its SHA-1 checksum, the author's name and email, the date written, and the commit message.

git log -p

-p option shows the difference (the patch output) introduced in each commit.

git log -p -3

You can also limit the number of log entries displayed, such as using -3 to show only the last three entries.

git log –stat

--stat option to see some abbreviated stats for each commit.

git log --oneline --all --graph

--oneline option to display the log messages in a single line.

--graph option to view the commit graph.

--all option to view all the branches.

NOTE: By default the git log command will only show the current working branch.

git log --oneline --all –graph

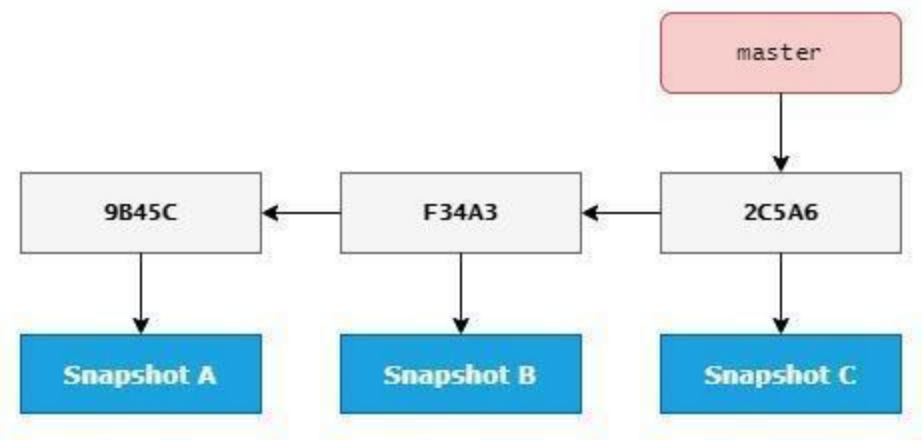

Figure 4

Step 9: Usage of diff/difftool to view the tracked file changes:

Command to compare what is in your working directory with what is in your staging area.

git diff <<File Name>>

or

git difftool <<File Name>> (Recommended)

NOTE: <<File Name>> is optional.

Command to compare the staging area with the most recent commit.

git diff --staged <<File Name>>

or

git difftool --staged <<File Name>>

NOTE: **--staged** and **--cached** are synonyms

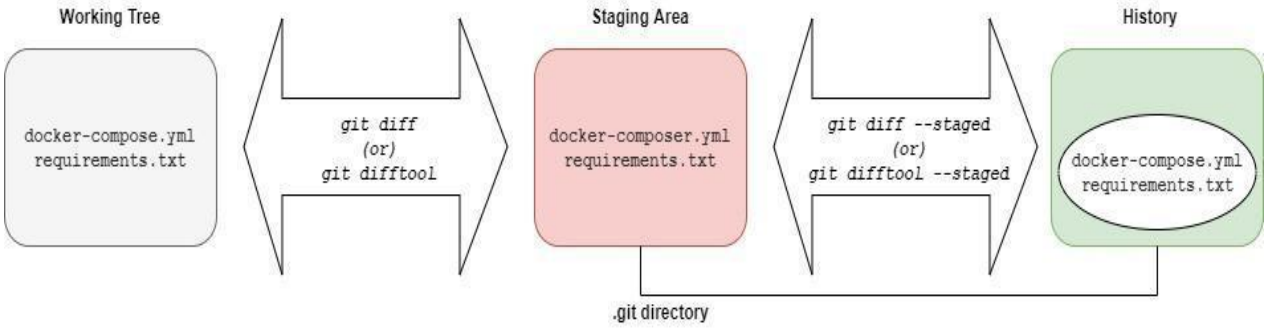

Figure 5

Do the following to understand difftool/diff

1. Change the first line in requirements.txt change from “numpy” to “numpy=1.18.2”.
2. Use git diff or git difftool to see the difference between working directory and Index area.

   git difftool requirements.txt
3. Index requirement.txt.

   git add requirements.txt
4. Use git diff --staged or git difftool --staged to see the difference between Index area and latest commit.

   git difftool --staged requirements.txt
5. Commit the changes.

   git commit -m “changed numpy version to 1.18.2”
6. Change the second line in requirements.txt change from “panda” to “pandas=1.0.3” and directly commit changes.

   git commit -a -m "changed pandas version in requirements.txt files"
7. Display commit log.

   git log --oneline --all --graph

Step 10: Removing a file:

- Removing from working directory

  rm <<File Name>>

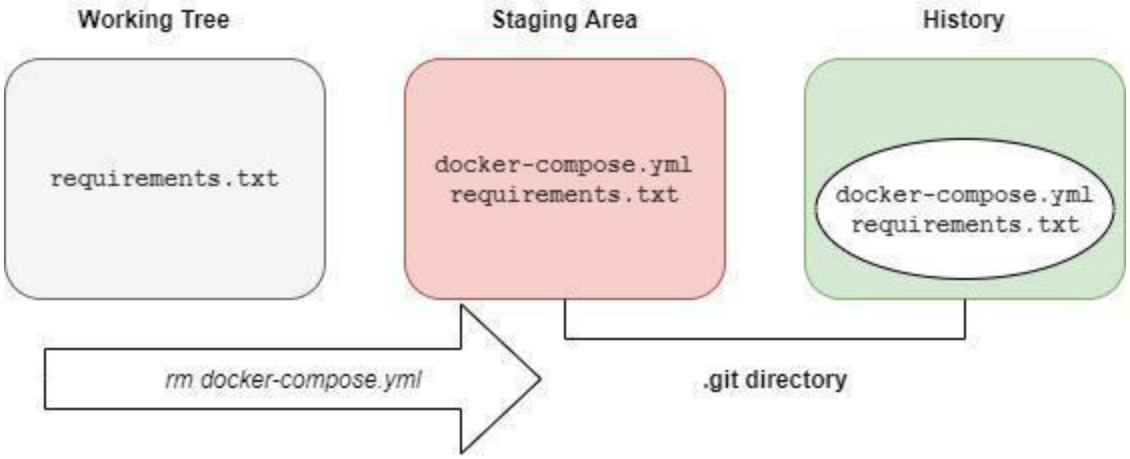

Figure 6

- Removing from index and working directory

  git rm <<File Name>>

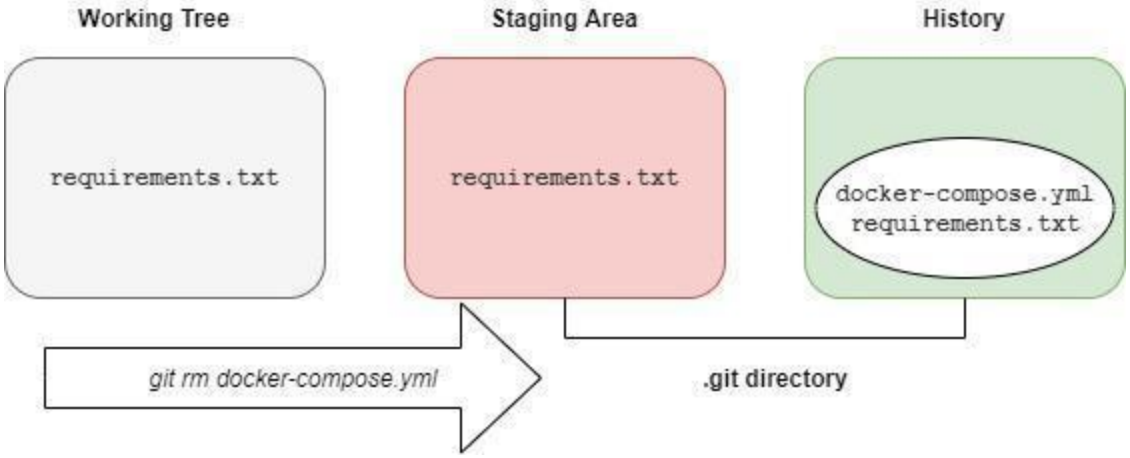

Figure 7

- Steps to removing a file for next commit

  To remove a file from Git, you have to remove it from your tracked files i.e. staging area and also remove it from your working directory. Then commit the changes.

E.g.:

git rm docker-composer.yml

git commit -m “Removing the docker-composer.yml file.”

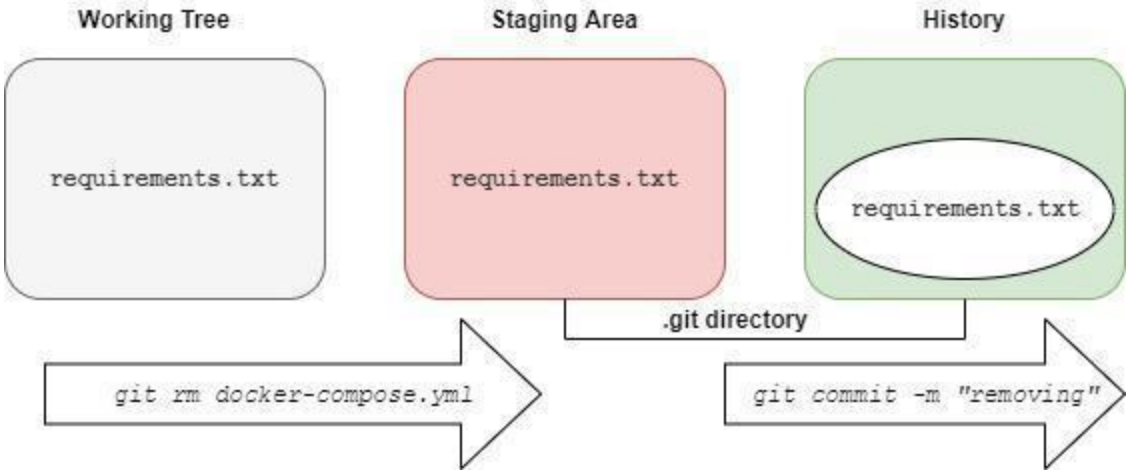

Figure 8

NOTE:

git rm -f <<File Name>>

If you modified the file or had already added it to the staging area, you must force the removal with the -f option

Step 11: Restoring files:

The restore command is available only from v2.23 onwards. If your Git version is older than that you need to use the checkout/reset command. If your version is newer than 2.22, restore is recommended.

git restore <<File Name>>

(or)

git checkout << File Name>>

E.g.:

git restore docker-compose.yml

This is to restore the modified file in the working directory to its previous state. The default restore source for the working tree is the staging area.

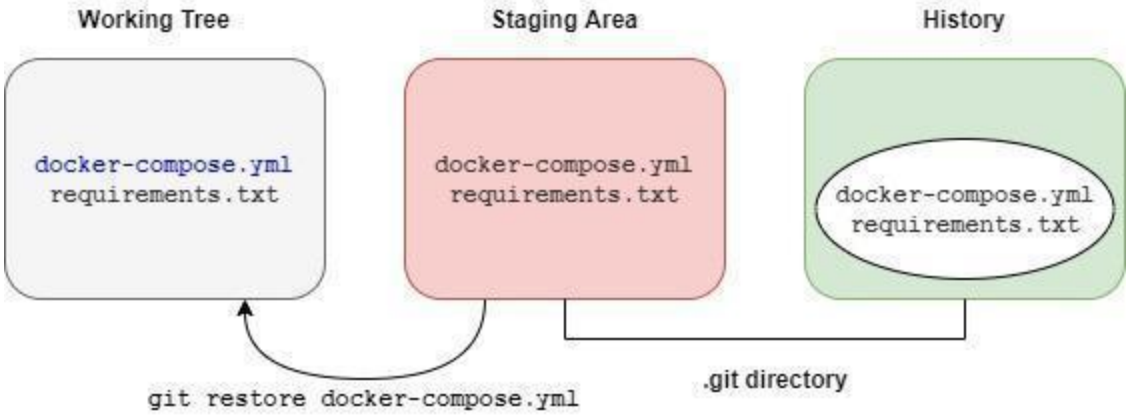

Figure 9

git restore --staged <<File Name>>

(or)

git reset <<File Name>>

E.g.:

git restore --staged docker-compose.yml

This is to restore the modified file in the staging area to its previous state. The default restore source for the index is the most recent commit (HEAD).

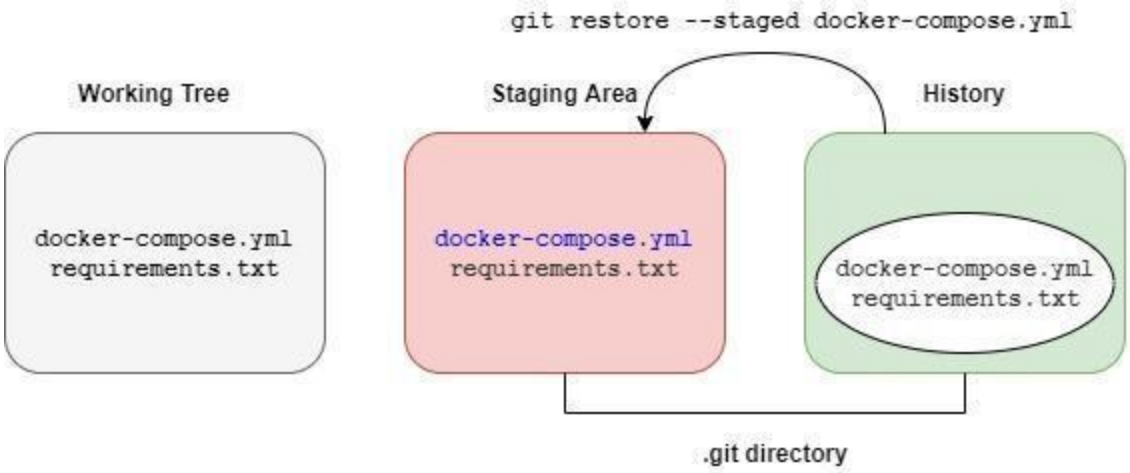

Figure 10

git restore --staged --worktree --source master <<File Name>>

E.g.:

git restore --staged --worktree --source master docker-compose.yml

This is to restore the modified file present in both the staging area and the working area to their previous state. When both --staged and --worktree are specified, --source must also be specified.

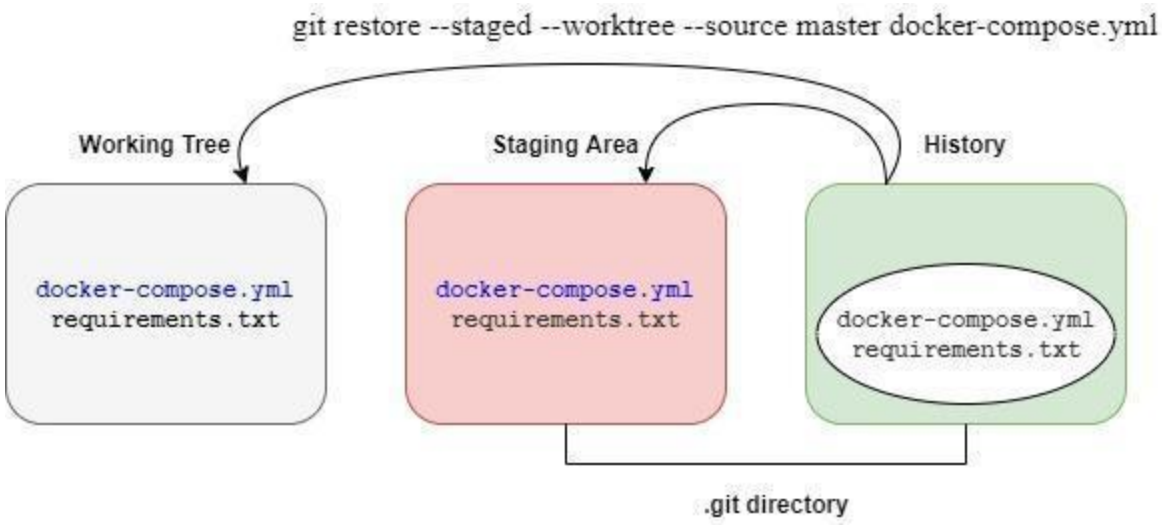

Figure 11

git restore --staged --worktree --source <<Commit ID>> <<File Name>>

(or)

git checkout <<Commit ID>> <<File Name>>

E.g.:

git restore --staged --worktree --source 638744614edf7 docker-compose.yml

git commit -m "Restoring the docker-compose.yml file."

This is to restore a file from an earlier commit.

Step 11: git-ls-files - Show information about files in the index and the working tree

git ls-files

Options

-c

--cached

Show cached files in the output (**default**)

-d

--deleted

Show deleted files in the output

-m

--modified

Show modified files in the output

-o

--others

Show other (i.e. untracked) files in the output

-i

--ignored

Show only ignored files in the output.

-s

--stage

Show staged contents' mode bits, object name and stage number in the output.

--directory

If a whole directory is classified as "other", show just its name (with a trailing slash) and not its whole contents.

-u

--unmerged

Show unmerged files in the output (forces --stage)

Do the following to understand delete and restore.

1. List the files in working tree/directory

   ls

2. Delete requirements.txt file from you working tree/directory

   rm requirement.txt

3. List the files in working tree/directory

   ls

4. List the file in stated area

   git ls-files

5. Restore the requirements.txt file in to working tree/directory from staged area

   git restore requirements.txt

6. List the files in working tree/directory

   ls

7. Delete requirements.txt file from both working tree/directory and staged area

   git rm requirements.txt

8. List the files in working tree/directory

   ls

9. List the file in stated area

   git ls-files

10. Restore the requirements.txt file in to staged area from most recent commit that current branch and HEAD are pointing to

    git restore —staged requirements.txt

11. List the files in working tree/directory

    ls

12. List the file in stated area

    git ls-files

13. Restore the requirements.txt file in working tree/directory from staged area

    git restore requirements.txt

14. Delete requirements.txt file from both working tree/directory and staged area

    git rm requirements.txt

15. Commit the changes i.e. requirements.txt file delete

    git commit -m "Deleted requirements.txt file"

16. List the commit graph

    git log —oneline —all —graph

17. Restore the requirements.txt file into both working tree/directory and staged area from the most recent commit (e.g. 023d9e5) that has requirements.txt

    git restore --staged --worktree --source 023d9e5 requirements.txt

18. List the files in working tree/directory

    ls

19. List the file in stated area

    git ls-files

20. Commit to add requirements.txt file

    git commit -m “Adding back requirement.txt”

Step 12: Ignoring Files:

If there are files in your working directory that you don’t want Git to automatically add or even show you as being untracked, you can create a file listing patterns to match them named .gitignore.

The rules for the patterns you can put in the .gitignore file are as follows:

- Blank lines or lines starting with # are ignored.
- Standard glob patterns (regular expressions in shell) work, and will be applied recursively throughout the entire working tree.
- You can start patterns with a forward slash (/) to avoid recursivity.
- You can end patterns with a forward slash (/) to specify a directory.
- You can negate a pattern by starting it with an exclamation point (!).

Examples:

a) Create a .gitignore file and add the following rule to ignore all the python files in this folder:

   *.py

b) Create a new file named test.py and check the status of the file using the git command.

   git status

c) The python files will be ignored by git inside this folder.

d) Add .gitignore file and commit changes.

   git add .gitignore

   git commit -m "Added .gitignore files"

Introduction to Git: Branching and Merging

Agenda:

1. To understand the concept of branches in Git.
2. To understand the purpose of the HEAD pointer and the meaning of detached HEAD.
3. Hands-on:
    - create, work on and delete branches
    - merging of branches i.e. fast-forward merge and 3-way merge
    - create and resolve merge conflict
    - work with git stash

Activity: Let's assume that there are two sprint teams working on two independent features that need to be implemented on the same codebase. Let's create two branches for them to work separately and later we will merge the branches.

1. Create new branches:

Branching is a process where you diverge from the main line of development and continue to do work without merging with that main line. When you make some changes in your files and commit again, the next commit stores a pointer to the commit that came immediately before it. Every time you commit, the master branch pointer moves forward automatically.

A branch in Git is simply a lightweight movable pointer to one of these commits. When you create a new branch in Git, you are creating a new pointer for you to move around.

Git has a special pointer called HEAD. HEAD is a pointer to the local branch you're currently working on.

NOTE: The "master" branch in Git is not a special branch. It is exactly like any other branch. The only reason nearly every repository has one is that the git init command creates it by default and most people don't bother to change it.

Step a) Create two branches named Feature1 and Feature2 from the master branch.

git branch <<Branch Name>>

E.g.:

git branch Feature1

git branch Feature2

The git branch command creates a new pointer to the same commit you're currently on. This command will only create a new branch and it doesn't switch to that branch.

Step b) Verifying the newly created branches

git branch

Step c) Verify the HEAD pointer by looking at log

git log --oneline --all --graph

NOTE: The HEAD is still pointing to the master branch. The master, Feature1 and Feature2 branches are pointing to the same commit.

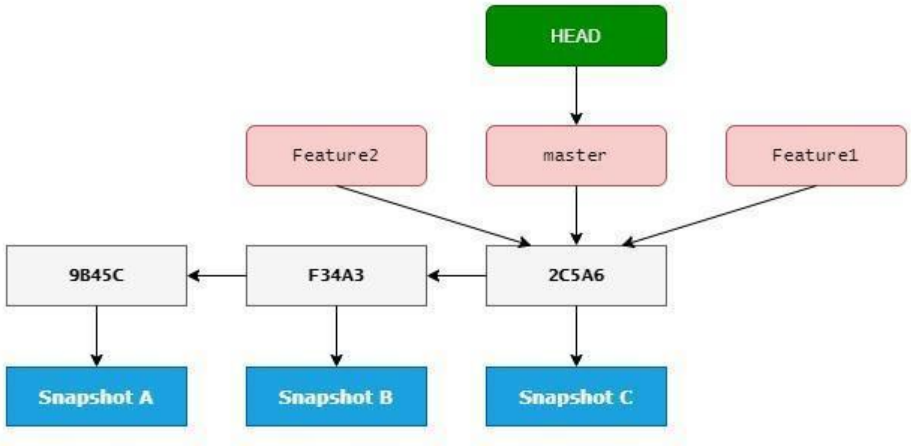

Figure 12

2 Working on the branches:

Step a) Switching to Feature1 branch

git checkout <<Branch Name>>

E.g.:

git checkout Feature1

The git checkout command is used to switch to a specified existing branch. This command moves the HEAD to point to the specified branch.

Step b) Verify the HEAD pointer

git log --oneline --all --graph

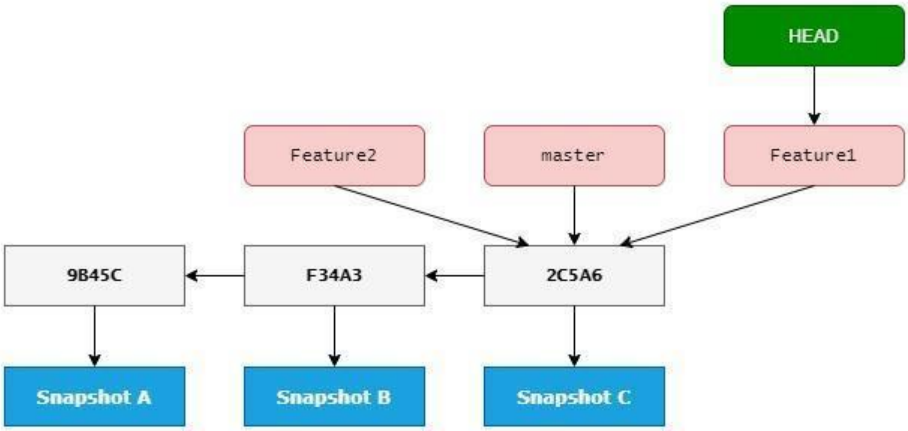

Figure 13

Step c) Modifying the requirements.txt file in the Feature1 branch and commit the changes

Let's open the requirements.txt and add the scipy package with version 1.5.0.

```
git commit -a -m "Adding the scipy package in requirements.txt file."
```

Step d) Verify the HEAD pointer

```
git log --oneline  --all --graph
```

NOTE: The Feature1 branch has moved forward, but your master and Feature2 branches are still pointing to the commit you were on when you ran git checkout to switch branches.

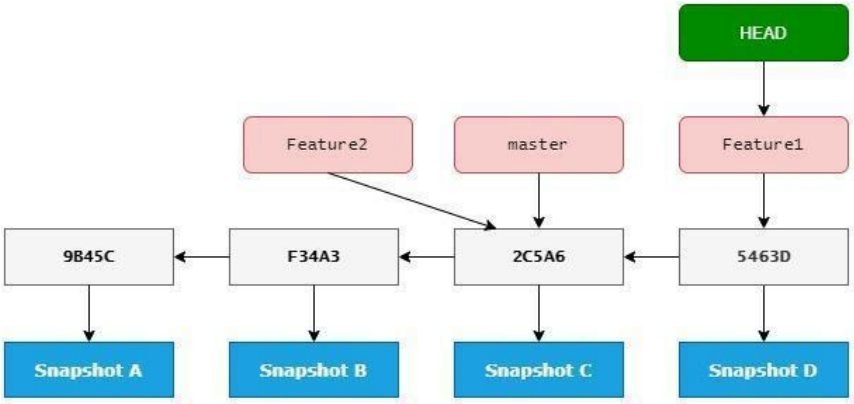

Figure 14

Step e) Switching to the Feature2 branch

```
git checkout Feature2
```

This command moves the HEAD to point to the Feature2 branch. Updates files in the working tree and index to match the version HEAD is pointed to i.e. to the commit the Feature2 branch is pointing to.

Step f) Modifying the docker-compose.yml file and commit the changes.

Open the docker-compose.yml file and update the port number from 1234 to 1235

```
git commit -a -m "Updating the port number in docker-compose.yml file"
```

Step g) Verify the HEAD pointer

```
git log --oneline --all --graph
```

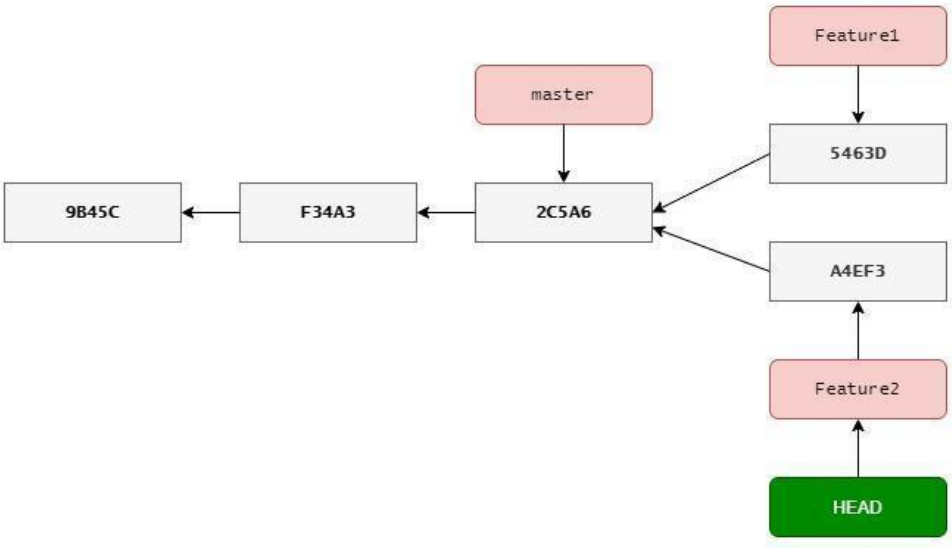

Figure 15

3 Fast-forward Merge:

Integrate the changes made in the Feature1 branch into the master branch.

Step a) Switching to the master branch

```
git checkout master
```

Step b) Check the difference between master branch and Feature1 branch

```
git difftool master Feature1
```

It shows that you have added the scipy package in the requirements.txt file in the Feature1 branch.

Step c) Do the merge operation

```
git merge <<Branch Name>>
```

E.g.:

```
git merge Feature1
```

This code will update all the changes in the Feature1 branch to the master branch. Fast-forward merge will work even if there were more than one commit in the Feature1 branch.

Step d) Verify the graph

```
git log --oneline --all --graph
```

The graph shows that the master branch has moved to the Feature1 branch.

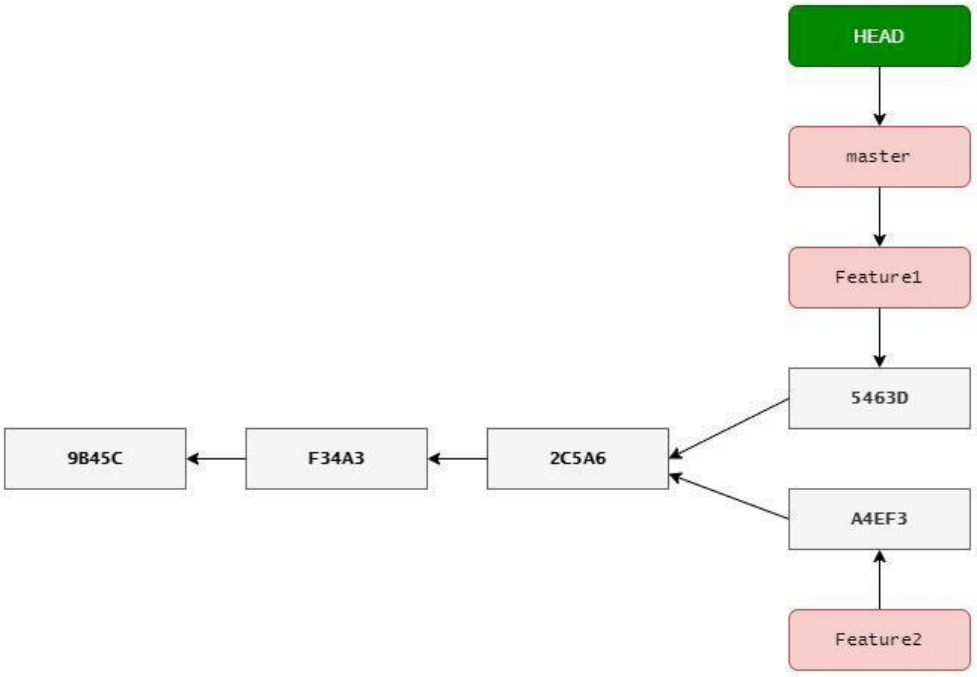

Figure 16

4 Deleting branches:

Since all the changes in the Feature1 branch have been merged into the master branch we can delete the Feature1 branch.

Step a) Check the branches that are merged

```
git branch --merged
```

This command is to check which branches are already merged with the current branch. It will show that the Feature1 branch is merged with the master branch.

Step b) Delete the Feature1 branch

```
git branch -d <<Branch Name>>
```

E.g.:

```
git branch -d Feature1
```

This command is used to delete an already merged branch.

Step c) Try to delete a branch that is not merged

git branch -d Feature2

Git will not allow you to delete the Feature2 branch, since it's not merged with the current branch.

NOTE: To force delete use the following command.

git branch -D <<Branch Name>>

Step d) Verify the graph

git log --oneline --all –graph

NOTE: Looking at the commit graph we see there is no direct path from the master branch to the Feature2 branch. Git cannot do a fast-forward merge.

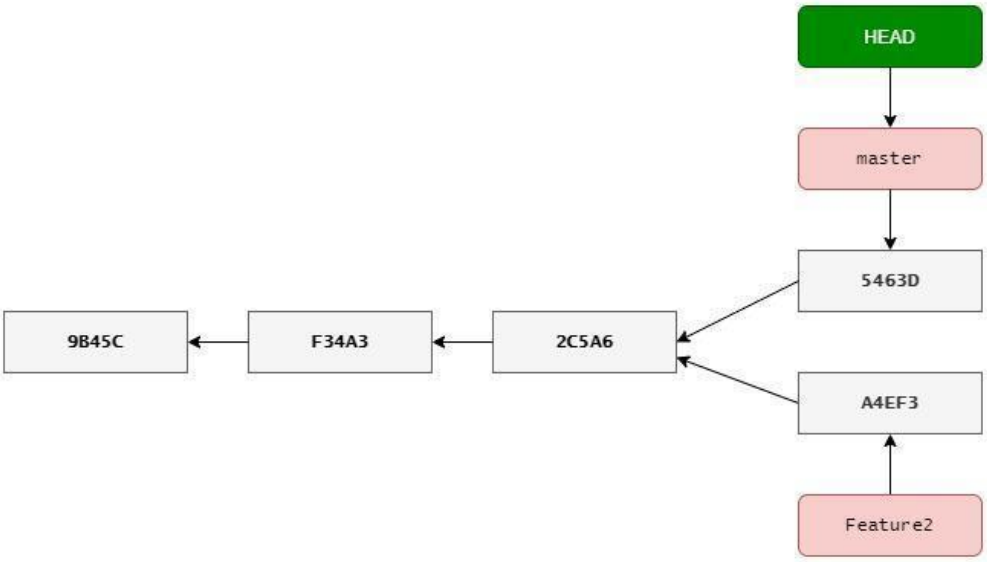

Figure 17

5 3-way Merge (Merge Without Conflict):

Current status of the branch is as in the above figure (Figure17). To merge the master and Feature2 branch we cannot simply move the master pointer to the Feature2 branch. If we do that we will be losing the Feature1 branch changes made in the master. We need to merge these branches together into a new commit called the merge commit. To make this merge commit, Git looks at three commits- i.e. the base commit from which the branches start from and the last commit of each branch.

Step a) Check the status of the master branch

git status

This shows that we are still on the master branch. Incase if master is not your current branch checkout the master branch using git checkout master

Step b) Do the merge operation

```
git merge Feature2
```

The output shows that the merge is made by recursive strategy.

Step c) Verify the graph

```
git log --oneline --all --graph
```

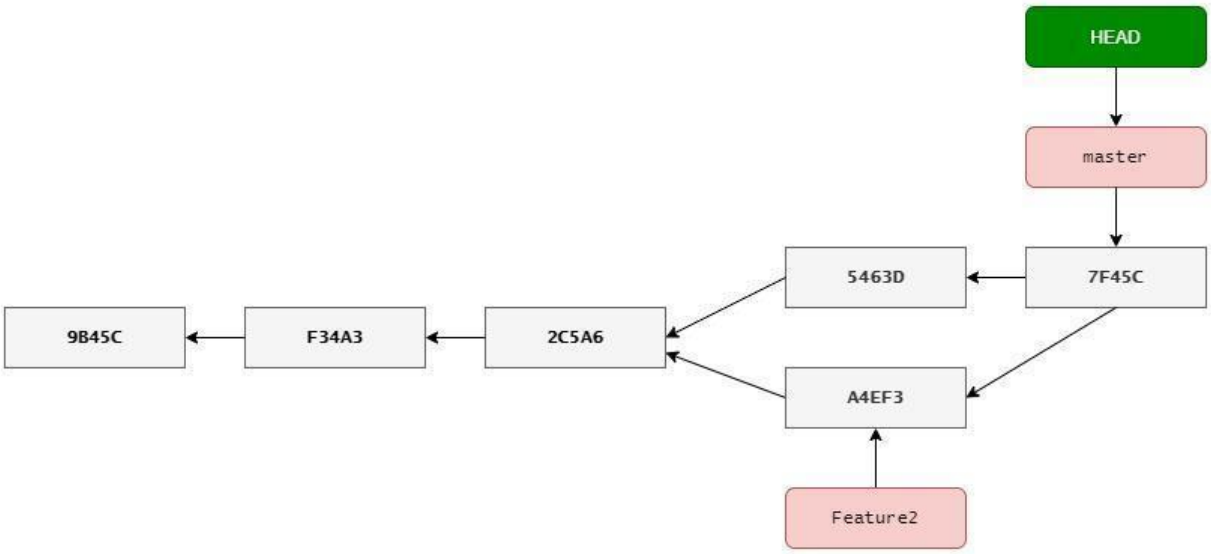

Figure 18

Step d) Delete the Feature2 branch

```
git branch --merged
git branch --d Feature2
```

Step e) Verify the graph

```
git log --oneline --all --graph
```

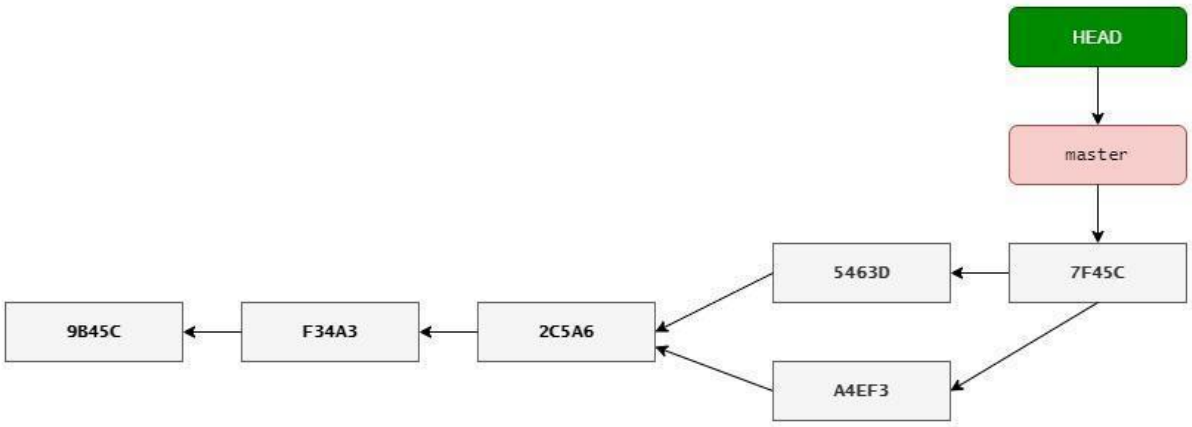

Figure 19

6 Merge With Conflicts:

A merge conflict occurs when we try to merge branches that have changed the same lines in the same files. Let's understand this in detail by simulating the same with an example.

Step a) Create and checkout a new branch named Feature1

```
git checkout -b Feature1
```

-b option creates a new branch if the branch doesn't exist.

Step b) Edit the requirements.txt file

Update the version for numpy from 1.18.2 to 1.15.4

Update the version for pandas from 1.0.0 to 1.0.3

Step c) Commit the changes to the Feature1 branch

```
git commit -a -m "Updating the versions for numpy and pandas"
```

Step d) Checkout the master branch

```
git checkout master
```

Step e) Edit the requirements.txt file

Update the version for pandas from 1.0.0 to 1.0.4

Step f) Commit the changes to the master branch

```
git commit -a -m "Updating the version for pandas"
```

Figure 20

Step g) Verify the commit graph

```
git log --oneline --all --graph
```

The commit graph shows that both the branches are diverged and we can't have a fast-forward merge. This is going to be a 3-way merge and there are conflicts to be resolved.

Step h) Try to merge the file in the master branch

```
git merge Feature1
```

The merge will fail with the merge conflict error.

Step i) Check the status of the branch

```
git status
```

Git status shows us that we are in the middle of the merge and says "You have unmerged paths". It also shows us the files that are having conflict.

Step j) Aborting the merge changes (Optional)

If you don't want to deal with resolving the conflicts here, we can abort the changes by running the following command:

```
git merge --abort
```

To abort the changes made in the files because of the merge conflict.

```
git status
git log --oneline --all --graph
```

The git status/log shows that all the changes have been reverted and we are back in the step (g).

Step k) Resolving the conflicts

If you want to resolve the conflicts, you should not execute the step (j).

Let's now open the requirements.txt file which is showing the conflicts. Here the set of equals signs separates out the state of the file in the two branches.

Above the equal sign you can see the lines of code which belong to the branch where the HEAD is currently pointing i.e. the master branch. Below the equal sign you can see the lines of code for the Feature1 branch. Manually resolve the conflicts, then stage and commit the file.

```
git add requirements.txt.
git commit -m "Merging the branches after removing the conflicts"
```

Step l) Delete the Feature1 branch

```
git branch --merged
```

git branch --d Feature1

Step m) Verify the commit graph

git log --oneline --all --graph

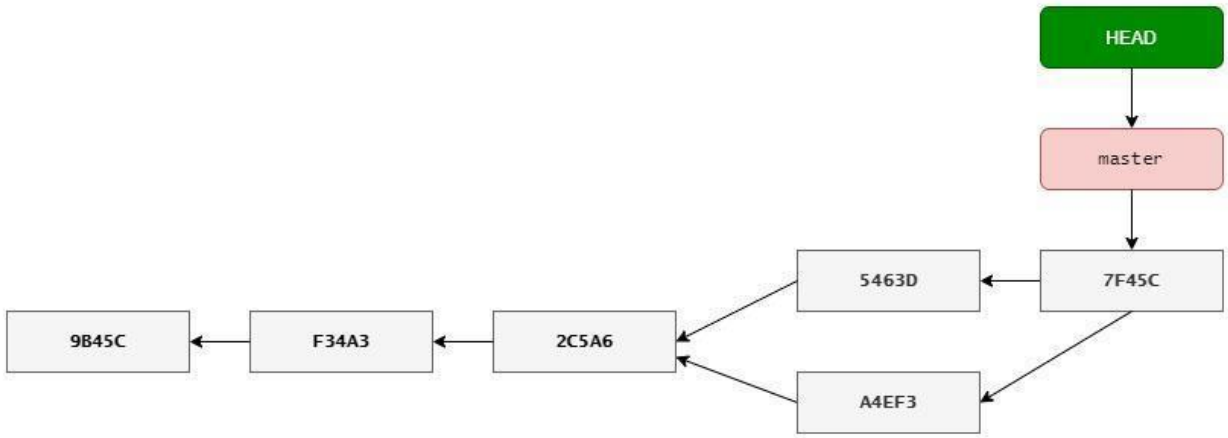

Figure 21

7 Detached HEAD:

Usually the HEAD points to a branch and the branch points to a commit. When the HEAD is pointing directly to a commit we say it as a detached HEAD.

git checkout <<Commit ID>>

E.g.:

git checkout 2c5a6a49048169ec64c277f2723a76260532c8d2

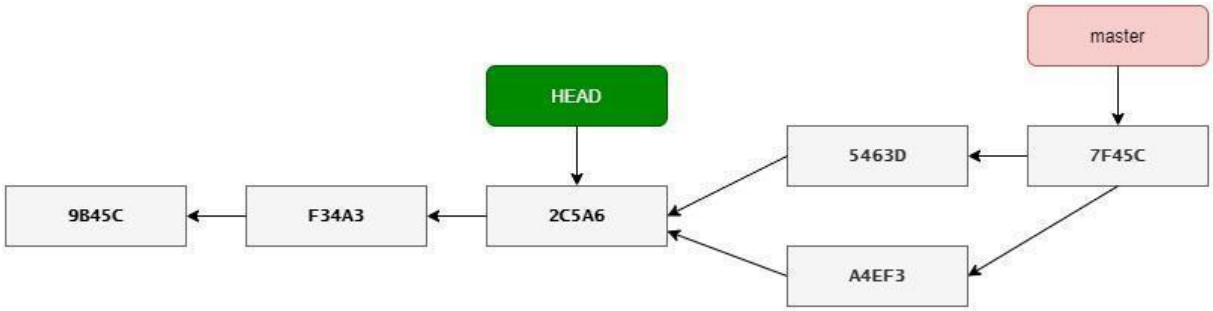

Figure 22

Step a) Resolving the detached HEAD issue by creating a new branch

git branch NewBranch

git checkout NewBranch

Step b) Verify the commit graph

git log --oneline --all --graph

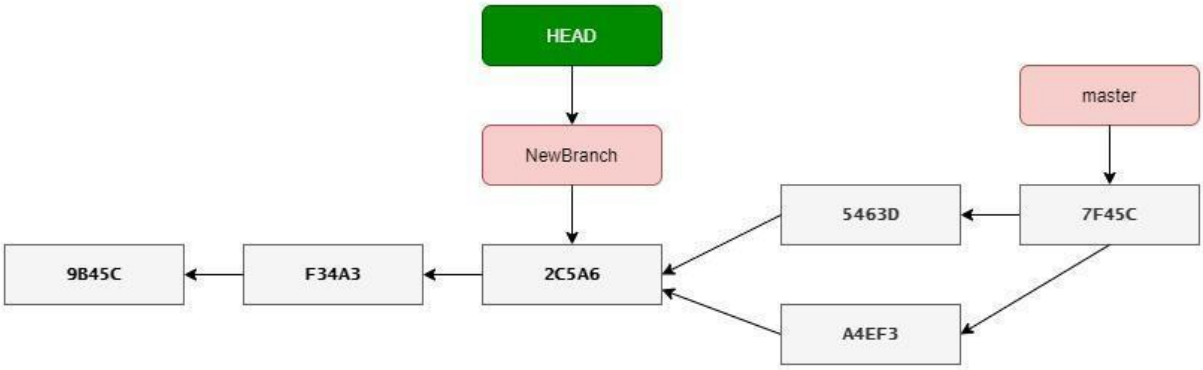

Figure 23

8 Git Stash:

Sometimes you want to switch branches to on the fly accommodate high priory work. Your current work is only half done and you don't want to commit the half-done work. The answer to this situation is **git stash** command. The **git stash** command enables you to switch branches without committing the current branch.

Generally, the stash's meaning is "store something safely in a hidden place." The sense in Git is also the same for stash; Git temporarily saves your data safely without committing.

Let's assume you started implementing new feature for the application in the new branch Feature1

1. Create and checkout new Feature1 branch

   git checkout -b Feature1

2. Update scipy version in requirements.txt file to 1.7.1, and stage the file.

   vim requirements.txt

   git status

   git add requirements.txt

   git status

3. Change port number in docker-compose.yml file to 1432

   vim docker-compose.yml

   git status

4. Now your code is in progress and suddenly a customer escalation comes. Because of this, you have to keep aside your changes for few hours. You cannot commit your partial code and also cannot throw away your changes. So you need some temporary space, where you can store your partial changes and later on commit it. In Git, the stash operation takes

your modified tracked files, stages changes, and saves them on a stack of unfinished changes that you can reapply at any time.

```
git stash save “Stashing half-done work”

git status
```

5. Now you can switch to the required branch and complete the high priority work. Once done, switch back to Feature1 branch, and check the Stored Stashes using

```
git stash list
```

6. Re-apply the changes that you just stashed by using the git stash apply command. This command restores the last stash if the <<stash index id>> is not specified. In case there are more than one stashes, use "git stash apply" command followed by <<stash id>> to apply the particular commit. i.e. **git stash apply <stash id>**

```
git stash apply

git status
```

- Git Stash Pop

  Git allows the user to re-apply the previous commits by using the git stash pop command. This command also removes the changes from stash and applies them to your working file.

- Git Stash Drop (Unstash)

  The git stash drop command is used to delete a stash from the queue. Generally, it deletes the most recent stash. To delete a particular stash from the available stashes, pass the stash id in the stash drop command i.e. git stash drop <stash id>

- git stash clear

  The git stash clear command allows deleting all the available stashes at once. To delete all the available stashes, operate below command:

7. Lets clear everything in the stash using git stash clear

```
git stash clear
```

8. Let’s assume now the work is fully completed, so add and commit.

```
git add .

git commit -m “Committing after stash”

git status

git log --oneline --all --graph
```

References:

1. Git-Book: https://git-scm.com/book/en/v2

2. https://www.youtube.com/watch?v=FyAAIHHClqI