**Git POC'S:**

1. **Explore .git folder.**

.git folder contains all the vital information about our project repository.

It contains info relating to commit and it also contains remote repo address.

All the scripts are stored here along with the staging and configuration details.

There are total 4 sub-directories inside a .git folder.

**Objects**: all objects

**Refs**: This folder contain information about tags & branches

**Hooks**: hooks folder contain script

**Info**: exclude file for ignored patterns

**Logs**: contains history for different branches

Also apart from this folders there are few files

**HEAD**: head contain the pointer or reference to your working directory or branch.

**Config** : Config contain all of your configurations including the credentials you use to connect to

Github your email & name that you used to configure your global config.

**Index**: index file is used for when we use git add to stage your files for a commit.

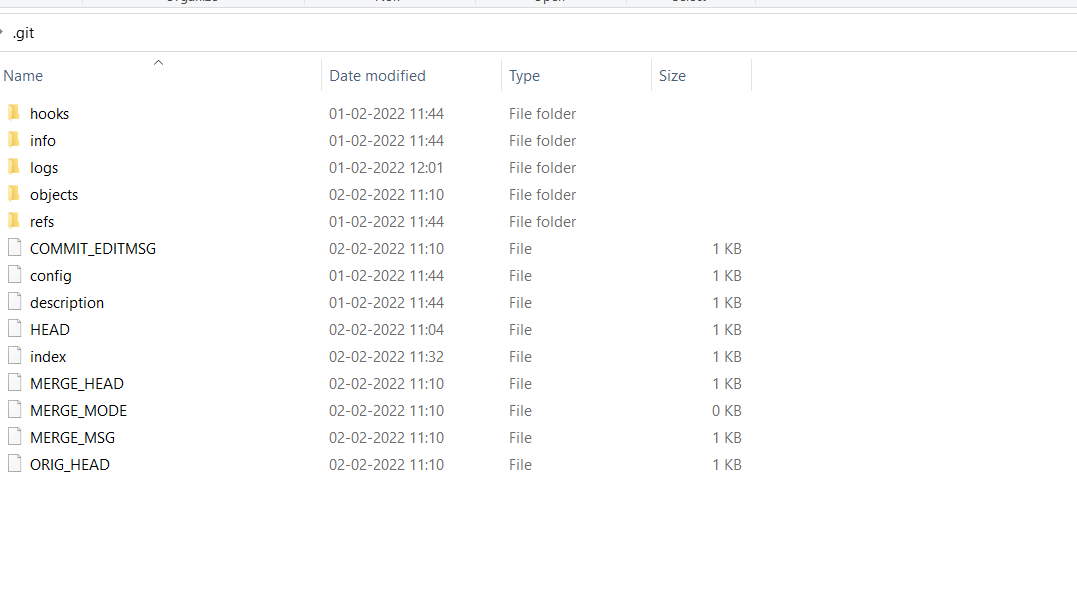
Every commit contains three main type of objects:

1) **Blob**: if we add the file to commit after the commit is made, these file are encrypted, compressed and store as object called blobs.

2) **Tree**: a hash algorithm is used to generate the identifier for each file & are put in a list called tree there is only one tree object per commit.

3) **Commit**: a hash of the tree object is generated using the hash algorithm to put in to the commit.

Below is the folder structure for the .git folder.



**b)What is git reset?(Theory + POC).**

git reset is use for undoing changes. The git reset command is use to reset the changes.git reset command has three core forms🡪

--Soft

--Mixed

--Hard

**Git reset hard** :

It will move the head and update the index with the contents of the commits. The –hard option changes the commit history and ref pointers are updated to the specified commit. Then the staging index and working directory need to reset to match that of the specified commit. Any previously pending commits to the staging index and the working directory gets reset to match commit tree.

**Git reset soft :**

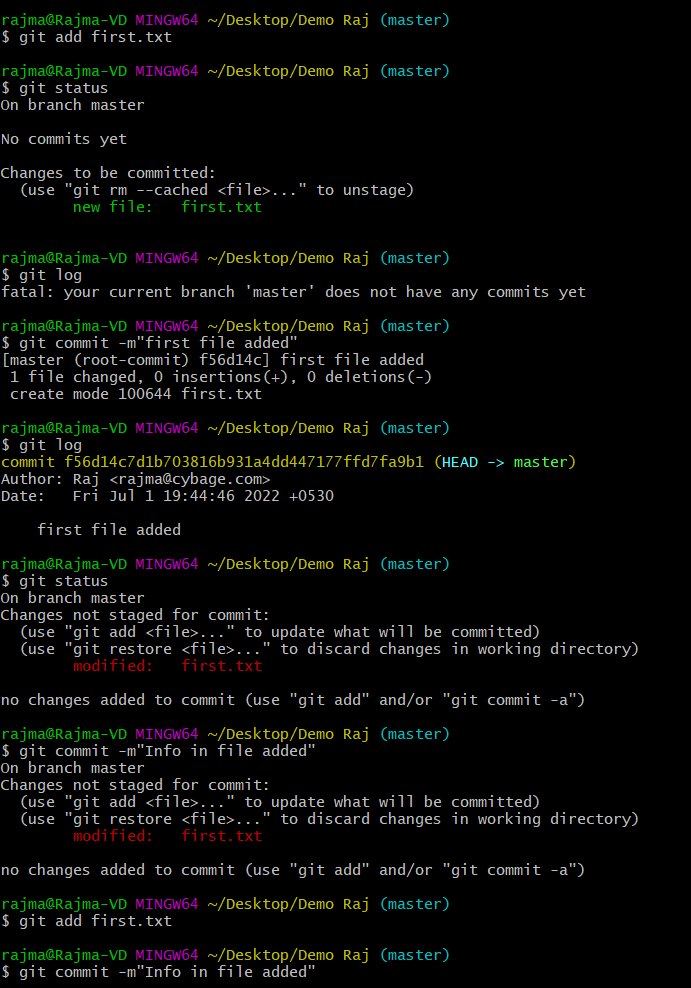
Undo the last git commit with reset. the easiest way to undo the last git commit is to execute the git reset command with –soft option that will preserve the changes done to your file.

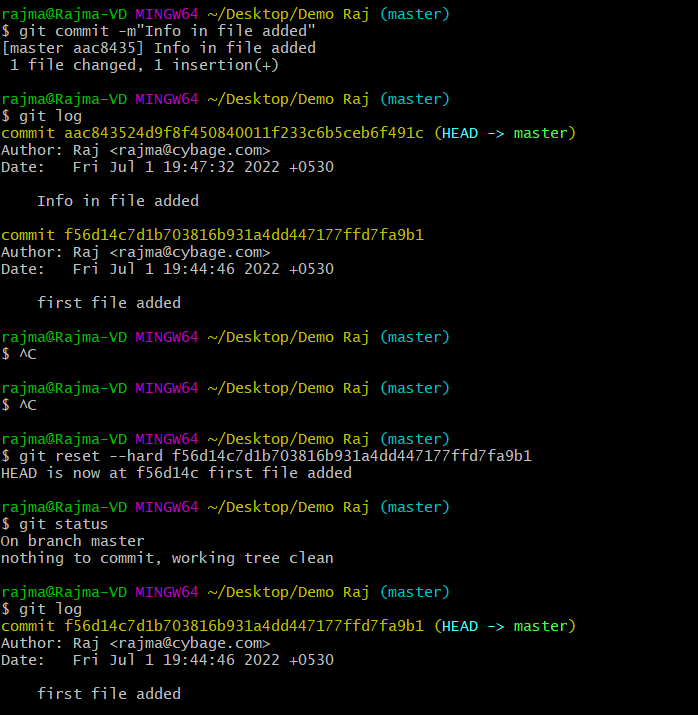
Command: **git reset --soft**

**Git reset mixed :**

A mixed option is a default option of the reset command. if we would not pass any argument then the git reset command considered as –mixed as default option. A mixed option updates the ref pointers. The staging area also reset to the state of a specified commit. the undone changes transferred the working directory.

Here we use command **git reset –mixed** or we can use only **git reset**

****

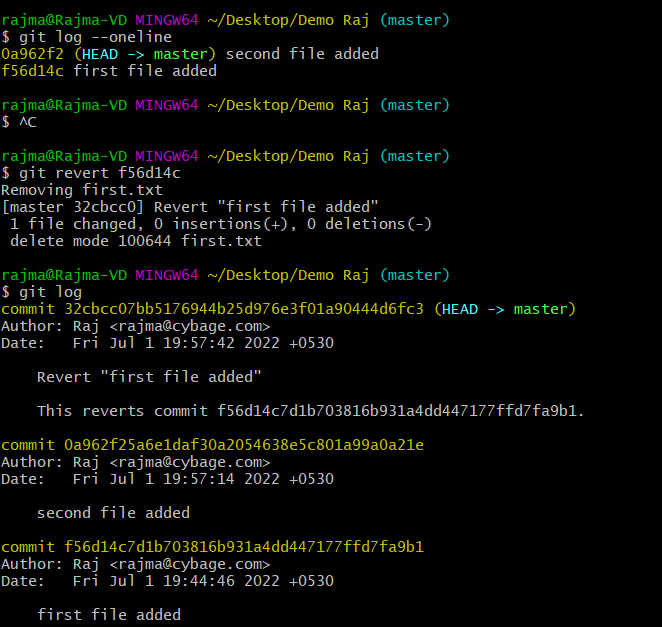
****

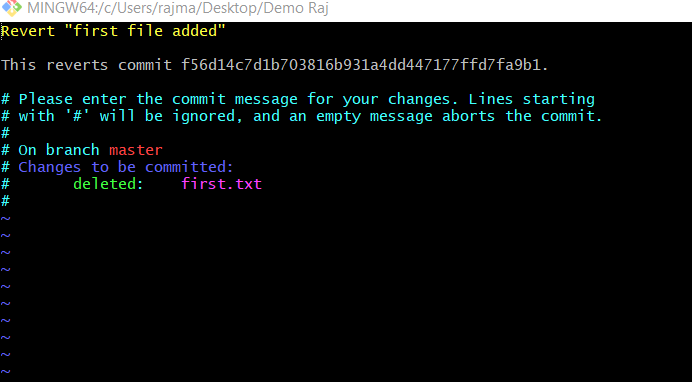
**c)What is git revert(Theory + POC)?**

Git revert is like undo operations.it will give safe method for undoing changes

Instead of deleting commits in the commit history a revert will create a new commit that inverse the changes specified.so that git revert is safer alternative to git reset in regards to losing work.

Command: git revert commit\_id





**d)what is difference between git reset and git revert?**

**Git revert:**

-Git revert command use on remote repo.

-we can perform git revert on specific commit.

-after revert new commit id is generated

-revert maintain original commit history.

Command: git revert commit\_id

**Git reset:**

-Git reset command use on local repo.

-after reset new commit is not generated.

Command: git reset commit\_id

**e) What is git cherrypick? (Theory + POC)?**

git cherry-pick commit-id: with the cherry-pick command , git allows you to integrate selected ,individual commits from any branch in to current HEAD branch. Which means any desired commit can be merged with our current head commit.

**f) Explain different types of merging strategies?**

**Fast forward merge**

In fast-forward merge, git simply moves the source branch pointer to the target branch pointer without creating an extra merge commit.

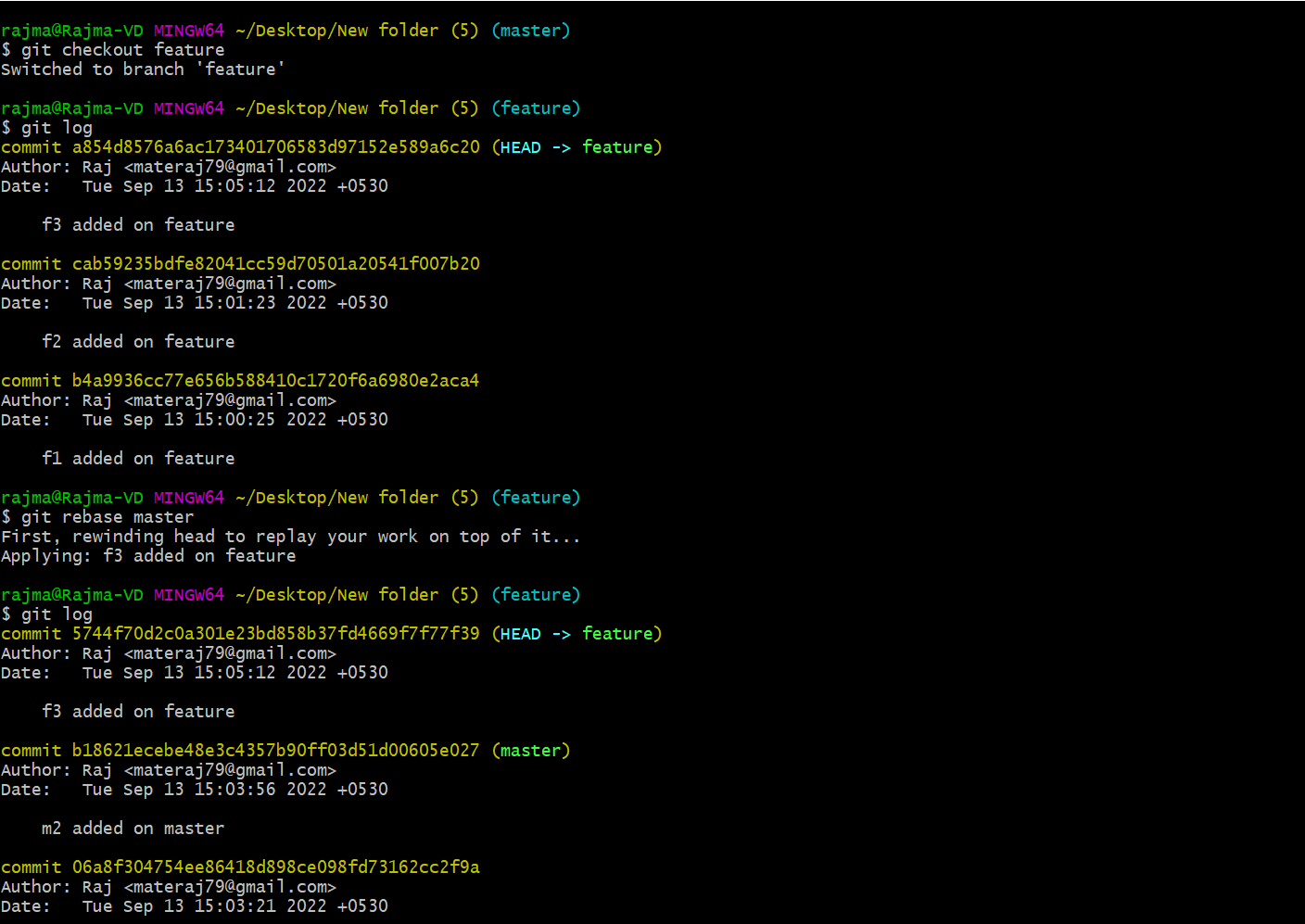
**Recursive merge**

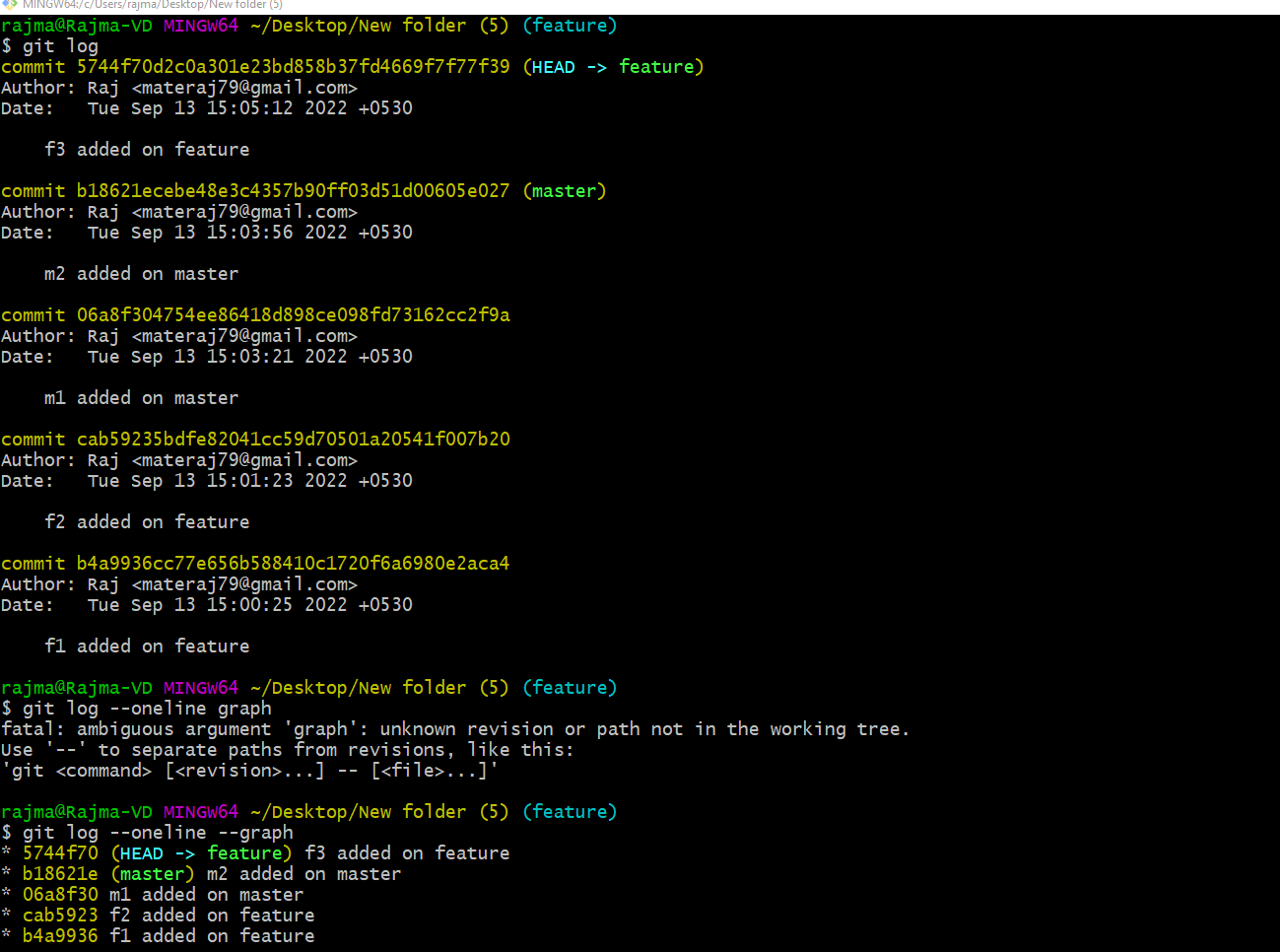
Recursive is the default merge strategy when pulling or merging one branch. additionally, this can detect and handle merges involving renames, but currently cannot make use of detected copies**.**

**Rebase merge**

Rebasing is a process to reapply commits on top of another base trip.it is used to apply a sequence of commits from distinct branches in to a final commit.it is an alternative of git merge command.

**g)What is git rebase(Theory + POC)?**

**🡪** This moves the entire feature branch to begin on the tip of the main branch, effectively incorporating all of the new commits in main. **Git rebase alters all the history of commits**. ****

****

**h)What is difference between fast forward and No fast forward merge?**

**🡪 Fast Forward Merge –** This merge occurs when we create a branch & there are no new commits in master & then if we merge both branches we will get fast forwards merge.

**No fast forward merge** :- A non-fast-forward merge is a merge where the main branch had intervening changes between the branch point and the merge back to the main branch.

**i)What is difference between normal rebase and interactive rebase(Theory + POC)?**

**Answer:-** Sometimes rebase is used as an alternative to merge.

Rebasing a branch updates one branch with another by applying the commits of one branch on top of the commits of another branch.

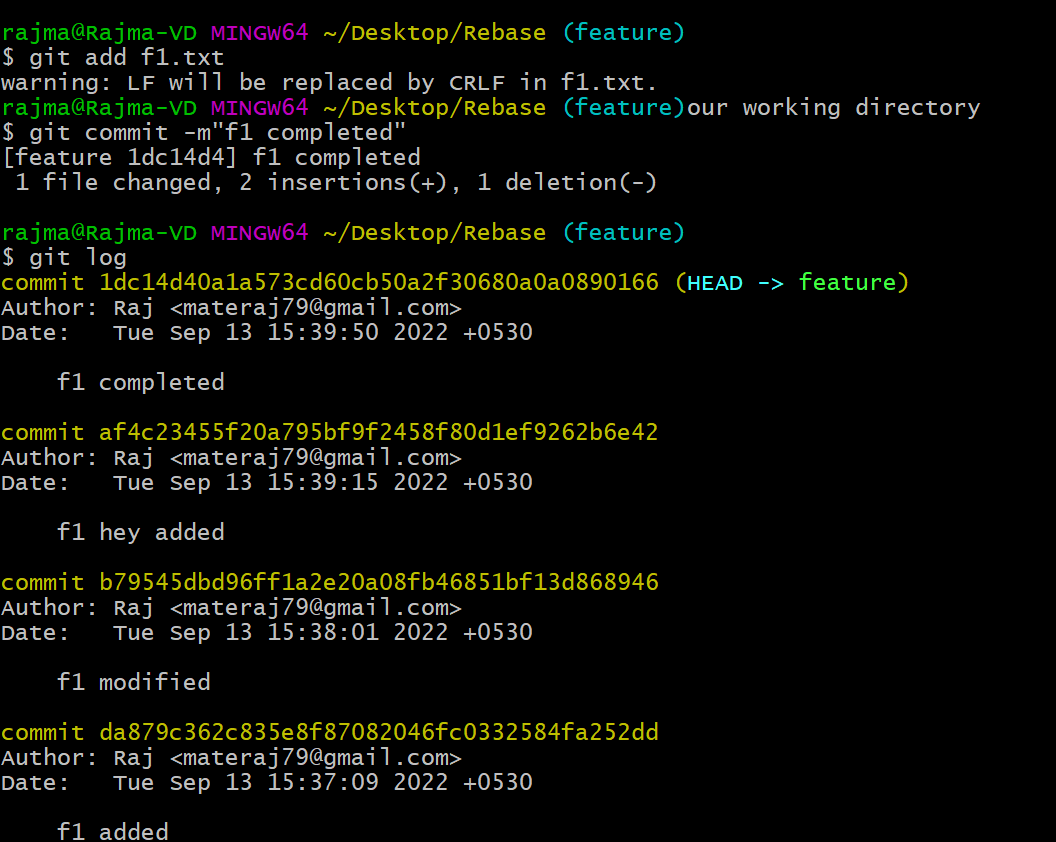
Rebase does not preserve history.

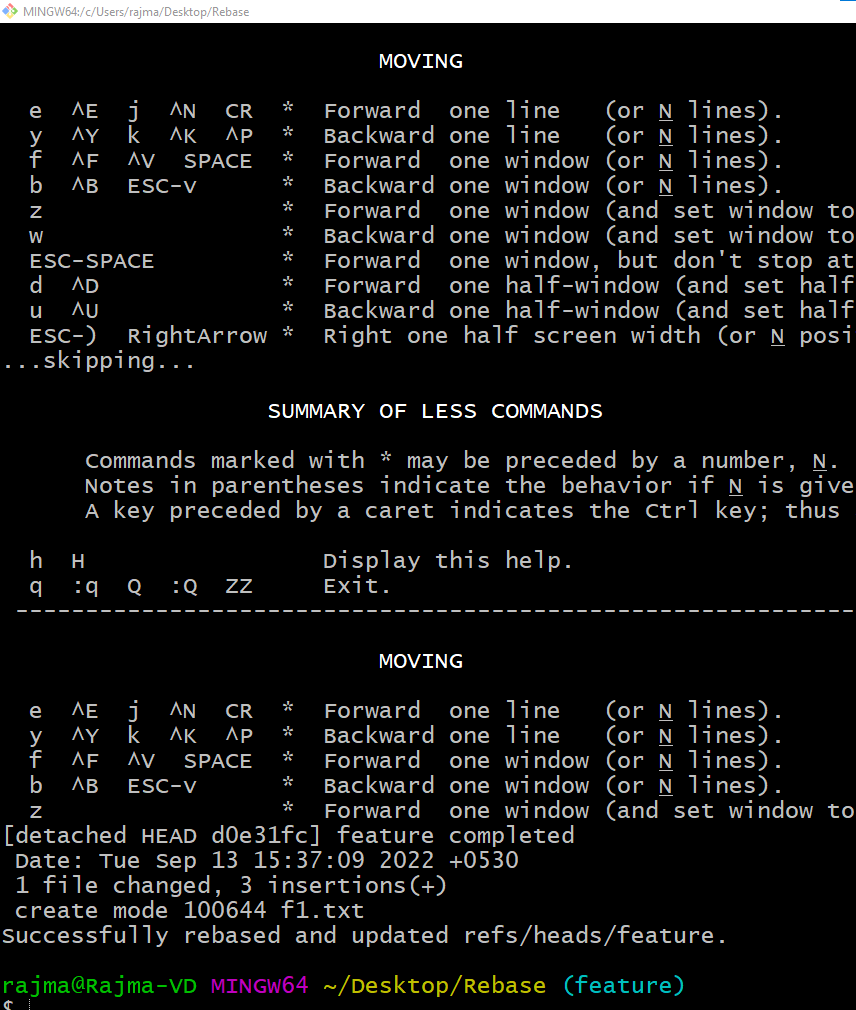
Difference –

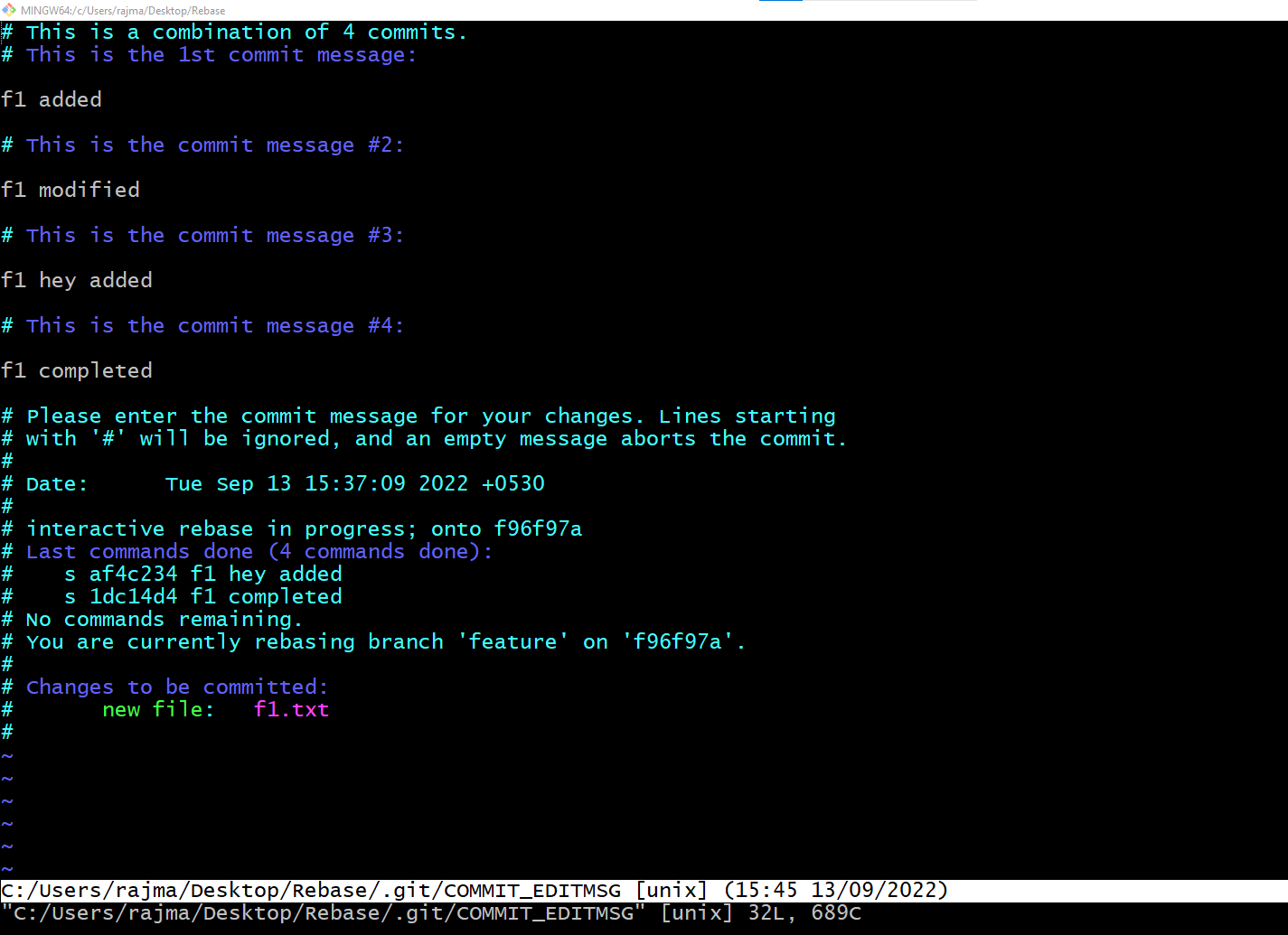
What happens in normal rebase is that all the commits of feature branch gets added to master branch as it is, but sometime when we need to modify commits while merging them in rebase operation we use interactive rebase.

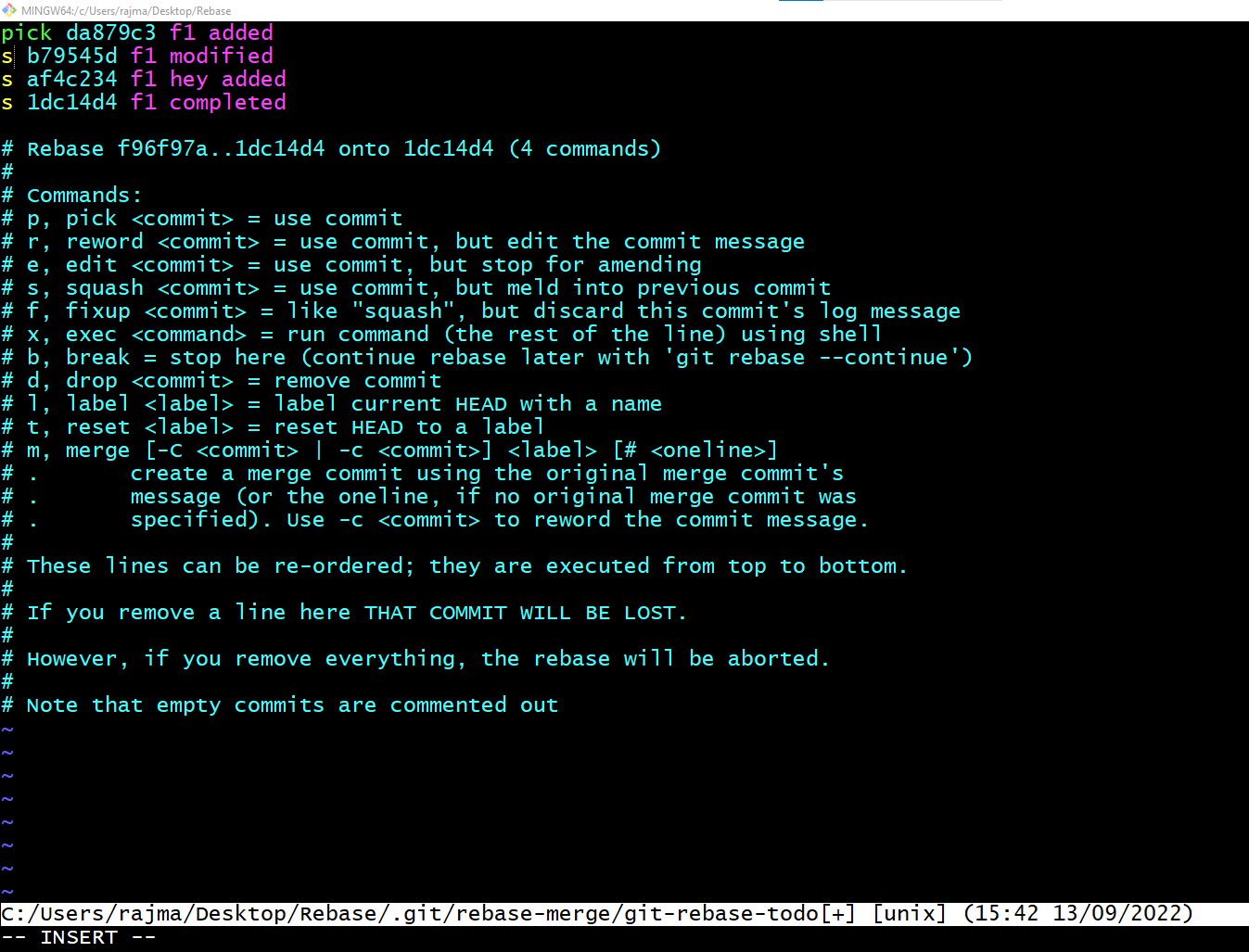
Command – git rebase –i master

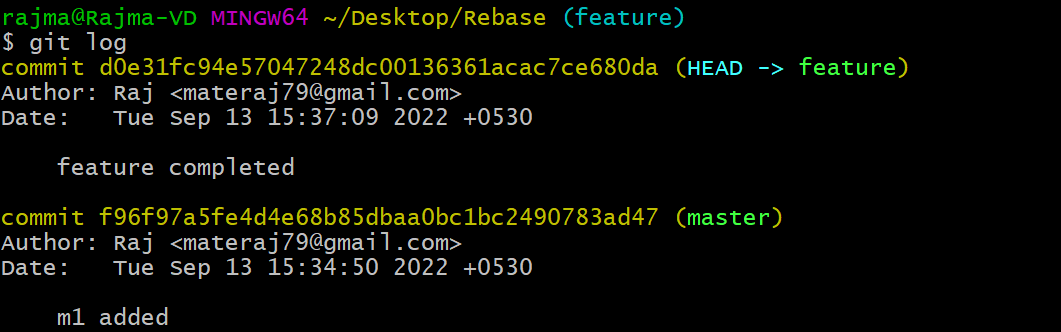
Here we can squash the commits and also do some more operations.

****

****

****

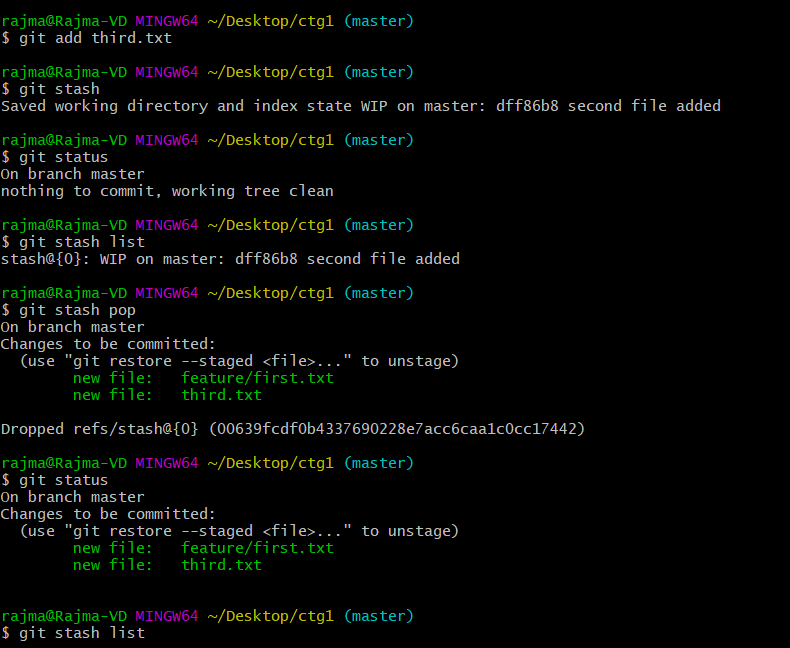
****

****

**j) What is git stash (Theory + POC)?**

We use stash command for save the incomplete task in stash area so that we can recall this later and do work on it again. We can use git stash pop to add the file to stage area again.

Command:git stash save “name”

****

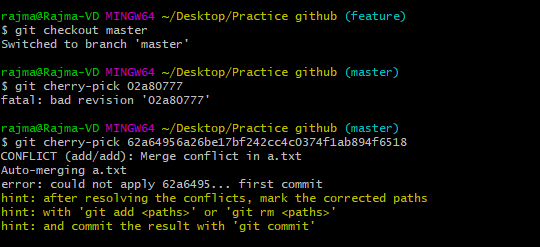
**k) What is merge conflict and how to resolve it (Theory + POC)?**

**Answer:** A merge conflict is an event that occurs when Git is unable to automatically resolve differences in code between two commits.

When all the changes in the code occur on different lines or in different files, Git will successfully merge commits without your help.

There are a few steps that could reduce the steps needed to resolve merge conflicts in Git.

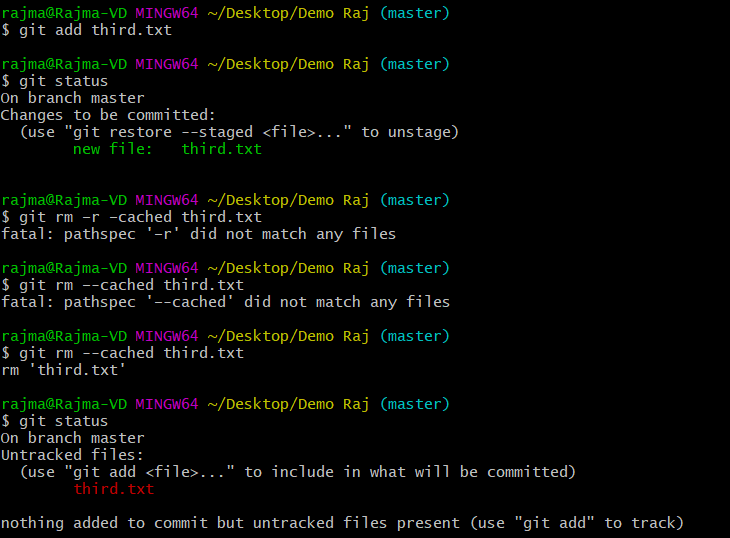
1. The easiest way to resolve a conflicted file is to open it and make any necessary changes
2. After editing the file, we can use the git add a command to stage the new merged content
3. The final step is to create a new commit with the help of the git commit command
4. Git will create a new merge commit to finalize the merge

****

**l) How can we untrack a file in git?**

By using following command we can untracked file from staging area.

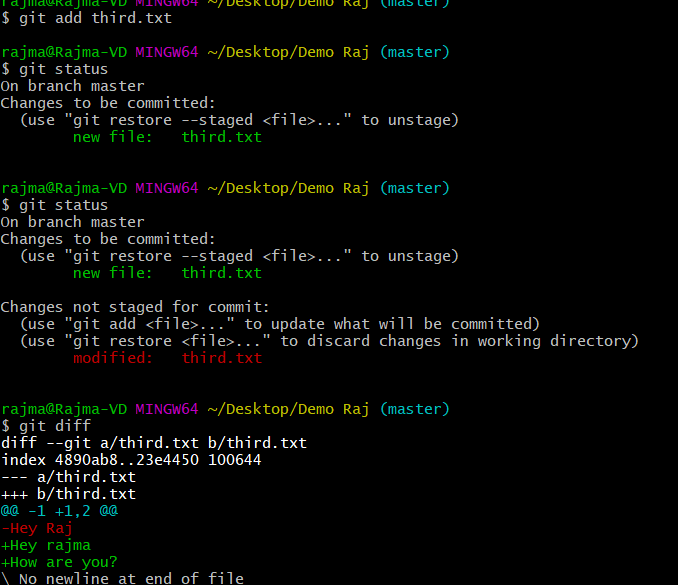
Command: git rm ––cached <file\_name>



**m) How can we see the difference between staging area and working directory?**

We can see the difference between staging area and working directory with the

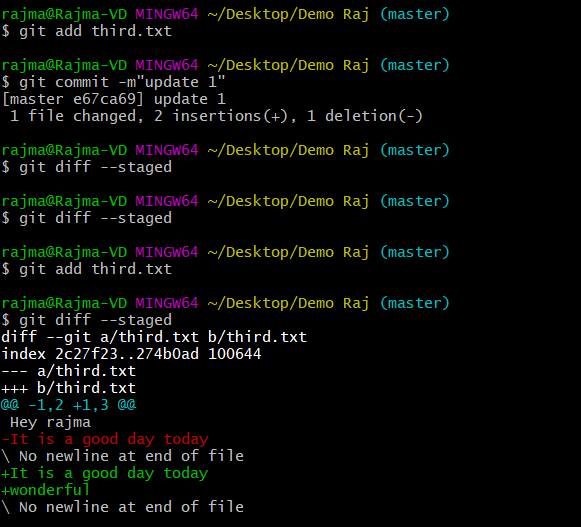
**Command - git diff**

****

**n) How can we see the difference between staging area and local repository?**

We can we see the difference between staging area and local repository with

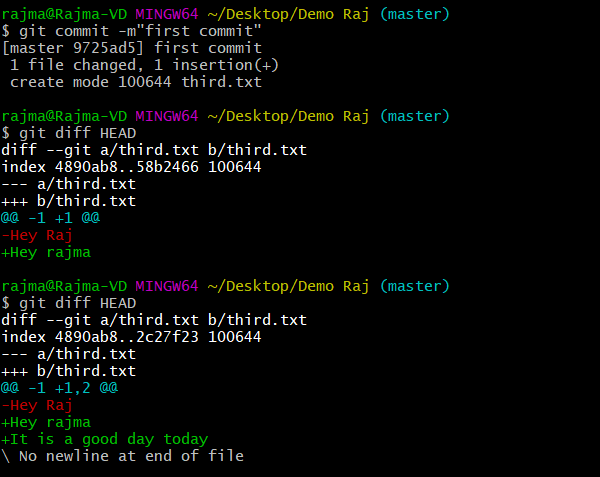
**Command: git diff –staged / git diff –cached**

****

**o) How can we see the difference between local repository and working directory?**

We see the difference between staging area and local repository by using

**Command: git diff HEAD**

****

**p) What is git-merge tool(Theory + POC)?**

A merge conflict is an event that occurs when git is unable to automatically resolve difference in code between two commit.

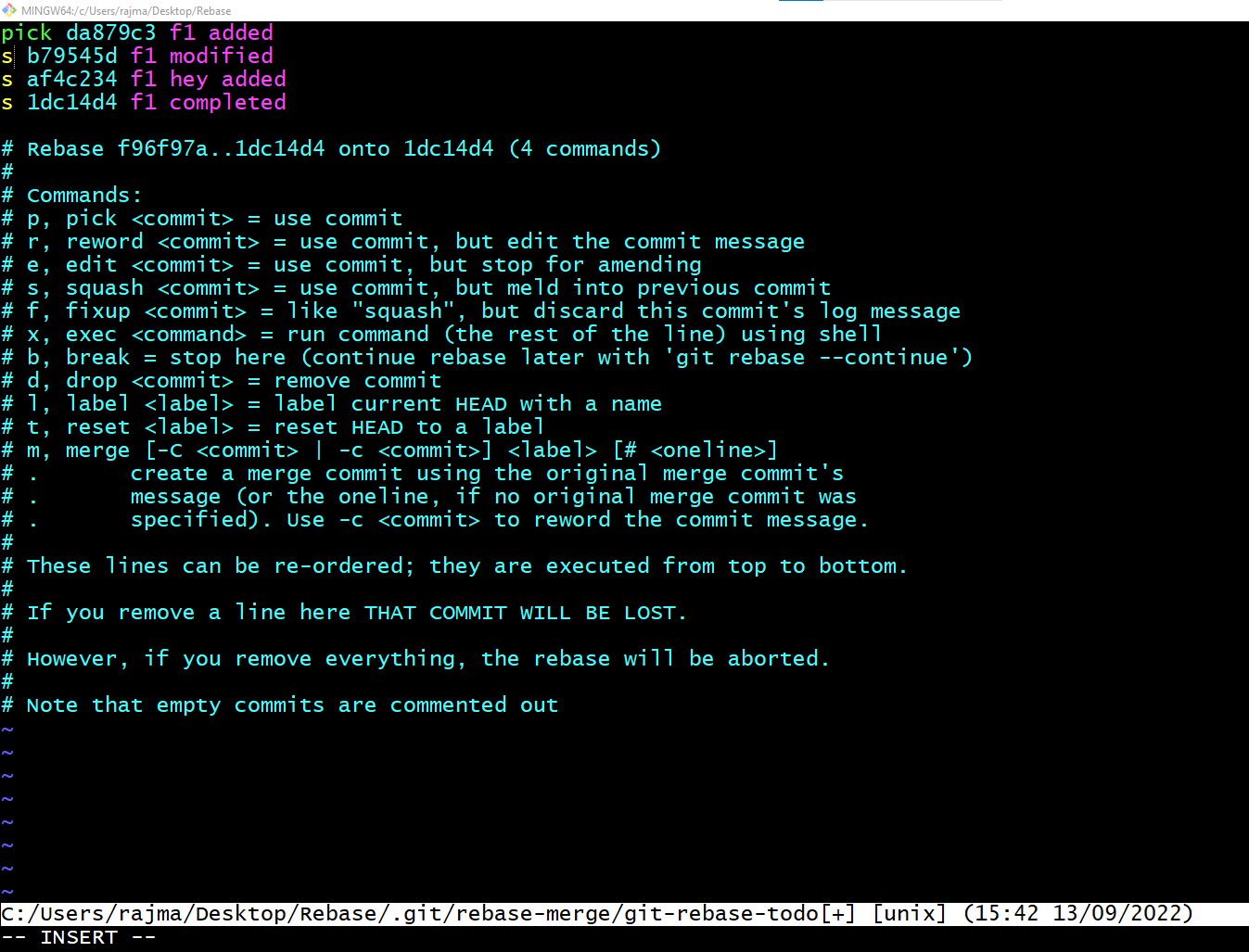
For example, I am in master branch I have one file on that file I added data and commit that file after that I create one feature branch and checkout on that branch and some changes on that file and when we merge that branch on master then we will get conflict.

So by using the **git mergetool** we resolve that conflict.

**q)Explore squash and merge(Theory + POC)?**

**Answer:-** Whenever we want to merge multiple commits into one commit we use git squash.

With squash we can change commit message also.

****

**s) What is pre-commit hook and Post-commit hook(Theory + POC)?**

**Answer-** Pre-commit hook is a script which runs even before we type the commit message.

It is used to inspect the code that is about to be committed.

Post-commit hook is a script which runs after the commit. It cannot change the outcome of the git command so it is primarily used for notification purpose.

**t) Write a pre-commit hook script for checking if the username and user email are configured correctly or not? If configured correctly commit should happen otherwise it should fail?**

**Answer:-**

#!/bin/sh

if [ “git config user.name” != “Rajmate16” ]

then

echo “incorrect username”

exit 1

fi

if [ “git config user.email” != “rajma@cybage.com” ]

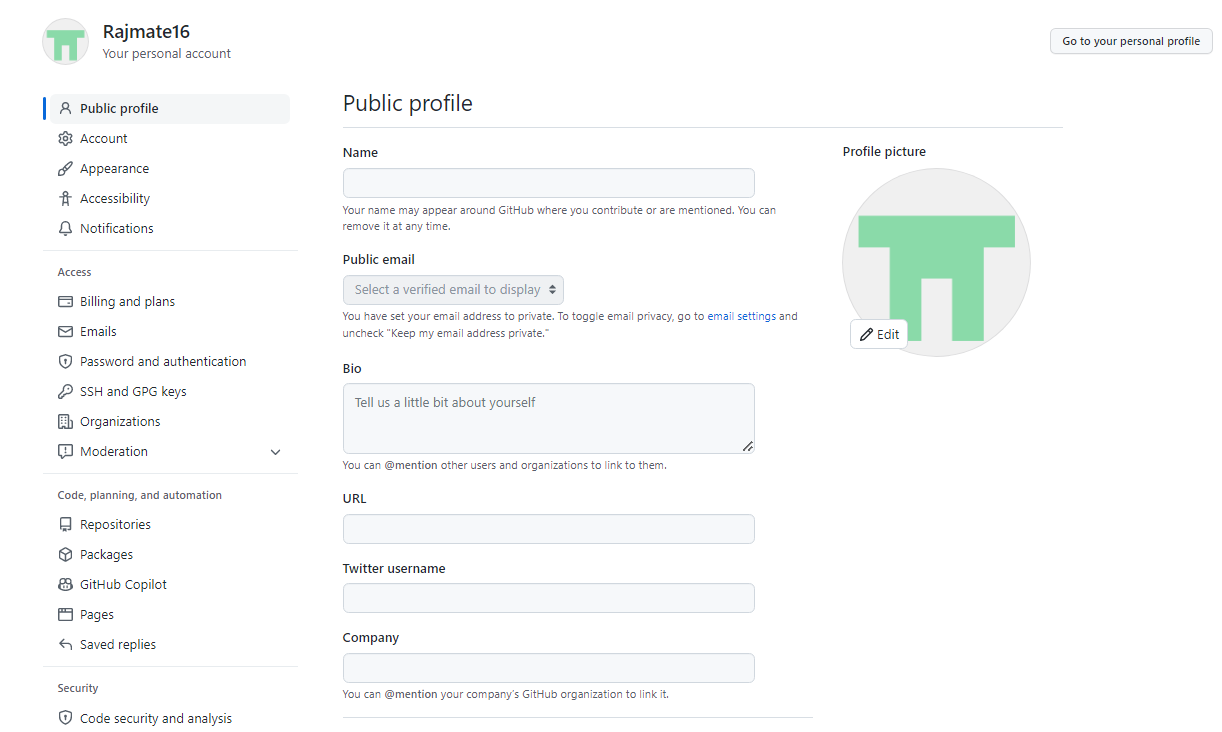
then

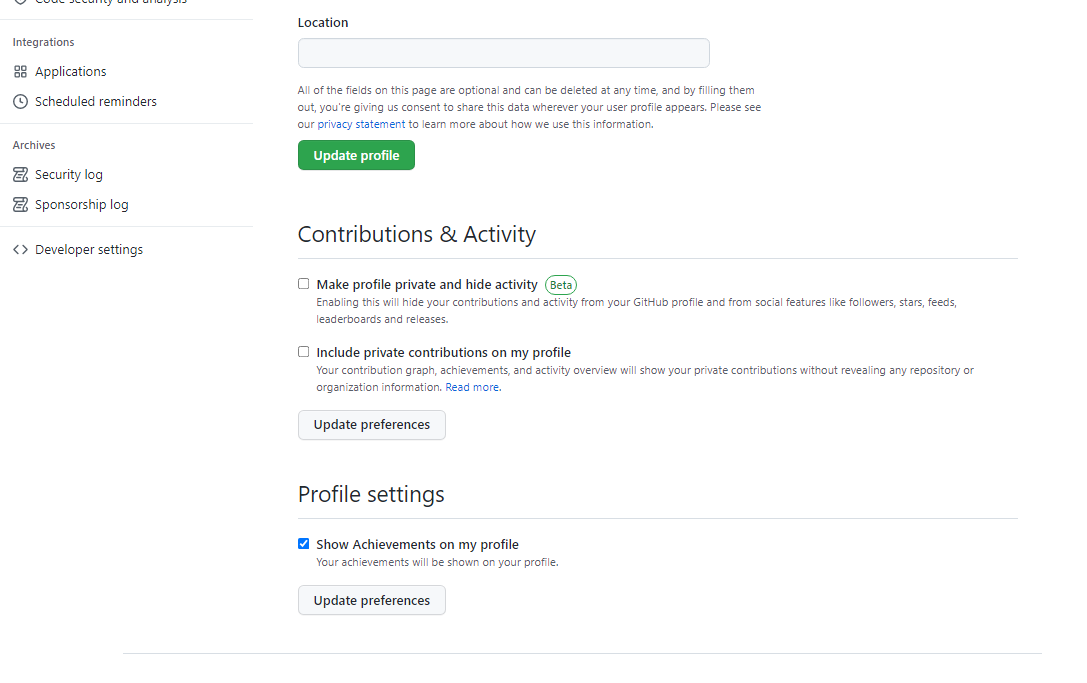
echo “Please add valid email”

exit 1

fi

**u)Explore git hub settings section?**

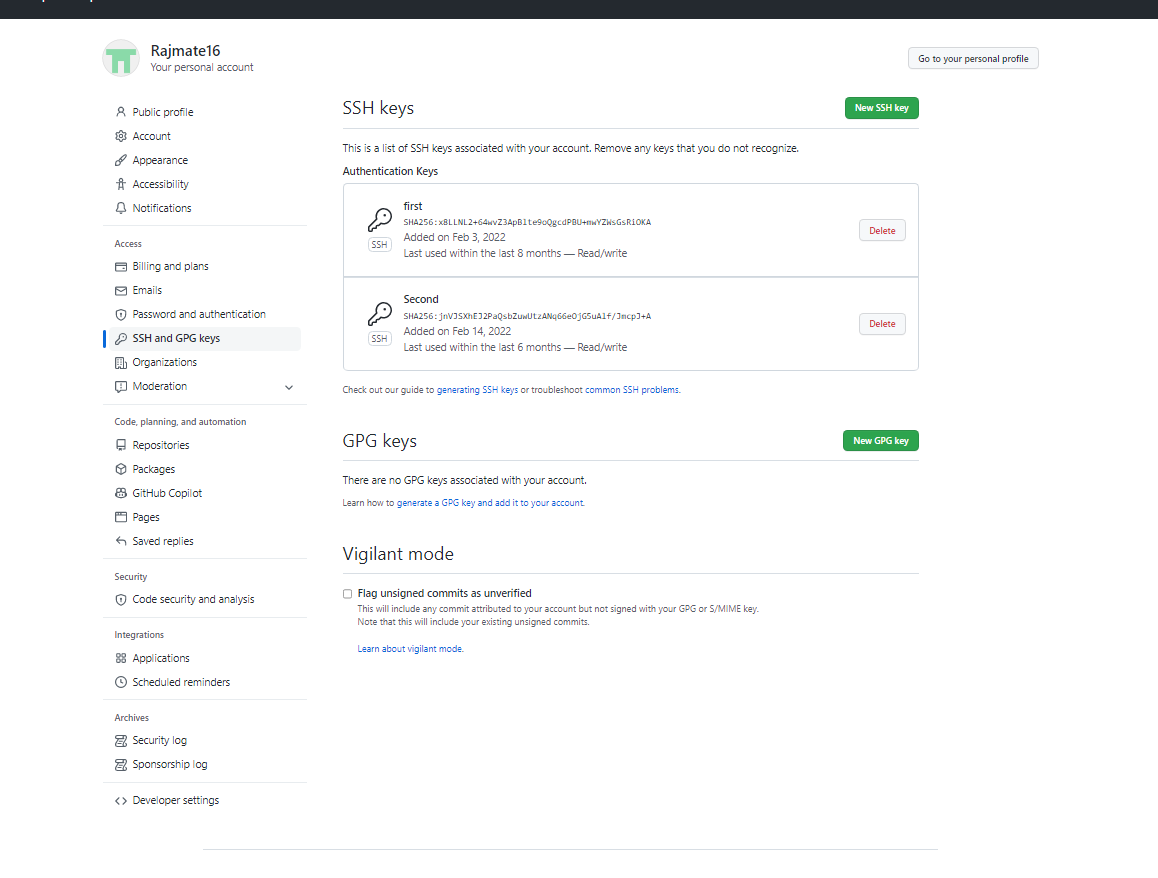
****

****

**v) clone the repository using ssh?**

**Answer :-**

Have already cloned a repository with ssh by generating a ssh key in github but unable to attach screenshot because port 22 is blocked.



**w) what are webhooks?**

A webhook is an HTTP request, triggered by an event in a source system and sent to a destination system, often with a payload of data. Webhooks are automated, in other words they are automatically sent out when their event is fired in the source system.

This provides a way for one system (the source) to "speak" (HTTP request) to another system (the destination) when an event occurs, and share information (request payload) about the event that occurred.

**x) What are different workflows of git?**

**Centralized workflow:**

The Centralized Workflow uses a central repository to serve as the single point-of-entry for all changes to the project. Instead of trunk, the default development branch is called main and all changes are committed into this branch. This workflow doesn’t require any other branches besides main.

**Feature Branching:**

Feature Branching is a logical extension of Centralized Workflow. The core idea behind the Feature Branch Workflow is that all feature development should take place in a dedicated branch instead of the main branch. This encapsulation makes it easy for multiple developers to work on a particular feature without disturbing the main codebase. It also means the main branch should never contain broken code, which is a huge advantage for continuous integration environments.

**Forking Workflow :**

The Forking Workflow is fundamentally different than the other workflows discussed in this tutorial. Instead of using a single server-side repository to act as the “central” codebase, it gives every developer a server-side repository. This means that each contributor has not one, but two Git repositories: a private local one and a public server-side one.

**Gitflow Workflow :**

The Gitflow Workflow defines a strict branching model designed around the project release. This workflow doesn’t add any new concepts or commands beyond what’s required for the Feature Branch Workflow. Instead, it assigns very specific roles to different branches and defines how and when they should interact.

**y) Can we fork a private repository if yes how?**

Yes we can fork the private repository by adding collaborator to the repository.

**z) Study gitflow workflowin detail?**

**Gitflow workflow has 5 branches, they are-**

Develop and Main Branch :- Instead of a single main branch, this workflow uses two branches to record the history of the project. The main branch stores the official release history, and the develop branch serves as an integration branch for features. It's also convenient to tag all commits in the main branch with a version number.

**Feature Branch :-** Each new feature should reside in its own branch, which can be [pushed to the central repository](https://www.atlassian.com/git/tutorials/syncing/git-push) for backup/collaboration. But, instead of branching off of main, feature branches use develop as their parent branch. When a feature is complete, it gets [merged back into develop](https://www.atlassian.com/git/tutorials/using-branches/git-merge). Features should never interact directly with main.

**Release Branch :-** Once develop has acquired enough features for a release (or a predetermined release date is approaching), you fork a release branch off of develop. Creating this branch starts the next release cycle, so no new features can be added after this point—only bug fixes, documentation generation, and other release-oriented tasks should go in this branch. Once it's ready to ship, the release branch gets merged into main and tagged with a version number.

**Hotfix Branch :-** Maintenance or “hotfix” branches are used to quickly patch production releases. Hotfix branches are a lot like release branches and feature branches except they're based on main instead of develop. This is the only branch that should fork directly off of main. As soon as the fix is complete, it should be merged into both main and develop (or the current release branch), and main should be tagged with an updated version number.