Git POC

a) Explore .git folder.

COMMIT_EDITMSG: This is the last commit's message. It's not actually used by Git at all, but it's there mostly for your reference after you made a commit.
config: Contains settings for this repository. Specific configuration variables can be dumped in here (and even aliases!) What this file is most used for is defining where remotes live and some core settings, such as if your repository is bare or not.
description: If you're using gitweb or firing up git instaweb, this will show up when you view your repository or the list of all versioned repositories.
FETCH_HEAD: The SHAs of branch/remote heads that were updated during the last git fetch
HEAD: The current ref that you're looking at. In most cases it's probably refs/heads/master
index: The staging area with meta-data such as timestamps, file names and also SHAs of the files that are already wrapped up by Git.
packed-refs: Packs away dormant refs, this is not the definitive list of refs in your repository (the refs folder has the real ones!) Take a look at gitster's comment to see more information on this.
ORIG_HEAD: When doing a merge, this is the SHA of the branch you're merging into.
MERGE_HEAD: When doing a merge, this is the SHA of the branch you're merging from.
MERGE_MODE: Used to communicate constraints that were originally given to git merge to git commit when a merge conflicts, and a separate git commit is needed to conclude it. Currently --no-ff is the only constraints passed this way.
MERGE_MSG: Enumerates conflicts that happen during your current merge.
RENAMED-REF: Still trying to track this one down. From a basic grep through the source, it seems like this file is related to errors when saving refs.
There's plenty of directories as well:
hooks: A directory that will fast become your best friend: this contains scripts that are executed at certain times when working with Git, such as after a commit or before a rebase. An entire series of articles will be coming about hooks.
info: Relatively uninteresting except for the exclude file that lives inside of it. We've seen this before in the ignoring files article, but as a reminder, you can use this file to ignore files for this project, but beware! It's not versioned like a .gitignore file would be.
logs: Contains history for different branches. Seems to be used mostly with the reflog command.
objects: Git's internal warehouse of blobs, all indexed by SHAs.
rebase-apply: The workbench for rebasing and for git am. You can dig into its patch file when it does not apply cleanly if you're brave.
refs: The master copy of all refs that live in your repository, be they for stashes, tags, remote tracking branches, or local branches.

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

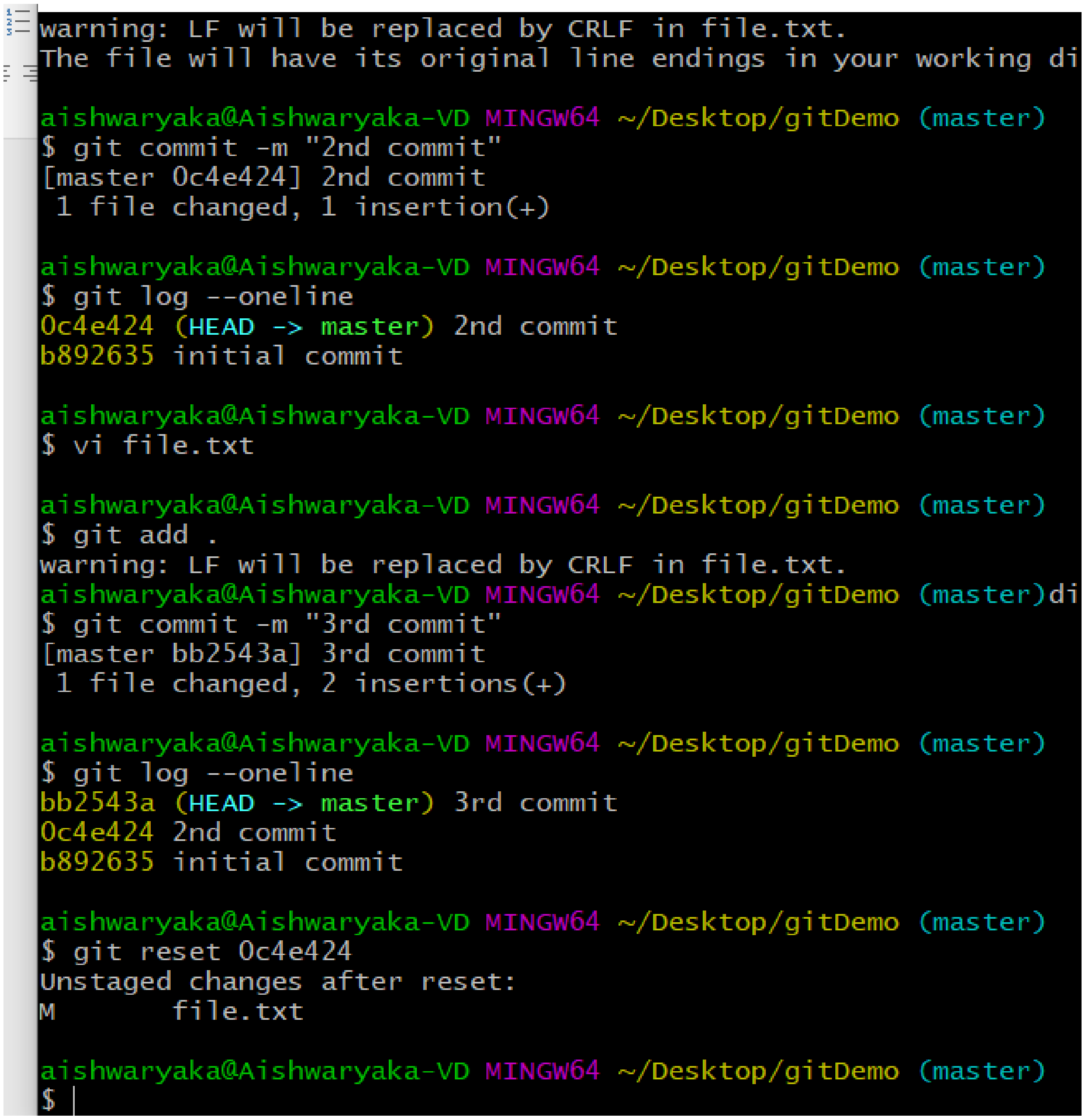
Reset is the command we use when we want to move the repository back to a previous commit, discarding any changes made after that commit. We need to find the point we want to return to. So, we can give the id where we are returning to.

This command is a little more complicated. It actually does a couple of different things depending on how it is invoked. It modifies the index (the so-called "staging area"). Or it changes which commit a branch head is currently pointing at. This command may alter existing history (by changing the commit that a branch references).

Git reset –soft head~1 – this command will remove the commit but would not unstage a file. Our changes still would be in the staging area.

Git reset –mixed head~1 or ***git reset head~1*** **–** this is the default command that we have used in the above example which removes the commit as well as unstages the file and our changes are stored in the working directory.

Git reset –hard head~1 **–** this command removes the commit as well as the changes from your working directory. This command can also be called destructive command as we would not be able to get back the changes so be careful while using this command.

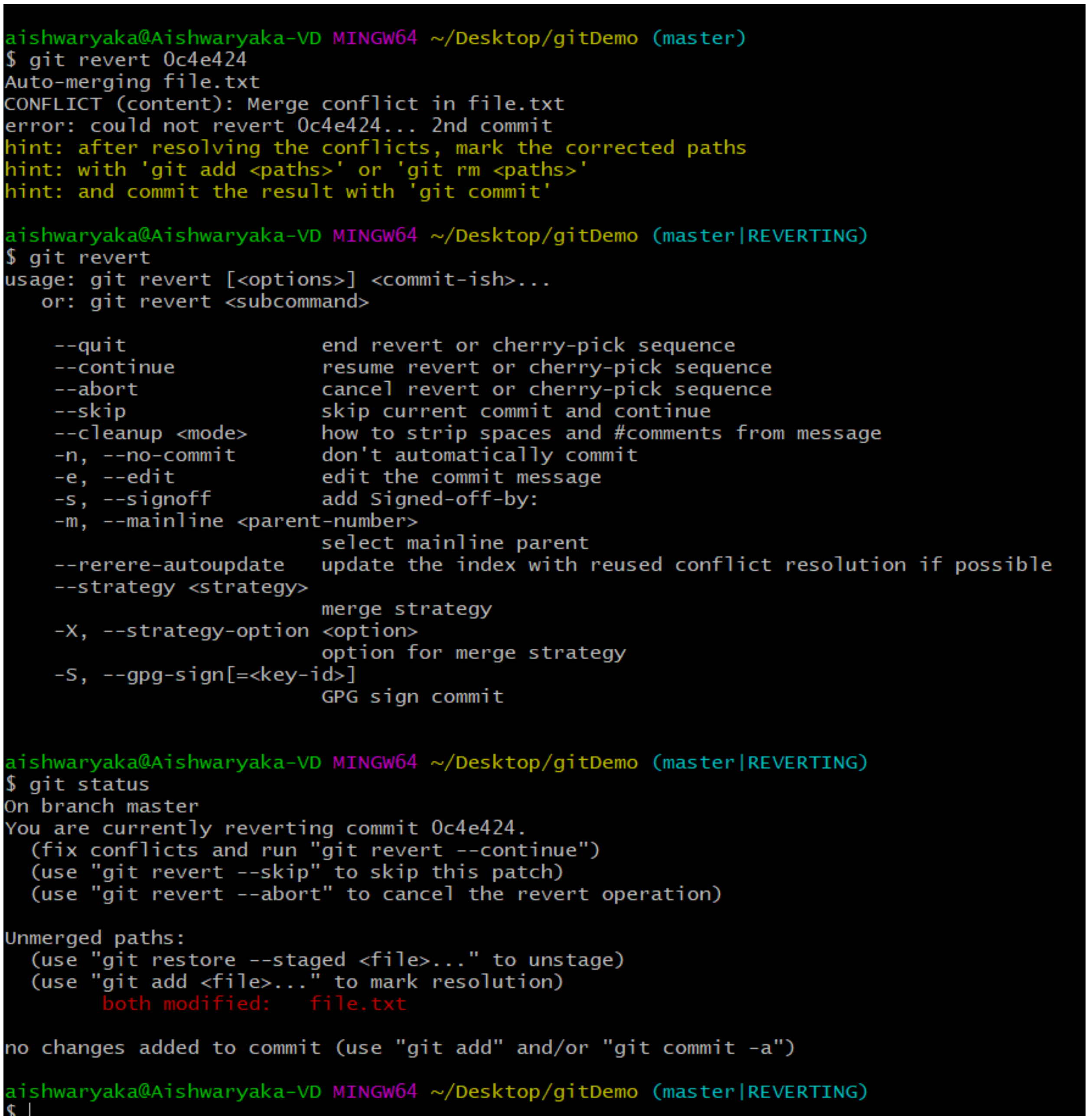


```
warning: LF will be replaced by CRLF in file.txt.
The file will have its original line endings in your working di

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git commit -m "2nd commit"
[master 0c4e424] 2nd commit
 1 file changed, 1 insertion(+)

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git log --oneline
0c4e424 (HEAD -> master) 2nd commit
b892635 initial commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ vi file.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git add .
warning: LF will be replaced by CRLF in file.txt.
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)di
$ git commit -m "3rd commit"
[master bb2543a] 3rd commit
 1 file changed, 2 insertions(+)

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git log --oneline
bb2543a (HEAD -> master) 3rd commit
0c4e424 2nd commit
b892635 initial commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git reset 0c4e424
Unstaged changes after reset:
M       file.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ |
```

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

It is an "undo" command, but technically it is much more than that. **Git revert** does not delete any commit in this project history. Instead, it inverts the changes implemented in a commit and appends new commits with the opposite effect. This process helps Git remove the unwanted commit from the codebase and retain the history of every commit and the reverted one. This makes it a handy command, especially when collaborating on a project.This command creates a new commit that undoes the changes from a previous commit. This command adds new history to the project (it doesn't modify existing history).

Now we want to delete the commit that we just added to the remote repository. We could have used the git reset command but that would have deleted the commit just from the local repository and not the remote repository. If we do this then we would get conflict that the remote commit is not present locally. So, we do not use git reset here. The best we can use here is git revert.



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git revert 0c4e424
Auto-merging file.txt
CONFLICT (content): Merge conflict in file.txt
error: could not revert 0c4e424... 2nd commit
hint: after resolving the conflicts, mark the corrected paths
hint: with 'git add <paths>' or 'git rm <paths>'
hint: and commit the result with 'git commit'

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|REVERTING)
$ git revert
usage: git revert [<options>] <commit-ish>...
   or: git revert <subcommand>

    --quit                end revert or cherry-pick sequence
    --continue            resume revert or cherry-pick sequence
    --abort               cancel revert or cherry-pick sequence
    --skip                skip current commit and continue
    --cleanup <mode>      how to strip spaces and #comments from message
    -n, --no-commit       don't automatically commit
    -e, --edit            edit the commit message
    -s, --signoff         add Signed-off-by:
    -m, --mainline <parent-number>
                          select mainline parent
    --rerere-autoupdate   update the index with reused conflict resolution if possible
    --strategy <strategy>
                          merge strategy
    -X, --strategy-option <option>
                          option for merge strategy
    -S, --gpg-sign[=<key-id>]
                          GPG sign commit


aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|REVERTING)
$ git status
On branch master
You are currently reverting commit 0c4e424.
  (fix conflicts and run "git revert --continue")
  (use "git revert --skip" to skip this patch)
  (use "git revert --abort" to cancel the revert operation)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both modified:   file.txt

no changes added to commit (use "git add" and/or "git commit -a")

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|REVERTING)
$ |
```

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

Git revert is used to undo a previous commit. In git, you can't alter or erase an earlier commit. (actually you can, but it can cause problems.) So instead of editing the earlier commit, revert introduces a new commit that reverses an earlier one.

Git revert is used to remove the commits from the remote repository and our changes are now in working directory.

Git reset is used to undo changes in your working directory that haven't been comitted yet

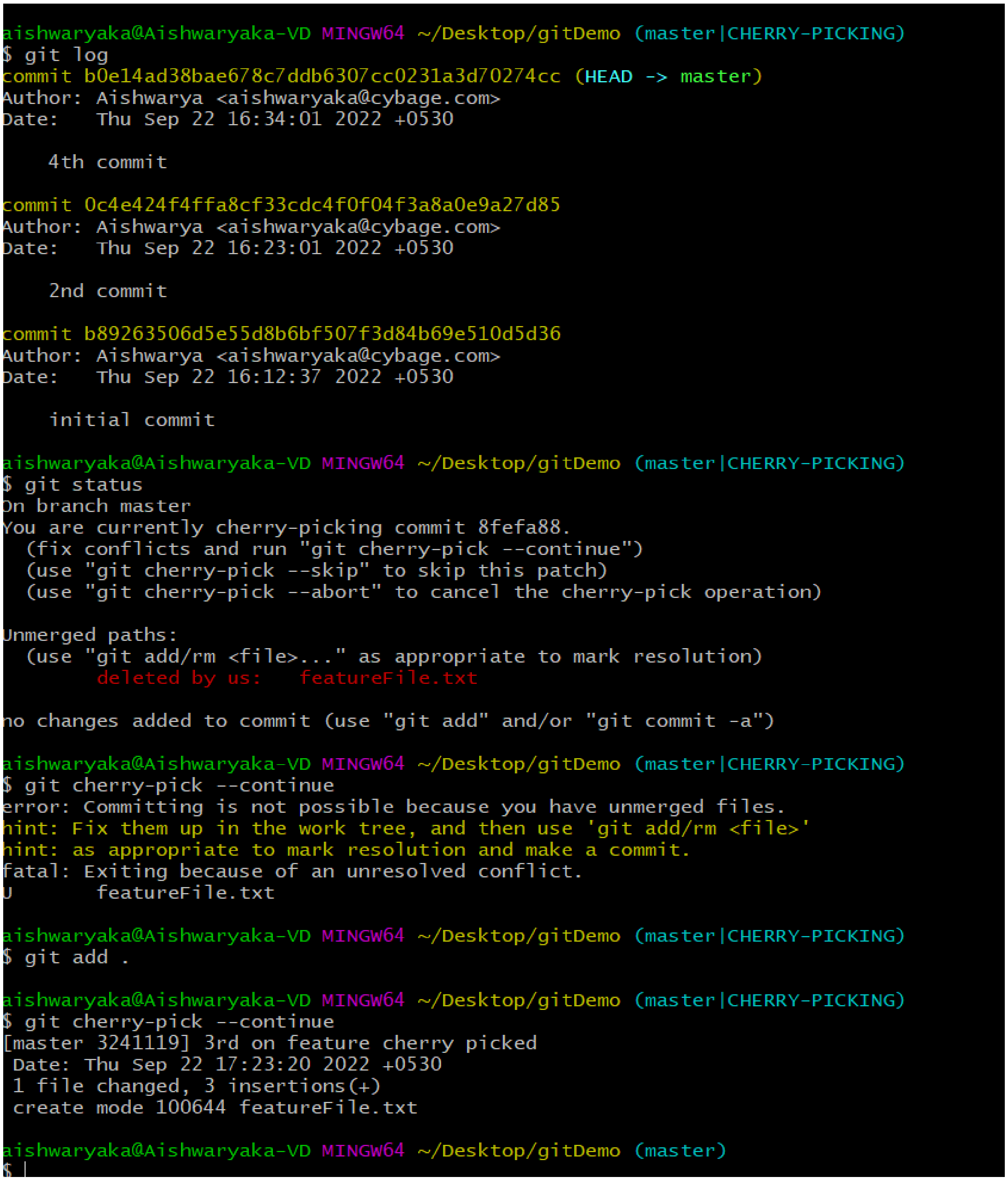
Git reset is used when we want to unstage a file and bring our changes back to the working directory. Git reset can also be used to remove commits from the local repository.

e) what is git cherrypick(Theory + POC)?

You can cherry-pick a commit on one branch to create a copy of the commit with the same changes on another branch. If you commit changes to the wrong branch or want to make the same changes to another branch, you can cherry-pick the commit to apply the changes to another branch. You can also use cherry-picking to apply specific changes before you are ready to create or merge a pull request. For example, if you commit a bug fix to a feature branch, you can cherry-pick the commit with the bug fix to other branches of your project.

Git cherry-pick is a powerful command that enables arbitrary Git commits to be picked by reference and appended to the current working HEAD. Cherry picking is the act of picking a commit from a branch and applying it to another. Git cherry-pick can be useful for undoing changes. For example, say a commit is accidently made to the wrong branch. You can switch to the correct branch and cherry-pick the commit to where it should belong.

Git cherry-pick is a useful tool but not always a best practice. Cherry picking can cause duplicate commits and many scenarios where cherry picking would work, traditional merges are preferred instead.



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|CHERRY-PICKING)
$ git log
commit b0e14ad38bae678c7ddb6307cc0231a3d70274cc (HEAD -> master)
Author: Aishwarya <aishwaryaka@cybage.com>
Date:   Thu Sep 22 16:34:01 2022 +0530

    4th commit

commit 0c4e424f4ffa8cf33cdc4f0f04f3a8a0e9a27d85
Author: Aishwarya <aishwaryaka@cybage.com>
Date:   Thu Sep 22 16:23:01 2022 +0530

    2nd commit

commit b89263506d5e55d8b6bf507f3d84b69e510d5d36
Author: Aishwarya <aishwaryaka@cybage.com>
Date:   Thu Sep 22 16:12:37 2022 +0530

    initial commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|CHERRY-PICKING)
$ git status
On branch master
You are currently cherry-picking commit 8fefa88.
  (fix conflicts and run "git cherry-pick --continue")
  (use "git cherry-pick --skip" to skip this patch)
  (use "git cherry-pick --abort" to cancel the cherry-pick operation)

Unmerged paths:
  (use "git add/rm <file>..." as appropriate to mark resolution)
        deleted by us:   featureFile.txt

no changes added to commit (use "git add" and/or "git commit -a")

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|CHERRY-PICKING)
$ git cherry-pick --continue
error: Committing is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
U       featureFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|CHERRY-PICKING)
$ git add .

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|CHERRY-PICKING)
$ git cherry-pick --continue
[master 3241119] 3rd on feature cherry picked
 Date: Thu Sep 22 17:23:20 2022 +0530
 1 file changed, 3 insertions(+)
 create mode 100644 featureFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ |
```

f)Explain different types of merging strategies?

Having multiple branches is extremely convenient to keep new changes separated from each other, and to make sure you don't accidentally push unapproved or broken changes to production. Once the changes have been approved, we want to get these changes in our production branch!

One way to get the changes from one branch to another is by performing a git merge! There are two types of merges Git can perform: a **fast-forward**, or a **no-fast-forward**.

This may not make a lot of sense right now, so let's look at the differences!

Fast-forward (--ff)

A **fast-forward merge** can happen when the current branch has no extra commits compared to the branch we're merging. Git is... *lazy* and will first try to perform the easiest option: the fast-forward! This type of merge doesn't create a new commit, but rather merges the commit(s) on the branch we're merging right in the current branch.

We now have all the changes that were made on the dev branch available on the master branch. So, what's the **no-fast-forward** all about?

No-fast-foward (--no-ff)

It's great if your current branch doesn't have any extra commits compared to the branch that you want to merge, but unfortunately that's rarely the case! If we committed changes on the current branch that the branch we want to merge doesn't have, git will perform a *no-fast-forward* merge.

With a no-fast-forward merge, Git creates a new *merging commit* on the active branch. The commit's parent commits point to both the active branch and the branch that we want to merge!

No big deal, a perfect merge! 🎉 The master branch now contains all the changes that we've made on the dev branch.

Merge Conflicts

Although Git is good at deciding how to merge branches and add changes to files, it cannot always make this decision all by itself ☺ This can happen when the two branches we're trying to merge have changes on the same line in the same file, or if one branch deleted a file that another branch modified, and so on.

In that case, Git will ask you to help decide which of the two options we want to keep! Let's say that on both branches, we edited the first line in the README.md.

If we want to merge dev into master, this will end up in a merge conflict.When trying to merge the branches, Git will show you where the conflict happens. We can manually remove the changes we don't want to keep, save the changes, add the changed file again, and commit the changes. Although merge conflicts are often quite annoying, it makes total sense: Git shouldn't just *assume* which change we want to keep.

Rebasing
We just saw how we could apply changes from one branch to another by performing a git merge. Another way of adding changes from one branch to another is by performing a git rebase.

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

Rebasing

We just saw how we could apply changes from one branch to another by performing a git merge. Another way of adding changes from one branch to another is by performing a git rebase.

A git rebase *copies* the commits from the current branch, and puts these copied commits on top of the specified branch.

Perfect, we now have all the changes that were made on the master branch available on the dev branch!

A big difference compared to merging, is that Git won't try to find out which files to keep and not keep. The branch that we're rebasing always has the latest changes that we want to keep! You won't run into any merging conflicts this way, and keeps a nice linear Git history.

This example shows rebasing on the master branch. In bigger projects, however, you usually don't want to do that. A git rebase **changes the history of the project** as new hashes are created for the copied commits!

Rebasing is great whenever you're working on a feature branch, and the master branch has been updated. You can get all the updates on your branch, which would prevent future merging conflicts!

Interactive Rebase

Before rebasing the commits, we can modify them! We can do so with an *interactive rebase*. An interactive rebase can also be useful on the branch you're currently working on, and want to modify some commits.

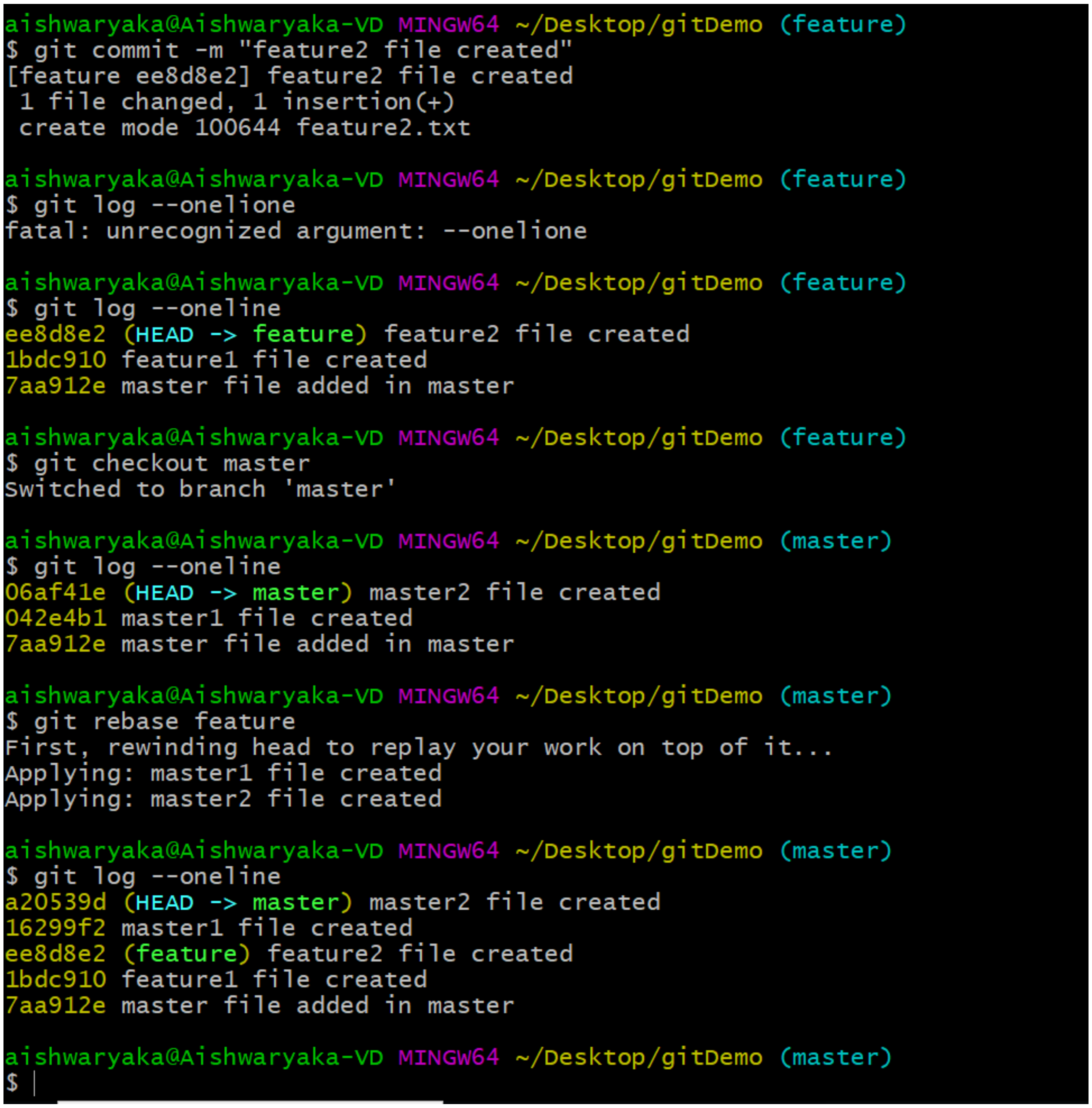
There are 6 actions we can perform on the commits we're rebasing:

- reword: Change the commit message
- edit: Amend this commit
- squash: Meld commit into the previous commit
- fixup: Meld commit into the previous commit, without keeping the commit's log message
- exec: Run a command on each commit we want to rebase
- drop: Remove the commit

This way, we can have full control over our commits. If we want to remove a commit, we can just drop it.

Or if we want to squash multiple commits together to get a cleaner history, no problem!

Interactive rebasing gives you a lot of control over the commits you're trying to rebase, even on the current active branch!



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git commit -m "feature2 file created"
[feature ee8d8e2] feature2 file created
 1 file changed, 1 insertion(+)
 create mode 100644 feature2.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git log --onelione
fatal: unrecognized argument: --onelione

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git log --oneline
ee8d8e2 (HEAD -> feature) feature2 file created
1bdc910 feature1 file created
7aa912e master file added in master

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git checkout master
Switched to branch 'master'

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git log --oneline
06af41e (HEAD -> master) master2 file created
042e4b1 master1 file created
7aa912e master file added in master

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git rebase feature
First, rewinding head to replay your work on top of it...
Applying: master1 file created
Applying: master2 file created

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git log --oneline
a20539d (HEAD -> master) master2 file created
16299f2 master1 file created
ee8d8e2 (feature) feature2 file created
1bdc910 feature1 file created
7aa912e master file added in master

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ 
```

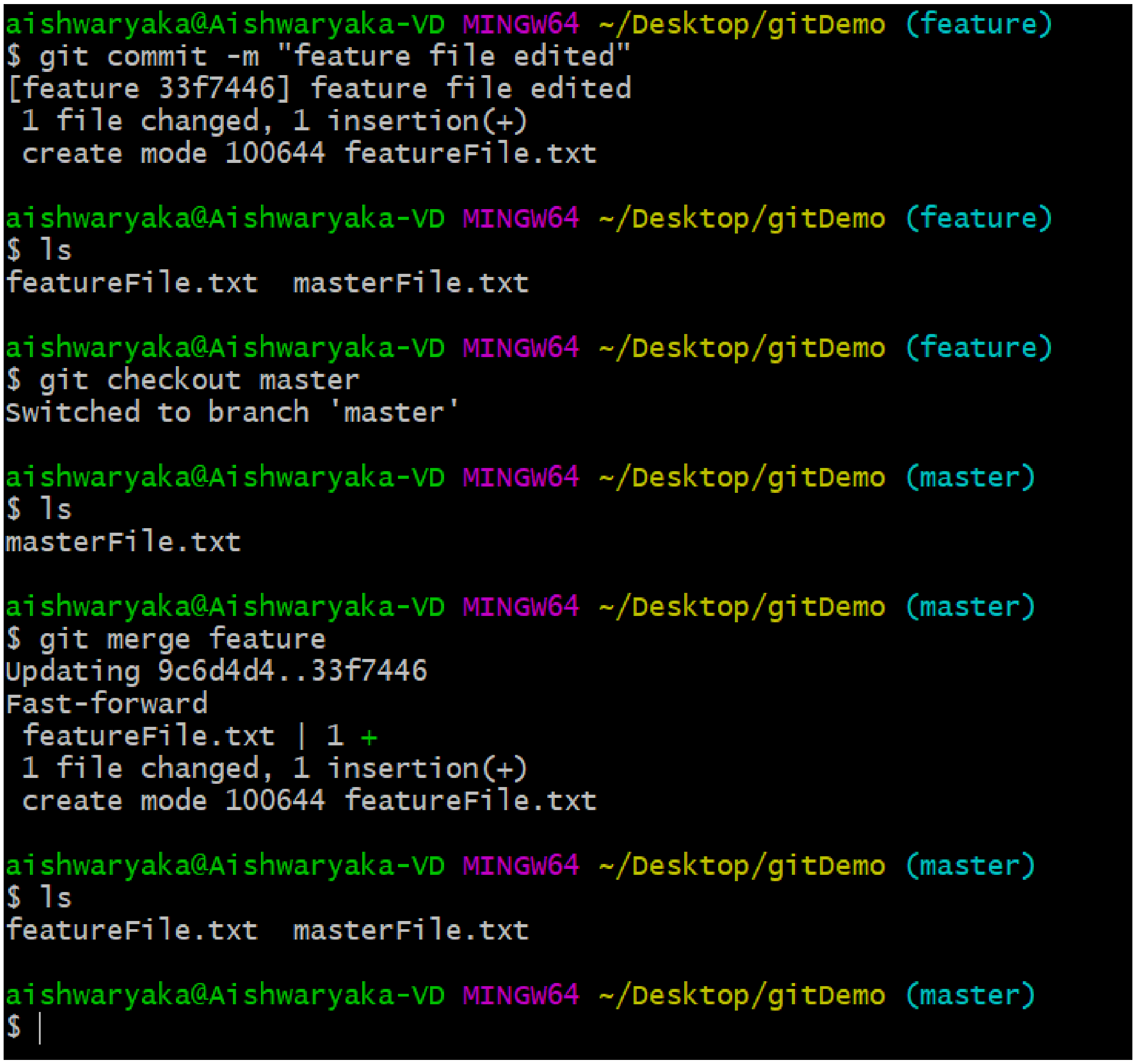
h)What is difference between fast forward and No fast forward merge(Theory + POC)?

A fast-forward merge can happen when the current branch has no extra commits compared to the branch we're merging.

No-fast-foward (--no-ff)

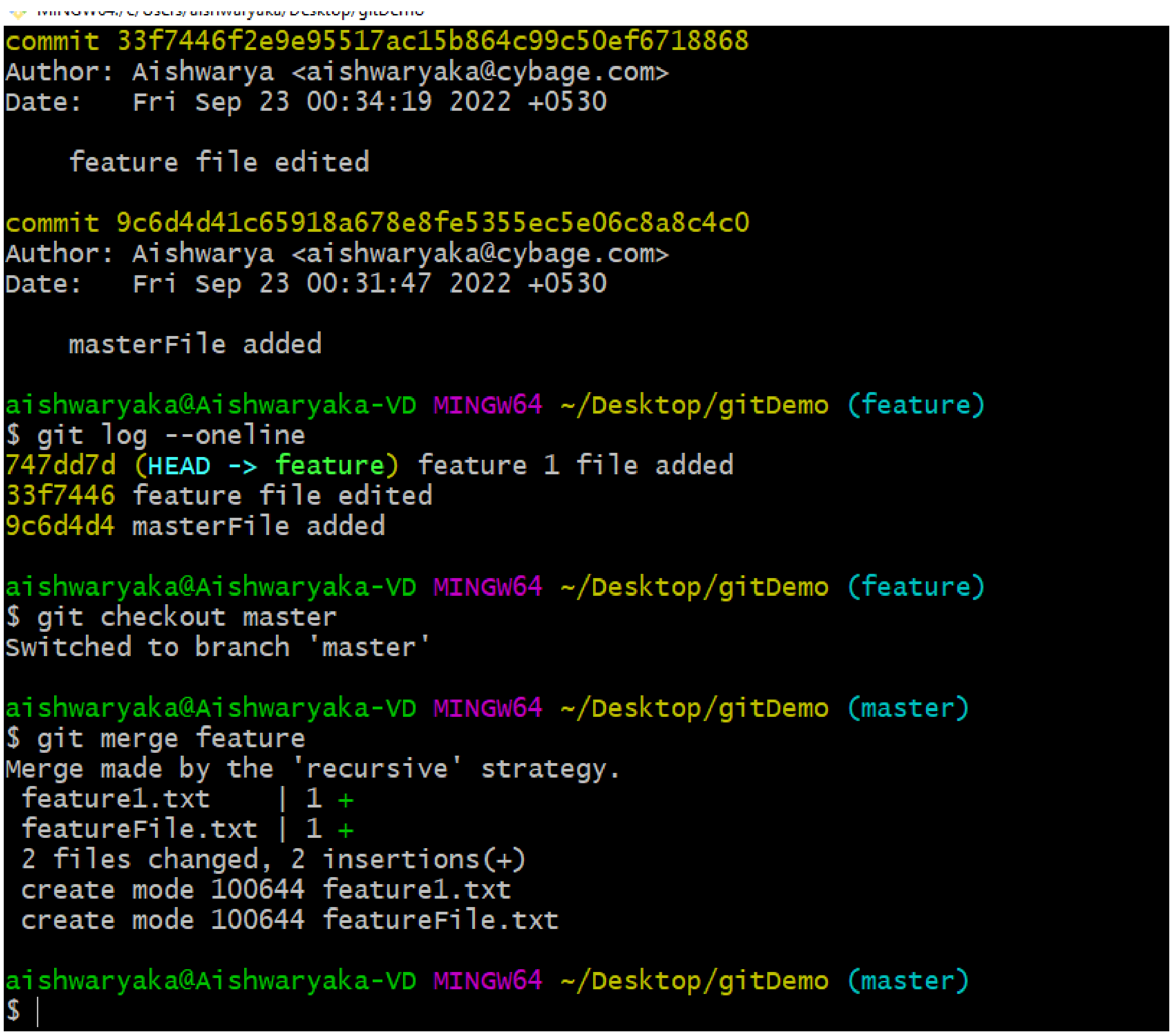
With a no-fast-forward merge, Git creates a new *merging commit* on the active branch. The commit's parent commits point to both the active branch and the branch that we want to merge!

Fast Forward merge:



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git commit -m "feature file edited"
[feature 33f7446] feature file edited
 1 file changed, 1 insertion(+)
 create mode 100644 featureFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ ls
featureFile.txt  masterFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git checkout master
Switched to branch 'master'

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ ls
masterFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git merge feature
Updating 9c6d4d4..33f7446
Fast-forward
 featureFile.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 featureFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ ls
featureFile.txt  masterFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ |
```

No fast forward merge (Recursive):



```
commit 33f7446f2e9e95517ac15b864c99c50ef6718868
Author: Aishwarya <aishwaryaka@cybage.com>
Date:   Fri Sep 23 00:34:19 2022 +0530

    feature file edited

commit 9c6d4d41c65918a678e8fe5355ec5e06c8a8c4c0
Author: Aishwarya <aishwaryaka@cybage.com>
Date:   Fri Sep 23 00:31:47 2022 +0530

    masterFile added

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git log --oneline
747dd7d (HEAD -> feature) feature 1 file added
33f7446 feature file edited
9c6d4d4 masterFile added

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git checkout master
Switched to branch 'master'

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git merge feature
Merge made by the 'recursive' strategy.
 feature1.txt    | 1 +
 featureFile.txt | 1 +
 2 files changed, 2 insertions(+)
 create mode 100644 feature1.txt
 create mode 100644 featureFile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ |
```

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

A git rebase *copies* the commits from the current branch, and puts these copied commits on top of the specified branch.

Interactive Rebase

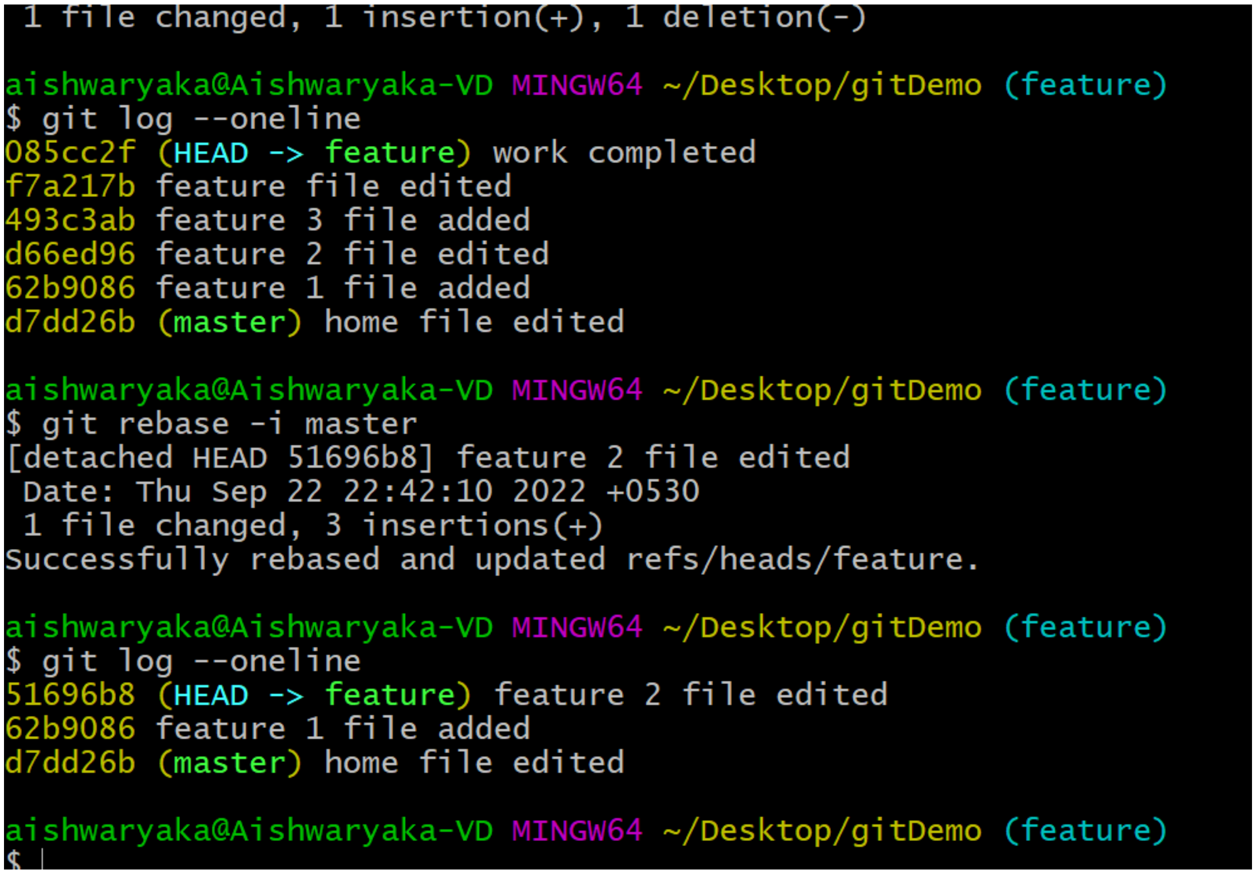
Before rebasing the commits, we can modify them! 😊 We can do so with an *interactive rebase*. An interactive rebase can also be useful on the branch you're currently working on, and want to modify some commits.

There are 6 actions we can perform on the commits we're rebasing:

- reword: Change the commit message

- edit: Amend this commit
- squash: Meld commit into the previous commit
- fixup: Meld commit into the previous commit, without keeping the commit's log message
- exec: Run a command on each commit we want to rebase
- drop: Remove the commit

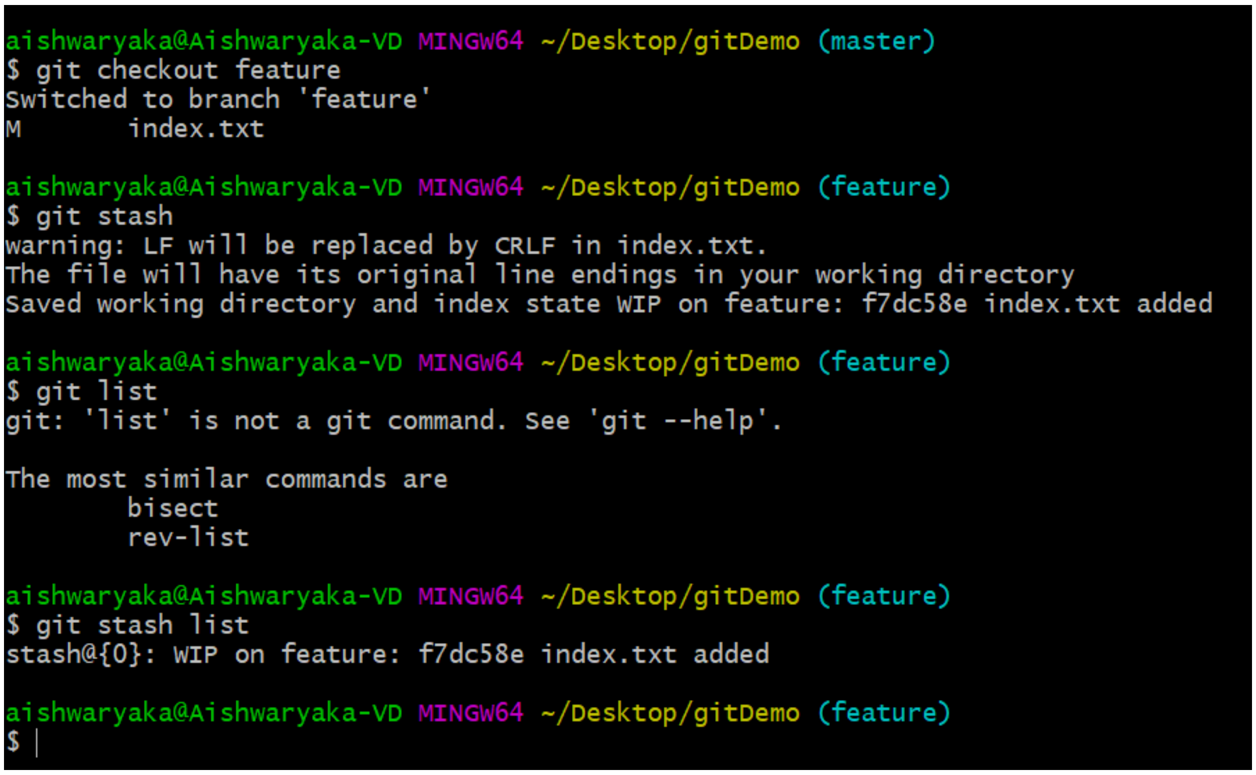
Awesome! This way, we can have full control over our commits. If we want to remove a commit, we can just drop it.



```
 1 file changed, 1 insertion(+), 1 deletion(-)

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git log --oneline
085cc2f (HEAD -> feature) work completed
f7a217b feature file edited
493c3ab feature 3 file added
d66ed96 feature 2 file edited
62b9086 feature 1 file added
d7dd26b (master) home file edited

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git rebase -i master
[detached HEAD 51696b8] feature 2 file edited
 Date: Thu Sep 22 22:42:10 2022 +0530
 1 file changed, 3 insertions(+)
Successfully rebased and updated refs/heads/feature.

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git log --oneline
51696b8 (HEAD -> feature) feature 2 file edited
62b9086 feature 1 file added
d7dd26b (master) home file edited

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ 
```

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

git stash temporarily shelves (or *stashes*) changes you've made to your working copy so you can work on something else, and then come back and re-apply them later on. Stashing is handy if you need to quickly switch context and work on something else, but you're mid-way through a code change and aren't quite ready to commit.



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git checkout feature
Switched to branch 'feature'
M       index.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git stash
warning: LF will be replaced by CRLF in index.txt.
The file will have its original line endings in your working directory
Saved working directory and index state WIP on feature: f7dc58e index.txt added

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git list
git: 'list' is not a git command. See 'git --help'.

The most similar commands are
        bisect
        rev-list

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git stash list
stash@{0}: WIP on feature: f7dc58e index.txt added

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ |
```

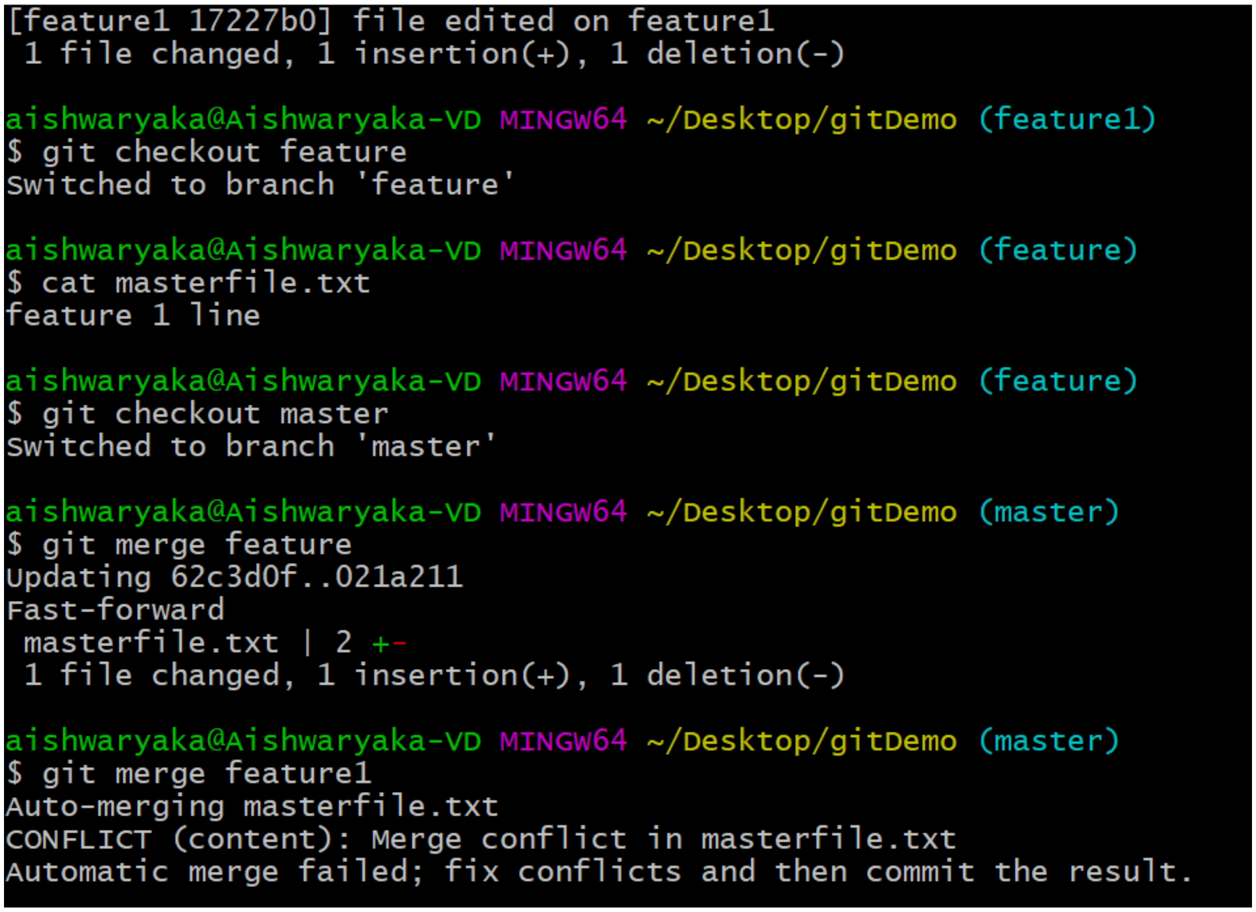
k)what is merge conflict and how to resolve it(Theory + POC)?

Although Git is good at deciding how to merge branches and add changes to files, it cannot always make this decision all by itself.This can happen when the two branches we're trying to merge have changes on the same line in the same file, or if one branch deleted a file that another branch modified, and so on.

If we want to merge dev into master, this will end up in a merge conflict.

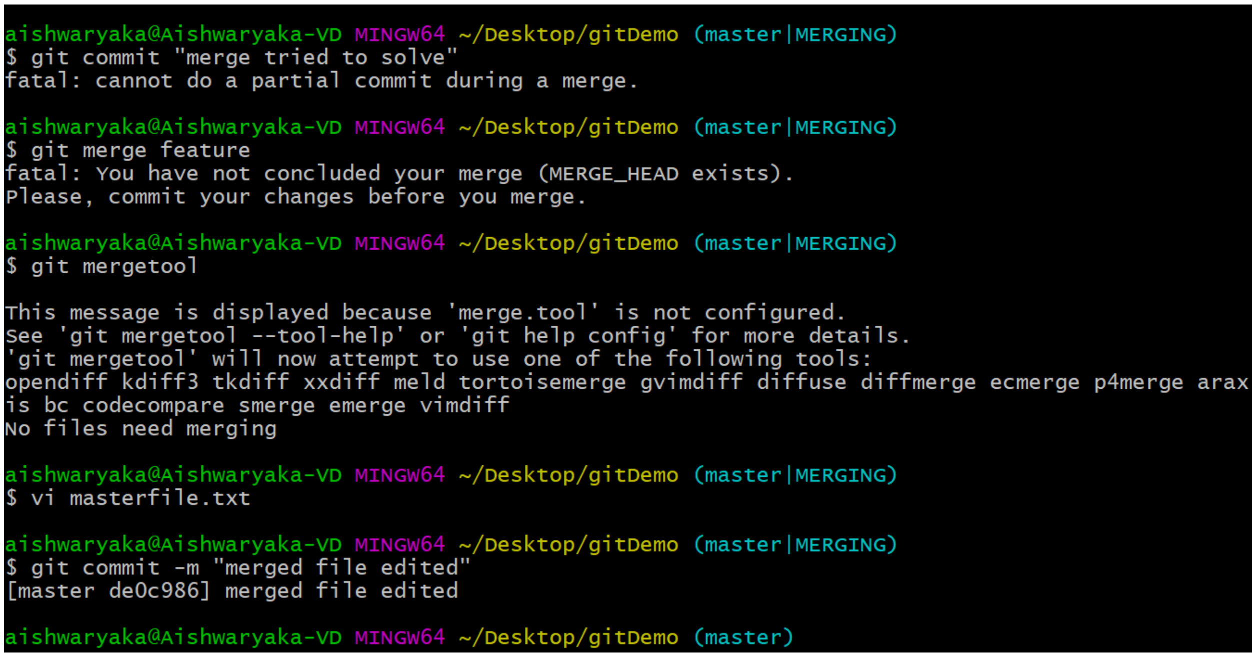
When trying to merge the branches, Git will show you where the conflict happens. We can manually remove the changes we don't want to keep, save the changes, add the changed file again, and commit the changes. Although merge conflicts are often quite annoying, it makes total sense: Git shouldn't just *assume* which change we want to keep.

Merge conflict added:



```
[feature1 17227b0] file edited on feature1
 1 file changed, 1 insertion(+), 1 deletion(-)

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature1)
$ git checkout feature
Switched to branch 'feature'

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ cat masterfile.txt
feature 1 line

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (feature)
$ git checkout master
Switched to branch 'master'

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git merge feature
Updating 62c3d0f..021a211
Fast-forward
 masterfile.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
$ git merge feature1
Auto-merging masterfile.txt
CONFLICT (content): Merge conflict in masterfile.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Merge conflict resolved:



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|MERGING)
$ git commit "merge tried to solve"
fatal: cannot do a partial commit during a merge.

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|MERGING)
$ git merge feature
fatal: You have not concluded your merge (MERGE_HEAD exists).
Please, commit your changes before you merge.

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|MERGING)
$ git mergetool

This message is displayed because 'merge.tool' is not configured.
See 'git mergetool --tool-help' or 'git help config' for more details.
'git mergetool' will now attempt to use one of the following tools:
opendiff kdiff3 tkdiff xxdiff meld tortoisemerge gvimdiff diffuse diffmerge ecmerge p4merge arax
is bc codecompare smerge emerge vimdiff
No files need merging

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|MERGING)
$ vi masterfile.txt

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master|MERGING)
$ git commit -m "merged file edited"
[master de0c986] merged file edited

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitDemo (master)
```

l)How can we untrack a file in git?

Git rm –cached filename

Or add the filename to .gitignore file

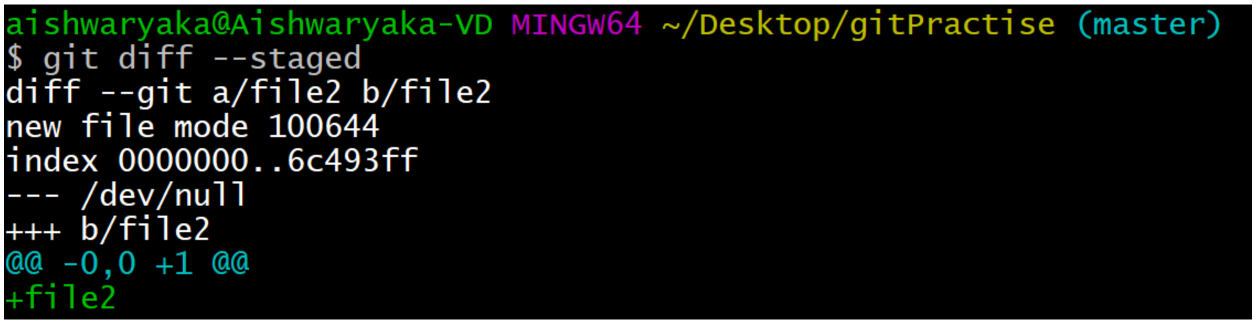
Untrack a folder – git rm –r –catched foldername

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

Git diff :– compares working directory to index or staging area.

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

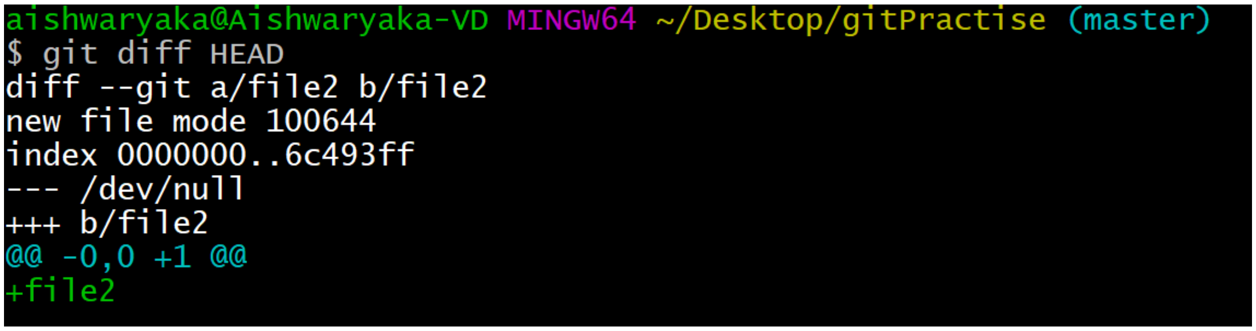
Git diff –staged :-- compares staging area or index to local repository.



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ git diff --staged
diff --git a/file2 b/file2
new file mode 100644
index 0000000..6c493ff
--- /dev/null
+++ b/file2
@@ -0,0 +1 @@
+file2
```

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

Git diff HEAD :-- compares working directory and local repository

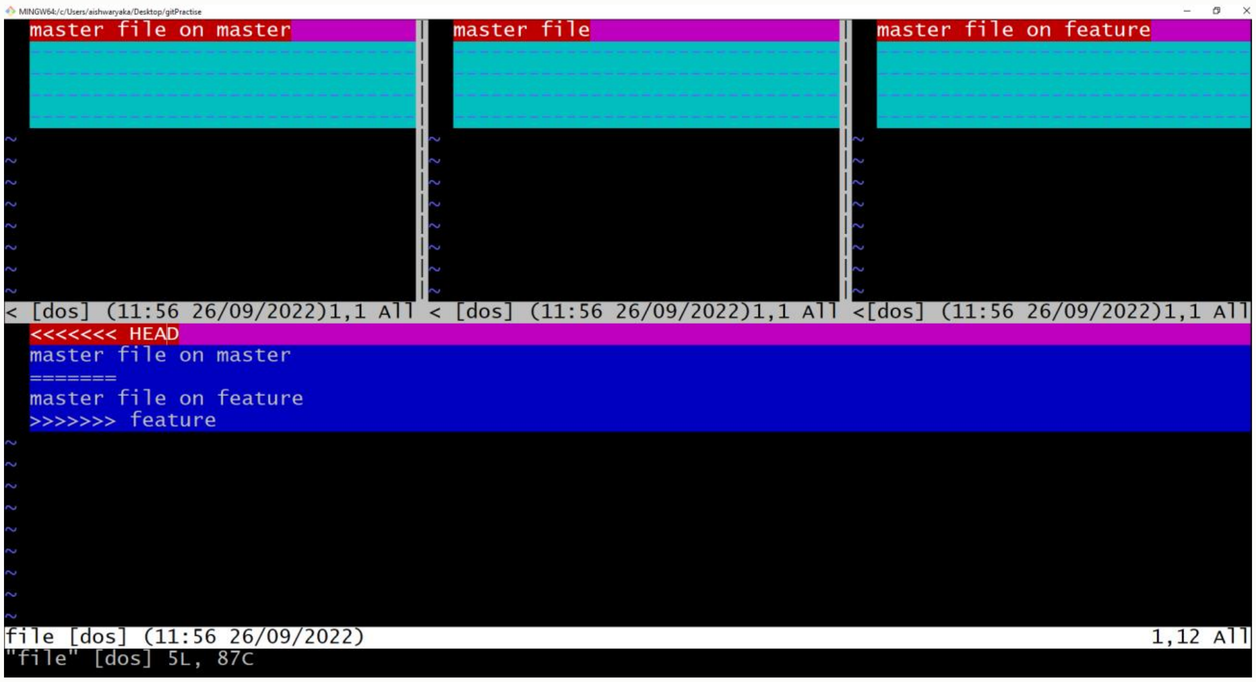


```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ git diff HEAD
diff --git a/file2 b/file2
new file mode 100644
index 0000000..6c493ff
--- /dev/null
+++ b/file2
@@ -0,0 +1 @@
+file2
```

p) What is git merge tool? (theory+poc)

Git mergetool is used to run one of several merge utilities to resolve merge conflicts. It is typically run after git merge.

If one or more <file> parameters are given, the merge tool program will be run to resolve differences on each file (skipping those without conflicts). Specifying a directory will include all unresolved files in that path. If no <file> names are specified, git mergetool will run the merge tool program on every file with merge conflicts.

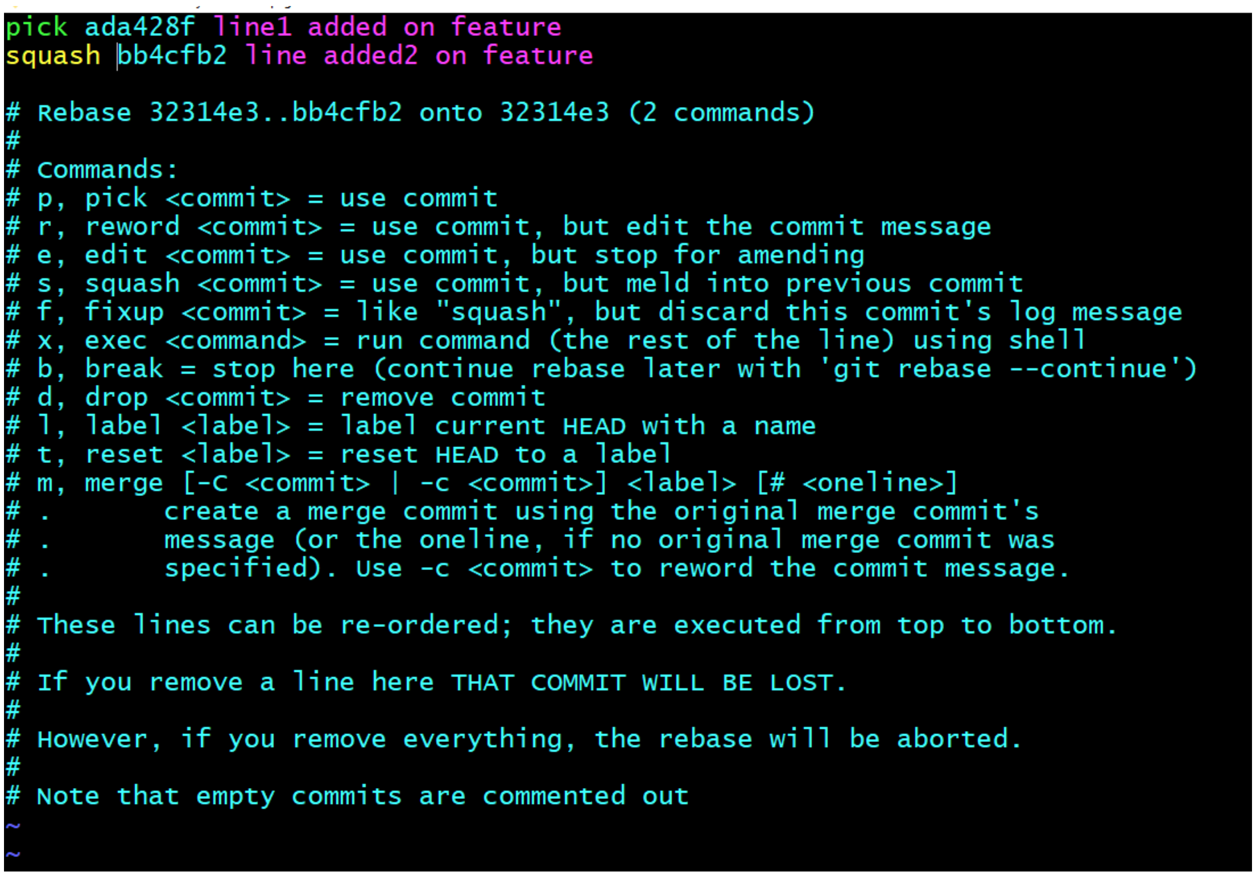


```
master file on master    | master file    | master file on feature
~                        | ~              | ~
< [dos] (11:56 26/09/2022)1,1 All < [dos] (11:56 26/09/2022)1,1 All <[dos] (11:56 26/09/2022)1,1 All
<<<<<<< HEAD
master file on master
=======
master file on feature
>>>>>>> feature
~
file [dos] (11:56 26/09/2022)                                   1,12 All
"file" [dos] 5L, 87C
```

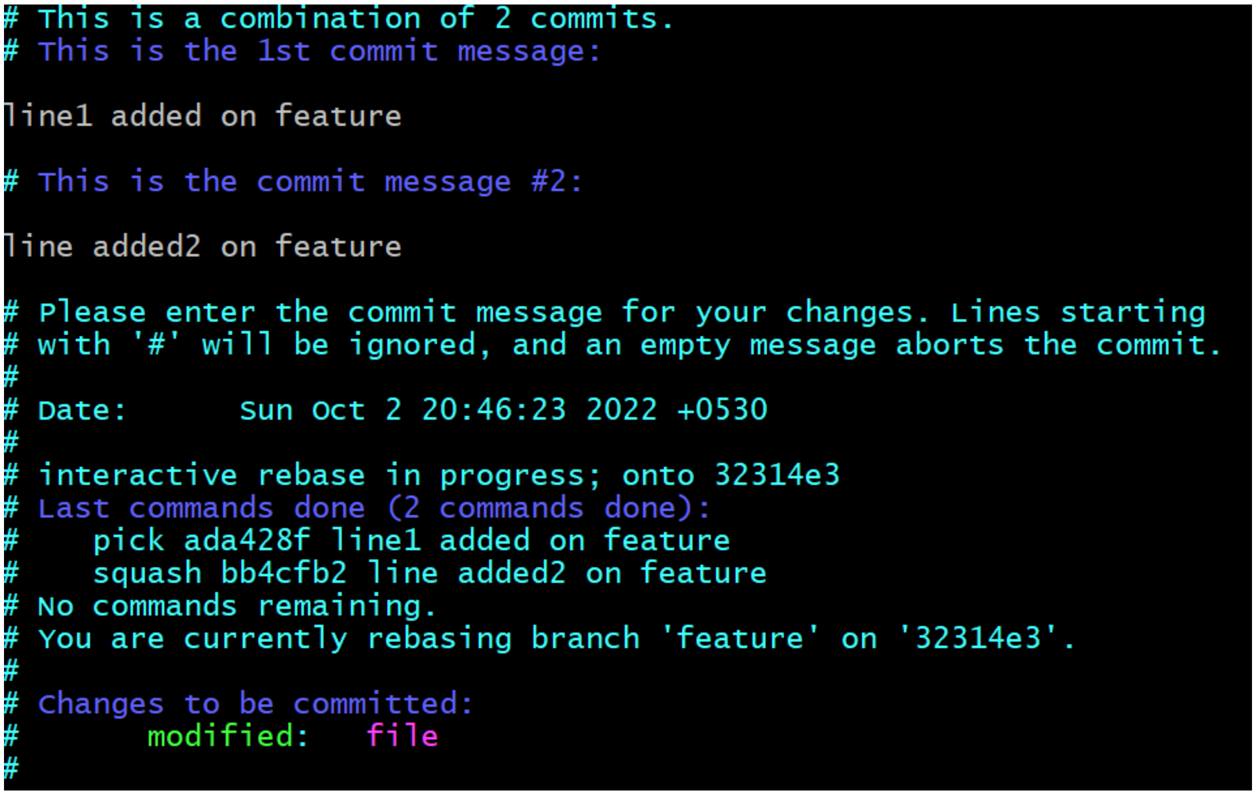
q)Explore squash and merge (theory +poc)?

To "squash" in Git means to combine multiple commits into one. You can do this at any point in time (by using Git's "Interactive Rebase" feature), though it is most often done when merging branches.

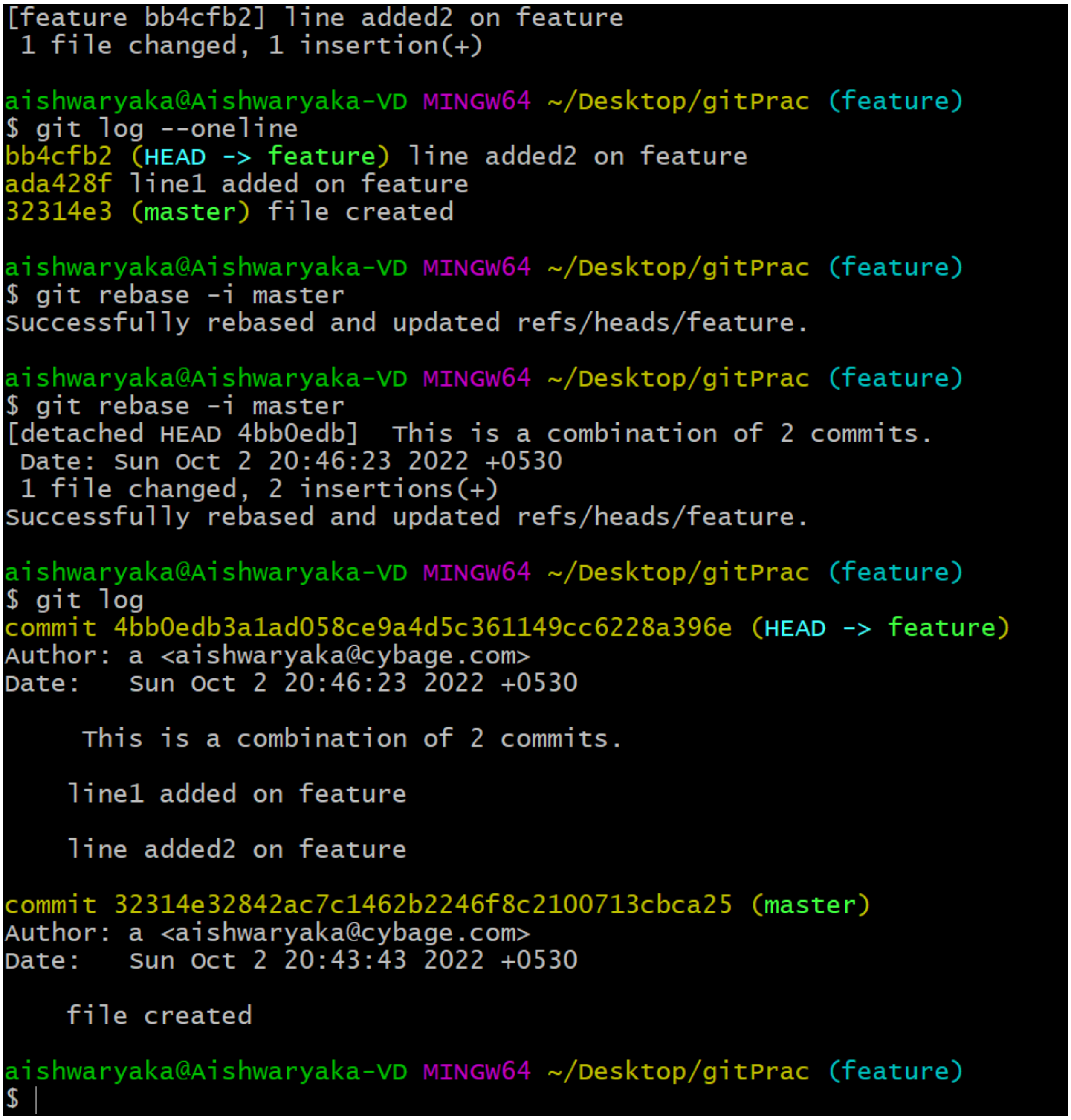
Please note that there is no such thing as a stand-alone git squash command. Instead, squashing is rather an *option* when performing other Git commands like interactive rebase or merge.



```
pick ada428f line1 added on feature
squash bb4cfb2 line added2 on feature

# Rebase 32314e3..bb4cfb2 onto 32314e3 (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .       create a merge commit using the original merge commit's
# .       message (or the oneline, if no original merge commit was
# .       specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
# Note that empty commits are commented out
~
~
```



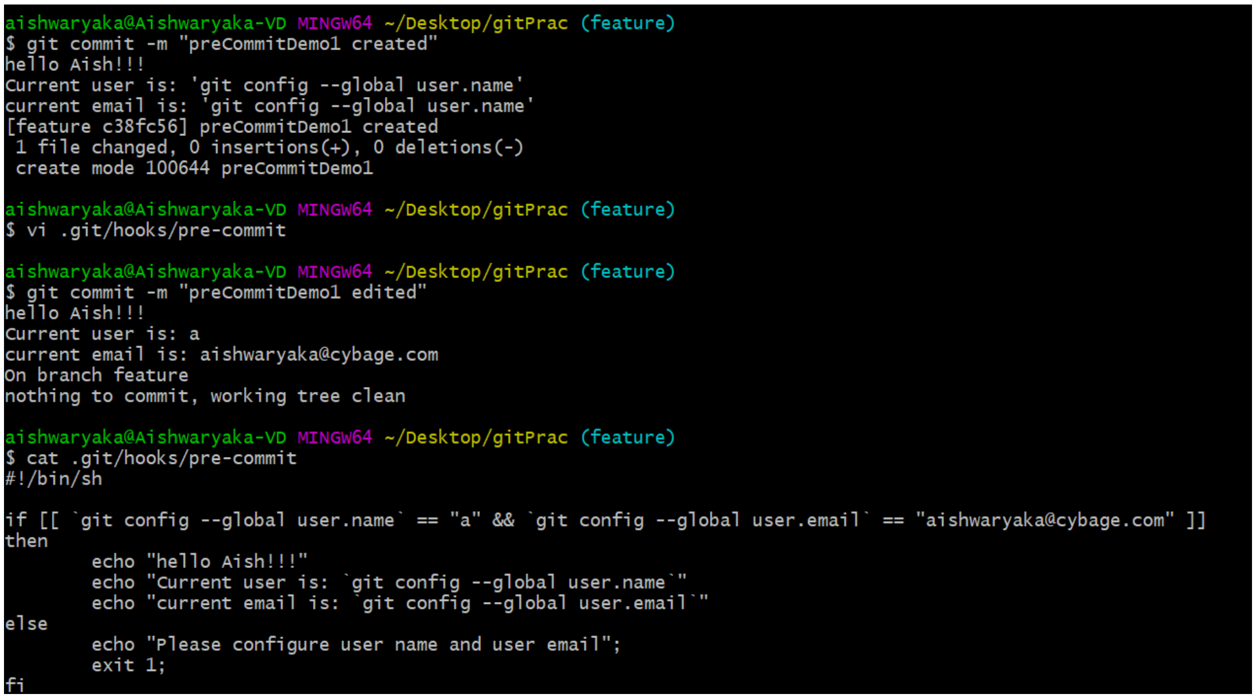
```
# This is a combination of 2 commits.
# This is the 1st commit message:

line1 added on feature

# This is the commit message #2:

line added2 on feature

# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Sun Oct 2 20:46:23 2022 +0530
#
# interactive rebase in progress; onto 32314e3
# Last commands done (2 commands done):
#    pick ada428f line1 added on feature
#    squash bb4cfb2 line added2 on feature
# No commands remaining.
# You are currently rebasing branch 'feature' on '32314e3'.
#
# Changes to be committed:
#       modified:   file
#
```



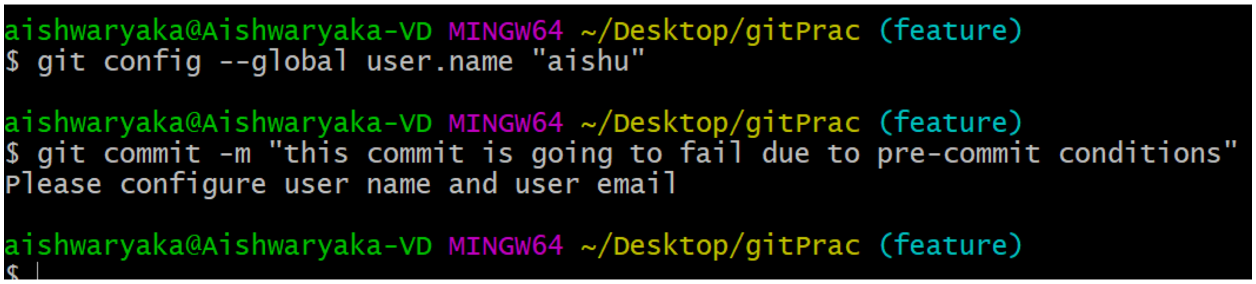
```
[feature bb4cfb2] line added2 on feature
 1 file changed, 1 insertion(+)

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git log --oneline
bb4cfb2 (HEAD -> feature) line added2 on feature
ada428f line1 added on feature
32314e3 (master) file created

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git rebase -i master
Successfully rebased and updated refs/heads/feature.

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git rebase -i master
[detached HEAD 4bb0edb]  This is a combination of 2 commits.
 Date: Sun Oct 2 20:46:23 2022 +0530
 1 file changed, 2 insertions(+)
Successfully rebased and updated refs/heads/feature.

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git log
commit 4bb0edb3a1ad058ce9a4d5c361149cc6228a396e (HEAD -> feature)
Author: a <aishwaryaka@cybage.com>
Date:   Sun Oct 2 20:46:23 2022 +0530

     This is a combination of 2 commits.

    line1 added on feature

    line added2 on feature

commit 32314e32842ac7c1462b2246f8c2100713cbca25 (master)
Author: a <aishwaryaka@cybage.com>
Date:   Sun Oct 2 20:43:43 2022 +0530

    file created

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ |
```

r)What is pre-commit hook and post-commit hook?(theory+poc)

Pre-commit:

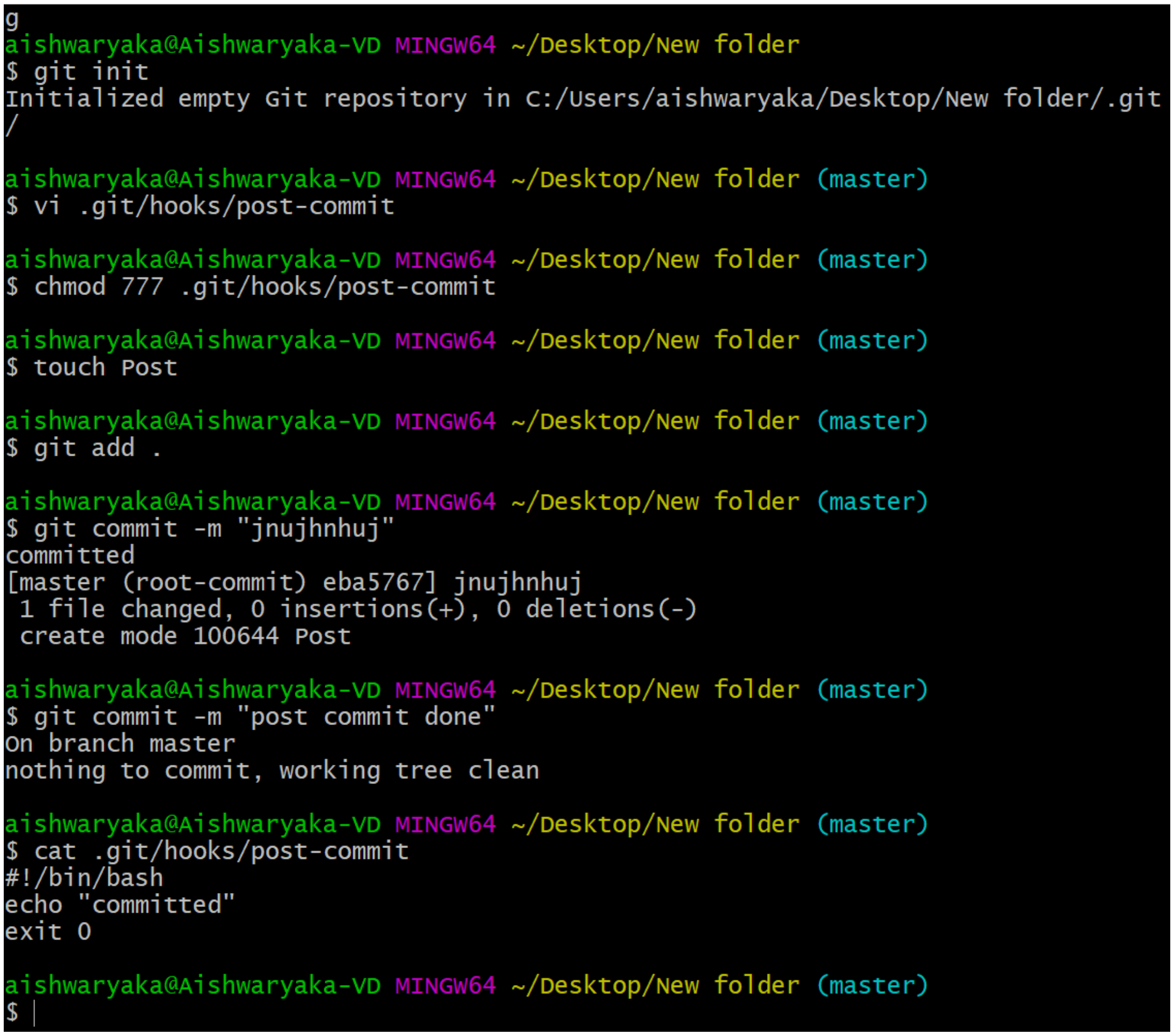


```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git commit -m "preCommitDemo1 created"
hello Aish!!!
Current user is: 'git config --global user.name'
current email is: 'git config --global user.name'
[feature c38fc56] preCommitDemo1 created
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 preCommitDemo1

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ vi .git/hooks/pre-commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git commit -m "preCommitDemo1 edited"
hello Aish!!!
Current user is: a
current email is: aishwaryaka@cybage.com
On branch feature
nothing to commit, working tree clean

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ cat .git/hooks/pre-commit
#!/bin/sh

if [[ `git config --global user.name` == "a" && `git config --global user.email` == "aishwaryaka@cybage.com" ]]
then
        echo "hello Aish!!!"
        echo "Current user is: `git config --global user.name`"
        echo "current email is: `git config --global user.email`"
else
        echo "Please configure user name and user email";
        exit 1;
fi
```



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git config --global user.name "aishu"

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git commit -m "this commit is going to fail due to pre-commit conditions"
Please configure user name and user email

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ 
```

Post-commit:



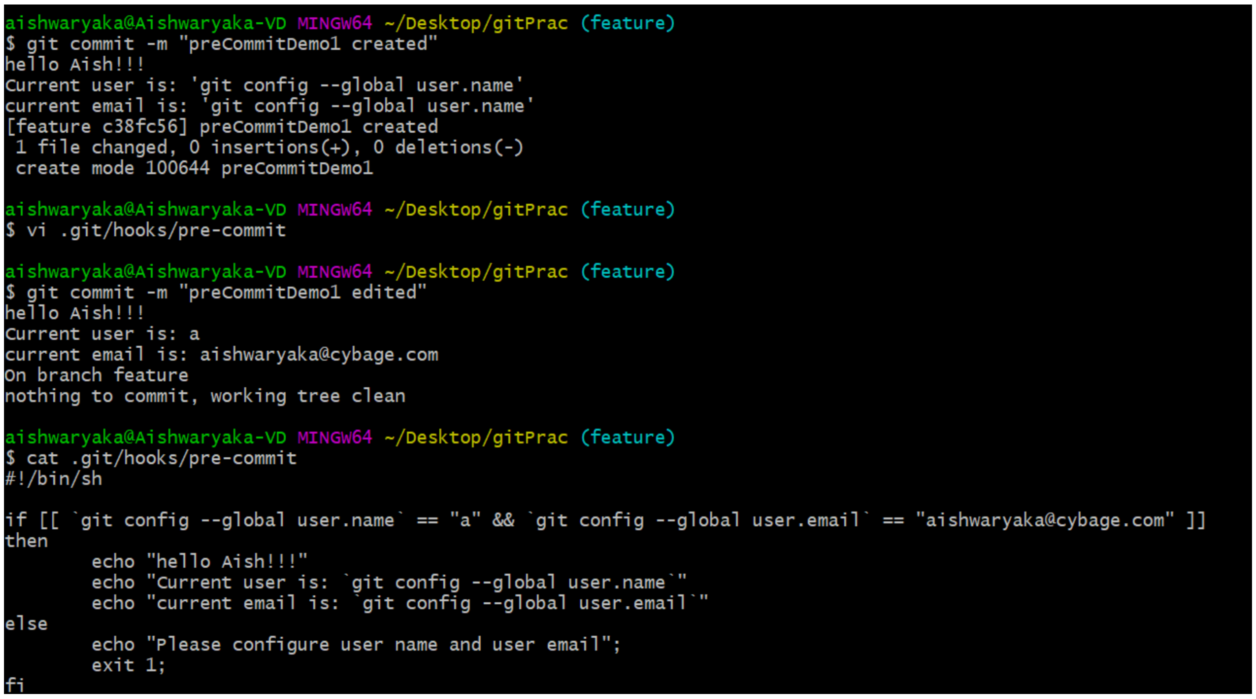
```
g
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder
$ git init
Initialized empty Git repository in C:/Users/aishwaryaka/Desktop/New folder/.git
/

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ vi .git/hooks/post-commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ chmod 777 .git/hooks/post-commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ touch Post

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ git add .

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ git commit -m "jnujhnhuj"
committed
[master (root-commit) eba5767] jnujhnhuj
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Post

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ git commit -m "post commit done"
On branch master
nothing to commit, working tree clean

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ cat .git/hooks/post-commit
#!/bin/bash
echo "committed"
exit 0

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ |
```

s)write a precommit 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?

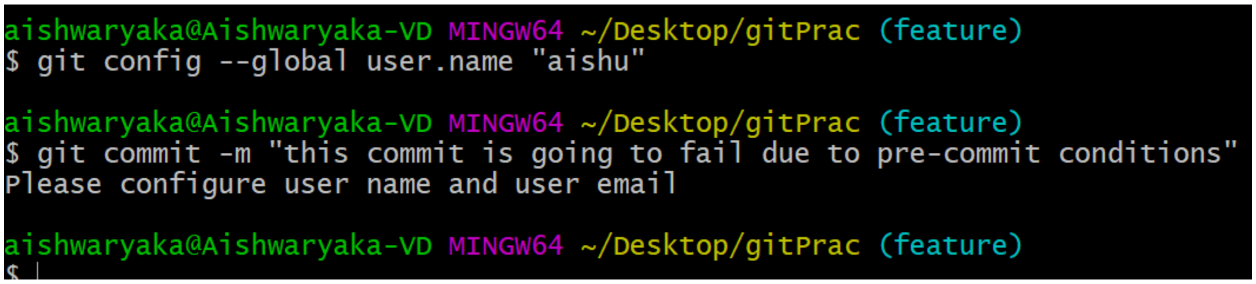
Git hooks are scripts that run automatically every time a particular event occurs in a Git repository. They let you customize Git's internal behavior and trigger customizable actions at key points in the development life cycle.

Pre-commit:

The pre-commit script is executed every time you run git commit before Git asks the developer for a commit message or generates a commit object. You can use this hook to inspect the snapshot that is about to be committed. For example, you may want to run some automated tests that make sure the commit doesn't break any existing functionality.



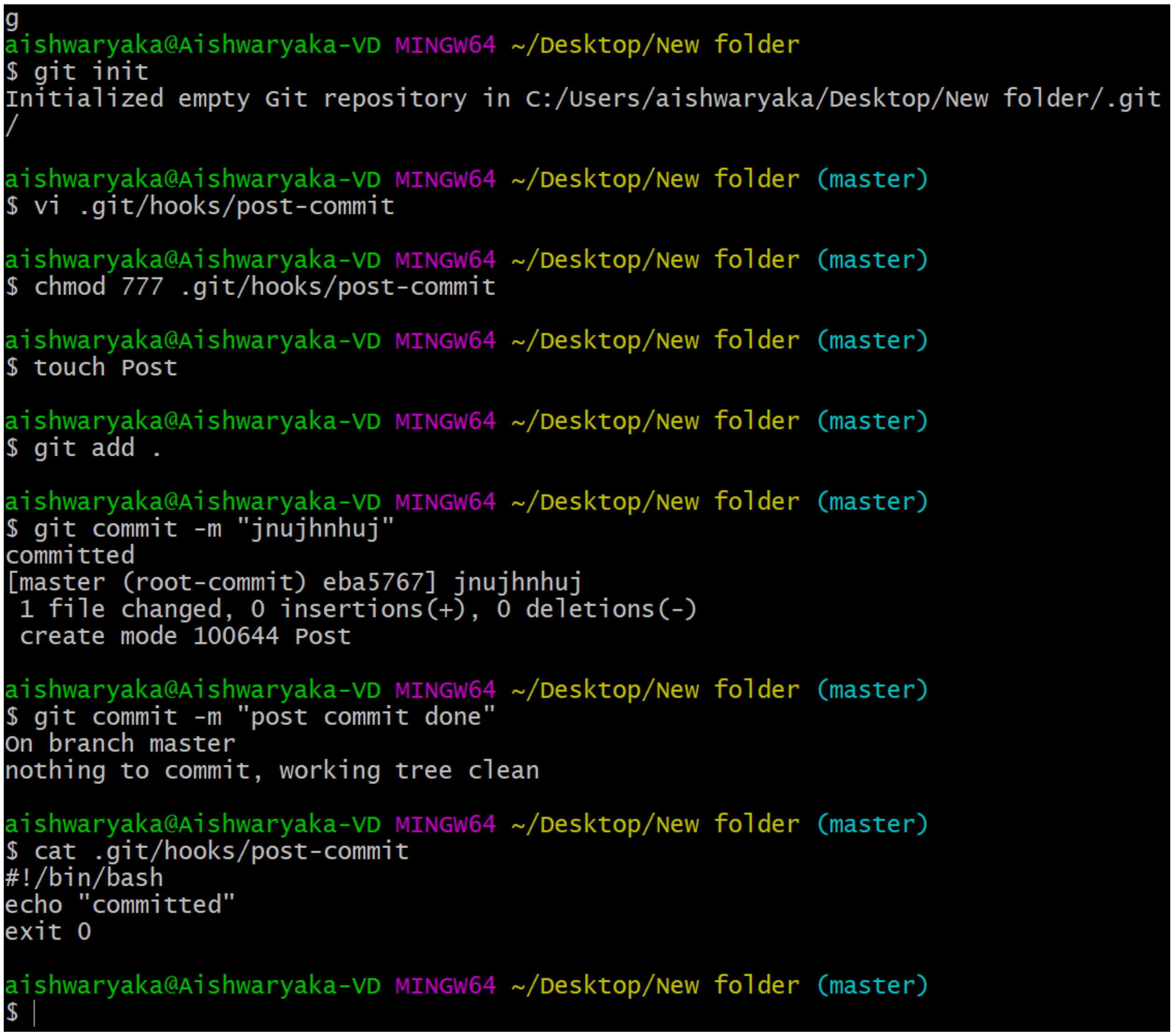
```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git commit -m "preCommitDemo1 created"
hello Aish!!!
Current user is: 'git config --global user.name'
current email is: 'git config --global user.name'
[feature c38fc56] preCommitDemo1 created
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 preCommitDemo1

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ vi .git/hooks/pre-commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git commit -m "preCommitDemo1 edited"
hello Aish!!!
Current user is: a
current email is: aishwaryaka@cybage.com
On branch feature
nothing to commit, working tree clean

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ cat .git/hooks/pre-commit
#!/bin/sh

if [[ `git config --global user.name` == "a" && `git config --global user.email` == "aishwaryaka@cybage.com" ]]
then
        echo "hello Aish!!!"
        echo "Current user is: `git config --global user.name`"
        echo "current email is: `git config --global user.email`"
else
        echo "Please configure user name and user email";
        exit 1;
fi
```



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git config --global user.name "aishu"

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$ git commit -m "this commit is going to fail due to pre-commit conditions"
Please configure user name and user email

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPrac (feature)
$
```

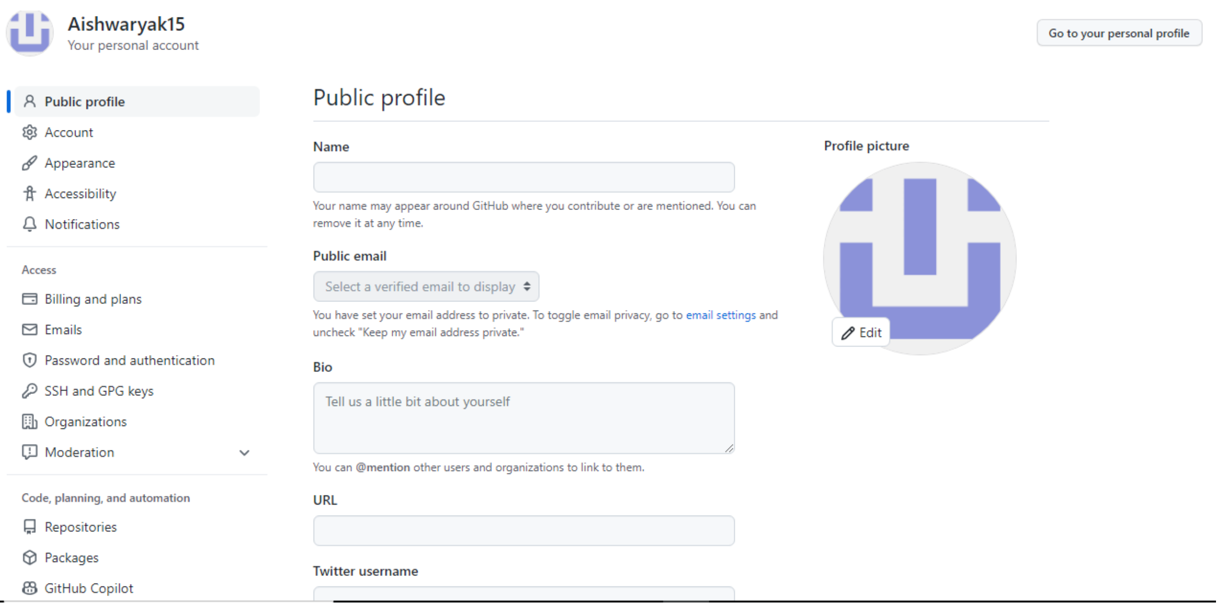
Post-commit:

The post-commit hook is called immediately after the commit-msg hook. It can't change the outcome of the git commit operation, so it's used primarily for notification purposes.The script takes no parameters and its exit status does not affect the commit in any way. For most post-commit scripts, you'll want access to the commit that was just created. You can use git rev-parse HEAD to get the new commit's SHA1 hash, or you can use git log -1 HEAD to get all of its information.



```
g
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder
$ git init
Initialized empty Git repository in C:/Users/aishwaryaka/Desktop/New folder/.git
/

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ vi .git/hooks/post-commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ chmod 777 .git/hooks/post-commit

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ touch Post

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ git add .

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ git commit -m "jnujhnhuj"
committed
[master (root-commit) eba5767] jnujhnhuj
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 Post

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ git commit -m "post commit done"
On branch master
nothing to commit, working tree clean

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ cat .git/hooks/post-commit
#!/bin/bash
echo "committed"
exit 0

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/New folder (master)
$ |
```

t)Explore git hub settings section?


Aishwaryak15
Your personal account

Go to your personal profile

Public profile

Name

Your name may appear around GitHub where you contribute or are mentioned. You can remove it at any time.

Public email

Select a verified email to display

You have set your email address to private. To toggle email privacy, go to email settings and uncheck "Keep my email address private."

Bio

Tell us a little bit about yourself

You can @mention other users and organizations to link to them.

URL

Twitter username

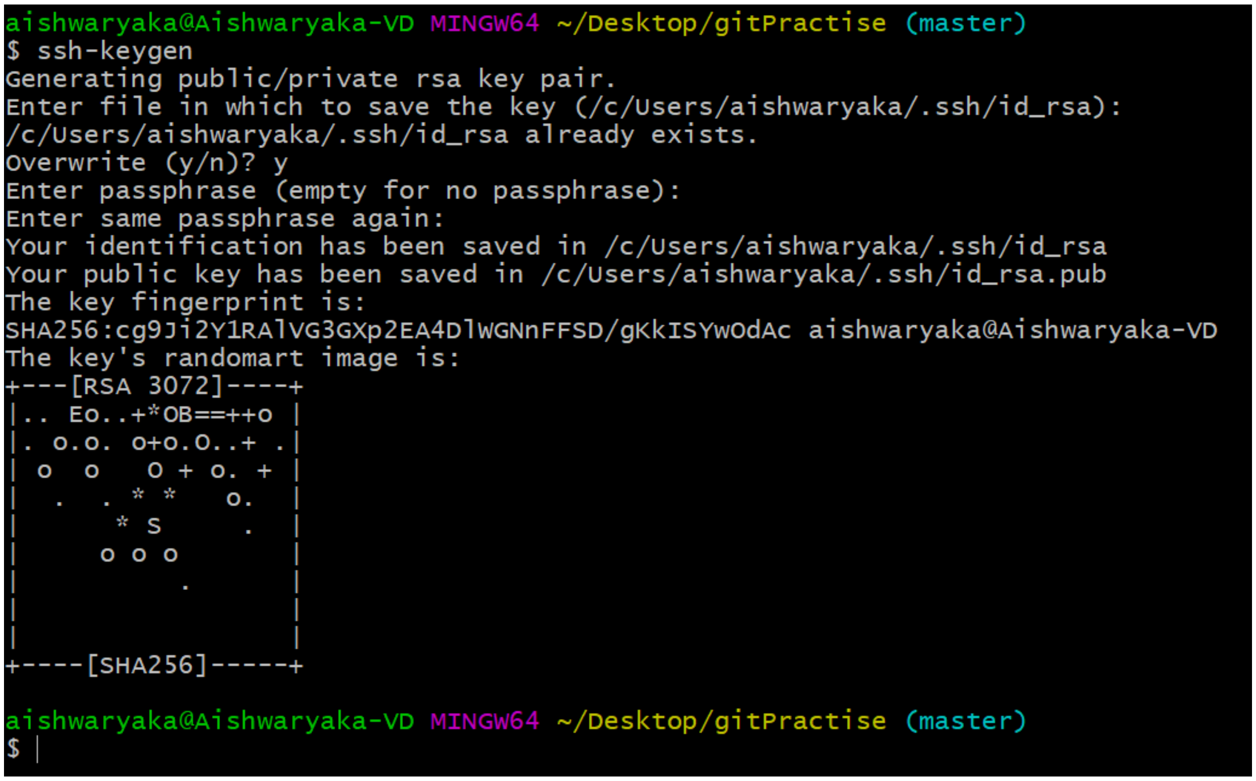
Profile picture

u)clone the repository using ssh?

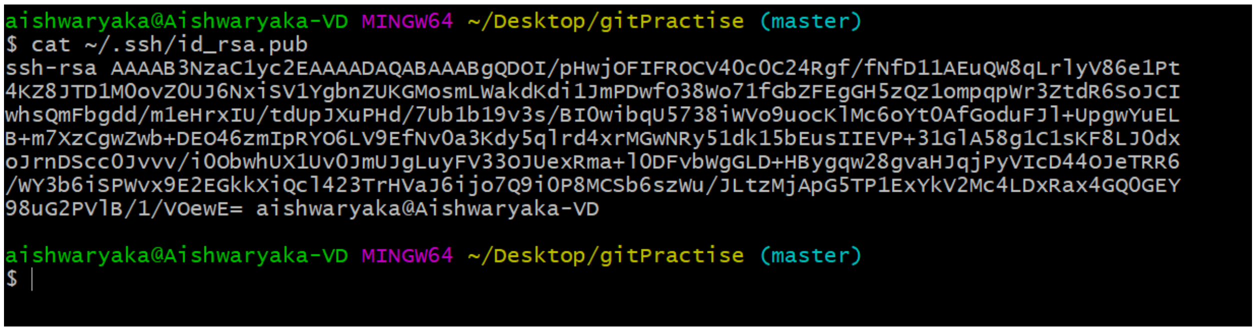
1. on client machine use ssh-keygen command to generate private and public keys inside /home/Ubuntu/.ssh/id_rsa for private key and /home/Ubuntu/.ssh/id_rsa.pub for public key

2. copy public key and paste it on github-> settings->SSH and GPG keys->new SSH key

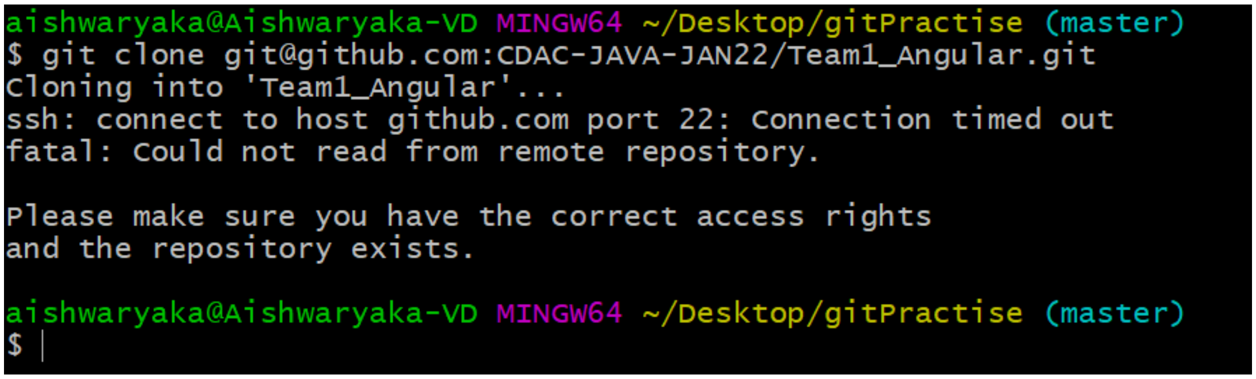
Copy repo’s ssh url from github repo’s clone section Use git command-> git clone



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ ssh-keygen
Generating public/private rsa key pair.
Enter file in which to save the key (/c/Users/aishwaryaka/.ssh/id_rsa):
/c/Users/aishwaryaka/.ssh/id_rsa already exists.
Overwrite (y/n)? y
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /c/Users/aishwaryaka/.ssh/id_rsa
Your public key has been saved in /c/Users/aishwaryaka/.ssh/id_rsa.pub
The key fingerprint is:
SHA256:cg9Ji2Y1RAlVG3GXp2EA4DlWGNnFFSD/gKkISYwOdAc aishwaryaka@Aishwaryaka-VD
The key's randomart image is:
+---[RSA 3072]----+
|.. Eo..+*OB==++o |
|. o.o. o+o.O..+ .|
| o  o   O + o. + |
|  .  . * *   o.  |
|      * S     .  |
|     o o o       |
|          .      |
|                 |
|                 |
+----[SHA256]-----+

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ |
```



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ cat ~/.ssh/id_rsa.pub
ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABgQDOI/pHwjOFIFROCV40c0C24Rgf/fNfD11AEuQW8qLrlyV86e1Pt
4KZ8JTD1MOovZOUJ6NxiSV1YgbnZUKGMosmLWakdKdi1JmPDwfO38Wo71fGbZFEgGH5zQz1ompqpWr3ZtdR6SoJCI
whsQmFbgdd/m1eHrxIU/tdUpJXuPHd/7Ub1b19v3s/BIOwibqU5738iWVo9uocKlMc6oYtOAfGoduFJl+UpgwYuEL
B+m7XzCgwZwb+DEO46zmIpRYO6LV9EfNvOa3Kdy5qlrd4xrMGwNRy51dk15bEusIIEVP+31GlA58g1C1sKF8LJOdx
oJrnDScc0Jvvv/iOObwhUX1UvOJmUJgLuyFV33OJUexRma+lODFvbWgGLD+HBygqw28gvaHJqjPyVIcD44OJeTRR6
/WY3b6iSPWvx9E2EGkkXiQcl423TrHVaJ6ijo7Q9iOP8MCSb6szWu/JLtzMjApG5TP1ExYkV2Mc4LDxRax4GQOGEY
98uG2PVlB/1/VOewE= aishwaryaka@Aishwaryaka-VD

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ |
```



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ git clone git@github.com:CDAC-JAVA-JAN22/Team1_Angular.git
Cloning into 'Team1_Angular'...
ssh: connect to host github.com port 22: Connection timed out
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/gitPractise (master)
$ |
```

v)what are webhooks?

Webhooks provide a way for notifications to be delivered to an external web server whenever certain actions occur on a repository or organization.

Webhooks can be triggered whenever a variety of actions are performed on a repository or an organization. For example, you can configure a webhook to execute whenever:

- A repository is pushed to
- A pull request is opened
- A GitHub Pages site is built
- A new member is added to a team

Using the GitHub API, you can make these webhooks update an external issue tracker, trigger CI builds, update a backup mirror, or even deploy to your production server.

w)what are different workflows of git?

Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules. These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.

It is referred to as Branching Model by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.

There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:
Master
Develop
Hotfixes
Release branches
Feature branches
Every branch has its meaning and standard. Let's understand each branch and its usage.
The Main Branches
Two of the branching model's branches are considered as main branches of the project. These branches are as follows:

master

develop

Master Branch

The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

Your local repository has its master branch that always up to date with the master of a remote repository.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

Develop Branch

It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "integration branch."

When the develop branch reaches a stable point and is ready to release, it should be merged with master and tagged with a release version.

Supportive Branches

The development model needs a variety of supporting branches for the parallel development, tracking of features, assist in quick fixing and release, and other problems. These branches have a limited lifetime and are removed after the uses.

The different types of supportive branches, we may use are as follows:

Feature branches

Release branches

Hotfix branches

Each of these branches is made for a specific purpose and have some merge targets. These branches are significant for a technical perspective.

Feature Branches

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.

To learn how to create a Feature Branch Visit Here.

Release Branches

The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.

Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. After all these tasks, it can be merged with the develop branch.

When all the targeted features are created, then it can be merged with the develop branch. Some usual standard of the release branch are as follows:

Generally, senior developers will create a release branch.

The release branch will contain the predetermined amount of the feature branch.

The release branch should be deployed to a staging server for testing.

Any bugs that need to be improved must be addressed at the release branch.

The release branch must have to be merged back into developing as well as the master branch.

After merging, the release branch with the develop branch must be tagged with a version number.

To create a release branch, visit Git Branching.

To tag branch after merging the release branch, Visit Git tag.

Hotfix Branches

Hotfix branches are similar to Release branches; both are created for a new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.

x)can we fork a private repository if yes how?

Cloning-

You can clone a private repository from your account and you can also clone a private repository from organization if you're its owner or member.

*pat *is* PAT(Personal Access Token).

git clone https://<pat>@github.com/<your account or organization>/<repo>.git
To clone a private repository from your account or organization, you need to generate a PAT(Personal Access Token) on your Github account, and add it to the command above. *Organization doesn't have PAT generator.

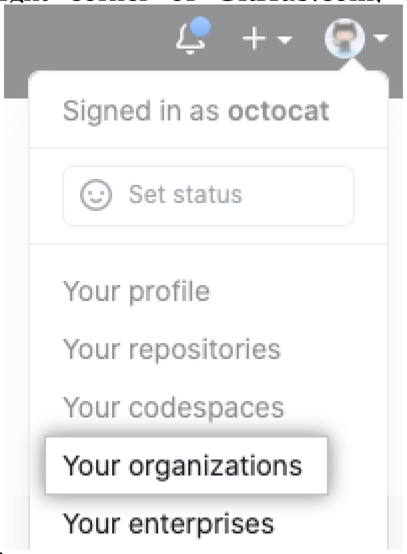
Forking-

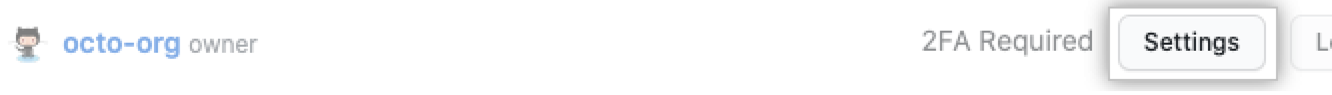
You can allow or prevent the forking of any private repositories owned by your organization.

Who can use this feature
Organization owners can manage the forking policy for an organization.

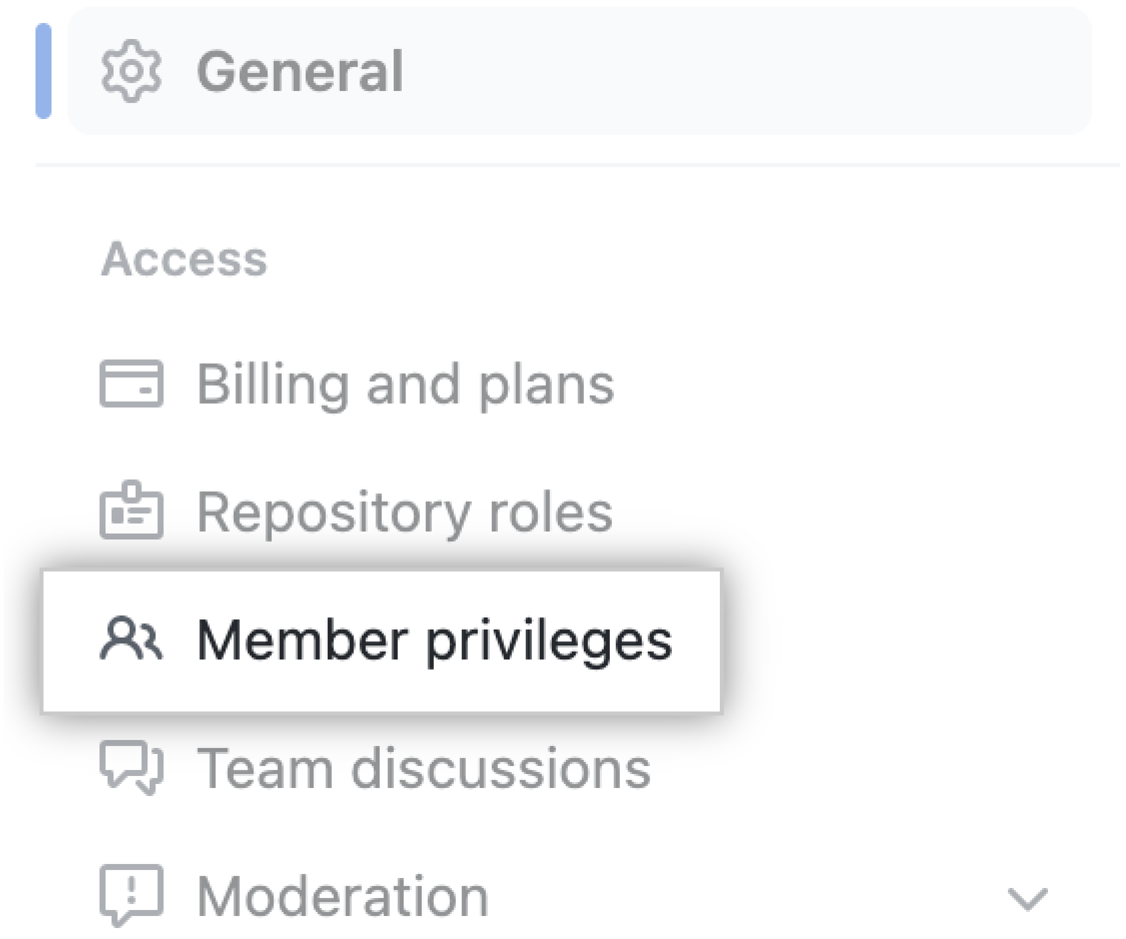
By default, new organizations are configured to disallow the forking of private repositories.

If you allow forking of private repositories at the organization level, you can also configure the ability to fork a specific private repository. For more information, see "Managing the forking policy for your repository."

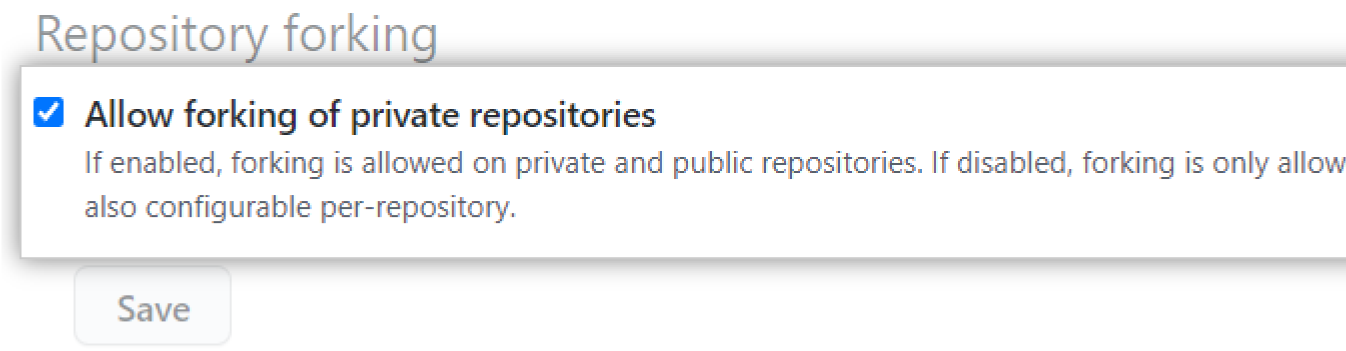
1. In the top right corner of GitHub.com, click your profile photo, then click Your 

organizations.
2. Next to the organization, click Settings.


3. Under "Access", click Member privileges.



4. Under "Repository forking", select Allow forking of private repositories.


Repository forking

Allow forking of private repositories

If enabled, forking is allowed on private and public repositories. If disabled, forking is only allow
also configurable per-repository.

5. Click Save.

y)Study gitflow workflowin detail?

Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules. These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.

It is referred to as Branching Model by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.

There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:
Master
Develop
Hotfixes
Release branches
Feature branches
Every branch has its meaning and standard. Let's understand each branch and its usage.
The Main Branches
Two of the branching model's branches are considered as main branches of the project. These branches are as follows:

master

develop

Master Branch

The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

Your local repository has its master branch that always up to date with the master of a remote repository.

It is suggested not to mess with the master. If you edited the master branch of a group project, your changes would affect everyone else, and very quickly, there will be merge conflicts.

Develop Branch

It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "integration branch."

When the develop branch reaches a stable point and is ready to release, it should be merged with master and tagged with a release version.

Supportive Branches

The development model needs a variety of supporting branches for the parallel development, tracking of features, assist in quick fixing and release, and other problems. These branches have a limited lifetime and are removed after the uses.

The different types of supportive branches, we may use are as follows:

Feature branches

Release branches

Hotfix branches

Each of these branches is made for a specific purpose and have some merge targets. These branches are significant for a technical perspective.

Feature Branches

Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.

To learn how to create a Feature Branch Visit Here.

Release Branches

The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.

Developers are allowed for minor bug fixing and preparing meta-data for a release on this branch. After all these tasks, it can be merged with the develop branch.

When all the targeted features are created, then it can be merged with the develop branch. Some usual standard of the release branch are as follows:

Generally, senior developers will create a release branch.

The release branch will contain the predetermined amount of the feature branch.

The release branch should be deployed to a staging server for testing.

Any bugs that need to be improved must be addressed at the release branch.

The release branch must have to be merged back into developing as well as the master branch.

After merging, the release branch with the develop branch must be tagged with a version number.

To create a release branch, visit Git Branching.

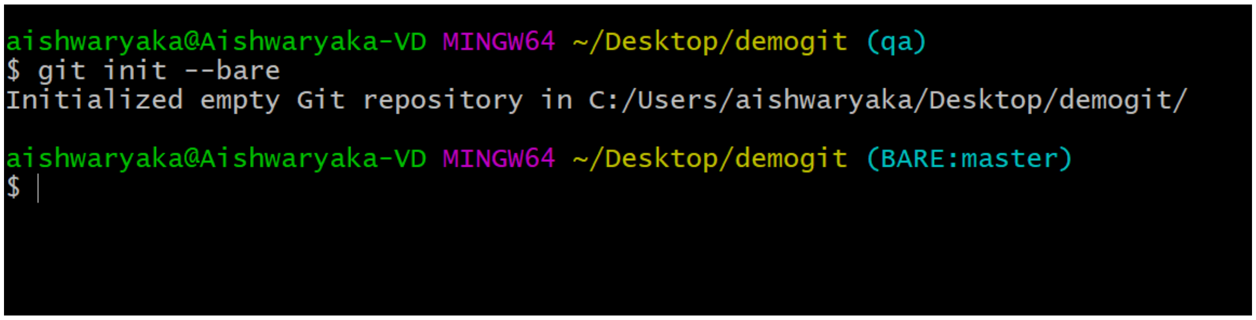
To tag branch after merging the release branch, Visit Git tag.

Hotfix Branches

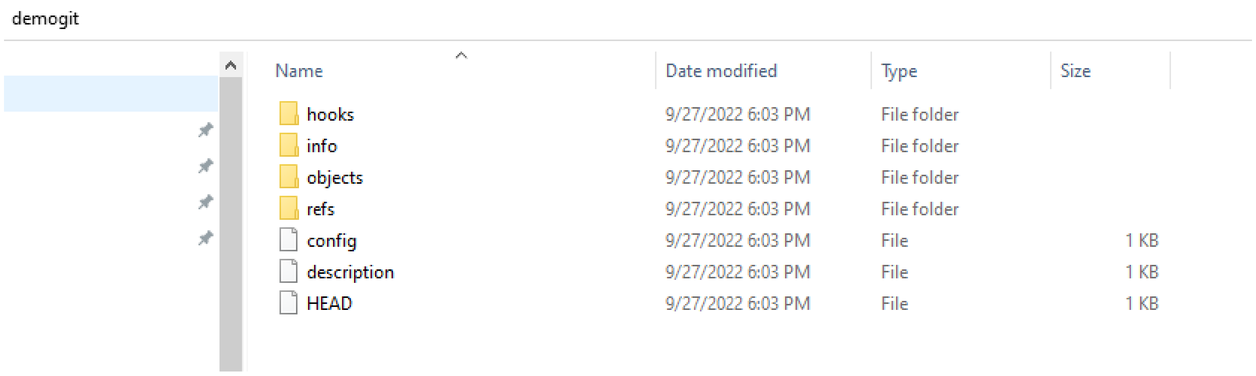
Hotfix branches are similar to Release branches; both are created for a new production release.

The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.

z)Create central repo on local machine(POC)



```
aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/demogit (qa)
$ git init --bare
Initialized empty Git repository in C:/Users/aishwaryaka/Desktop/demogit/

aishwaryaka@Aishwaryaka-VD MINGW64 ~/Desktop/demogit (BARE:master)
$ |
```



demogit

Name	Date modified	Type	Size
hooks	9/27/2022 6:03 PM	File folder	
info	9/27/2022 6:03 PM	File folder	
objects	9/27/2022 6:03 PM	File folder	
refs	9/27/2022 6:03 PM	File folder	
config	9/27/2022 6:03 PM	File	1 KB
description	9/27/2022 6:03 PM	File	1 KB
HEAD	9/27/2022 6:03 PM	File	1 KB