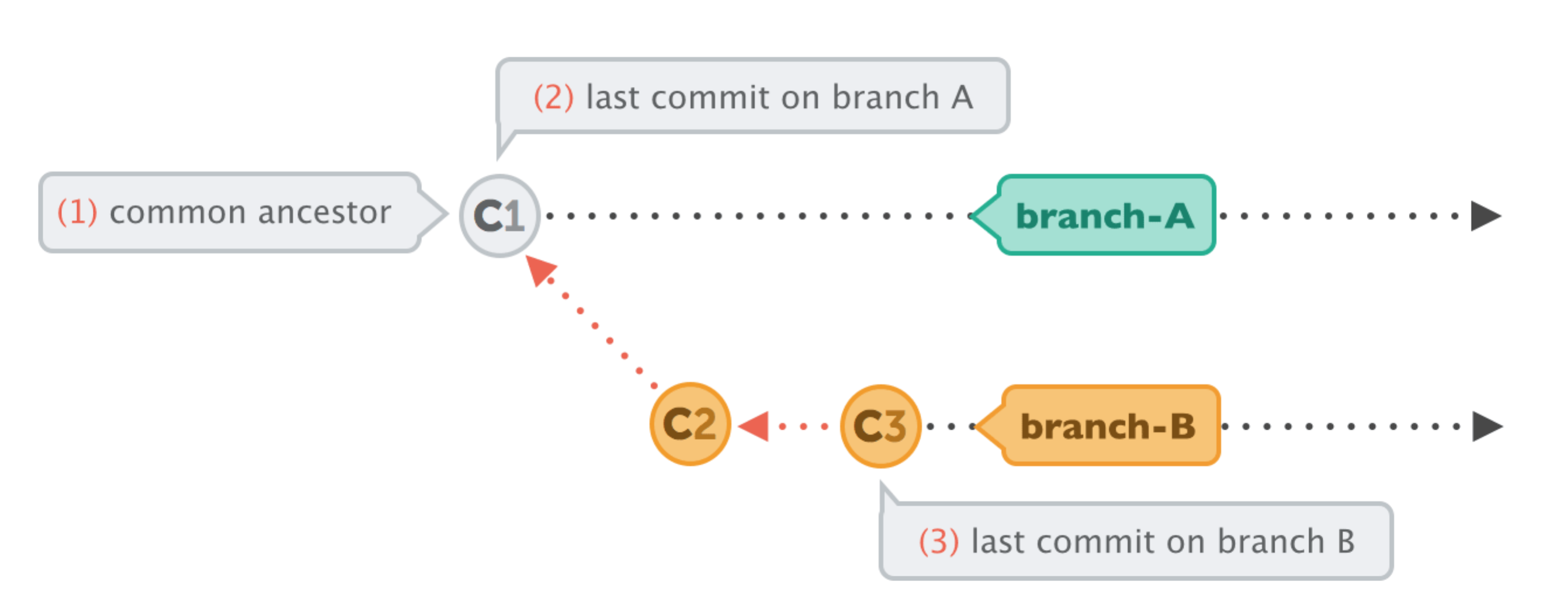
**Github: https://github.com/muralicloud/git\_final.git**

**ITU SWE 525 Version Control with Git Final Examination 4/30/2016**

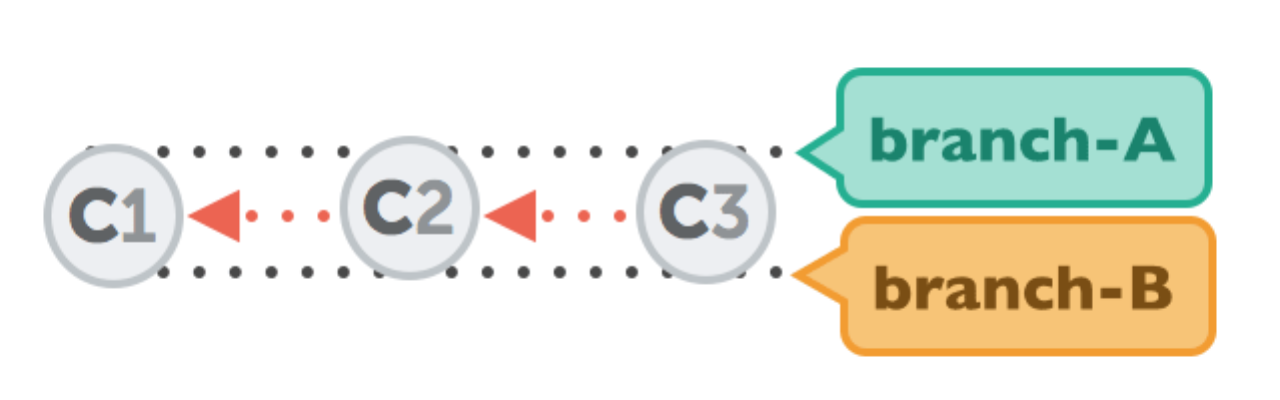
Name: **Murali Krishna Gogula** ITU ID: **90584**

1. Please explain the git Flow and write set of commands that produces following flow. Give tag to each subquestion and push it to git repository. Put the link to your repo.
2. **Fast Forward or Merge commit**

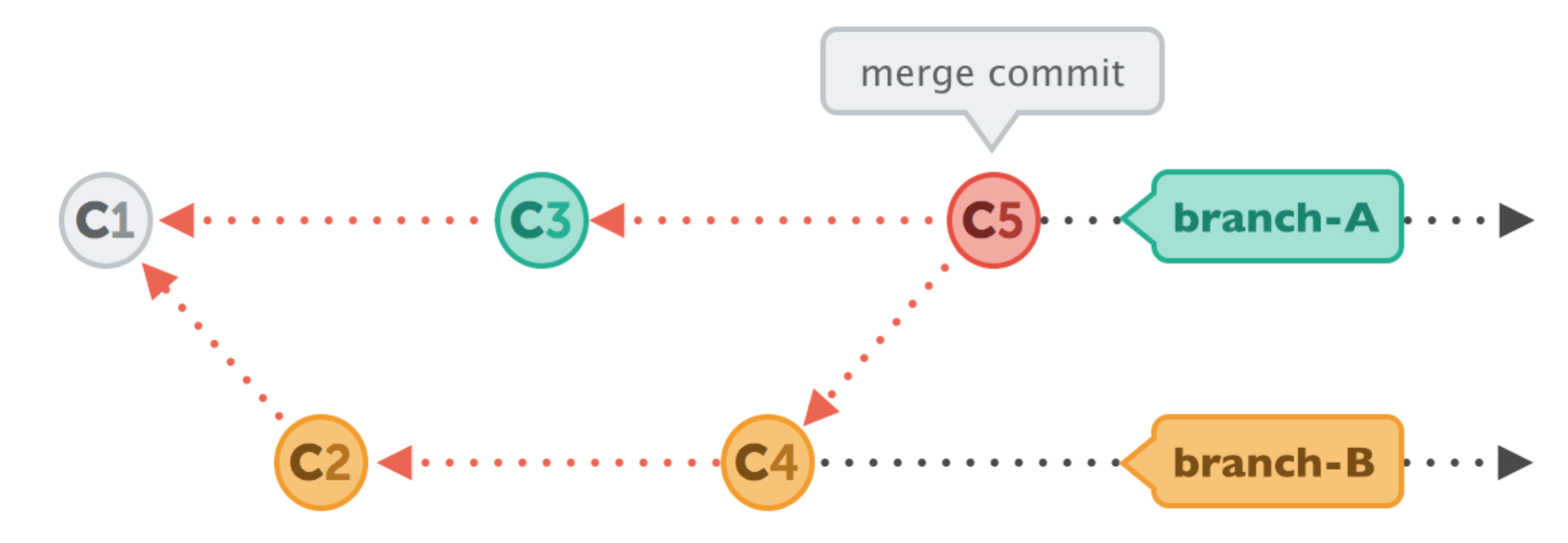
In very simple cases, one of the two branches doesn't have any new commits since the branching happened - its latest commit is still the common ancestor.



**B)** In this case, performing the integration is dead simple: Git can just add all the commits of the other branch on top of the common ancestor commit. In Git, this simplest form of integration is called a "fast-forward" merge. Both branches then share the exact same history.

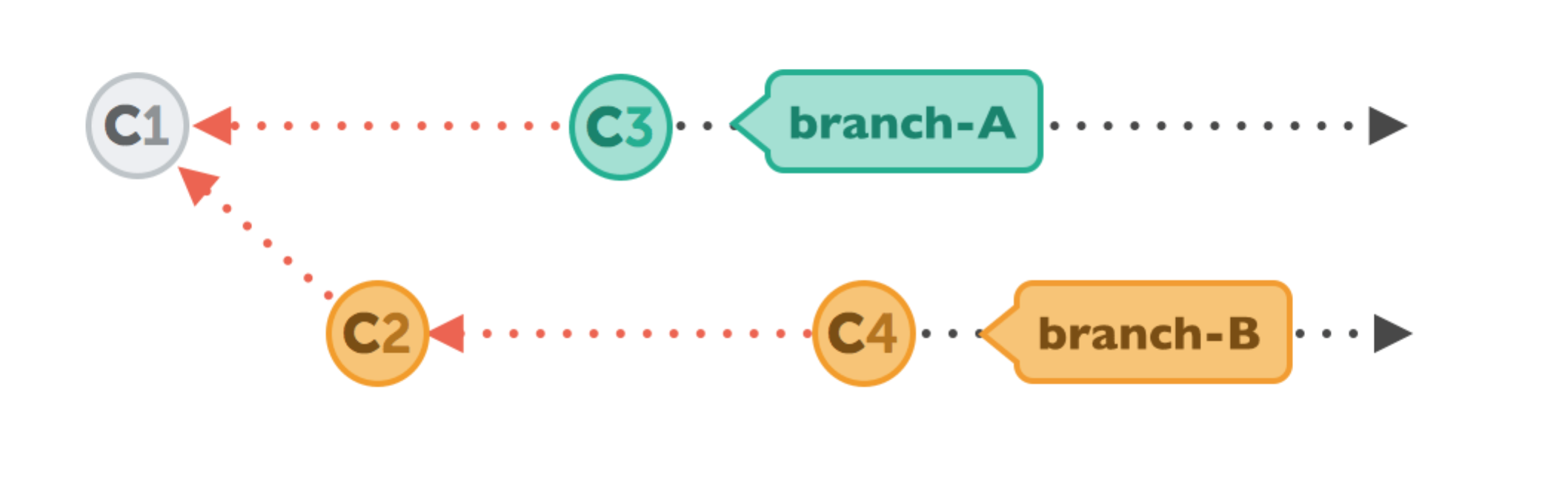


c.  **Merge Commit**



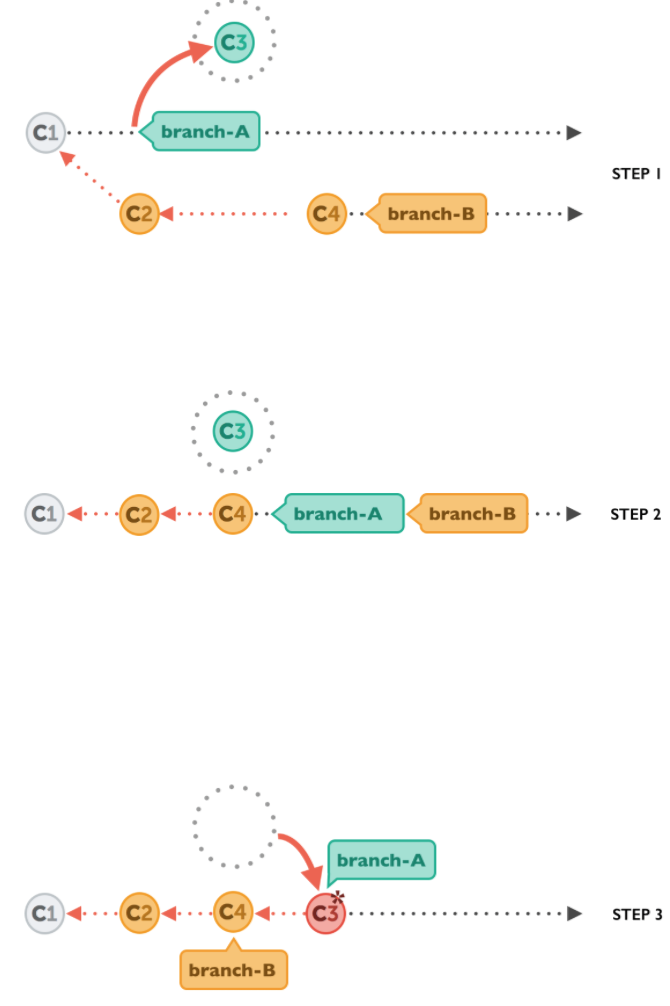
To make an integration, Git will have to create a new commit that contains the differences between them - the merge commit.

d. **Integrating with Rebase.**



Let's walk through a rebase operation step by step. The scenario is the same as in the previous examples: we want to integrate the changes from branch-B into branch-A, but now by using rebase.

**e. Integrating with rebase**



**Fig 1**) First, Git will "undo" all commits on branch-A that happened after the lines began to branch out (after the common ancestor commit). However, of course, it won't discard them: instead you can think of those commits as being "saved away temporarily".

**Fig 2)** Next, it applies the commits from branch-B that we want to integrate. At this point, both branches look exactly the same.

**Fig 3)** In the final step, the new commits on branch-A are now reapplied - but on a new position, on top of the integrated commits from branch-B (they are re-based).  
The result looks like development had happened in a straight line. Instead of a merge commit that contains all the combined changes, the original commit structure was preserved.

**2. Solve the following exercise. Write your code, add screenshots where applicable. Push your flow into a git hub repository. Put your link here.**

1. Make a commit, and make a silly typo in the commit message.
2. Use the --amend flag to enable you to fix the commit message.
3. Look at the log and notice how the mistake is magically gone.
4. Now make a commit where you make a typo in one of the files. Once again, use --amend to magic away your problems.
5. Create a branch. Make a commit.
6. Now switch back to your master branch. Make a (non-conflicting) commit there also.
7. Now switch back to your branch.
8. Use the rebase command in your branch. Look at the DAG in gitk, and note that you have the commit from the master branch, but no merge commit.
9. Make one more commit in your branch.
10. Return to master. Merge your branch. Notice how, thanks to the rebase, this is a fast- forward merge.
11. Find somebody from your team from the previous exercise. Have them push a commit to the central repository.
12. Make a commit locally yourself also. Note that you should not have pulled their commit at this point.
13. Try to push, and watch it fail.
14. Now, pull but using the --rebase flag.
15. Use git log and gitk to verify that there is no merge commit, and the DAG is linear.
16. Notice that your commit is the latest one, even though temporally the other member of your team made their commit afterwards. Why is this?

Answer)

muralikrishna@Shrita MINGW64 ~ (master)

$ mkdir gitfinal

muralikrishna@Shrita MINGW64 ~ (master)

$ cd gitfinal

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ vi gitfinal.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git add --all

fatal: Not a git repository: AppData/Local/GitHub/IgnoreTemplates\_a66c3719071da6d865a984bb8d6bfb5bcd775ec8/../.git/modules/gitignore

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git add gitfinal.txt

warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git commit -m "jfflaf"

[master 89aa6e2] jfflaf

warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

1 file changed, 1 insertion(+)

create mode 100644 gitfinal/gitfinal.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git commit --amend -m "final"

[master 881ea6d] final

Date: Sat Apr 30 17:43:35 2016 -0700

warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

1 file changed, 1 insertion(+)

create mode 100644 gitfinal/gitfinal.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git branch

checkout

chekout

exp

masster

\* master

mehta

murali

newone

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ vi gitfinal.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git diff

**diff --git a/gitfinal/gitfinal.txt b/gitfinal/gitfinal.txt**

**index a72bdb0..af1eee9 100644**

**--- a/gitfinal/gitfinal.txt**

**+++ b/gitfinal/gitfinal.txt**

@@ -1 +1,4 @@

-jnnzxvnnjvznvznvnzmnzmmnzmm.zxm,jzjj

+jnnzxvnnjvznvznvnzmnzmmnzmm.zxm,jzjan

+aja,fn

+x,cxc,njf

+zkcnzlncj

warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git branch rough

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git checkout -b rough

fatal: A branch named 'rough' already exists.

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git branch

checkout

chekout

exp

masster

\* master

mehta

murali

newone

rough

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git checkot rough

git: 'checkot' is not a git command. See 'git --help'.

Did you mean this?

checkout

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git checkout rough

M gitfinal/gitfinal.txt

Switched to branch 'rough'

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ vi rough.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ git add -all

error: did you mean `--all` (with two dashes ?)

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ git add --all

fatal: Not a git repository: AppData/Local/GitHub/IgnoreTemplates\_a66c3719071da6d865a984bb8d6bfb5bcd775ec8/../.git/modules/gitignore

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ git add rough.txt

warning: LF will be replaced by CRLF in gitfinal/rough.txt.

The file will have its original line endings in your working directory.

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ git commit -m "second"

[rough e137d1e] second

warning: LF will be replaced by CRLF in gitfinal/rough.txt.

The file will have its original line endings in your working directory.

1 file changed, 2 insertions(+)

create mode 100644 gitfinal/rough.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ ls

gitfinal.txt rough.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (rough)

$ git checkout master

M gitfinal/gitfinal.txt

Switched to branch 'master'

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ ls

gitfinal.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ gti merge rough

bash: gti: command not found

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git merge rough

Updating 881ea6d..e137d1e

Fast-forward

gitfinal/rough.txt | 2 ++

1 file changed, 2 insertions(+)

create mode 100644 gitfinal/rough.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git push

fatal: No configured push destination.

Either specify the URL from the command-line or configure a remote repository using

git remote add <name> <url>

and then push using the remote name

git push <name>

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git push https://github.com/muralicloud/git\_final.git

warning: push.default is unset; its implicit value has changed in

Git 2.0 from 'matching' to 'simple'. To squelch this message

and maintain the traditional behavior, use:

git config --global push.default matching

To squelch this message and adopt the new behavior now, use:

git config --global push.default simple

When push.default is set to 'matching', git will push local branches

to the remote branches that already exist with the same name.

Since Git 2.0, Git defaults to the more conservative 'simple'

behavior, which only pushes the current branch to the corresponding

remote branch that 'git pull' uses to update the current branch.

See 'git help config' and search for 'push.default' for further information.

(the 'simple' mode was introduced in Git 1.7.11. Use the similar mode

'current' instead of 'simple' if you sometimes use older versions of Git)

fatal: The current branch master has no upstream branch.

To push the current branch and set the remote as upstream, use

git push --set-upstream https://github.com/muralicloud/git\_final.git master

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git push https://github.com/muralicloud/git\_final.git master

\nThis repository is configured for Git LFS but 'git-lfs' was not found on your path. If you no longer wish to use Git LFS, remove this hook by deleting .git/hooks/pre-push.\n

error: failed to push some refs to 'https://github.com/muralicloud/git\_final.git'

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ ls

gitfinal.txt rough.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ vi gitfinal.txt

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git add -all

error: did you mean `--all` (with two dashes ?)

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git add gitfinal.txt

warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git commit -m "fns"

[master warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

1294aa9] fns

warning: LF will be replaced by CRLF in gitfinal/gitfinal.txt.

The file will have its original line endings in your working directory.

1 file changed, 4 insertions(+), 1 deletion(-)

muralikrishna@Shrita MINGW64 ~/gitfinal (master)

$ git pull https://github.com/muralicloud/git\_final.git

warning: no common commits

remote: Counting objects: 3, done.

remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0

Unpacking objects: 100% (3/3), done.

From https://github.com/muralicloud/git\_final

\* branch HEAD -> FETCH\_HEAD

Auto-merging README.md

CONFLICT (add/add): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.

muralikrishna@Shrita MINGW64 ~/gitfinal (master|MERGING)

$ git fetch https://github.com/muralicloud/git\_final.git

From https://github.com/muralicloud/git\_final

\* branch HEAD -> FETCH\_HEAD

muralikrishna@Shrita MINGW64 ~/gitfinal (master|MERGING)

$

1. What is Degenerate, speciality and normal merge?

**Degenerate Merge:**

There are two common degenerate scenarios that lead to merges and are called already up-to-date and fast forward. Because neither or these scenarios actually introduces a new merge commit after performing the **git merge,** some might consider them not to be true merge strategies

* **Already up-to-date**: when all the commits from the other branch are already present in your target branch, even if it has advanced on its own, the target branch is said to be already up to date. As a result, no new commits are added to your branch.
* **Fast Forward:** A fast forward merge happens when your branch Head is already fully present and represented in the other branch. This is the inverse of the already up-to-date case.

Because your HEAD is already present in the other branch (likely due to a common ancestor), git simply tacks on to your HEAD the new commits from the other branch. Git then moves tour branch HEAD to point to the final, new commit. Naturally the index and your working directory also adjusted accordingly to reflect the view, final commit state.

**Normal Merge:**

These merge strategies all produce a final commit, added to your current branch that represents the combined state of the merge.

* **Resolve:** the resolve strategy operates on only two branches, locating the common ancestor as the merge basis and performing a direct three-way merge by applying the changes from the merge base to the tip of the other branch HEAD on to the current branch. This method makes intuitive sense.
* **Recursive:** The recursive strategy is similar to the resolve strategy In that it can only join two branches at once. However, it is designed to handle the scenario where there is more than one merge base between the two branches. In these cases, git forms a temporary merge of all common merge bases and then uses that as the base from which to deserve resulting merge of the two given branches via a normal three way merge algorithm.
* **Octopus:** The octopus strategy is specifically designed to merge together more than two branches simultaneously. Conceptually, it is fairly simple ; internally, it calls the recursive merge strategy multiple times, once for each branch you are merging.

**Specialty Merges:**

There are two merge strategies that you should be aware of because they can sometimes help you to solve strange problems. The two special strategies are **Ours** and **subtree.**

These merge strategies each produce a final commit, added to your current branch, that represents the combined state of the merge.

* **Ours:** the ours strategy merges in any number of other branches, but it actually discards changes from those branches and uses only the files from the current branch. The result of an ours merge is identical to the current HEAD, but any other named branches are also recorded as commit parents.
* **Subtree:**  The subtree strategy merges in another branch, but everything in that branch is merged into a particular subtree of the current tree. You don’t specify which subtree; git determines that automatically.

1. **Study the following case study. Read through it. Perform the steps noted in this case. Push your work in a repository in github. Put your github repository link in this document. Explain what you understand from this case in this document in detail. Add your screenshots where applicabl.e**

**Git Rebase**

Imagine you are working on that radical new feature. It’s going to be brilliant but it takes a while. You’ve been working on that for a couple of days now, maybe weeks.

Your feature branch is already six commits ahead of master. You’ve been a good developer and have crafted meaningful semantical commits. But there’s the thing: you are slowly realizing that this beast will still take some more time before it’s really ready to be merged back into master.

m1-m2-m3-m4 (master)

\

f1-f2-f3-f4-f5-f6(feature)

What you also realize is that some parts are actually less coupled to the new feature. They could land in master earlier. Unfortunately, the part that you want to port back into master earlier is in a commit somewhere in the middle of your six commits. Even worse, it also contains a change that relies on a previous commits of your feature branch. One could argue that you should have made that two commits in the first place, but then nobody is perfect.

m1-m2-m3-m4 (master)

\

f1-f2-f3-f4-f5-f6(feature)

^

|

mixed commit

At the time that you crafted the commit, you didn’t foresee that you might come into a situation where you want to gradually bring the feature into master. Heck! You wouldn’t have guessed that this whole thing could take us so long.

What you need is a way to go back in history, open up the commit and split it into two commits so that you can separate out all the things that are safe to be ported back into master by now.

Speaking in terms of a graph, we want to have it like this.

m1-m2-m3-m4 (master)

\

f1-f2-f3a-f3b-f4-f5-f6(feature)

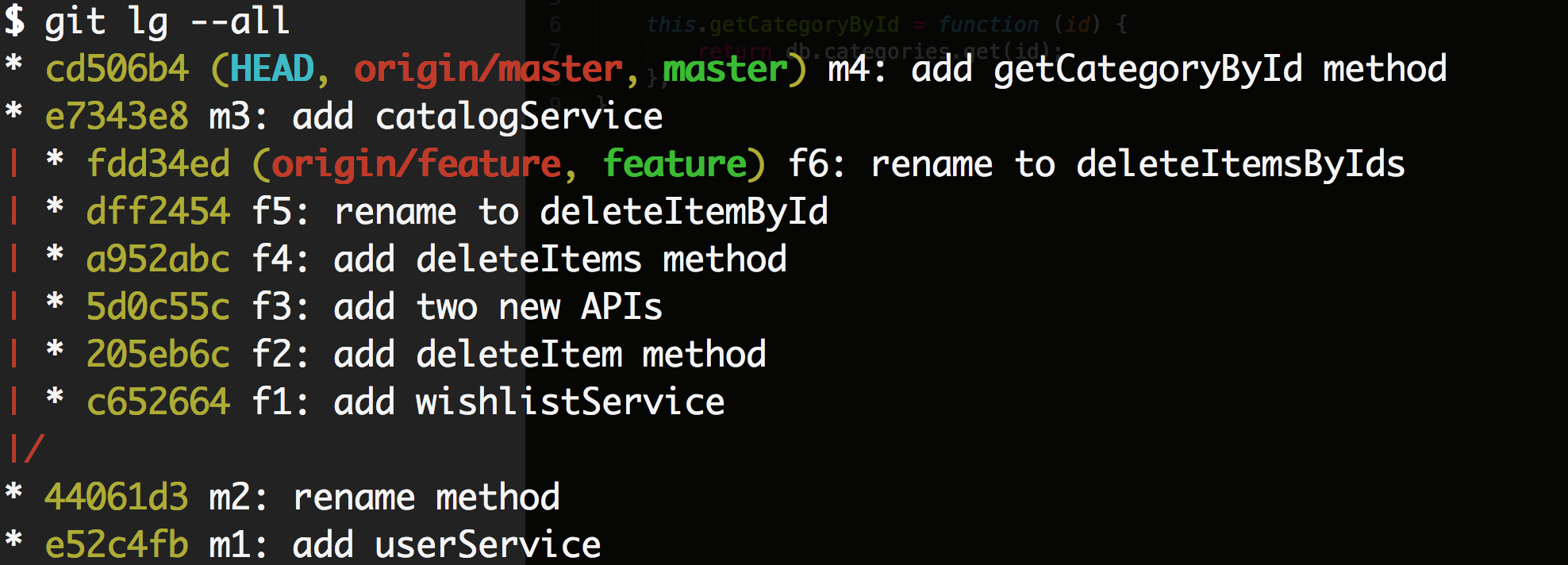
With the work split into two commits, we could just cherry-pick the precious bits into master.

Turns out, git comes with a powerful command git rebase -i which lets us do exactly that. It lets us change the history. Changing the history can be problematic and as a rule of thumb should be avoided as soon as the history is shared with others. In our case though, we are just changing history of our local feature branch. Nobody will get hurt. Promised!

Ok, let’s take a closer look at what exactly happened in commit f3. Turns out we modified two files: userService.js and wishlistService.js. Let’s say that the changes to userService.js could go straight back into master whereas the changes to wishlistService.js could not. Because wishlistService.js does not even exist in master. It was introduced in commit f1.

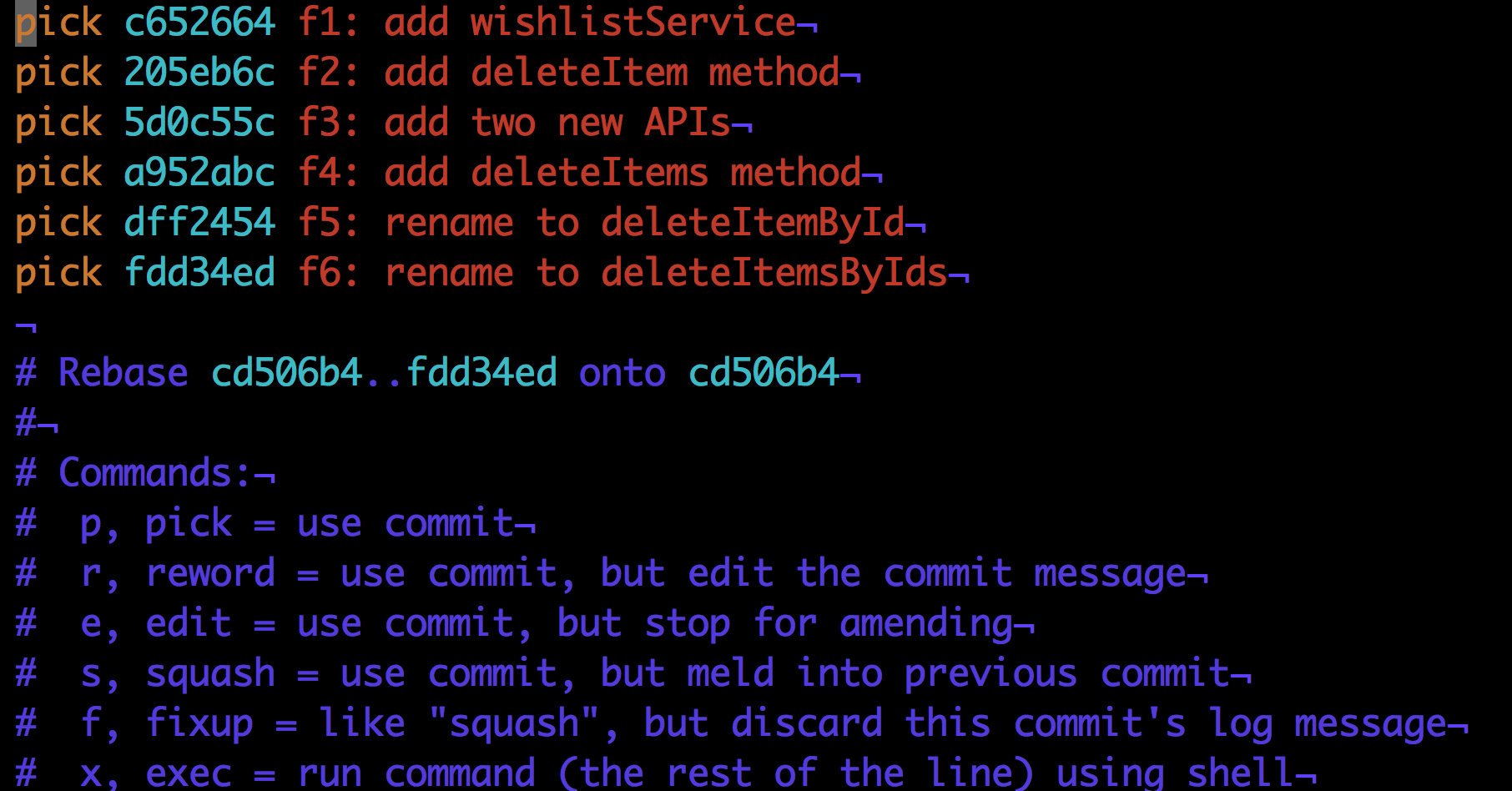
*Pro Tip: even if the changes would have been in* one *file, git could handle that. We keep things simple for this blog post though.*

We’ve set up a [public demo repository](https://github.com/thoughtram/interactive-rebase-demo) that we will use for this exercise. To make it easier to follow, each commit message is prefixed with the pseudo SHAs used in the graphs above. What follows is the branch graph as printed by git before we start to split the commit f3.



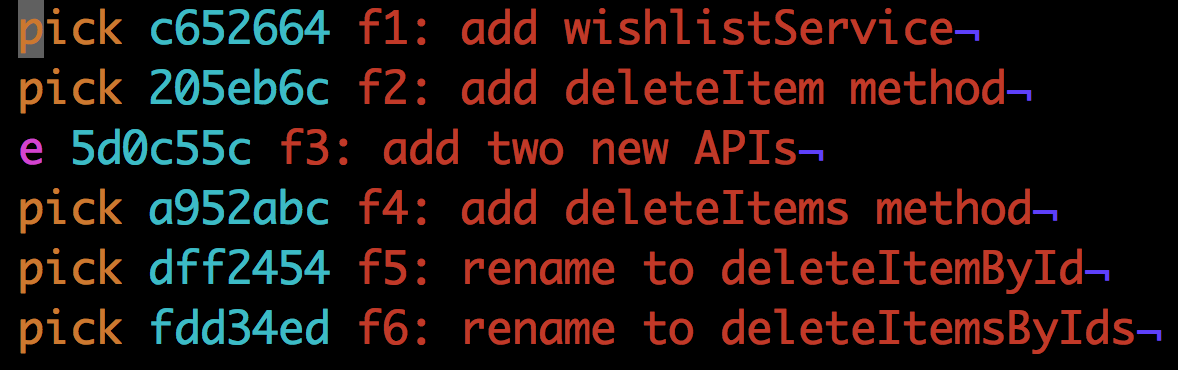
Now the first thing we want to do is to checkout our feature branch with git checkout feature. To get started with the rebase we run git rebase -i master.

Now what follows is that git opens a temporary file in the configured editor (defaults to Vim).

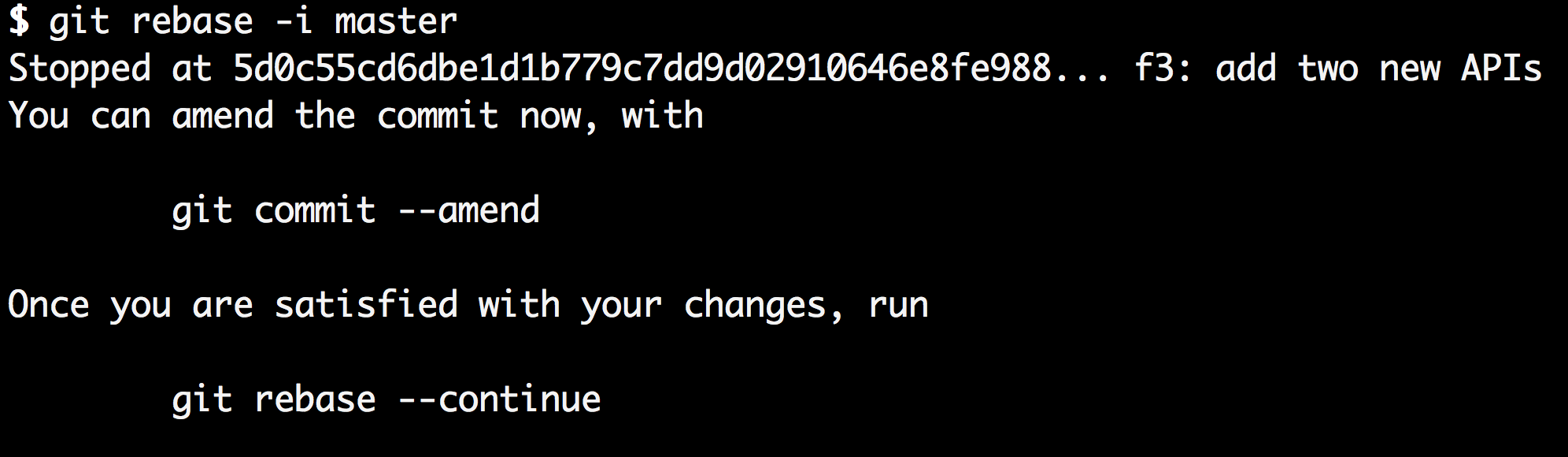


This file is meant to provide you some options for the rebase and it comes with a little cheat sheet (the blue text). For each commit we could choose between the actions pick, reword, edit, squash, fixup and exec. Each action can also be referred to by its short form p, r, e, s, f and e. It’s out of the scope of this article to describe each and every option so let’s focus on our specific task.

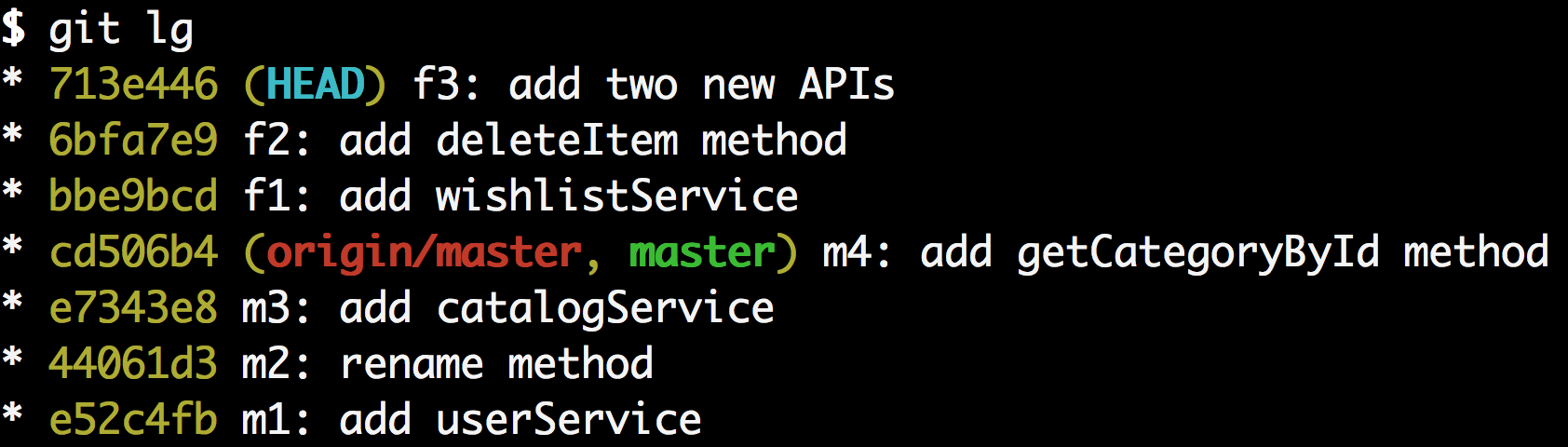
We want to choose the edit option for our f3 commit hence we change the contents to look like that.



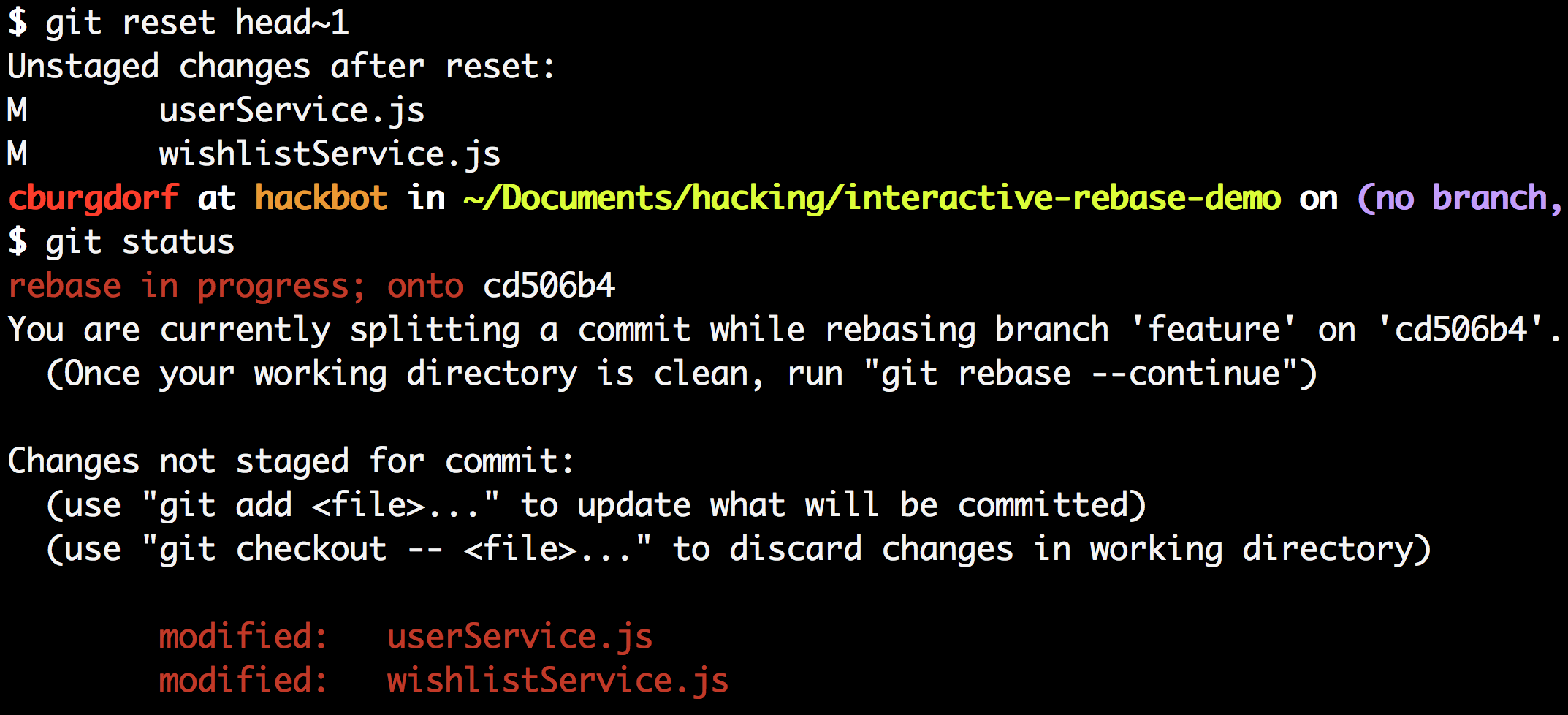
Now we save the file (in Vim <ESC> followed by :wq, followed by <RETURN>). The next thing we notice is that git stops the rebase at the commit for which we choose the edit option.



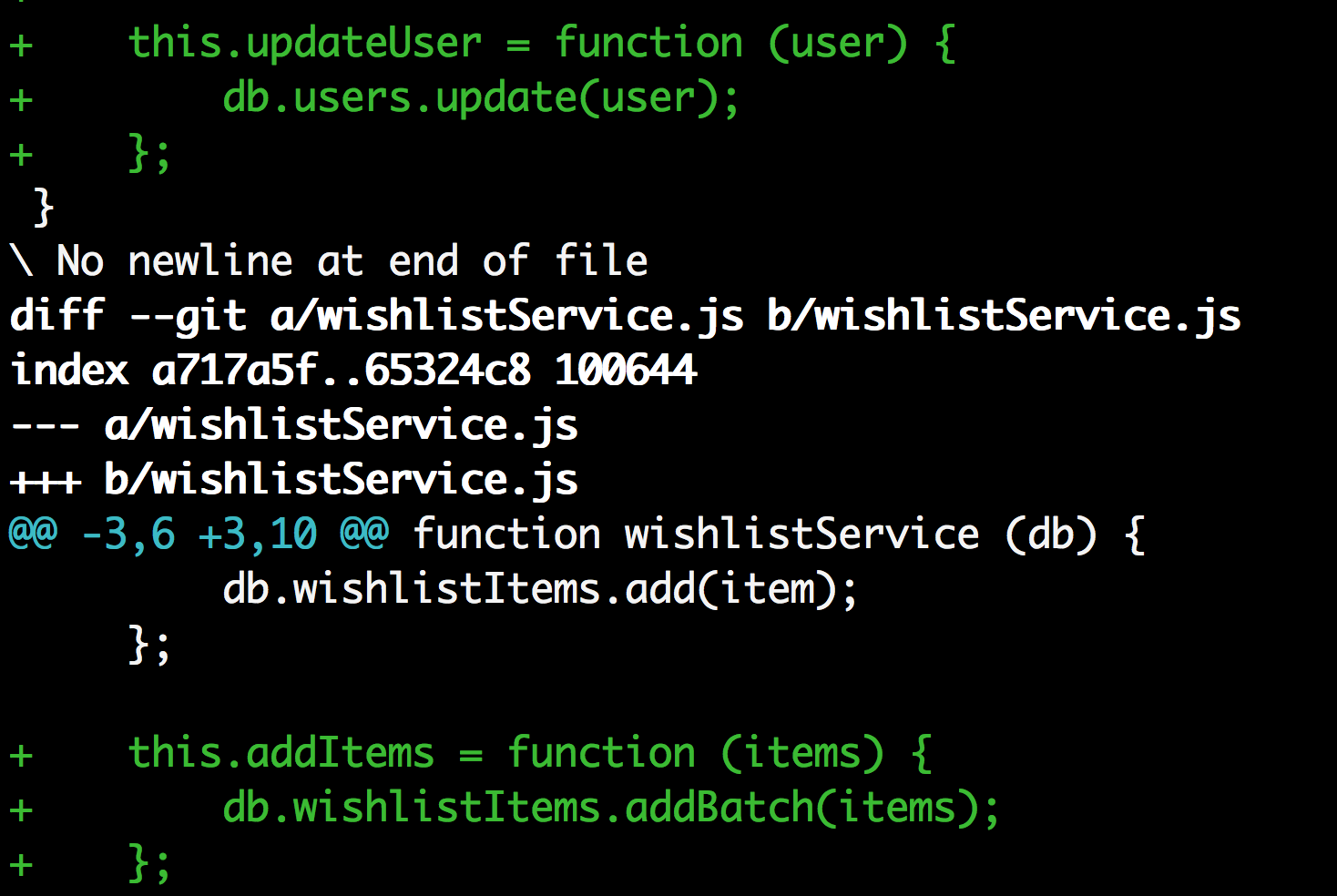
What this means is that git started to apply f1, f2 and f3 as if it was a regular rebase but then stopped **after** applying f3. In fact, we can prove that if we just look at the log at the point where we stopped.



To split our commit f3 into two commits, all we have to do at this point is to reset gits pointer to the previous commit (f2) while keeping the working directory the same as it is right now. This is exactly what the mixed mode of git reset does. Since mixed is the default mode of git reset we can just write git reset head~1. Let’s do that and also run git status right after it to see what happened.



The git status tells us that both our userService.js and our wishlistService.js are modified. If we run git diff we can see that those are exactly the changes of our f3 commit.



If we look at the log again at this point we see that the f3 is gone though.

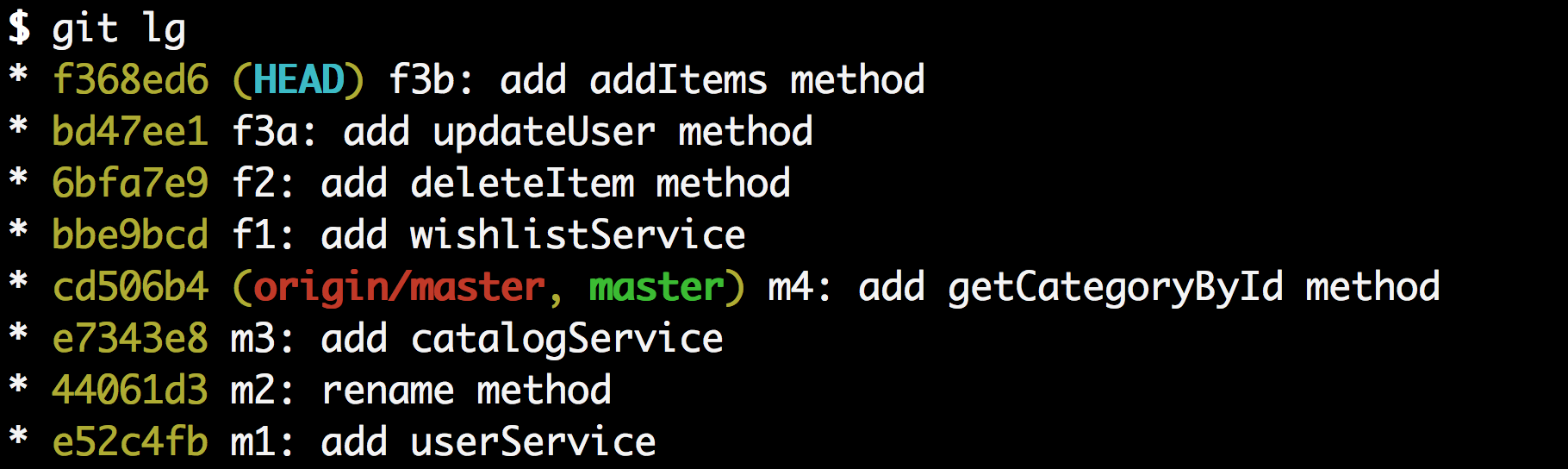


We are now at the point that we have the changes of our previous f3 commit ready to be committed whereas the original f3 commit itself is gone. Keep in mind though that we are still in the middle of a rebase. Our f4, f5 and f6 commits are not lost, they’ll be back in a moment.

Let’s make two new commits: Let’s start with the commit for the changes made to the userService.js which are fine to get picked into master. Run git add userService.js followed by git commit -m "f3a: add updateUser method".

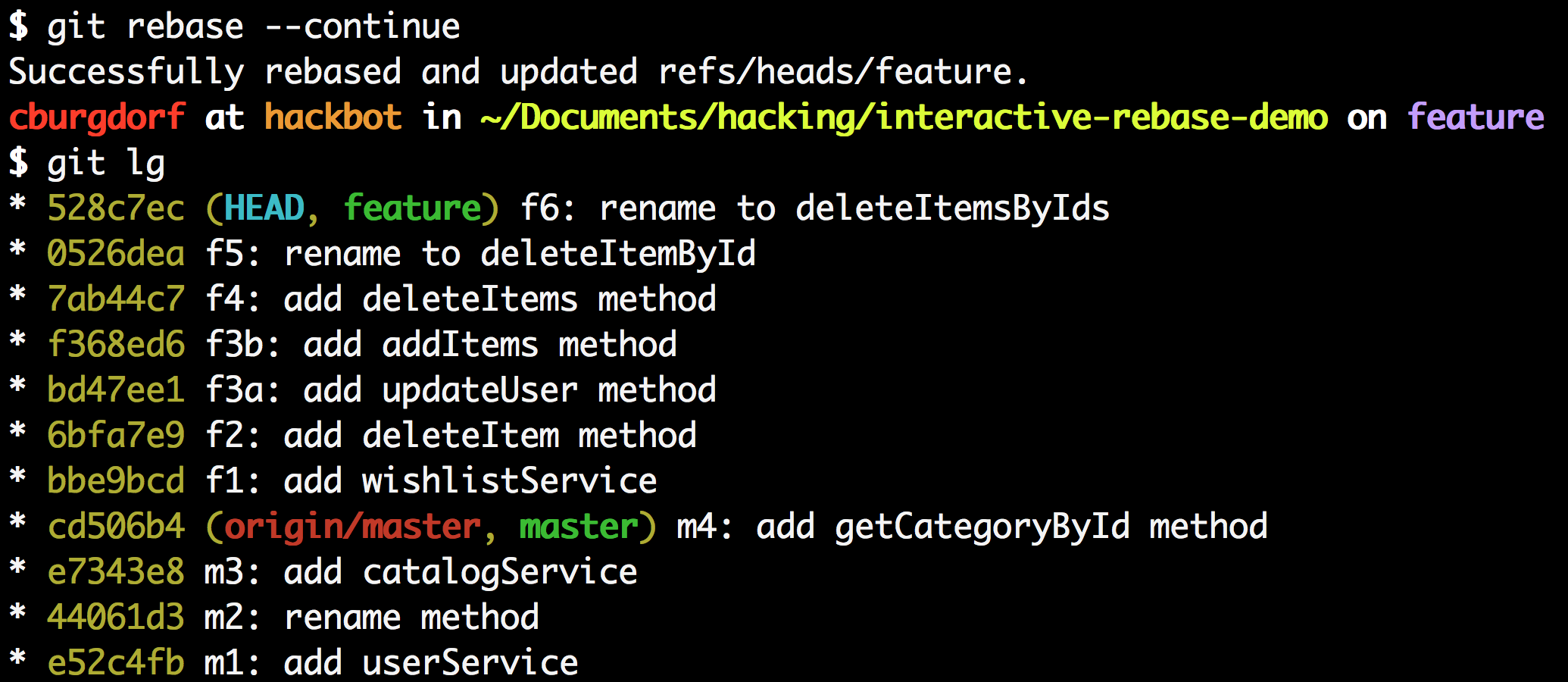
Great! Let’s create another commit for the changes made to wishlistService.js. Run git add wishlistService.js followed by git commit -m "f3b: add addItems method".

Let’s take a look at the log again.



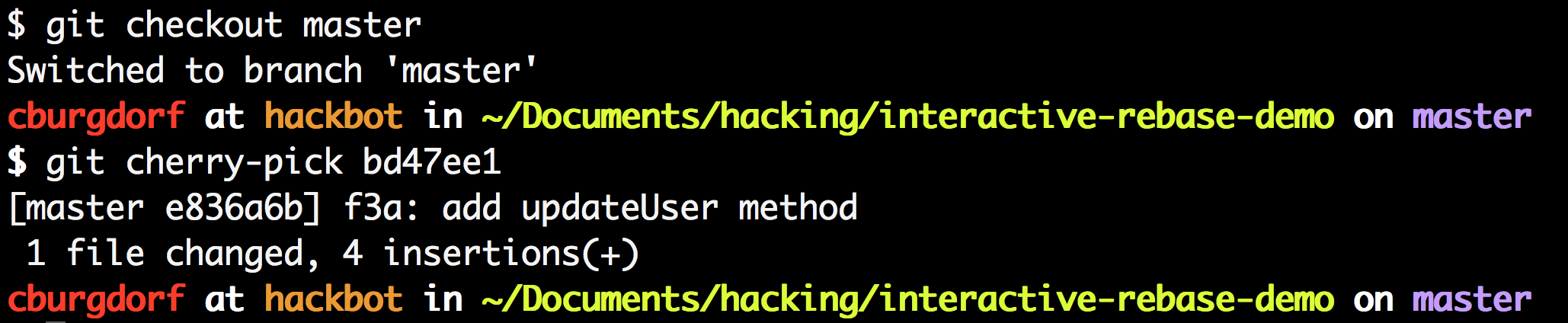
This is exactly what we wanted except our commits f4, f5 and f6 are still missing. This is because we are still in the middle of the interactive rebase and we need to tell git to continue with the rebase. This is done with the command git rebase --continue.

Let’s check out the log again.

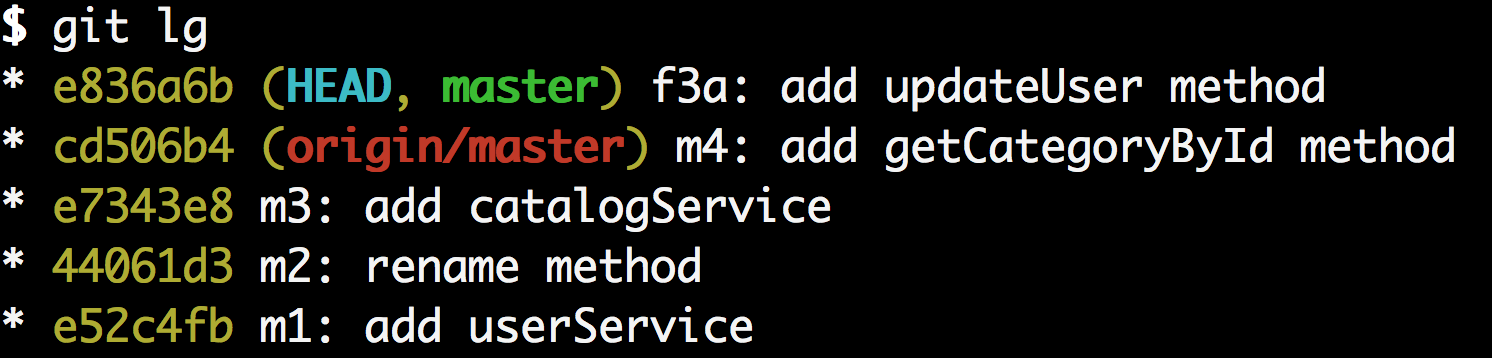


And that’s it. We now have the history we wanted. The previous f3 commit is now split into two commits f3a and f3b. The only thing left to do is to cherry-pick the f3a commit over to the master branch.

To finish the last step we first switch to the master branch. We do this with git checkout master. Now we can pick the f3a commit with the cherry-pick command. We can refer to the commit by its SHA key which is bd47ee1 in this case.



We now have the f3a commit sitting on top of latest master. Exactly what we wanted!



Given the length of the post this may seem like a lot of effort but it’s really only a matter of seconds for an advanced git user.

**5. Answer following short questions:**

**Question: How to delete a Git branch both locally and remotely?**

**Ans)** git branch -d bugfix # Must delete local branch manually

**To remove a local branch from your local system.**

**Ans)** git branch -d Mybranch # Deletes remote branch

If you are sure you want to delete it, run

git branch -D bugfix

Now to clean up deleted remote branches run

git remote prune origin

**To remove a remote branch from the server.**

**Question: How do you undo the last commit?**

$ git commit -m "Something terribly misguided" (1)

$ git reset --soft HEAD~ (2)

<< edit files as necessary >> (3)

$ git add ... (4)

$ git commit -c ORIG\_HEAD (5)

1. This is what you want to undo
2. This is most often done when you remembered what you just committed is incomplete, or you misspelled your commit message1, or both. Leaves working tree as it was before git commit.
3. Make corrections to working tree files.
4. git add whatever changes you want to include in your new commit.
5. Commit the changes, reusing the old commit message. reset copied the old head to .git/ORIG\_HEAD; commit with -c ORIG\_HEAD will open an editor, which initially contains the log message from the old commit and allows you to edit it. If you do not need to edit the message, you could use the -C option instead.

**Question: How to Edit an incorrect commit message in Git?**

git commit --amend

Will open your editor, allowing you to change the commit message of the most recent commit. Additionally, you can set the commit message directly in the command line with:

git commit --amend -m "New commit message"

**Question: What are the differences between 'git pull' and 'git fetch'?**

In the simplest terms, git pull does a git fetch followed by a git merge.

You can do a git fetch at any time to update your remote-tracking branches under refs/remotes/<remote>/. This operation never changes any of your own local branches under refs/heads, and is safe to do without changing your working copy.   
  
I have even heard of people running git fetch periodically in a cron job in the background (although I wouldn't recommend doing this).  
  
  
A git pull is what you would do to bring a local branch up-to-date with its remote version, while also updating your other remote-tracking branches.

**Question: How do you rename the local branch?**

If you want to rename a branch while pointed to any branch, do :

git branch -m <oldname> <newname>

If you want to rename the current branch, you can do:

git branch -m <newname>

**For Detailed answer:**

If you have named a branch incorrectly AND pushed this to the remote repository follow these steps before any other developers get a chance to jump on you and give you shit for not correctly following naming conventions.

**1. Rename your local branch.**  
If you are on the branch you want to rename:

|  |  |
| --- | --- |
| 1 | git branch -m new-name |

If you are on a different branch:

|  |  |
| --- | --- |
| 1 | git branch -m old-name new-name |

**2. Delete the old-name remote branch and push the new-name local branch.**

|  |  |
| --- | --- |
| 1 | git push origin :old-name new-name |

**3. Reset the upstream branch for the new-name local branch.**  
Switch to the branch and then:

|  |  |
| --- | --- |
| 1 | git push origin -u new-name |

**Question: How do I remove local files (Not in Repo) from my current Git**

**Git clean –f –d** to be sure that also **directories** are gone! you can check with git status if they are really gone.

**Question: How to Checkout remote Git branch?**

Before you can start working locally on a remote branch, you need to fetch it as called out in answers below.

To fetch a branch, you simply need to:

git fetch origin

This will fetch all of the remote branches for you. You can see the branches available for checkout with:

git branch -v -a

With the remote branches in hand, you now need to check out the branch you are interested in, giving you a local working copy:

git checkout -b test origin/test

**Question: How do you create a remote Git branch?**

First, you create your branch locally:

git checkout -b your\_branch

The remote branch is automatically created when you push it to the remote server. So when you feel ready for it, you can just do:

git push <remote-name> <branch-name>

Where <remote-name> is typically origin, the name which git gives to the remote you cloned from. Your colleagues would then just pull that branch, and it's automatically created locally.

Note however that formally, the format is:

git push <remote-name> <local-branch-name>:<remote-branch-name>

**Question: How to Change the URL for a remote Git repository?**

git remote set-url origin git://new.url.here

**Question: How to Change the author of a commit in Git?**

Interactive rebase off of a point earlier in the history than the commit you need to modify (git rebase -i <earliercommit>). In the list of commits being rebased, change the text from pick to edit next to the hash of the one you want to modify. Then when git prompts you to change the commit, use this:

git commit --amend --author="Author Name <email@address.com>"

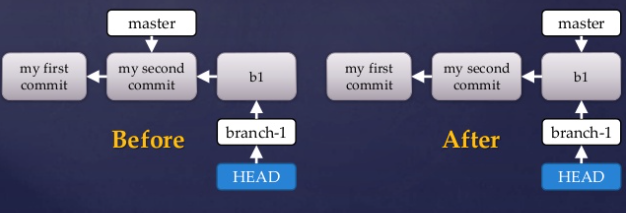
For example, if your commit history is A-B-C-D-E-F with F as HEAD, and you want to change the author of C and D, then you would...

1. Specify git rebase -i B
2. change the lines for both C and D to edit
3. Once the rebase started, it would first pause at C
4. You would git commit --amend --author="Author Name <email@address.com>"
5. Then git rebase --continue
6. It would pause again at D
7. Then you would git commit --amend --author="Author Name <email@address.com>" again
8. git rebase --continue
9. The rebase would complete.

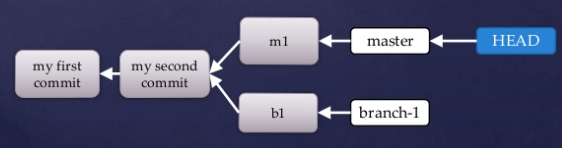
6.  **Complete the following steps using git commands on any repository of your choice. You can create your own. Please note the steps and git commands**

1. Make a fast forward merge
2. Undo fast forward merge
3. Make Commit on master
4. Make 3-way merge
5. Undo Merge commit
6. Make rebase

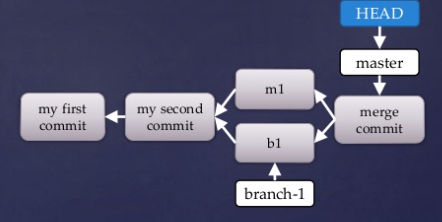
Following views will help you solve the problem

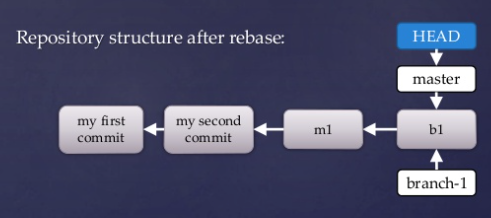
1. 

1. Step 2 &3



1. Step 4 &5



1. After Rebase

|  |
| --- |
| 6. commands |
|  | 1. Make a fast forward merge |
|  | 2. Undo fast forward merge |
|  | 3. Make Commit on master |
|  | 4. Make 3-way merge |
|  | 5. Undo Merge commit |
|  | 6. Make rebase |
|  |  |
|  | 1. git merge branch -a |
|  |  |
|  | 2. git reset -hard HEAD |
|  |  |
|  | 3. git commit -am "message for commit" |
|  |  |
|  | 4. git merge branch -a |
|  |  |
|  | 5. git reset -hard HEAD |
|  |  |
|  | 6. git rebase |

1. **Complete the following steps using git commands on any repository of your choice. You can create your own. Please note the steps and git commands. Push the repo to github. Put your link here.**
2. Create empty repo
3. Make changes in the filesystem and add to index
4. Commit
5. Make changes to file system
6. Reset to clean state
7. Make changes in the file system and add to the index
8. Commit
9. Diff between commits
10. Checkout the first commit
11. Create a new branch
12. Commit changes
13. Create Another branch
14. Diff between branches
15. Delete branch
16. Copy local repo to remote repo
17. Add remote to your repo
18. Push change to remote repo
19. Clone remote repo

Ans)

|  |
| --- |
| clear |
|  | 594 mkdir Q7 |
|  | 595 cd Q7 |
|  | 596 echo "Question 7" >> fileQ7 |
|  | 597 clear |
|  | 598 git status |
|  | 599 ls |
|  | 600 git add . |
|  | 601 git commit -am "Q 7" |
|  | 602 clear |
|  | 603 touch Q7part2 |
|  | 604 git reset --hard |
|  | 605 touch fileQ7part3 |
|  | 606 git add . |
|  | 607 git commit -am"Another commit after clean state" |
|  | 608 git log |
|  | 609 git diff 7c61d5e90c793a8d82d2b7db319c8dc1a0c5e097 83fb74d12d36364aa6b6d4834828edddbaafc984 |
|  | 610 git checkout 83fb74d12d36364aa6b6d4834828edddbaafc984 |
|  | 611 clear |
|  | 612 git branch |
|  | 613 git branch brach2 |
|  | 614 git checkout brach2 |
|  | 615 touch fileonBranc2 |
|  | 616 git add . |
|  | 617 git commit -am"file on bracnh" |
|  | 618 git diff master brach2 |
|  | 619 git branch -D brach2 |
|  | 620 git checkout master |
|  | 621 git branch -D brach2 |
|  | 622 git add . |
|  | 623 git commit -am"finished q7" |
|  | 624 ls |
|  | 625 cd .. |
|  | 626 ls |
|  | 627 git add . |
|  | 628 git commit -am |
|  | 629 git commit -am"fixed Q7" |
|  | 630 git push origin master |
|  | 631 ls |
|  | 632 cd .. |
|  | 633 mkdir q7clone |
|  | 634 cd q7clone/ |
|  | 635 git clone https://github.com/salimben60/GitFinal.git |
|  | 636 cd .. |
|  | 637 cd GitFinal2/ |
|  | 638 ls |
|  | 639 cd Q7 |
|  | 640 clear |
|  | 641 git log |
|  | 642 clear |
|  | 643 history |
|  | 644 history tail -55 |
|  | 645 history | tail -55 |
|  | 646 ls |
|  | 647 history | tail -55 >> Q7historyfile |

**Draw a flow diagram showing the above changes and give your remote repo git address here. Explain with screenshots.**

1. **What is git cherrypicking and git stash. Execute the following steps. Push it to github and give repo address. Explain. Draw the flow if necessary.**

**Git Cherrypick:**

git checkout -b exp

echo Not Cherry1 >> afile.txt

git commit -a -m NotCherry1

echo Cherry >> afile.txt

git commit -a -m Cherry

echo Not Cherry2 >> afile.txt

git commit -a -m NotCherry2

git checkout master

echo masterchange >> afile\_master.txt

git add afile\_master.txt

git commit -a -m "Master Change"

git log exp

git cherry-pick 3c8f

muralikrishna@Shrita MINGW64 ~ (master)

$ git checkout -b exp

M murali/gogula

Switched to a new branch 'exp'

muralikrishna@Shrita MINGW64 ~ (exp)

$ echo not cherry1 >> afile.txt

muralikrishna@Shrita MINGW64 ~ (exp)

$ git commit -a -m notcherry1

[exp 0730343] notcherry1

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

muralikrishna@Shrita MINGW64 ~ (exp)

$ echo cherry >> afile.txt

muralikrishna@Shrita MINGW64 ~ (exp)

$ git commit -a -m cherry

On branch exp

Untracked files:

.VirtualBox/

.bash\_history

.docker/

.eclipse/

.gitconfig

.idlerc/

.metadata/

.oracle\_jre\_usage/

.ssh/

.vim/

.viminfo

3D Objects/

AppData/

BullseyeCoverageError.txt

Cloud4/

Contacts/

Creative Cloud Files/

Desktop/

Documents/

Downloads/

Favorites/

ITU-/

IntelGraphicsProfiles/

LICENSE

Links/

Midterm/

Music/

NOTICE

NTUSER.DAT

NTUSER.DAT{174c13b1-a57d-11e5-9203-d7fe1cc80fb7}.TxR.0.regtrans-ms

NTUSER.DAT{174c13b1-a57d-11e5-9203-d7fe1cc80fb7}.TxR.1.regtrans-ms

NTUSER.DAT{174c13b1-a57d-11e5-9203-d7fe1cc80fb7}.TxR.2.regtrans-ms

NTUSER.DAT{174c13b1-a57d-11e5-9203-d7fe1cc80fb7}.TxR.blf

NTUSER.DAT{174c13b2-a57d-11e5-9203-d7fe1cc80fb7}.TM.blf

NTUSER.DAT{174c13b2-a57d-11e5-9203-d7fe1cc80fb7}.TMContainer000000000000 00000001.regtrans-ms

NTUSER.DAT{174c13b2-a57d-11e5-9203-d7fe1cc80fb7}.TMContainer000000000000 00000002.regtrans-ms

OneDrive/

Others/

Pictures/

Protek1/

RELEASE-NOTES

RUNNING.txt

RemoteSystemsTempFiles/

Saved Games/

Searches/

Tracing/

Videos/

VirtualBox VMs/

afile.txt

bin/

cloud3

cloudrep/

conf/

first.js/

git/

gitday2/

index.html

ituday2/

lib/

libgit2/

md

murali.txt

muralione/

my\_project/

mycode/

myrepo/

newdoc.txt/

newproject/

newrepo/

nrepo/

ntuser.dat.LOG1

ntuser.dat.LOG2

ntuser.ini

practise/

project/

repo/

saitejak/

sample.txt

temp/

webapps/

workspace/

nothing added to commit but untracked files present

muralikrishna@Shrita MINGW64 ~ (exp)

$ echo not cherry2 >> afile.txt

muralikrishna@Shrita MINGW64 ~ (exp)

$ git checkout master

M murali/gogula

Switched to branch 'master'

muralikrishna@Shrita MINGW64 ~ (master)

$ echo masterchange >> afile\_master.txt

muralikrishna@Shrita MINGW64 ~ (master)

$ git add afile\_master.txt

warning: LF will be replaced by CRLF in afile\_master.txt.

The file will have its original line endings in your working directory.

muralikrishna@Shrita MINGW64 ~ (master)

$ git commit -a -m "master change"

[master 8a49c8f] master change

warning: LF will be replaced by CRLF in afile\_master.txt.

The file will have its original line endings in your working directory.

2 files changed, 2 insertions(+), 1 deletion(-)

create mode 100644 afile\_master.txt

muralikrishna@Shrita MINGW64 ~ (master)

$ git log exp

commit 0730343fc432bd516bc324c8f0d3ea27fb2b608f

Author: murali <murali.protek@gmail.com>

Date: Sat Apr 30 17:29:29 2016 -0700

notcherry1

commit cadff03d90694fa241cafa1ad791338ded12c2e0

Author: murali <murali.protek@gmail.com>

Date: Sun Apr 10 16:31:50 2016 -0700

dsglsngs

commit a25a9472aa1825a9303fc75d3a7ba275a95b9f74

Merge: 3f430cb bc8dbe5

Author: murali <murali.protek@gmail.com>

Date: Mon Mar 28 10:59:34 2016 -0700

Merge https://github.com/muralicloud/gogula

commit bc8dbe5686fa8f83e79d239500dc7645f204407e

Author: murali <murali.protek@gmail.com>

Date: Mon Mar 28 10:58:32 2016 -0700

...skipping...

**SUMMARY** **OF** **LESS** **COMMANDS**

Commands marked with \* may be preceded by a number, N.

Notes in parentheses indicate the behavior if N is given.

A key preceded by a caret indicates the Ctrl key; thus ^K is ctrl-K.

h H Display this help.

q :q Q :Q ZZ Exit.

---------------------------------------------------------------------------

**MOVING**

e ^E j ^N CR \* Forward one line (or N lines).

y ^Y k ^K ^P \* Backward one line (or N lines).

f ^F ^V SPACE \* Forward one window (or N lines).

b ^B ESC-v \* Backward one window (or N lines).

z \* Forward one window (and set window to N).

w \* Backward one window (and set window to N).

ESC-SPACE \* Forward one window, but don't stop at end-of-file.

d ^D \* Forward one half-window (and set half-window to N).

u ^U \* Backward one half-window (and set half-window to N).

ESC-) RightArrow \* Left one half screen width (or N positions).

ESC-( LeftArrow \* Right one half screen width (or N positions).

...skipping...

**SUMMARY** **OF** **LESS** **COMMANDS**

Commands marked with \* may be preceded by a number, N.

Notes in parentheses indicate the behavior if N is given.

A key preceded by a caret indicates the Ctrl key; thus ^K is ctrl-K.

h H Display this help.

q :q Q :Q ZZ Exit.

---------------------------------------------------------------------------

**MOVING**

e ^E j ^N CR \* Forward one line (or N lines).

y ^Y k ^K ^P \* Backward one line (or N lines).

f ^F ^V SPACE \* Forward one window (or N lines).

b ^B ESC-v \* Backward one window (or N lines).

z \* Forward one window (and set window to N).

w \* Backward one window (and set window to N).

ESC-SPACE \* Forward one window, but don't stop at end-of-file.

d ^D \* Forward one half-window (and set half-window to N).

u ^U \* Backward one half-window (and set half-window to N).

ESC-) RightArrow \* Left one half screen width (or N positions).

ESC-( LeftArrow \* Right one half screen width (or N positions).

...skipping...

**SUMMARY** **OF** **LESS** **COMMANDS**

Commands marked with \* may be preceded by a number, N.

Notes in parentheses indicate the behavior if N is given.

A key preceded by a caret indicates the Ctrl key; thus ^K is ctrl-K.

h H Display this help.

q :q Q :Q ZZ Exit.

---------------------------------------------------------------------------

**MOVING**

e ^E j ^N CR \* Forward one line (or N lines).

y ^Y k ^K ^P \* Backward one line (or N lines).

f ^F ^V SPACE \* Forward one window (or N lines).

b ^B ESC-v \* Backward one window (or N lines).

z \* Forward one window (and set window to N).

w \* Backward one window (and set window to N).

ESC-SPACE \* Forward one window, but don't stop at end-of-file.

d ^D \* Forward one half-window (and set half-window to N).

u ^U \* Backward one half-window (and set half-window to N).

ESC-) RightArrow \* Left one half screen width (or N positions).

HELP -- Press RETURN for more, or q when done...skipping...

commit 0730343fc432bd516bc324c8f0d3ea27fb2b608f

commit 0730343fc432bd516bc324c8f0d3ea27fb2b608f

Author: murali <murali.protek@gmail.com>

Date: Sat Apr 30 17:29:29 2016 -0700

notcherry1

commit cadff03d90694fa241cafa1ad791338ded12c2e0

Author: murali <murali.protek@gmail.com>

Date: Sun Apr 10 16:31:50 2016 -0700

dsglsngs

commit a25a9472aa1825a9303fc75d3a7ba275a95b9f74

Merge: 3f430cb bc8dbe5

Author: murali <murali.protek@gmail.com>

Date: Mon Mar 28 10:59:34 2016 -0700

Merge https://github.com/muralicloud/gogula

commit bc8dbe5686fa8f83e79d239500dc7645f204407e

Author: murali <murali.protek@gmail.com>

Date: Mon Mar 28 10:58:32 2016 -0700

muralikrishna@Shrita MINGW64 ~ (master)

$

muralikrishna@Shrita MINGW64 ~ (master)

$ git cherry-pick ^C

muralikrishna@Shrita MINGW64 ~ (master)

$ git cherry-pick a25a9472aa1825a9303fc75d3a7ba275a95b9f74

error: Commit a25a9472aa1825a9303fc75d3a7ba275a95b9f74 is a merge but no -m opti on was given.

fatal: cherry-pick failed

muralikrishna@Shrita MINGW64 ~ (master)

**Git Stash**

git-stash to save the current state of your work

git stash save "work in progress for foo feature"

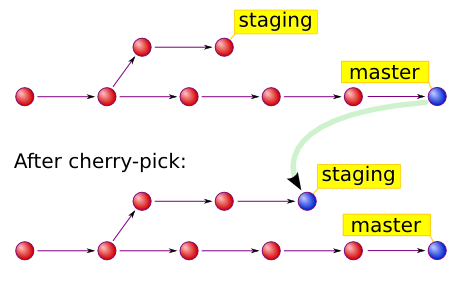
echo “bug fix” >> afile.txt

git commit -a -m "fixed the bug"

git stash pop

Cherry picking in Git is designed to apply some commit from one branch into another branch. It can be done if you eg. made a mistake and committed a change into wrong branch, but do not want to merge the whole branch.

Cherry picking in git means to choose a commit from one branch and apply it onto another.

This is in contrast with other ways such as merge and rebase which normally applies many commits onto a another branch.

Git Stashing takes the dirty state of your working directory – that is, your modified tracked files and staged changes – and saves it on a stack of unfinished changes that you can reapply at any time.

