|  |  |
| --- | --- |
| Purpose | Git commands |
| To Configure the username and email to be used with your commits. | git config –global [user.name](http://user.name/) "Ankur Jain" git config --  global user.email[ankurjain@gmail.com](mailto:ankurjain@gmail.com) |
| Transform the current directory into Git repository | git init |
| To create a working copy of a local repository | git clone /path/to/repository  git clone https://github.com/venkywarriors/C-Shap- Repository/ |
| Pretty colours to fit terminal | Git config –global color.ui true |
| To view git configuration | Git config --list |
| To Add one staging | git add <filename> |
| Add list of files in current directory | git add . |
| To Add only one or more txt to staging | git add \*.txt |
| Add all new and updated files throughout project | Git add -A |
| Add files in all directory level | Git add –all |
| Add all modifications and deleted files not new created files | Git add -u  Git add --update |

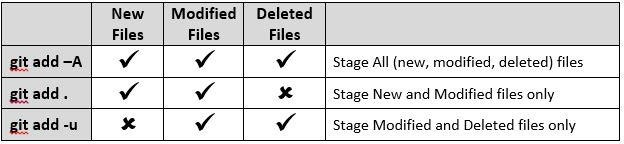
|  |  |
| --- | --- |
| Viewing information about remote repository | Git remote -v |
| List no of branches | Git branch -a |
| Add new files + modifications but ignore deleted files | Git add --ignore-removal |
| Remove one file from staging area | Git rm –cached FILE\_NAME |
| Remove a directory | Git rm -r –cached directory |
| Remove a file | Git rm FILE\_NAME or git rm -f <filename> |
| Remove/Discard unstaged changes in current directory | Git checkout -- . |
| Remove all unstaged files | Git reset Head/git reset |
| To unstage one file | Git reset Head FILE\_NAME |
| Remove commit | Git reset –soft <Hashcode> |
| To commit Changes | git commit -m "Commit message" |
| Add files to staging and commit | Git commit -a -m "Commit message" |
| Changed a commit message | Git commit –amend -m "Commit message" |
| Push code to the master branch of remote repository | git push -u origin master |
| To check git status | git status |
| To push the code to the remote server | git remote add origin https://GitHub.com.\*\*\*.git |

|  |  |
| --- | --- |
| To get the list of all currently configured remote repositories | git remote -v |
| To update local repo from global repository | Git pull origin master |
| To Create and switch to a new branch | git checkout -b <branchname> |
| To Switch from one branch to another: | git checkout <branchname> |
| To List all the branches in the repository | git branch |
| Create a new branch | Git branch new-branch |
| Clone a branch | Git clone -b BRANCH\_NAME URL “new  branch name” |
| To Delete the particular feature branch: | git branch -d <branchname> |
| To Push the branch to the remote repository. | git push origin <branchname> |
| To Push all branches to remote repository: | git push --all origin |
| To Delete a branch on a remote repository: | git push origin :<branchname> |
| To merge a branch into an active branch: | git merge <branchname> |
| To view the changes in the file which was staged | Git diff – cached filename |
| To view difference in two commits | Git difftool Head~2 Head~1  git diff <Hashcode> <hashcode> |

|  |  |
| --- | --- |
| To view all the merge conflicts:  View the conflicts against the base file:  Preview changes, before merging: | git diff  git diff --base <filename>  git diff <sourcebranch> <targetbranch> |
| To view changes in local repository and remote repository | Git diff master origin/master |
| Print only merge commits | Git log --merges |
| Limits the number of commits to show to 3. | git log -3 |
| --stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end Show statistics for files modified in each commit. | git log --stat |
| To add a new file to the previous commit | $ git add new-file  $ git commit --amend |

|  |  |
| --- | --- |
| will remove the last commit from the current branch, but the file changes will stay in  your [working tree](https://mijingo.com/blog/what-is-the-working-tree-in-git). Also the changes will stay on your index, so following with a git commit will create a commit with the exact same changes as the commit you "removed" before | git reset -- soft HEAD~1 |
| will lose all uncommited changes in addition to the changes introduced in the last commit. The changes won't stay in your working tree so doing a git status command will tell you that you don't have any changes in your repository | git reset -- hard HEAD~1 |
| To view difference between two commits | Git diff hash1 hash2 |
| Comparing staging area with the repository | Git diff –staged |
| How to commit directly to the repository without staging | Git commit -am “commit msg” |
| Git revert file to a specific branch | git checkout branchName\_Which\_Has\_stable\_Commit fileName |
| Git revert file to a specific commit | git checkout Last\_Stable\_commit\_Number -- fileName |
| git-show is a command line utility that is used to view expanded details on Git objects such as blobs, trees, tags, and commits. git-show has specific behavior per object type | |  |  | | --- | --- | | git show commitA...commitD |  | | This will output all commits in the range from commitA to commit D | |   Git show head |

|  |  |
| --- | --- |
| Comparing staging area with repository | Git diff –staged |
| Discard changes in file | Git checkout -- filename |
| Show the whole commit history, but skip any merges | Git log --no-merges |
| To get the information about the previous commits to the project. | git log |



**[How to copy a branch from one GitHub repository to another?](https://stackoverflow.com/questions/34384642/how-to-copy-a-branch-from-one-github-repository-to-another)**

Simply add the new remote (Organization) to your old repository (master). Once you did it simply push the branch A to the new (organization) repository.

cd <old repository>

git remote add origin2 <new\_url> git push origin2 <branch A>

Now you should have the new branch A in your new repository.

The point is to add new remote and to push the branch to your new repository.

Second way id to update the current repository remote to point to the new location:

git remote set-url origin <new url>

And then push the branch.



#### How can I reset or revert a file to a specific revision?

**You can do it in 4 steps:**

* 1. revert the entire commit with the file you want to specifically revert - it will create a new commit on your branch
  2. soft reset that commit - removes the commit and moves the changes to the working area
  3. handpick the files to revert and commit them
  4. drop all other files in your work area

**What you need to type in your terminal**:

1. git revert <commit\_hash>
2. git reset HEAD~1
3. git add <file\_i\_want\_to\_revert> && git commit -m 'reverting file'
4. git checkout .

I have a commit abc1 and after it I have done several (or one modification) to a file file.txt. **Now say that I messed up something in the file file.txt and I want to go back to a previous commit abc1.**

1.git checkout file.txt : this will remove local changes, if you don't need them 2.git checkout abc1 file.txt : this will bring your file to your *wanted* version

1. git commit -m "Restored file.txt to version abc1" : this will commit your reversion.
2. git push : this will push everything on the remote repository

#### Copy files between Git branches

To copy file(s) from from another branch to the current one in Git, there are two possible options:

1. use the git show command:
2. $ git show <branch\_name>:path/to/file >path/to/local/file
3. use the git checkout command:
4. $ git checkout <branch\_name> path/to/new/file

the latter form checkouts a file from another branch and adds it to this branch; the file will still need to be added to the branch with git add command, but the file is already present.

**How .gitignore Works**

.gitignore file is a plain text file where each line contains a pattern for files/directories to ignore. Generally, this is placed in the root folder of the repository

**Negation**

You can use a prefix of ! to negate a file that would be ignored.

\*.log

!example.log

[**How can I reset or revert a file to a specific revision?**](https://stackoverflow.com/questions/215718/how-can-i-reset-or-revert-a-file-to-a-specific-revision)

Assuming the hash of the commit you want is c5f567: git checkout c5f567 -- file1/to/restore

What Is Git Rebase?

Git rebase is a command that allows developers to integrate changes from one branch to another.

How Does Git Rebase Work?

Git rebase compresses all the changes into a single “patch.” Then it integrates the patch onto the target branch. Unlike merging, rebasing flattens history. It transfers the completed work from one branch to another. In the process, unwanted history is eliminated.(No commit hashcode after merging)

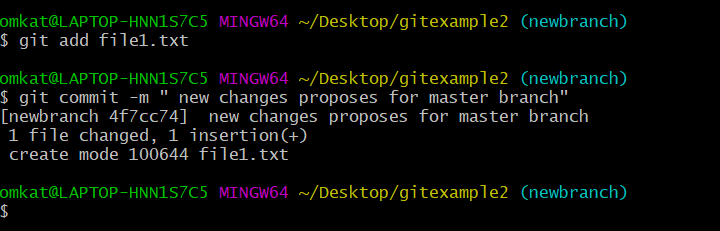
Git Rebase vs. Merge: Similarities and Differences

Git rebase and merge both integrate changes from one branch into another. Where they differ is how it's done. Git rebase moves a feature branch into a master. Git merge adds a new commit, preserving the history.

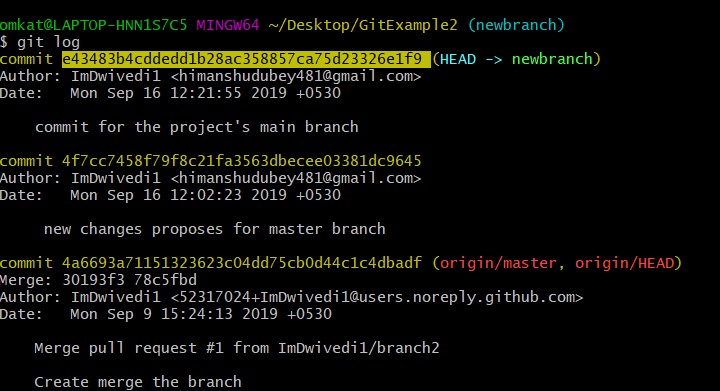
|  |  |
| --- | --- |
| Table 2. Common options to git log | |
| **Option** | **Description** |
| --name-only | Show the list of files modified after the commit information. |
| --name-status | Show the list of files affected with added/modified/deleted information as well. |
| --abbrev-commit | Show only the first few characters of the SHA-1 checksum instead of all 40. |
| --relative-date | Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format. |
| --graph | Display an ASCII graph of the branch and merge history beside the log output. |
| --pretty | Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format). |
| --oneline | Shorthand for --pretty=oneline --abbrev-commit used together |
| 3. Options to limit the output of git log | |
| **Option** | **Description** |
| -<n> | Show only the last n commits |
| --since, --after | Limit the commits to those made after the specified date. |
| --until, --before | Limit the commits to those made before the specified date. |
| --author | Only show commits in which the author entry matches the specified string. |
| --committer | Only show commits in which the committer entry matches the specified string. |
| --grep | Only show commits with a commit message containing the string |
| -S | Only show commits adding or removing code matching the string |

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

Scenerio1: Accidently make a commit in a wrong branch.

Git cherry-pick is helpful to apply the changes that are accidentally made in the wrong branch. Suppose I want to make a commit in the master branch, but by mistake, we made it in any other branch. See the below commit.

In the above example, I want to make a commit for the master branch, but accidentally I made it in the new branch. To make all the changes of the new branch into the master branch, we will use the git pull, but for this particular commit, we will use git cherry-pick command.

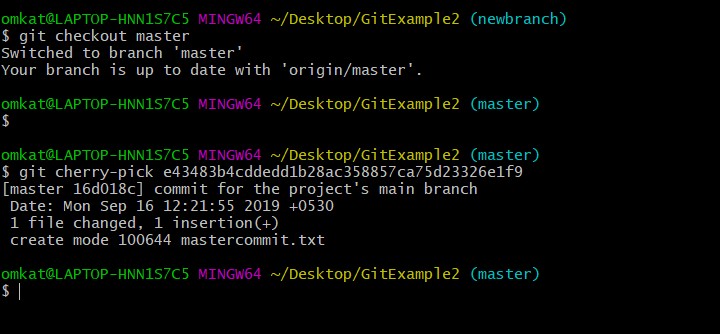


In the given output, I have used the git log command to check the commit history. Copy the particular commit id that you want to make on the master branch. Now switch to master branch and cherry-pick it there. See the below output:

Syntax:

* $ git cherry-pick **<commit** id**>**

Output:



From the given output, you can see that I have pasted the commit id with git cherry-pick command and made that commit into my master branch. You can check it by git log command.

What happens if we want to **roll back to a previous commit**. For example, if we want to reset master to point to the commit two back from the current commit, we could use either of the following methods

**$ git reset 9ef9173** (using an absolute commit SHA1 value 9ef9173) or

**$ git reset current~2** (using a relative value -2 before the "current" tag)

**Clean command**

Step 1 is to show what will be deleted by using the -n option:

# Print out the list of files which will be removed (dry run) git clean -n

# Delete the files from the repository git clean -f

* + To remove directories, run git clean -f -d or git clean -fd
  + To remove ignored files, run git clean -f -X or git clean -fX
  + To remove ignored and non-ignored files, run git clean -f -x or git clean -fx

Use git clean -f -d to make sure that **directories** are also removed.

* 1. Don’t actually remove anything, just show what would be done.
  2. git clean -n or git clean --dry-run
  3. Remove untracked directories in addition to untracked files. If an untracked directory is managed by a different Git repository, it is not removed by default. Use the -f option twice if you really want to remove such a directory.
  4. git clean -fd

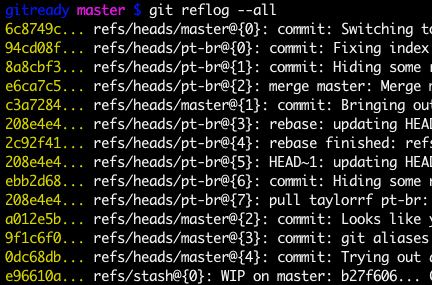
You can then check if your files are really gone with git status.

**git reflog**

Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." (you can see what you have done so far)

### You can also use the option to get more

--all

detailed information about different branches and even the stash

git revert a72ef023

|  |  |  |  |
| --- | --- | --- | --- |
| $ git revert HEAD |  | | |
| [master b9cd081] Revert "prepend content to demo file" | | |  |
| 1 file changed, 1 deletion(-) | |  | |
| Git revert expects a commit ref was passed in and will not execute without one. Here we have passed in the HEAD ref. This will revert the latest commit. This is the same behavior as if we reverted to commit 3602d8815dbfa78cd37cd4d189552764b5e96c58. Similar to a merge, a revert will create a new commit which will open up the configured system editor prompting for a new commit message. Once a commit message has been entered and saved Git will resume operation. We can now  examine the state of the repo using git log and see that there is a new commit added to the previous log: | | | |

|  |  |
| --- | --- |
| $ git log --oneline  1061e79 Revert "pre 86bb32e prepend con 3602d88 add new con  299b15f initial com | pend content to demo file" tent to demo file  tent to demo file  mit |
| Note that the 3rd commit is still in the project history after the revert. Instead of deleting it, git  revert added a new commit to undo its changes. | |

$ git fetch origin

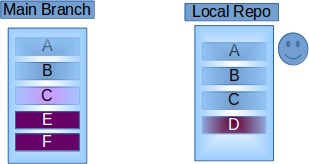
##### git fetch really only downloads new data from a remote repository - but it doesn't integrate any of this new data into your working files.

##### Git fetch origin <branch>. It would fetch all the changes into your computer, but keep it separate from your local development/workspace.

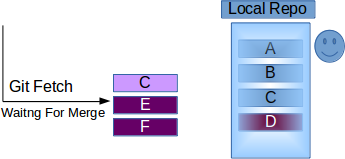
$ git pull origin master

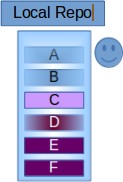
##### git pull, in contrast, is used with a different goal in mind: to update your current HEAD branch with the latest changes from the remote server. This means that pull not only downloads new data; it also directly integrates it into your current working copy files.

Git fetch + git merge = git pull

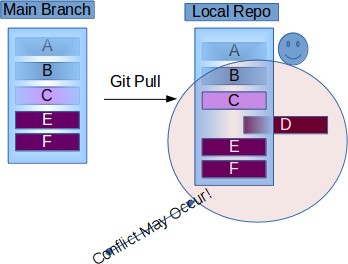


1. **Git Fetch-** This will Download all the changes that have been made to the origin/main branch project which are not present in your local branch. And will wait for the Git Merge command to apply the changes that have been fetched to your Repository or branch.



So now You can carefully monitor the files before merging it to your repository. And you can also modify D if required because of Modified C.

1. **Git Pull-** This will update your local branch with the origin/main branch i.e. actually what it does is a combination of Git Fetch and Git merge one after another. *But this may Cause Conflicts to occur, so it’s recommended to use Git Pull with a clean copy.*



Let's say you had commits:

C B A

git revert B, will create a commit that undoes changes in B.

git revert A, will create a commit that undoes changes in A, but will not touch changes in B

Note that if changes in B are dependent on changes in A, the revert of A is not possible. git reset --soft A, will change the commit history and repository; staging and working directory will still be at state of C.

git reset --mixed A, will change the commit history, repository, and staging; working directory will still be at state of C.

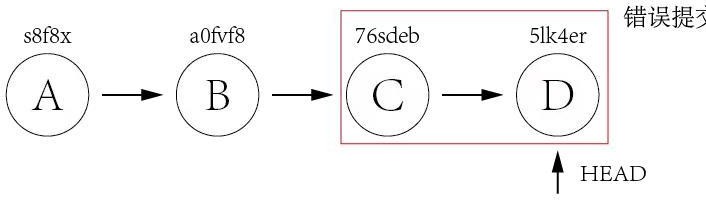
git reset --hard A, will change the commit history, repository, staging and working directory; you will go back to the state of A completely.

git reset vs git revert

When maintaining code using version control systems such as git, it is unavoidable that we need to rollback some wrong commits either due to bugs or temp code revert.

## git reset

Assuming we have below few commits.

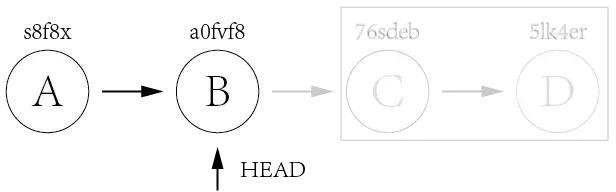


Commit A and B are working commits, but commit C and D are bad commits. Now we want to rollback to commit B and drop commit C and D. Currently HEAD is pointing to commit D **5lk4er**, we just need to point HEAD to commit B **a0fvf8** to achieve what we want.

It's easy to use git reset command.

git reset --hard a0fvf8

After executing above command, the HEAD will point to commit B.



But now the remote origin still has HEAD point to commit D, if we directly

use git push to push the changes, it will not update the remote repo, we need to add a -f option to force pushing the changes.

git push -f

The drawback of this method is that all the commits after HEAD will be gone once the reset is done.

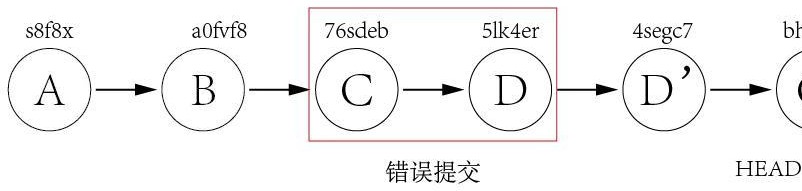
## git revert

The use of git revert is to create a new commit which reverts a previous commit. The HEAD will point to the new reverting commit.

For the example of git reset above, what we need to do is just reverting commit D and then reverting commit C.

git revert 5lk4er git revert 76sdeb

Now it creates two new commit D' and C'

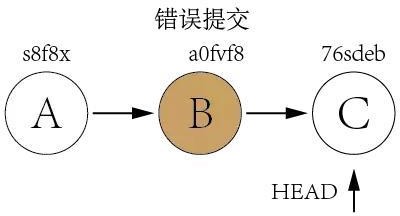


In above example, we have only two commits to revert, so we can revert one by one. But what if there are lots of commits to revert? We can revert a range indeed.

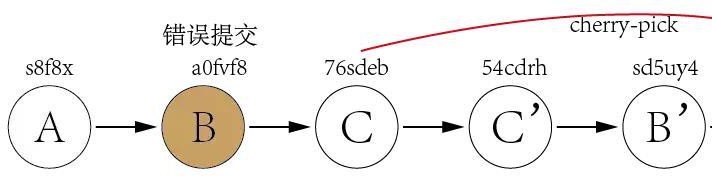
git revert OLDER\_COMMIT^..NEWER\_COMMIT

This method would not have the disadvantage of git reset, it would point HEAD to newly created reverting commit and it is ok to directly push the changes to remote without using the -f option.

Now let's take a look at a more difficult example. Assuming we have three commits but the bad commit is the second commit.



It's not a good idea to use git reset to rollback the commit B since we need to keep commit C as it is a good commit. Now we can revert commit C and B and then use cherry-pick to commit C again.



From above explanation, we can find out that the biggest difference between git reset and git revert is that git reset will reset the state of the branch to a previous state by dropping all the changes post the desired

commit while git revert will reset to a previous state by creating new reverting commits and keep the original commits. It's recommended to use git revert instead of git reset in enterprise environment.

Git Checkout File

Checking out a file is similar to using git reset with a file path, except it updates the working

directory instead of the stage. Unlike the commit-level version of this command, this does not move the HEAD reference, which means that you won’t switch branches.

For example, the following command makes foo.py in the working directory match the one from the 2nd-to-last commit:

git checkout HEAD~2 foo.py

Just like the commit-level invocation of git checkout, this can be used to inspect old versions of a project—but the scope is limited to the specified file.

If you stage and commit the checked-out file, this has the effect of “reverting” to the old version of that file. Note that this removes all of the subsequent changes to the file, whereas the git

revert command undoes only the changes introduced by the specified commit.

Like git reset, this is commonly used with HEAD as the commit reference. For instance, git checkout HEAD foo.py has the effect of discarding unstaged changes to foo.py. This is similar behavior to git reset HEAD -- hard, but it operates only on the specified file.

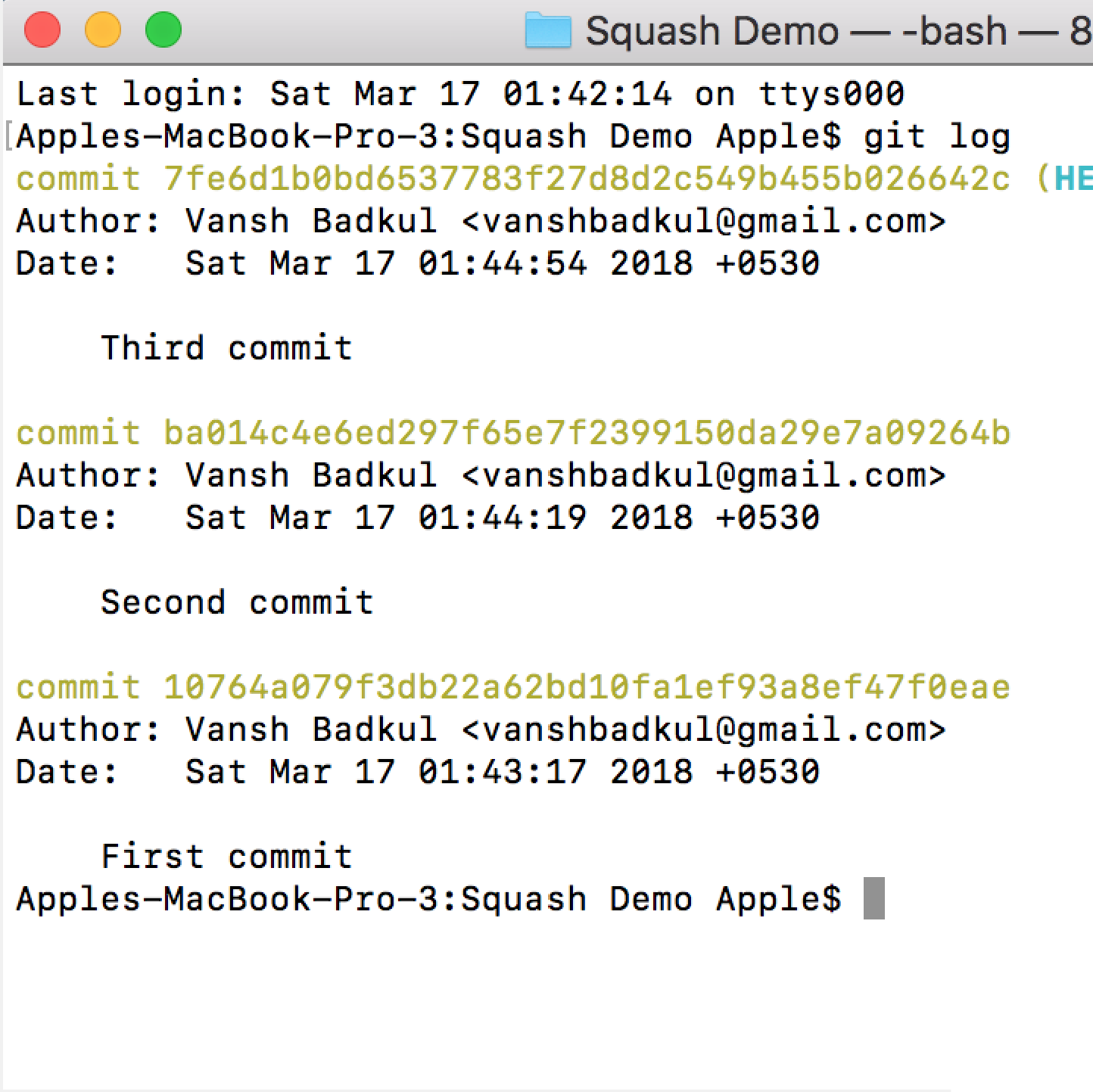
|  |  |  |
| --- | --- | --- |
| **Command** | **Scope** | **Common use cases** |
| git reset | Commit- level | Discard commits in a private branch or throw away uncommited changes |
| git reset | File-level | Unstage a file |
| git checkout | Commit- level | Switch between branches or inspect old snapshots |
| git checkout | File-level | Discard changes in the working directory |
| git revert | Commit- level | Undo commits in a public |
| git revert | File-level | (N/A) |

# Squashing

###### In Git you can merge several commits into one with the

powerful interactive rebase.

1. Check your commit history:*git log*



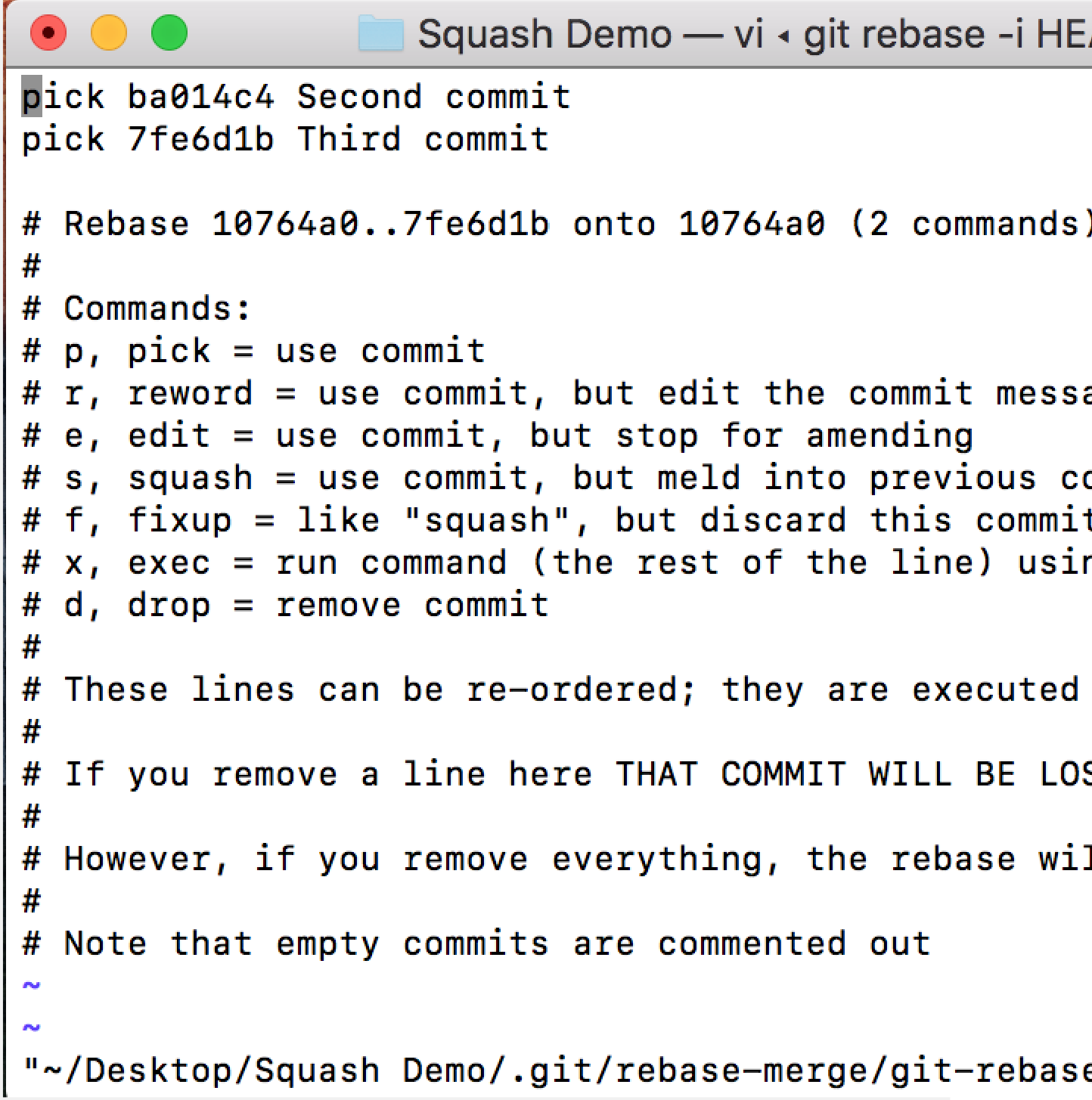
1. At this moment the current HEAD is at the last commit, in my case it’s the “Third commit”.

Now comes the squashing part! Let’s say you want to squash the last 3 commits

(i.e commit pointing to the current HEAD + the previous 2 commits) : Get the previous 2 commits from the current HEAD, use:

*git rebase -i HEAD~2*

Note: You can get previous “n” commits, depends on your case.



1. Choose the commits that you want to squash. For that, replace “pick” with “squash” to the commits you want to squash.



Note: Press “i” button to edit in the nano editor.

Notice the third line in the editor : It says Rebase <commit-id-1>..<commit-id- 2> onto <commit-id-3> which means that you can choose to squash your commits from commit-id-2 to commit-id-3 into commit-id-1

In my case : It will mean that I’m squashing “Second commit” and “Third commit” into my “First commit”, which will result in a combination of 3 commits.

Save and exit the editor :

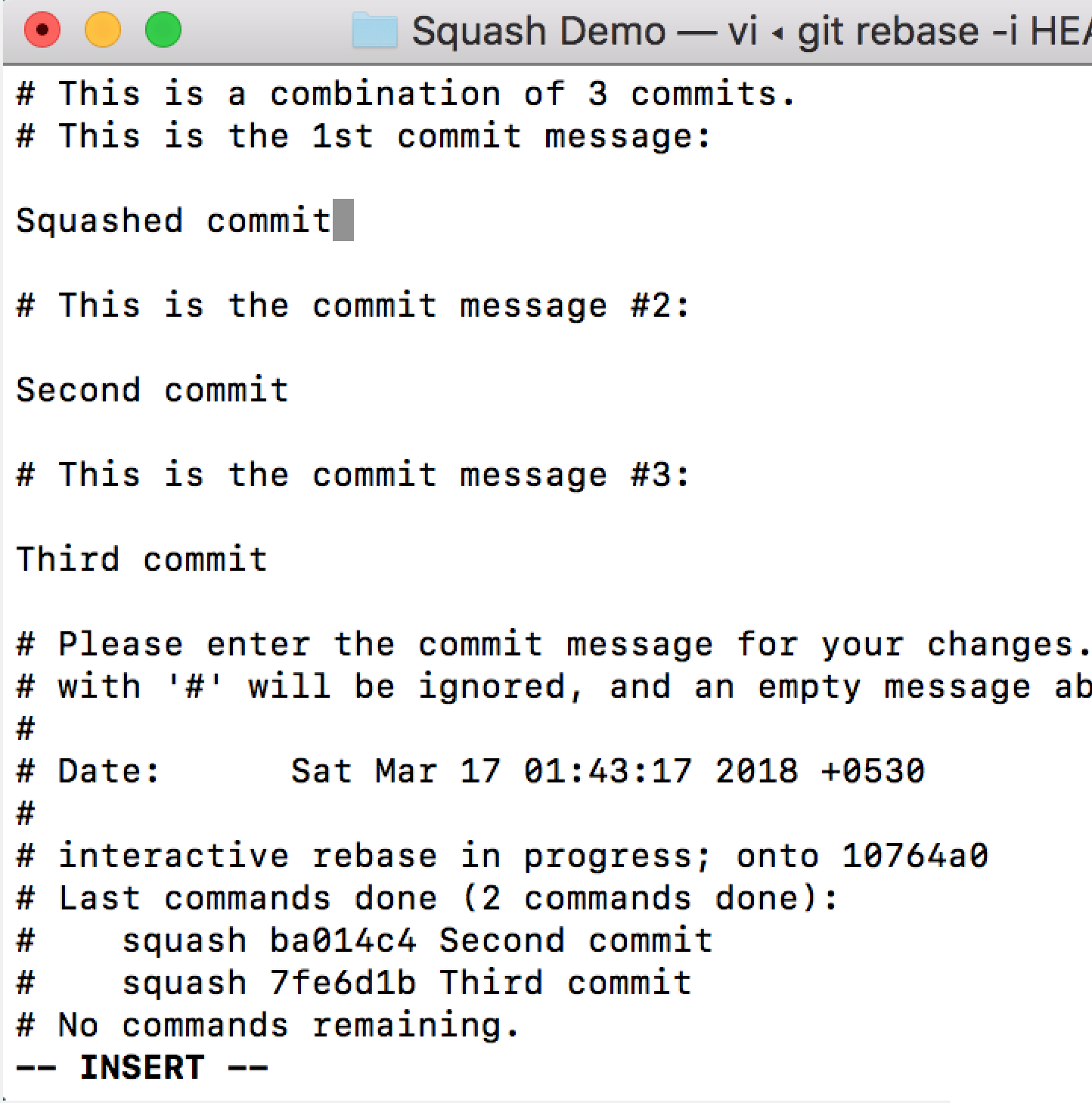
*:wq*

1. On pressing enter, you’ll get a screen with the editor where you can edit your

commit messages.

I’m editing my 1st commit message to (“Squashed commit”) because this will

be the combined commit message.

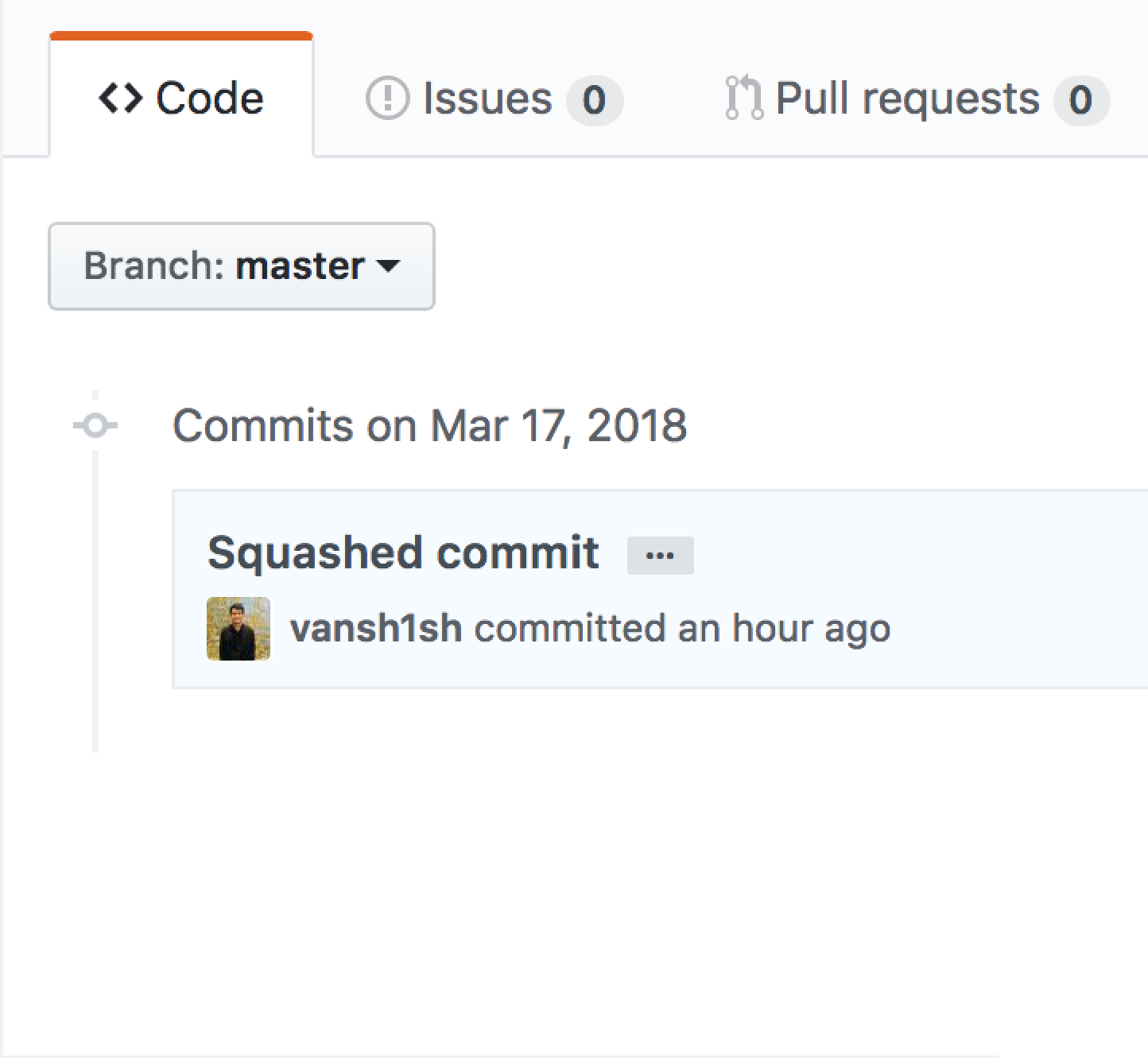


Save and exit the editor.

1. Use this command to push the changes to your github repository:

*git push -f origin master*

You can check the squashed commit on your online Github repository. Also now, when you check your PR, the multiple commits will be updated to a single commit automatically.



#### git stash - How to Save Your Changes Temporarily

##### The "git stash" command can help you to (temporarily but safely) store your uncommitted local changes - and leave you with a clean working copy

$ git stash

Saved working directory and index state WIP on master:

2dfe283 Implement the new login box

HEAD is now at 2dfe283 Implement the new login box

Your working copy is now clean: all uncommitted local changes have been saved on this kind of "clipboard" that Git's Stash represents. You're ready to start your new task (for example by pulling changes from remote or simply switching branches).

Continuing Where You Left Off

##### As already mentioned, Git's Stash is meant as a temporary storage. When you're ready to continue where you left off, you can restore the saved state easily:

$ git stash pop

##### The "pop" flag will reapply the *last saved* state and, at the same time, delete its representation on the Stash (in other words: it does the clean-up for you).

In case you want to apply a specific Stash item (not the most recent one), you can provide the index name of that item in the "pop" option:

$ git stash pop stash@{2}

* 1. Git stash save
  2. Git stash list
  3. Git stash apply
  4. Git stash pop
  5. Git stash show
  6. Git stash branch <name>
  7. Git stash clear
  8. Git stash drop

***Git stash save***

This command is like Git stash. But this command comes with various options. I will discuss some important options in this post.

**Git stash with message**

git stash save “Your stash message”.

The above command stashes with a message. We will see how this is helpful in a bit.

**Stashing untracked files**

You can also stash untracked files.

git stash save -u

or

git stash save --include-untracked

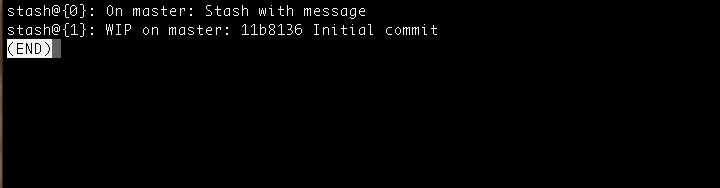
***Git stash list***

Before discussing this command, let me tell you something about how stash works.

When you Git stash or Git stash save, Git will actually create a Git commit object with some name and then save it in your repo.

So it means that you can view the list of stashes you made at any time. See the example below:

git stash list



git stash list example

You can see the list of stashes made. And the most recent stash made is in the top.

***Git stash apply***

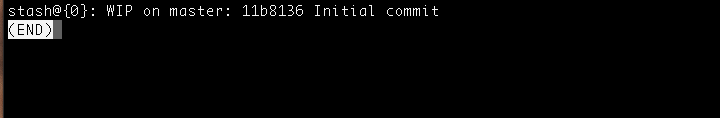
This command takes the top most stash in the stack and applies it to the repo. In our case it is **stash@{0}**

If you want to apply some other stash you can specify the stash id.

git stash apply stash@{1}

***Git stash pop***

This command is very similar to stash apply but it deletes the stash from the stack after it is applied.



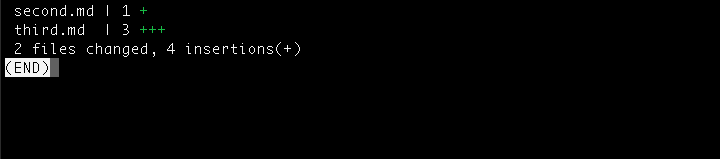
Git stash pop example

As you can see the top stash is deleted and **stash@{0}** is updated with older stash. Likewise, if you want a particular stash to pop you can specify the stash id.

git stash pop stash@{1}

***Git stash show***

This command shows the summary of the stash diffs. The above command considers only the latest stash.

Here’s the example:

Git stash show example

If you want to see the full diff, you can use

git stash show -p

Likewise with other commands, you can also specify the stash id to get the diff summary.

git stash show stash@{1}

***Git stash branch <na****me>*

This command creates a new branch with the latest stash, and then deletes the latest stash ( like stash pop).

If you need a particular stash you can specify the stash id.

git stash branch <name> stash@{1}

This will be useful when you run into conflicts after you’ve applied the stash to the latest version of your branch.

***Git stash clear***

This command deletes all the stashes made in the repo. It maybe impossible to revert.

***Git stash drop***

This command deletes the latest stash from the stack. But use it with caution, it maybe be difficult to revert.

git stash drop stash@{1}

