**License Statement**

**Creative Commons Attribution 4.0**

This work is licensed under the Creative Commons Attribution 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA

**About**

This workshop style self-learning material is to getting you from zero knowledge about ‘git’ to being a hero in using it. Git is an important software configuration tool used for software versioning popularly used in open-source development. So, the target audience of this material is for any aspiring individuals (college students or engineers) who wants to acquire the skill. The material assumes that you have Linux basic shell commands and bash knowledge.

Just like any material created, there could be mistake, typo or grammar (since I am not native speaker). So, I would appreciate that you share with me so that I can continue to update it over time.

**Author**

**Ong Boon Leong** works for Intel as Principal Software Engineer under Internet of Things Group (IOTG). He is passionate about developing local talents in open-source software development and over these years have been giving adjunct lectures in local and regional educational institution and universities. He has been in the embedded software industries for close to 20 years now (since 2001) and worked on numerous different technologies ranging from IXP network processor, Windows CE/MSAuto, I/O, Audio, Low Power IA processor series, Yocto Project BSPs for Intel’s Xeon/Core/Atom/ Quark product series and in recent years are leading the software development for Ethernet and Time-Sensitive Networking.

**Email: boon.leong.ong@intel.com  
Linked-In:** https://www.linkedin.com/in/boon-leong-ong-81255b1/

**Prepared by Ong Boon Leong**

Linux-based development machine preparation:

* The symbol ‘$’ marks the shell command prompt and symbol ‘#’ indicates a comment.
* The term ‘repo’ refers to repository (a software configuration management system that tracks changes to source code)
* This workshop assumes that there is no network firewall or proxy between your development machine and Internet. However, most universities or enterprises deploy network firewall or proxy, so you will need to configure the network settings accordingly. **Appendix A** provides guidance on configuration required on a Linux-based Ubuntu OS.
* <https://www.yoctoproject.org/docs/current/ref-manual/ref-manual.html#required-packages-for-the-build-host> provides a list of essential software packages for different type of Linux distro such as Ubuntu/Debian and Fedora. Please make sure that you have installed them on your development machine. For example, Ubuntu development machine will need below softwares:

|  |
| --- |
| $ sudo apt-get install gawk wget git-core diffstat unzip texinfo \  gcc-multilib build-essential chrpath socat cpio python3 \  python3-pip python3-pexpect xz-utils debianutils iputils-ping \  python3-git python3-jinja2 libegl1-mesa libsdl1.2-dev \  pylint3 xterm |

* To use auto-completion feature when typing a bash or git command, please install bash-completion package.

|  |
| --- |
| $ sudo apt-get install bash-completion |

* It is important to configure your git before you start. An example is shown in **Appendix B**.
* An example of vim configuration is in **Appendix D**.
* We will use <https://git.kernel.org/pub/scm/network/ethtool/ethtool.git> for this lab-work.
* Finally, use ‘git help’ to check its version is relative new (v2.25.1).

|  |
| --- |
| $ git help |

**Basic Git - 101**

**To git clone an open-source (remotely hosted) software over Internet to your work directory**

|  |
| --- |
| $ mkdir -p ~/repos; cd ~/repos  $ git clone git://git.kernel.org/pub/scm/network/ethtool/ethtool.git  $ cd ethtool |

**Note**: “master” branch is automatically checked out after ‘git clone’ operation has completed.   
**Note**: If the git clone operation fails, it is likely that you are behind firewall and you need to setup your network proxy setting accordingly. Please refer to Appendix A.

**To show current working branch and all available remote branch**

|  |
| --- |
| # To show current working git branch  $ git branch  # To show all (working, local and remote git branches)  $ git branch -a |

**Note**: The ‘\*’ besides master branch indicates that it is the currently check-out branch  
**Note**: In Appendix A, PS1 environment variable (inside ~/.bashrc) is set to contains $(\_\_git\_ps1 " (%s)") to have shell prompt automatically shows git branch.



**To show the commit history/log in full or in one-liner format**

|  |
| --- |
| # To show full log (both commit title and message)  # Note: to exit git log, simply type “q” and “Enter” key.  $ git log  # To show only commit title (one-liner)  $ git log --oneline  # To show commit title for a specific file  $ git log --oneline -- ethtool.c |

**Note**: Navigate to older commit history using down/up key. Also, observe that each commit has unique commit ID (SHA-1 hash value).   
**Note**: We have set git alias (inside ~/.gitconfig) for ‘git log --oneline’ as ‘git ol’, so you may also use ‘git ol’ as short command.   
**Note**: The file filter search separator ‘ -- ’ is very useful to limit the scope of the git command (include git show, git diff and other commands). So, ‘git log --oneline -- ethtool.c’ is used to list one-liner git log related to the file ethtool.c.

**To show a git commit (message and change)**

|  |
| --- |
| $ git show 124a3c06d1c3 |

**Note**: Observe that commit title is one-liner and usually less than 74-character. A good commit title is crisp in explaining the intention of the commit. A good commit body explains the what, why and how about the changes. Lastly, a good commit also has clear ‘signed-off-by’, a.k.a. S.O.B.

Note: In “Sign your work” section in <https://www.kernel.org/doc/html/v5.10/process/submitting-patches.html>, there are good materials that explain how numerous tags (Signed-off-by, Tested-by, Reviewed-by) are used Linux kernel community and the practices are often adopted in other open source projects other than Linux. However, it is always good to scroll through commit history/log or read through mailing list of a software project to understand the community practices.

**To make change on source code, simply use any text editor, such as vscode or vim.**

|  |
| --- |
| # In this example, we use ‘vim’ editor to add extra text  # “This is my edit” to the end of the README file.  $ vim README  # Create a new file with one-liner ‘Hello World’ text.  $ echo “Hello World” > new\_file.txt |

**To show status of working directory (tracked and untracked file by git)**

|  |
| --- |
| $ git status |

Note: Observe that there are both “changes not staged for commit” for tracked file (in this case README) and “untracked files” (for new\_file.txt)

**To show changes made on tracked file**

|  |
| --- |
| $ git diff -- README |

Note: ‘+’ and ‘-’ marks the lines that have addition/modification/ delete.

**To restore the change of a tracked file to its previous state**

|  |
| --- |
| $ git restore README |

**To add changes to git indexed staging area (‘stage the change’ before a commit)**

|  |
| --- |
| $ git add README new\_file.txt |

**To show the changes added to the staging area**

|  |
| --- |
| $ git diff --cached  # To show per-file changes in staging  $ git diff --cached -- README |

**If you are not satisfied with the change, you can continue to edit the file and use ‘git add’ to add the changes to the staging area.**

**Finally, to commit (make it a change history) the staged change into repository**

|  |
| --- |
| $ git commit -s |

Note: A good “git commit” write-up: (1) one-liner, maximum 74-character width in the format ‘subject: crisp title description’, (2) commit body that has details explaining the why, what and how of the changes (3) the SOB (automatically added because of ‘-s’).

**Now that you have committed a new change, you can view the new commit as follow:**

|  |
| --- |
| $ git log --oneline  $ git show |

**Let’s assume that you made a mistake in README and you want to change to commit,**

|  |
| --- |
| # Edit your change by using editor  $ vim README  # Then, add the new change into staging and commit the amendment  $ git add README  $ git commit --amend |

Note: The above “vim README” step is not needed if you just want to edit the commit message.   
Note: The above steps are only applicable for the most recent commit (a.k.a. HEAD commit)

**Observe that the updated commit also now has a new commit ID.**

|  |
| --- |
| $ git log --oneline |

**Basic Git - 102**

**To show the list of tracked git repositories**

|  |
| --- |
| $ git remote -v |

**Note**: the ‘origin’ is the default name of a tracked repo and has two URLs (often the same) for git ‘fetch’ and ‘push’ operation (to be explained later).

**Let’s assume that you are interested to follow a fork of ethtool repo contributed by another developer (**[**https://github.com/pamolloy/ethtool**](https://github.com/pamolloy/ethtool)**), you will add it to your tracked repository list.**

|  |
| --- |
| $ git remote add pamo <https://github.com/pamolloy/ethtool.git> |

**Note**: The repo name ‘pamo’ is just an example and can be any sensibly chosen name.   
**Note**: In real-world, all ethtool (a tool used for controlling Ethernet controller) is developed on the upstream repository at <https://git.kernel.org/pub/scm/network/ethtool/ethtool.git>

**To show the newly added tracked repo:**

|  |
| --- |
| $ git remote -v |

Note: At this stage, the git objects (commit, labels, etc) for ‘pamo’ repo has not been fetched (downloaded) into your local repo yet.

**To synchronize a tracked repo with its upstream (remotely hosted) repo:**

|  |
| --- |
| $ git fetch pamo |

**Note**: Observe that during git fetch operation, the difference between local ‘pamo’ repo and its upstream repo is calculated, compressed, downloaded, and finally unpacked. This makes your local repo to have same states as its upstream repo.   
**Note**: If you perform another ‘git fetch pamo’ command now, you will observe that there is no more new update (in fact the ‘pamo’ repo is actually inactive years ago as observable from its commit history in github).

**To show all git branches tracked by both ‘origin’ and ‘pamo’ repositories:**

|  |
| --- |
| $ git branch -a |

Note: Observe that ‘pamo’ repo only has one ‘master’ branch and the ‘origin’ repo has 3 branches (ethtool-3.4.y, next and master).

**To check-out the ‘master’ branch from ‘pamo’ repository:**

|  |
| --- |
| $ git checkout -b pamo-master --track pamo/master  $ git branch -vv |

**To show the upstream repo of local branch:**

|  |
| --- |
| $ git branch -vv |

**Note**: It is important to know that ‘the upstream repo’ here is as update as the last git fetch operation. **Note**: If you have added commits on-top of local branch and you wish to update the local branch to be the same state as the upstream branch, you will need to use ‘git pull --rebase’ so that your local commits are rebased on-top of the latest commit history from the upstream branch.

**To switch to another git branch, say ‘master’ branch:**

|  |
| --- |
| $ git checkout master  # Now, let look for the new git commit that we added in 101 section  $ git log --oneline |

**Note**: The commit ‘d605b92d9a94’ will be different in your working directory.

**Now, switch back to ‘pamo-master’ branch.**

|  |
| --- |
| $ git checkout pamo-master |

**To cherry-pick a commit from ‘master’ to ‘pamo-master’ branch:**

|  |
| --- |
| $ git cherry-pick -s -x d605b92d9a94  $ git show |

**Note**: The option ‘-s’ means automatically adds a signed-off-by tag and ‘-x’ means adding cherry-pick origin commit ID before the signed-off-by. The use ‘-x’ is very important if you are cherry-picking from public open-source repo so that other developers know the origin of the changes. It is often used as a record to for your fellow developer to cherry-pick the same commit into their own repo. ‘-x’ is of no use (in fact does not have any meaning) if you are not using ‘upstream’ repo. This is because the commit ID (in cherry picked …tag) is only unique to your local branch and not traceable to anyone else.

**Now, let’s add two more changes/commits to ‘pamo-master’.**

|  |
| --- |
| # Make the 2nd change according to below ‘git show’.  $ vi README  $ git add README  $ git commit -s -m “README: my 2nd edit”  $ git show    # Make the 3rd change according to below ‘git show’  $ vi README  $ git add README  $ git commit -s -m “README: my 3rd edit”  $ git show |

**Note**: The option ‘-m’ allows us to input the one-liner git title and this is often used for git commit that is obvious and does not need lengthy git commit body (description). In another words, if you have a multi-line commit message, please use ‘git commit -s’ instead.

**To show the git commit history:**

|  |
| --- |
| $ git log --oneline |

**Next, we want to create a new branch and to try out ‘git rebase’ command later.**

|  |
| --- |
| $ git branch pamo-master-test  $ git checkout pamo-master-test  $ git log --oneline    $ git branch -vv |

**Note**: The ‘pamo-master-test’ branch does not have upstream branch. It is simply a new branch that at this point has the same commit history as ‘pamo-master’ branch.

**Note**: In git, ‘pamo-master’ and ‘pamo-master-test’ branches use their respective HEAD to point to commit. Since a commit has parent commit and the relationship goes on up the lineage of git history, any change to a particular commit along the lineage will mean the entire commit history now has different commit ID. As a matter of fact, any change to a git commit (code change, authorship, date/time, parent of the commit and other attributes) will produce a new SHA-1 hash computation that is essentially the commit ID.

**Let’s say that we want to discard the 3rd edit on README from the history tracked ‘pamo-master-test’ branch.**

|  |
| --- |
| # The commit-ID before 3rd README edit is 7dae1df23115.  $ git reset --hard 7dae1df23115  $ git log --oneline |

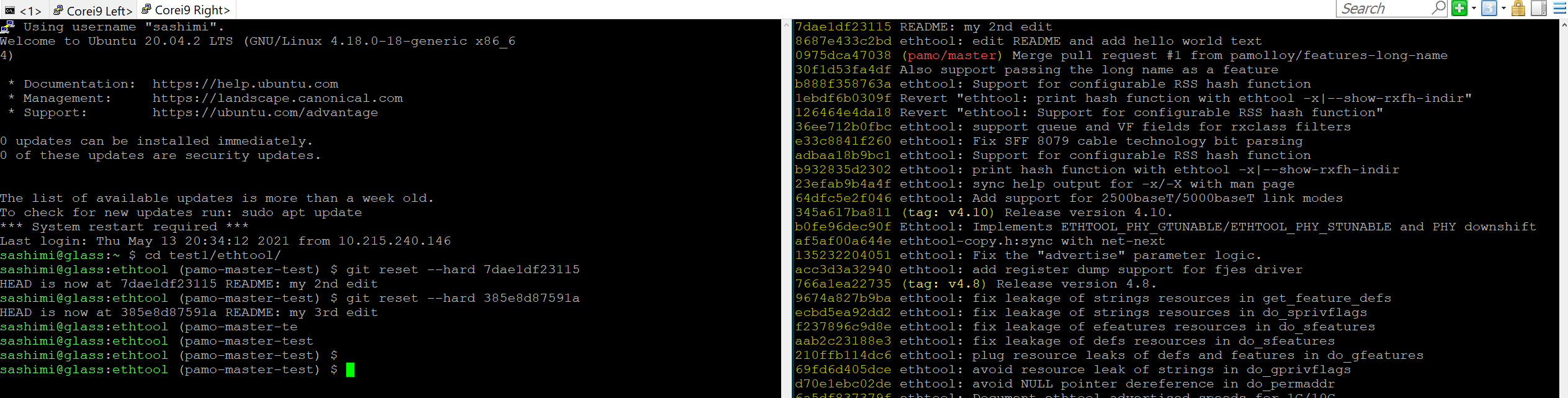
**Note**: The commit ‘7dae1df23115’ for the 3rd README commit will be different in your working git repo. So, you will need to use the correct commit ID in your case.

**Now, you regret about removing the 3rd README edit commit and you want to reverse the removal.**

|  |
| --- |
| # The commit-ID before 3rd README edit is 385e8d87591a.  $ git reset --hard 385e8d87591a  $ git log --oneline |

**Note**: By now, you should see the nature of git branch ‘pamo-master-test’ here. The HEAD of ‘pamo-master-test’ branch is really a pointer and the ‘git reset --hard’ command can be used to reset the HEAD to point at a specific commit ID in a git repo (this includes commit that is also tracked by another git branch).

**Tip**: If you are worried of messing up your git history, you could always just make a copy of “git log --oneline” before you perform any “git reset --hard <other commit>” or simply just open two terminals side-by-side, the right terminal is ‘git log --oneline’ which allows you to scroll up and down to obtain the desired commit ID. In addition, until you exit the “git log --oneline” by entering ‘q’ + ‘Enter’, the git history does not change even though you have done multiple “git reset --hard” on left terminal on the same git repo.



**Advanced Git - 201**

We have learned about ‘git commit --amend’ in Basic Git 101 and this command is used to edit the HEAD commit (git commit write-up or code changes). In this section, we will learn about ‘git rebase -i <commit-ID>’ that is used to reapply commits (a change to make commit change) on top of a commit base.

**Let’s show the commit history on ‘pamo-master-test’ branch:**

|  |
| --- |
| $ git log --oneline |

**To change commits including and on-top of a commit base, say “8687e433c2bd ethtool: edit README and add hello world text”:**

|  |
| --- |
| $ git rebase -i 8687e433c2bd^ |

**Note**: The character ‘^’ is used here so that the commit ‘8687e433c2bd’ is included in the commits apply list.

**Note**: The order of the commits apply order is such that the most recent commit at the bottom of the list. It is completely in reverse order in comparison to the list by ‘git log --oneline’.

**Note**: The commit apply list shows all the commits from the base (in this case 8687e433c2bd till the HEAD). Each of the commit is default to “pick” command which, as its name suggests, picks the commit for apply. There are other commands such as reword, edit, squash, etc. Instead of typing the whole command name, you can just use the character, e.g. for ‘rewording git commit message’, use ‘r’ or ‘reword’.

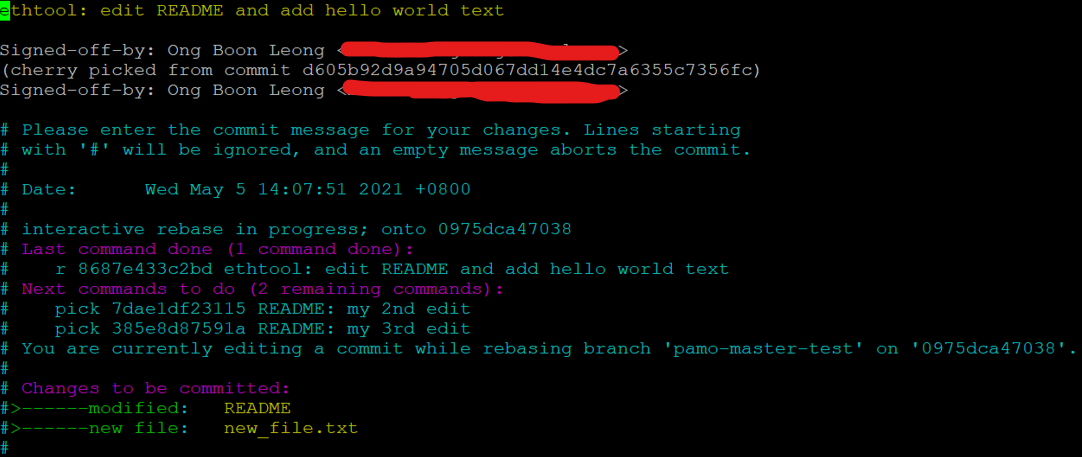
1. **git rebase: “reword” command**

**To reword commit message of ‘8687e433c2bd ethtool: edit README and add hello world text’, change ‘pick’ to ‘r’. Then, save the commit apply list (for vim, press “ESC” then “:wq”).**

|  |
| --- |
|  |

**Then, the terminal will use ‘vim’ to open the git commit message for “ethtool: edit README and add hello world text” as shown below. So, make the change for the commit message as shown below.**

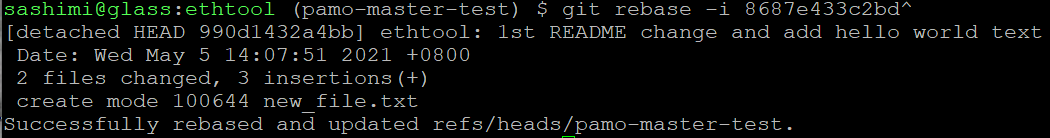
**Original commit message:**



**New commit message:**



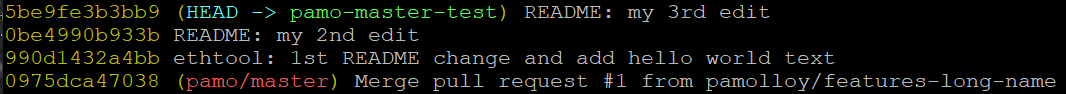
**After the commit message has been saved, the rest of the git commits will be applied according to the commits apply list.**



**Show the git history log, then take note that the commit ID for all the 3 commits have changed.**

|  |
| --- |
| $ git log --oneline |

**Note**: What you have just performed is to use git rebase to change git commit message that is not HEAD. Please remember that if you intend to change the git commit message for HEAD commit, you can simply just use ‘git commit --amend’.



**Now, let’s look at the new git commit that we have just changed before:**

|  |
| --- |
| $ git show 990d1432a4bb |

**Note**: Observe that this commit is not done in accordance to software configuration good practices whereby the change of a commit should have natural scope (in this case it is a mixed bag of README and new\_file.txt). So, in the next section, we will use ‘edit’ command to fix it.

1. **git rebase – “edit” command**

**To edit a (non-HEAD) commit, we start with git rebase command:**

|  |
| --- |
| $ git rebase -i 990d1432a4bb^  # replace ‘pick’ to ‘e’ or ‘edit’ and save the commit apply list    # When the git rebase started, you will be presented at the  # “ethtool: 1st README change and add hello world text” commit.  # To show the commit now,  $ git show |

**Note**: The terminal prompt now shows “REBASE-i 1/3” which means it is operating at the 1st of the 3 commits of the apply list.

**Let’s just say we want to make two changes (README and new\_file.txt). Recognize that the commit is also at new HEAD, we can use “git reset HEAD~" to the previous state before the change was committed.**

|  |
| --- |
| $ git reset HEAD~ # At this point, you are at the point before the commit was performed.  $ git status |

**Tip:** if you want to save some time from re-writing a commit message, simply open two terminals with one of the terminal has ‘git show’ command used to show the git commit message.

**Next, we add and commit the change for README.**

|  |
| --- |
| $ git add README  $ git commit -s  # If we just paste the previous commit message as part of the  # new commit message, we need to be aware that the cherry picked # tag (from -x) earlier is no longer valid. Besides, there are multiple # SOBs in the commit message that is not relevant.    # So, we will change commit message as follow and save the commit:    # Now, we have gotten the first part of what we wanted to achieve.  # i.e. to split the original commit (990d1432a4bb) into one commit for  # the README change  $ git show |

**Next, we add another commit for ‘new\_file.txt’.**

|  |
| --- |
| $ git status    $ git add new\_file.txt  $ git commit -s  # Add the commit message as below and save it.    # Next, we show the commit history again. There are our two new commits  $ git log --oneline    # At this point, if we want to add another new file or make  # another commit we can do the usual ‘edit + git add + git commit’  # sequence that we have been using a lot by now  $ echo ‘Good Morning to Alexa’ > alexa.txt  $ git add alexa.txt $ git commit -s -m “alexa: Good morning message”    # Take note that we are still at “REBASE-i 1/3” and to continue # applying the rest of commit list  $ git rebase –continue |

1. **git rebase – “squash” command and commits reordering with “pick”**

**Let’s look at the commit list that we want to operate on again:**

|  |
| --- |
| $ git log --oneline |

**Let’s say we want to move the order of ‘c4cdb86c0db1 ethtool: 1st README change’ to be after ‘b18eb8037fe4 alexa: Good morning message’. We can use the ‘git rebase’ apply commit list again.**

|  |
| --- |
| $ git rebase -i c4cdb86c0db1^  # The original commit apply list    # The new commit apply list # Note: For ‘vim’ editor, under command mode (press ESC):  # to cut a line, type ‘cc’ then move your cursor to  # ‘alexa: Good morning message’, then type ‘p’ to paste a cut line.  # To delete the previous cut line, type ‘dd’. # To save the apply list, press “:wq”    # Show the new git history again $ git log --oneline |

**Let’s continue to squash (combine) commits related to README:**

|  |
| --- |
| $ git rebase -i f23e892efbbe^  # The original commit apply list    # The new commit apply list, to combine both 2nd and 3rd commit  # README change    # When we are squashing git commits, we will be presented with commit # message editing automatically that combines older commit messages. # Note there are 3 SOBs and you really just need one SOB for the new # commit.    # Let’s just edit commit message as follow:    # Observe that commit history now only has one commit for README # instead 3 commits earlier.    # Show the commit for README now  $ git show |

**As summary, we have use ‘squash’ and ‘pick’ command in git rebase to combine commits and rearrange the order of commits. This technique is important because as you are developing/debugging software, often your working commits are not in the right order to be sent upstream. So, it is important to rearrange and clean-up your commits before they are shared with upstream communities so that they appear in logical sense.**

**Advanced Git - 202**

We have seen in Advanced Git 201 earlier that the ‘git rebase’ allows us to reapply a series of git commit list (with changes, commit reordering, squashed commits) on-top of a base. In this section, we will learn about aborting a git rebase command or resolving a conflict that happen when applying commits.

**Let’s look at the commit list that we want to operate on and now we use ‘ec46a7ba1865 new\_file: initial version of the new\_file that says hello world’ as the commit base:**

|  |
| --- |
| $ git log –oneline    $ git rebase -i ec46a7ba1865^  # Let’s use “edit” to change the ‘ec46a7ba1865’ commit    # Then, change the new\_file.txt as follow    # View the difference (change) that you just made  $ git diff    # Now, add and commit the change.  $ git add new\_file.txt  $ git commit --amend  # Also, edit the git commit message as follow: |

|  |
| --- |
| # Show the git commit at REBASE-i 1/3  $ git show    # At this point, git commit history has been changed.  $ git log --oneline |

**Now that we are done with REBASE-i 1/3 commit, normally we will use “git rebase --continue” to apply the rest of the commits. However, let’s say we just want to abort the whole git rebase operation and revert to previous git commit history.**

|  |
| --- |
| # To abort a git rebase operation  $ git rebase --abort    # Observe that commit history and their IDs have not changed  $ git log --oneline |

**Next, we will go through an example of git conflict and resolve it. A conflict happens when the git tool does not know how to merge the commit change (added/deleted).**

|  |
| --- |
| $ git rebase -i ec46a7ba1865^  # We set the apply list to “edit cb3fe007855a alexa: Good morning message” and save the apply list    # Take note that we are at REBASE-i 2/3 commit just before the commit  # “README: Add additional”. Now, let’s add some extra text to README # as shown below  $ vi README   # Next, git add and commit the change above  $ git add README $ git commit -s -m “README: extra changes that causes conflict next” $ git log --oneline    # Now, let’s continue with git rebase and we got a merge conflict!! $ git rebase --continue |

**To show merge conflict:**

|  |
| --- |
| # To list file that has merge conflict $ git diff --stat    # To show the merge conflict  $ git diff |

**Note**: The delimiters ‘<<<<<<<’ and ‘=======’ shows the content in HEAD and the delimiters ‘=======’ and ‘>>>>>>>’ shows the content of the commit that is supposed to be applied (in this case REBASE-i 3/3 commit).

**Now, let’s resolve the conflict (pick the commit in HEAD, or pick the commit in the commit 3/3 or combine them) by editing the file that has merge conflict.**

|  |
| --- |
| # Edit the README as shown below.  $ vi README    # It is always good to check what we have resolve so far  $ git diff    # Now, since we no longer have any conflict as marked by the three  # delimiters (‘<<<<<<<’, ‘=======’ and ‘>>>>>>>’) earlier. If you are # not sure what is the next step, you can use git status to cue you.  $ git status    # To mark the conflict in README file has been resolved  $ git add README  # To check if there is any more conflict is left out, use either one # of below command  $ git diff $ git diff --stat    # Finally, continue with the git rebase operation  $ git rebase –continue  # Then, you will be presented to edit the 3/3 commit message. In this # case, we just simply save the commit message as it is.    # Let’s list the commit history again  $ git log --oneline |

**We have just learned how to abort a git rebase operation and how to resolve conflict. If you feel intimidated by resolving a complex conflict, the rule of thumbs is to do the change in new git branch and try to resolve the conflicts incremental fashion. The purpose of resolving conflict is to make the the state of your working git branch to a desired state. Finally, check the change between two branches are the same or as desired:**

|  |
| --- |
| # List git branch  $ git branch    # To see the difference between git branches $ git diff pamo-master pamo-master-test    # The git history for “pamo-master” and “pamo-master-test”  $ git checkout pamo-master  $ git log --oneline    $ git checkout pamo-master-test  $ git log --oneline |

**Note**: Comparing the difference between git branches is good way to cross check the changes that you have made on the “pamo-master-test” branch. If you are not happy with it, you can continue to use the “git rebase” operation to change the source code or commit message until you are satisfied with it.

**Advanced Git - 203**

Now that you have acquired the skill to make commit change, it is time to learn about how share your commits with other developers. If you are the maintainer or owner to a public repository hosted on Internet such as under www.github.com, you can push your local git branch to the remote repository to publish your changes to the project follower. If you are a fellow developer of a project, you send a pull request or send patch-series via email.  
Note: before you proceed, you need to have a GitHub account so that you can practice the steps. Please go to <https://github.com/> and sign-up your own account.

1. **As the owner of a remote repo, you can push local branch to it.**

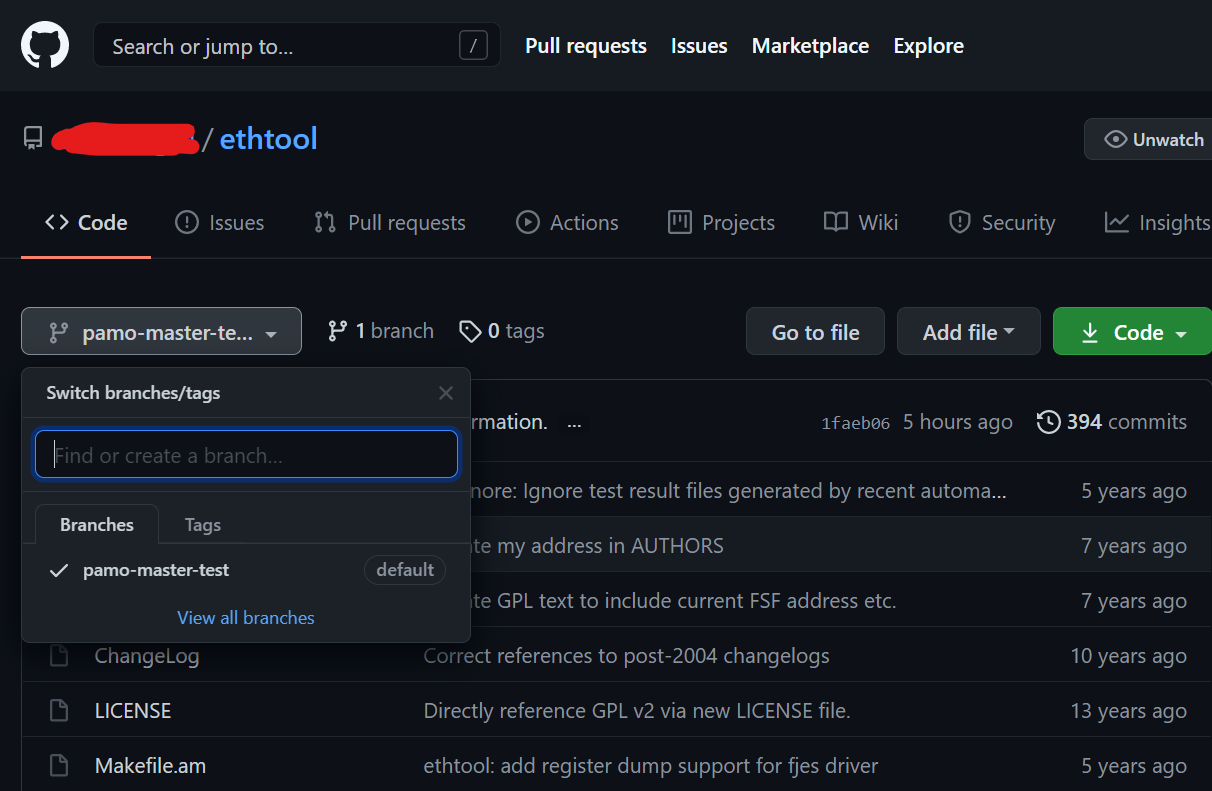
**To check the remote repo name:**

|  |
| --- |
| $ git remote -v |

**To add upstream repo for working branch and push it to upstream branch:**

|  |
| --- |
| # List the git branch and their upstream branch information  $ git branch -vv    # Since we are at “pamo-master-test” branch and it does not have # an upstream branch, it is not allowed to perform git push here. $ git push    # In your GitHub account, create a repo called ‘ethtool’. Then, # add a new tracked repo to it $ git remote add mygithub https://github.com/<user>/ethtool.git  $ git remote -v    # Then, push “pamo-master-test” branch to “mygithub”  $ git push --set-upstream mygithub pamo-master-test |

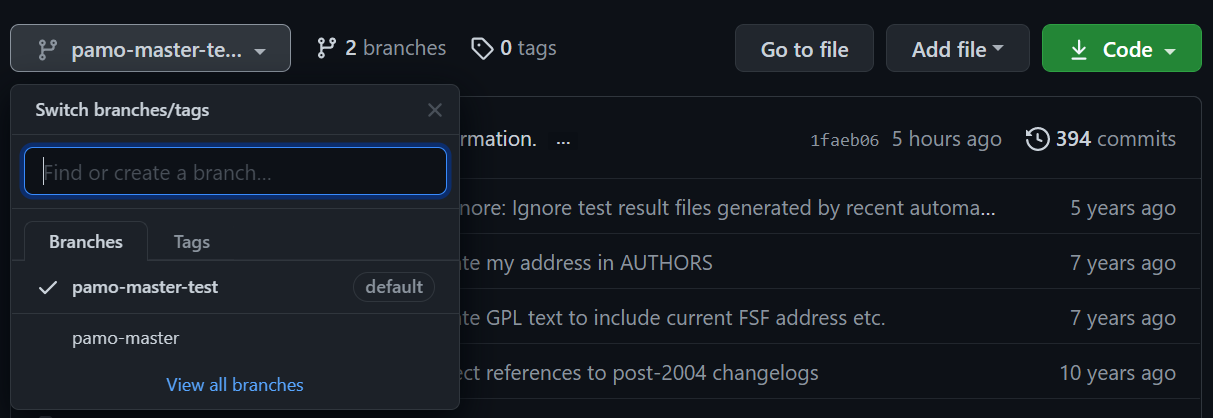
**You can cross check what you have just pushed in your GitHub account.**



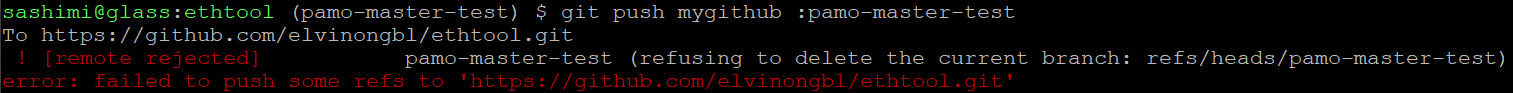
**You can also push different upstream branch name instead of local branch name.**

|  |
| --- |
| # To push ‘pamo-master-test’ as ‘pamo-master’ to upstream repo $ git push mygithub pamo-master-test:pamo-master |

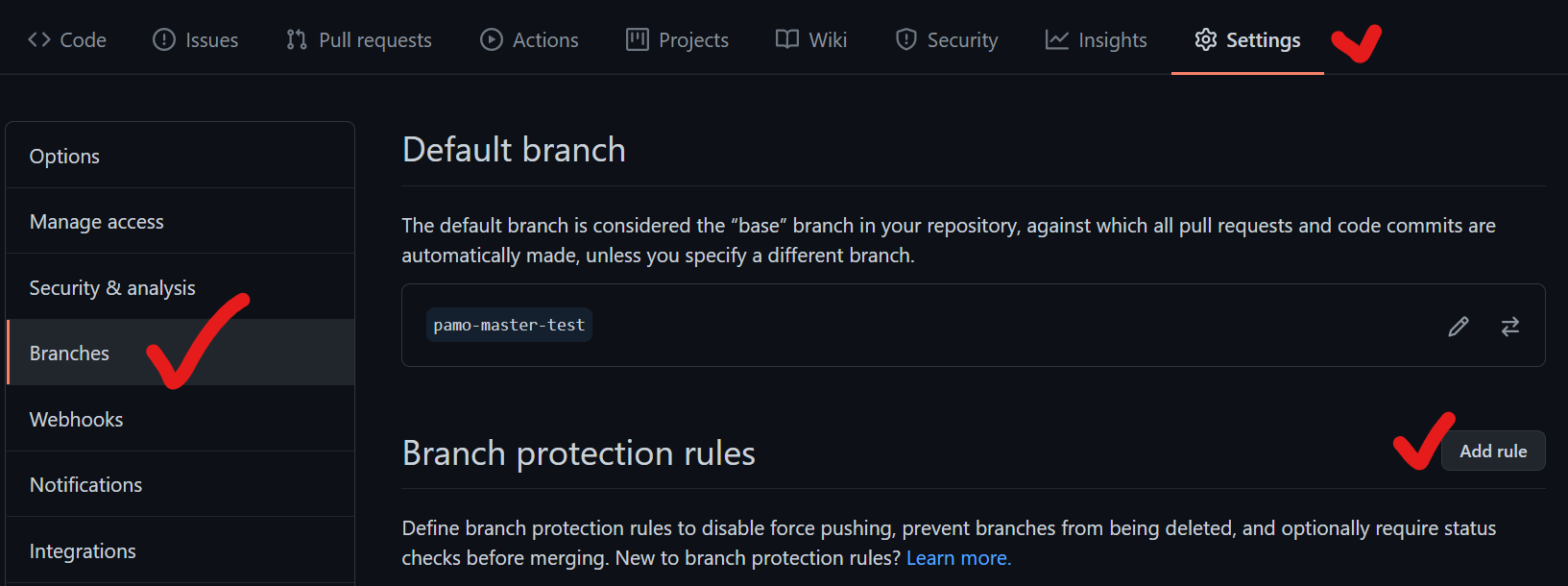
**Now, your GitHub account has the new “pamo-master” branch**



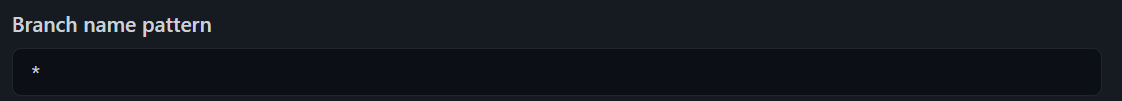
**As the owner to the upstream repo, you can delete a branch, say “pamo-master-test”. However, GitHub default setting prevents you from doing so. At this point of time, you will get below warning when you try to perform ‘git push myhithub :pamo-master-test’ to delete the ‘pamo-master-test’ branch.**

****

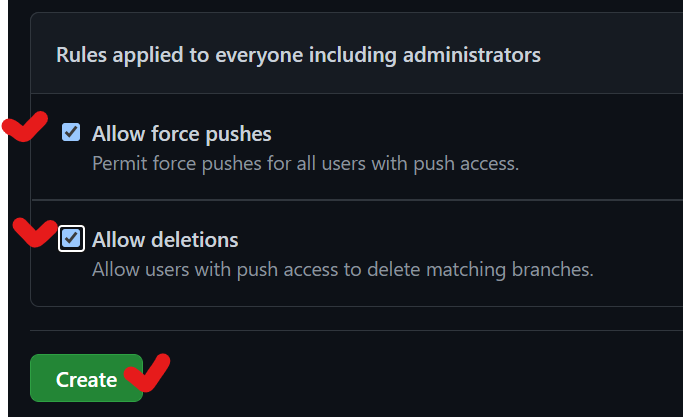
**We will need to change the “Branch protection rules” under the repo “Settings” page.**

****

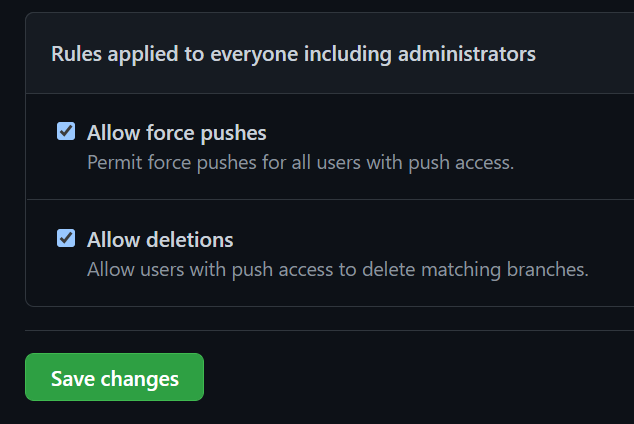
**Use \* in ‘Branch name pattern’ to apply the rules to all branches.**

****

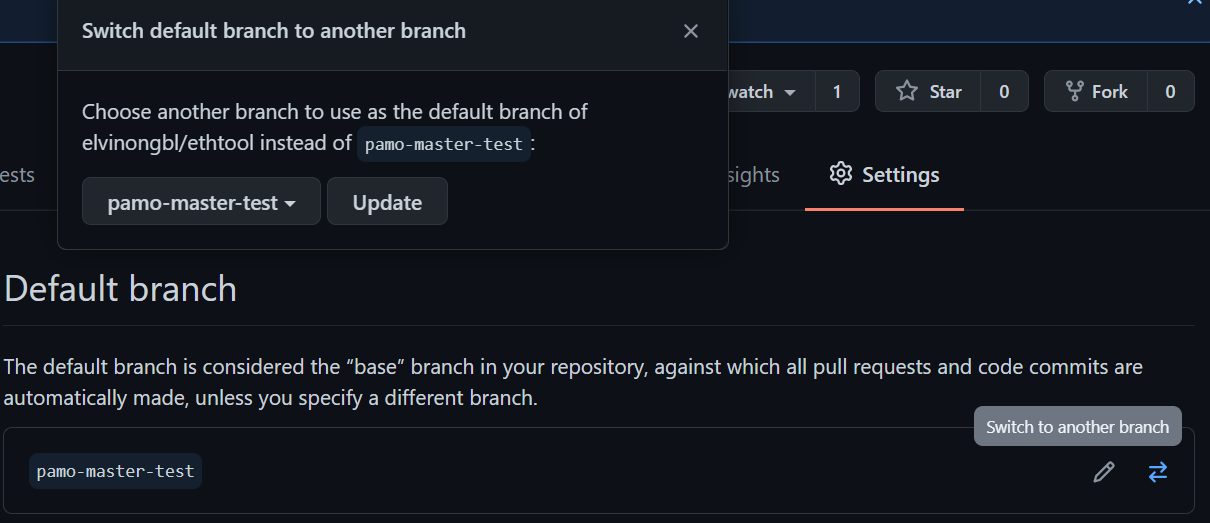
**Mark the below options to allow owner to perform force pushes or delete branch**

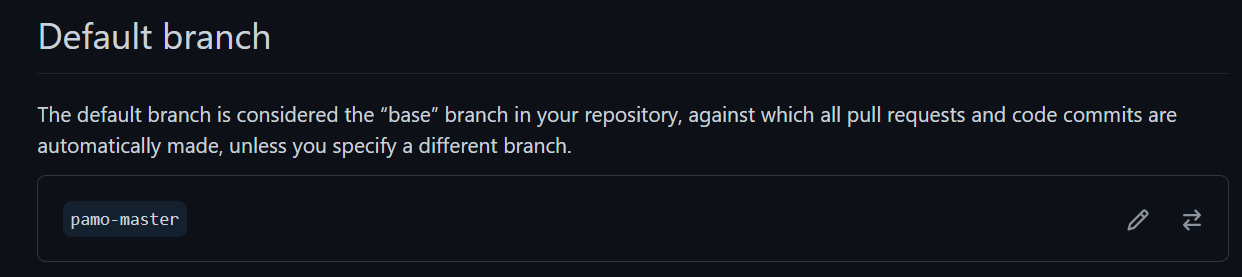
****

**Then, press the ‘Save changes’ button**

****

**Next, you also need to change the default branch that was set to ‘pamo-master-test’ to ‘pamo-master’ because you are not allowed to delete a default branch.**

****

**Default branch is now ‘pamo-master’. **

**Let’s delete the ‘pamo-maste-test’ branch again.**

|  |
| --- |
| # To delete ‘pamo-master-test’, simply push empty branch to  # ‘pamo-master-test’ branch $ git push --force mygithub :pamo-master-test    # Update mygithub to sync with upstream repo  $ git fetch mygithub $ git branch -a |

**In local git repo (after git getch), local ‘pamo-master-test’ branch that was tracking an already deleted upstream branch ‘pamo-master-test’ with be marked as ‘gone’.**

**So, we can change the upstream branch for ‘pamo-master-test’ as follow.**

|  |
| --- |
| # List the git branch upstream branch  $ git branch -vv    # To change the local ‘pamo-master-test’ branch’ to track upstream # ‘mygithub/pamo-master’ branch  $ git branch -u mygithub/pamo-master pamo-master-test  $ git branch -vv |

**Let’s say we make change on HEAD commit and then want to git push it to upstream repo:**

|  |
| --- |
| # Change git commit meesage  $ git commit --amend    # push the HEAD commit to upstream repo $ git push mygithub HEAD:pamo-master    # Obviously, the HEAD commit of local branch is in conflict  # with HEAD commit in upstream branch, so to overwrite the  # HEAD commit at upstream branch we will need to force push it over $ git push --force mygithub HEAD:pamo-master |

**In this section, we have learned about deleting git branch or force push (over-writing) commit history by enabling this special permission in GitHub repo setting. In large project whereby a repo is followed by many developers, tampering with git branch and commit history is often not desirable. So, please understand the implication and use these techniques with care.**

1. **As “contributor” to an upstream repo, you can send pull-request to share your commit changes.**

**Let’s create to new commits as follow:**

|  |
| --- |
| # Make a new commit by changing alexa.txt as follow: $ vim alexa.txt  $ git add alexa.txt $ git commit -s -m “alexa: add additional description”  # Make a new commit by changing new\_file.txt as follow:  $ vim new\_file.txt    $ git add new\_file.txt  $ git commit -s -m “new\_file: Change to Good Night World”  # List git history  $ git log --oneline    # Show git status  $ git status    # Note that git status informs us that local branch is ahead of upstream # branch by 2 commits. This matches what was shown in git log above too |

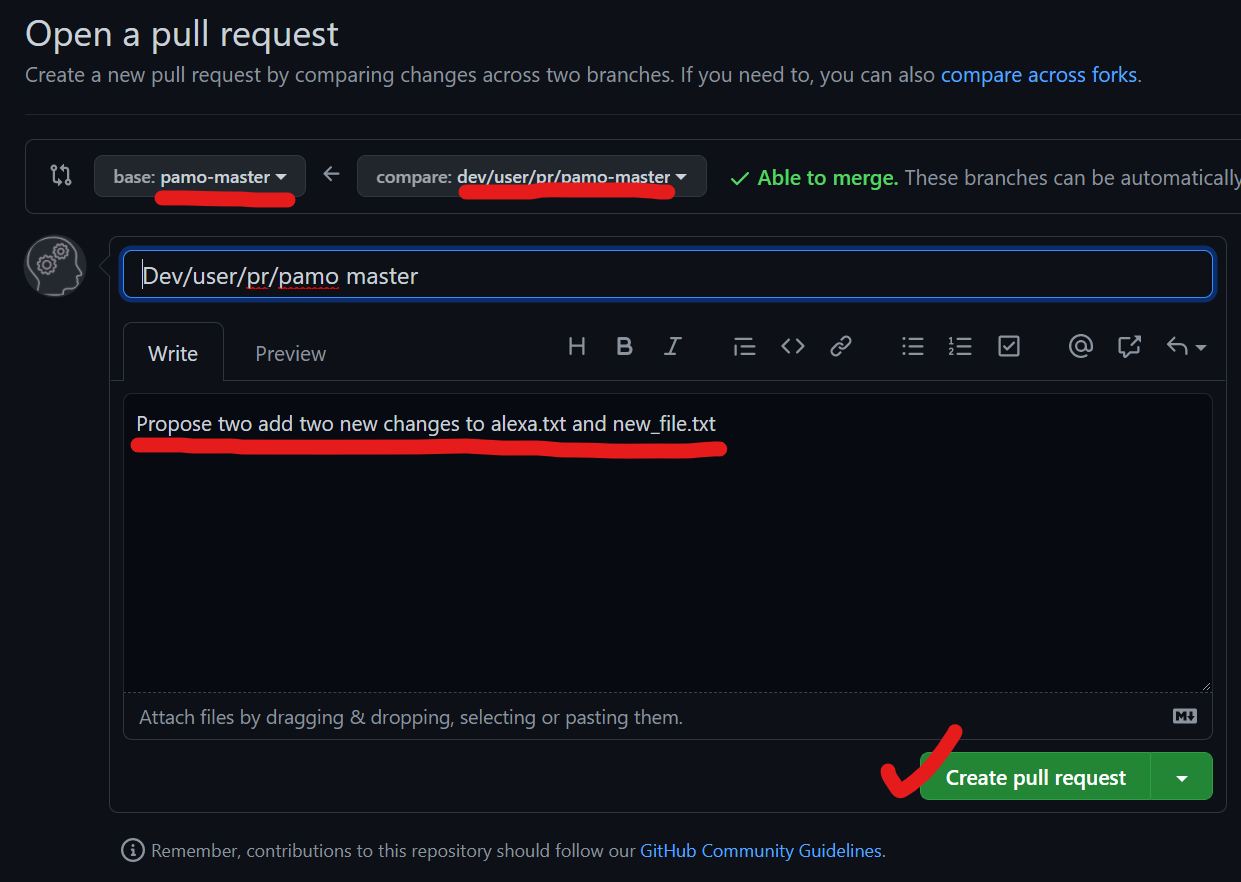
**Push a developer’s topic branch to GitHub**

|  |
| --- |
| # Push a developer’s topic branch to GitHub $ git push mygithub HEAD:dev/user/pr/pamo-master |

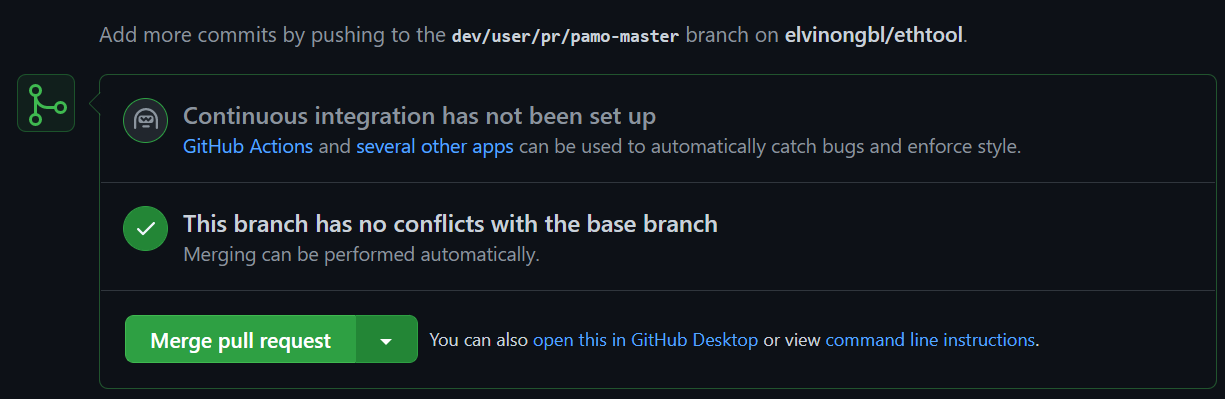
**Then, got to GitHub to create a pull-request for the developer’s topical branch at https://github.com/<user>/ethtool/pulls**

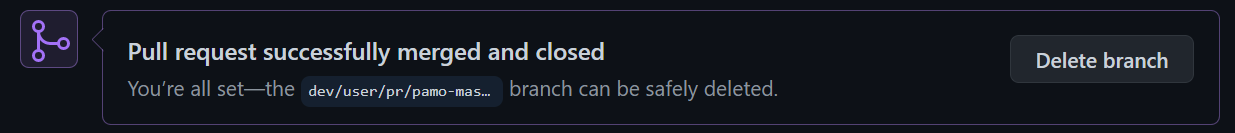
****

**Write-up the pull-request and create pull request.**

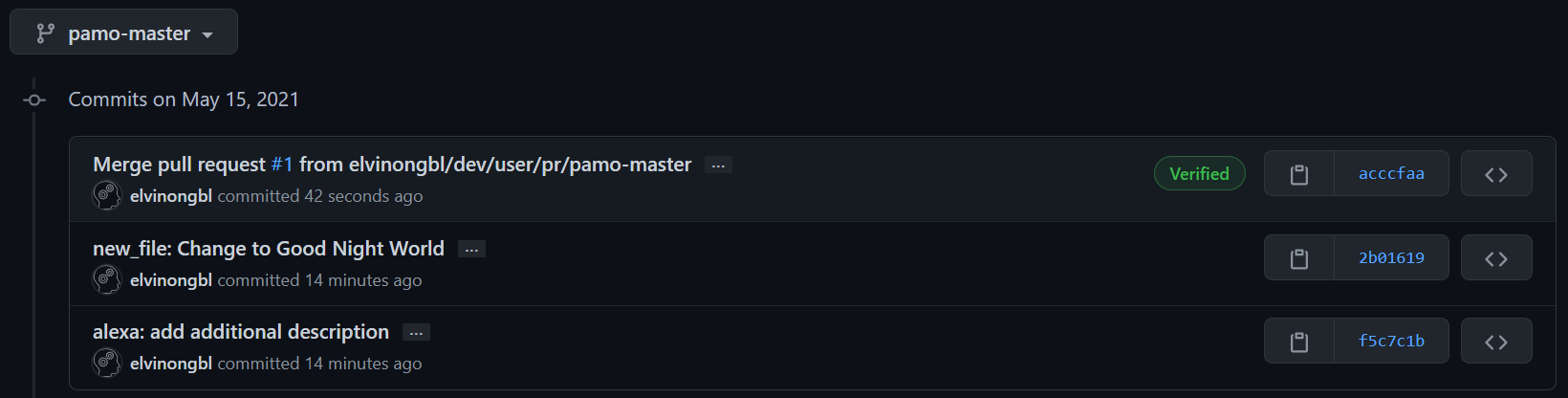
****

**Here, since you are the owner of the GitHub repo, you can comment on the changes to request the developer to clarify/fix the pull request or if you are satisfied with the PR you can just click “Merge pull request” then “Confirm nerge”.**

****

****

**With merge approved, now the git commit history shows the two commits that have been added.**

****

**Now, go back to your local git repo and check the git status again.**

|  |
| --- |
| # Show git status  $ git status    # Now, you see that your local repo is yet to be in sync with  # upstream repo. So, we fetch the latest git states $ git fetch mygithub    # Show git status  $ git status    # Now, git pull from updated upstream branch  $ git pull mygithub pamo-master    # List git history  $ git log --oneline    # Instead of doing all the above, to keep your local branch in-syn # you could just perform ‘git pull --rebase’ which include git fetch. $ git pull --rebase |

1. **As “contributor” to an upstream repo, you can send patch-series to share your commit changes.**

In Appendix C, there is an example of bash script that I used for sending patch-series to Linux mailing list. You are encouraged to read through the simple bash script to get an idea what it is doing and after that change it to your own need.

Sending patch-series to mailing list involve below steps:

1. Generate patch series (using ‘git format-patch’)
2. If there are more than 1 patches, write-up cover-letter (in 0000-cover-letter.patch)
3. Check your cover-letter and patch series are in good shape
4. Send the patch series using ‘git send-email’. Note also we have setup [sendemail] configuration in ~/.gitconfig earlier.

**Identify the commit ID for the patch-series.**

|  |
| --- |
| # List git history  $ git log --oneline |

**Generate patch-series and send-email bash script**

|  |
| --- |
| # Let’s say we want to send patches from ec46a7ba1865 to 2b016196295e # Note the ‘^’ at the back of ec46a7ba1865 to include.  # Next, we use the convenient bash script to generate  # cover-letter, patches and send-email bash script  # ./bformat-patch.sh $START\_ID $END\_ID $SERIES\_VER $TOPIC  $ ./bformat-patch.sh ec46a7ba1865^ 2b016196295e 1 ethtool-text  $ ls # Note that the bformat-patch.sh script generates a folder “EXT-<topic>-v1” and sendemail bash script ‘EXT-ethtool-text-sendemail.sh’. |

**Write the cover-letter (EXT-ethtool-text-v1/0000-cover-letter.patch)**

|  |
| --- |
| # Add description to cover letter $ vim EXT-ethtool-text-v1/0000-cover-letter.patch  # The original cover letter format.  # Add your email title at \*\*\* SUBJECT HERE \*\*\* # Add your patch series description at \*\*\* BLURB HERE \*\*\*    # Example of cover letter write-up |

**Check your cover-letter and patches are in good-shape. Basically, this means there is no unwanted trailing white-space. The cover letter and commit messages in the patch should normally not exceed 74-characters and they are written with proper indentation. It is also good to see how existing git log on how a good commit looks like.**

|  |
| --- |
| # Inspect cover letter $ cat 0000-cover-letter.patch  # Inspect patches  $ cat 0001-new\_file-initial-version-of-the-new\_file-that-says-h.patch |

**Note**: Observe that a patch is really a git commit. So, sending patch-series in email is really just another way of communicating your commits to the recipient.

**Send the cover letter and patch-series to mailing-list.   
Note, the sendemail bash script will require some update from you before you use it to send the email and patch-series out.**

|  |
| --- |
| # It is always good to send the patch-series to yourself.  $ vim EXT-ethtool-text-sendemail.sh    # Change it to yourself now and then use the script to send to yourself.  # Note that the comment gives you hint on how to send all patch series or # just one patch.  # So, you can use the bformat-patch.sh to generate one patch and use the # the sendemail.sh script to send one patch. # Finally, to send the cover-letter and patch series  $./EXT-ethtool-text-sendemail.sh EXT-ethtool-text-v1    # Always remember to check:  # 1) To add CC list  # 2) Cover letter and the patch series are listed.  # 3) Since this is 1st version, the [PATCH net 0/6] does not show ‘v1’. # 4) Note that in this example ‘net’ is used in [PATCH net 0/6] and # this is for Linux networking subsystem label.  # Note: always remember you can change the bformat-patch.sh to fit your # need before using it. # Finally, to confirm that you want to send the cover-letter and patch # series, press ‘a’.  # If you found issue, and you want to abort, press ‘q’ to quit.  # Since we send the patches to ourself, we can look them up in our # email account.    # After you are happy with it, update the sendemail.sh emails and send # the patches out to the intended recipient. |

**Lastly, assuming you have downloaded the patches into your working repo and you want to apply the patches to try this out.**

|  |
| --- |
| # Let’s create a new test branch $ git branch test-branch $ git checkout test-branch  $ git log --oneline    # Reset the history to “0975dca47038 (pamo/master) Merge pull  # request #1 from pamolloy/features-long-name” so that we can apply # the 6 patches we just created earlier  $ git reset --hard 0975dca47038    # List the patch series  $ ls -1 EXT-ethtool-text-v1/  # Note that 0000-cover-letter.patch is actually not a patch.  # Finally to apply 0001 patch  $ git am EXT-ethtool-text-v1/0001-new\_file-initial-version-of-the-new\_file-that-says-h.patch    # list git history  $ git log --oneline    # You can apply more patches to try out ‘git am’ here. # For now, let’s just remove the ‘test-branch’. In order to do that # you need to switch to another branch  $ git branch    $ git checkout pamo-master-test  $ git branch -D test-branch  $ git branch |

**Finally, congratulations! You are now ready to be an expert user of git tool.**

**If you like this git workshop material, please drop me a thank you note!**

**Appendix A: Network Proxy Setting at ~/.bashrc**

|  |
| --- |
| # Add below global environment exports to local bash session  $ vi ~/.bashrc  # You need to check with your network administration for # network proxy information |
| export SOCKS\_SERVER=<proxy-server.com>:<SOCKS Port>  export HTTP\_PROXY=<proxy-server.com>:<HTTP Port>  export HTTPS\_PROXY=<proxy-server.com>:<HTTPS Port>  export FTP\_PROXY=<proxy-server.com>:<FTP Port>  export HTTP\_DIRECT=localhost,127.0.0.0/8,172.169.0.0/20,192.168.0.0/16  export SOCKS\_DIRECT=$HTTP\_DIRECT  export NO\_PROXY=$HTTP\_DIRECT  export ALL\_PROXY=$HTTP\_DIRECT  export socks\_server=$SOCKS\_SERVER  export http\_proxy=$HTTP\_PROXY  export https\_proxy=$HTTPS\_PROXY  export ftp\_proxy=$FTP\_PROXY  export http\_direct=$HTTP\_DIRECT  export socks\_direct=$SOCKS\_DIRECT  export no\_proxy=$NO\_PROXY  export all\_proxy=$ALL\_PROXY  PS1='${debian\_chroot:+($debian\_chroot)}\[\033[01;32m\]\u\[\033[00m\]@\[\033[01;32m\]\h\[\033[00m\]:\[\033[01;32m\]\W\[\033[00m\]$(\_\_git\_ps1 " (%s)") \$ ' |
| Note: PS1 is the global environment variable that defines shell prompt format.  Note: $(\_\_git\_ps1 " (%s)") is a useful shell prompt augmentation to print the git branch if your current working directory is a git repository. |

**Appendix B: Git configuration at ~/.gitconfig**

|  |
| --- |
| $ vim ~/.gitconfig |
| [user]  email = felix.ong@gmail.com  name = Felix Ong  [sendemail]  from = “Felix <felix.ong@gmail.com>"  smtpServer = smtp.gmail.com  smtpUser = felix.ong@gmail.com  smtpEncryption = tls  smtpServerPort = 587  smtpPass = <gmail token>  signedoffcc = false  suppresscc = all  chainreplyto = false  assume8bitEncoding = utf-8  confirm = always  [tig "color"]  # A strange looking cursor line  cursor = yellow default bold  # Title colors <foreground background attribute>  title-blur = black white  title-focus = red white bold  # Diff colors <foreground background attribute>  diff-header = yellow default  diff-index = green default  diff-chunk = red default  date = white black  author = green black  directory = white black  file = green black  # View-specific color  [tig "color.tree"]  date = green default bold  [color]  diff = auto  ui = auto  interactive = auto  grep = always  [color.branch]  current = green  local = white  remote = red  upstream = yellow  plain = white  [color "grep"]  match = red  [core]  abbrev = 12  editor = vim  # Uncomment for network proxy  #gitproxy = /usr/bin/git\_proxy\_command  whitespace = trailing-space,tab-in-indent  pager = sed 's/\t/>-------/g' | less -R  [alias]  co = checkout  dc = describe --contains  br = branch -v  ci = commit  st = status  sts = status -sb  ol = log --oneline  olf = log --pretty=format:\"%h [%ci %cn]: %s\"  cp = cherry-pick  showf = show --format=fuller  logf = log --format=fuller  merge-log = "!f() { git log --stat \"$1^..$1\"; }; f"  merge-ol = "!f() { git log --oneline \"$1^..$1\"; }; f"  merge-olg = "!f() { git log --oneline --graph \"$1^..$1\"; }; f"  bl = blame --abbrev=11 |
| Note: <gmail token> is obtained from Google Account -> Security -> Signing in to Google : App passwords.  Note: “editor = vim” set the default git commit editor to ‘vim’. I encourage you to learn it. If you are not familiar with ‘vim’, you could change it to ‘editor = nano’ or ‘editor = gedit’.  Note: [alias] contains a list of git command alias (short-cut) that you can use in bash shell prompt. For example, instead of typing ‘git log --oneline’ which often you may mistype as ‘online’ (because our brain trick us there all the times), we could just type ‘git ol’. |

|  |
| --- |
| # Add below global environment exports to local bash session  $ vim ~/common/bin/socatproxy |
| #!/bin/sh  # Please choose your site proxy server accordingly  exec socat stdio SOCKS:<proxy-server@somename.com>:$1:$2 |
| Note: [proxy-server@somename.com](mailto:proxy-server@somename.com) is the secure socket network server |

**Appendix C: bash script for patch-series**

|  |
| --- |
| $ vim bformat-patch.sh |
| #!/bin/bash  # ./bformat-patch.sh $START\_ID $END\_ID $SERIES\_VER $TOPIC  #  # For version 1 of patch-series  # ./bformat-patch.sh fec4d42724a1 ae4393dfd472 1 vlan-filter  # For version 2 of patch-series  # ./bformat-patch.sh 5afe69c2ccd0 38318f23a7ef 2 vlan-filter  START\_ID=$1  END\_ID=$2  SERIES\_VER=$3  TOPIC=$4  # Usually, the send email list is fixed, so  # we just need to generate once for version 1  gen\_sendemail() {  cat << EOF > $1  #!/bin/bash  # ./EXT-<topic>.sh <folder> to send all patch series  # ./EXT-<topic>.sh <folder>/0001-xyz.patch to send a patch  git send-email \$1 \\  --to="Some VIP <vip.some@somecopr.com>" \\  --to="Another Person <person.another@somecopr2.com>" \\  --cc="My Self <self.my@gmail.com>"  EOF  }  SENDMAIL=EXT-${TOPIC}-sendemail.sh  PATCHDIR=EXT-${TOPIC}-v${SERIES\_VER}  # ‘net’ is Mailing List(ML) tag for Linux networking subsystem ML=net  # In Linux M/L, we can send Request For Comment(RFC) or Patch  PATCHTYPE=PATCH  #PATCHTYPE=RFC  # We can reuse the send-email list  if [ x"$SERIES\_VER" == x"1" ]; then  git format-patch ${START\_ID}..${END\_ID} --cover-letter \  --subject-prefix="$PATCHTYPE $ML" -o $PATCHDIR  gen\_sendemail $SENDMAIL  chmod a+x $SENDMAIL  else  git format-patch ${START\_ID}..${END\_ID} --cover-letter \  --subject-prefix="$PATCHTYPE $ML v${SERIES\_VER}" -o $PATCHDIR  fi  FOR\_LINUX=false  if [ x"$FOR\_LINUX" == x"true" ]; then  # Linux project has get\_maintainer.pl script to list the mailing list  # recipients for a particular patch. So, it is good to cross-check  # the list in gen\_sendemail() is accurate  echo "Show maintainer list and PLEASE cross check if $SENDMAIL is READY"  echo "================================================================="  SERIES=$(ls -1 $PATCHDIR -I0000-cover-letter.patch)  for patch in $SERIES; do  echo -e "++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++"  ./scripts/get\_maintainer.pl $PATCHDIR/$patch | sed -e "s# (#\t\t\t(#g"  echo -e "------------------------------------------------------------"  done  echo "================================================================="  echo "After you have sent, you check at patchwork at \ https://patchwork.kernel.org/project/netdevbpf/list/"  echo "=================================================================" fi |

**Appendix D: vim editor configuration ~/.vimrc**

|  |
| --- |
| $ vim ~/.vimrc |
| colorscheme desert  set background=dark  set nonumber  set listchars=tab:>-,nbsp:#  set list  set hlsearch  set tabstop=8  highlight ExtraWhitespace ctermbg=red guibg=red  match ExtraWhitespace /\s\+$/  set tw=74  set fo+=t  " No annoying sound on errors  set noerrorbells  set novisualbell |
| **To mark TAB as >-------**  listchars=tab:>-,nbsp:#  **To highlight trailing and leading white-space**  highlight ExtraWhitespace ctermbg=red guibg=red  match ExtraWhitespace /\s\+$/  **To limit text width to 74 character.**  set tw=74 |