



Git for Geeks By Kevin

မာတိကာ

အခန်း (၁) Git Basic Setup

အခန်း (၂) အသုံးအများဆုံး command များ

အခန်း (၃) reset command အသုံးဝင်ပုံ

အခန်း (၄) git flow သို့မဟုတ် လှပတဲ့ လမ်းကြောင်းများ

အခန်း (၅) squash အကြောင်း သိကောင်းစရာ

အခန်း (၆) cherry-pick ဆိုတာဘာလဲ

အခန်း (၇) rebase အသုံးပြုပုံ

အခန်း (၈) မလိုချင်တဲ့ commit တွေကို ဘယ်လို drop လုပ်မလဲ နှင့် commit တွေကို ပြန်စီမံ

အခန်း (၉) git revert အကြောင်း သိကောင်းစရာ

အခန်း (၁၀) Log များအကြောင်းနှင့် git tag

အခန်း(၁) Git Basic Setup

Basic ဆိုပေမဲ့ installation ပိုင်းကို ထည့်မပြောတော့ပါဘူး။ (Installation ပိုင်းမသိရင် <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git> သွားလေ့လာနိုင်ပါတယ်)။ အရင်ဆုံး GitHub Or GitLab မှာ account တစ်ခုတော့ လိုပါလိမ့်မယ်။ ပြီးတော့ နောက်ပိုင်း push or pull လုပ်တိုင်းမှာ username နဲ့ password မတောင်းအောင် ssh key ကို ကိုယ့်ရဲ့ GitHub or GitLab profile နဲ့ ချိတ်ပေးဖို့လိုပါတယ်။ key generation ကနောက်ပိုင်းမှာ ၂ မျိုးရှိလာပါတယ်။

- ED25519 Key
- RSA Key

ED25519 key က RSA Key ထက်ပိုပြီး security ပိုင်းမှာ စိတ်ချရသလို၊ စွမ်းဆောင်မှုကလည်း ပိုကောင်းတယ်လို့ ပြောကြပါတယ်။ ပြီးတော့ key length က RSA Key လို မဟုတ်ပဲ ကျစ်လစ်ပါတယ်။ Generate လုပ်တဲ့ပုံစံက

```
ssh-keygen -t ed25519 -C "<comment>"
Generating public/private ed25519 key pair.
Enter file in which to save the key (/home/user/.ssh/id_ed25519):
```

RSA SSH Key ပဲသုံးမယ်ဆိုရင် command က

```
ssh-keygen -t rsa -b 2048 -C "email@example.com"
Generating public/private rsa key pair.
Enter file in which to save the key (/home/user/.ssh/id_rsa):
```

အဓိပ္ပါယ်ပြည်စုံတဲ့ နာမည် ဖြစ်မယ်ဆို ပိုကောင်းပါတယ်။ နောက်ပိုင်း key တွေများလာရင် ဘာအတွက်ဆိုတာ အလွယ်တကူမှတ်မိအောင်ပါ။ ကျွန်တော်ပေးတတ်တဲ့ naming convention နာမူနာ ကိုပြပါမယ်။

- git_ed25519 staging_ed25519 production_ed25519

သက်ဆိုင်ရာ information နဲ့ keytype ပါ။ ကိုယ်က key တစ်ခုပဲသုံးချင်တယ်ဆိုလည်း ရပါတယ်။ ဒါကတော့ ကိုယ့်ရဲ့သဘောပါ။ Key များရင် maintain လုပ်ရတာ များတတ်ပါတယ်။ ဒါပေမဲ့ incase of security breach ဆို တစ်ခုပဲ ပါသွားပါမယ်။

ကဲ ထားချင်တဲ့ folder လေးလည်း ရွေးပြီးသွားပြီဆို ဘာမှမထည့်ပဲ Enter key ပဲ သုံးကြိမ် ခေါက်သွားလိုက်ပါ။ ဒါဆို .ssh folder ထဲမှာ private key နဲ့ public key (.pub) ၂ ခုတွေပါလိမ့်မယ်။ Private

key ဆိုတာက ကိုယ့် local စက်ထဲမှာ ထားရမယ့် key ဖြစ်ပြီး၊ public key ကတော့ GitHub or GitLab profile မှာ တင်ရမယ့် key ဖြစ်ပါတယ်။ ပြီးတာနဲ့ command line ကနေ `cat yourkey.pub` command ကိုရိုက်ပြီးဖွင့်၊ အကုန် copy လုပ်ပြီး Profile ထဲက SSH Keys ဆိုတဲ့ tag ကနေထည့်ပေးလိုက်ပါ။

ပြီးတာနဲ့ .ssh folder ထဲမှာ config တစ်ဖိုင် ထည့်ပါမယ်။ GitHub သုံးသူက GitHub၊ GitLab သုံးသူက GitLab ပေါ့

```
Host github.com
  User git
  IdentityFile ~/.ssh/your_private_key
Host gitlab.com
  User git
  IdentityFile ~/.ssh/your_private_key
```

ဒါဆို pull or push အတွက် ready ဖြစ်ပါပြီ။

Repository တစ်ခု GitHub မှာဖြစ်ဖြစ်၊ GitLab မှာဖြစ်ဖြစ်ဆောက်လိုက်တိုင်း URL နှစ်ခုရပါတယ်။ တစ်ခုက HTTPS အတွက်ရယ်၊ နောက်တစ်ခု SSH အတွက်ရယ်။ တကယ်လို့ ကိုယ်က SSH key မထည့်ထားဘူးဆိုရင် HTTPS သုံးပြီး၊ pull or push လုပ်လို့ရပါတယ်။ ဒါပေမဲ့ ကိုယ့်ရဲ့ username နဲ့ password အမြဲရိုက်ထည့်ရပါတယ်။ အမြဲမရိုက်ထည့်ချင်လို့ ssh key သုံးလိုက်တာပါ။

```
git --version
git version 2.24.3 (Apple Git-128)
```

Installation လုပ်ပြီးသွားရင် git ဆိုတဲ့ command က ရိုက်လို့ရပြီးလို့ ယူဆပါတယ်။ folder အသစ်တစ်ခုယူပြီး git ကို initialize လုပ်ပြီး remote ရဲ့ repo ထည့်ကြည့်ရအောင်။ repository ကိုနောက်ပိုင်း repo လိုပဲ ပြောင်းခေါ်သွားပါမယ်။

`mkdir git_tutorial && cd git_tutorial` #mkdir က command နဲ့ folder အသစ်ဆောက်ပြီး && က ဆက်တာ၊ ပြီးတော့ အသစ်ဆောက်လိုက်တဲ့ folder ထဲသွားတာ

```
git init
git remote add origin git@github.com:kelvinkyaw/git\_tutorial.git
```

တကယ်လို့ ကိုယ်က `remote add origin` ရဲ့ နေရာမှာ `ssh` ရဲ့ URL မထည့်ပဲ `HTTPS` ရဲ့ URL ထည့်မယ်ဆိုလည်းရပါတယ်။ Project folder အသစ်ဆောက်လိုက်တိုင်းမှာ `git init` ဆိုတဲ့ command ကလိုအပ်ပါတယ်။ ဒါမှသာ သက်ဆိုင်ရာ `git` ရဲ့ default folder တွေကို hidden files တွေအနေနဲ့ create လုပ်ပေးသွားမှာဖြစ်ပါတယ်။ ကိုယ်က `GitHub` or `GitLab` ကနေ project ကို clone လုပ်ရင်တော့ `git init` ဆိုတာကို သက်သက်ထပ်ရိုက်စရာမလိုတော့ပါဘူး။

`git` ရဲ့ `config` folder ကိုဖွင့်လိုက်ပြီဆို ဒီလိုမျိုးတွေ့နိုင်ပါတယ်။ `config` folder က `.git/` ဆိုတဲ့ hidden ဖြစ်နေတဲ့ အောက်မှာ ရှိတတ်ပါတယ်။ အဲဒီ file ထဲမှာဆိုရင် `remote` ရဲ့ `host repo` တွေ၊ `local` နဲ့ `remote` ရဲ့ `branch` တွေ ရှိနေနိုင်ပါတယ်။ `branch` အသစ်ယူလိုက်တိုင်း ဒါမှမဟုတ် `remote` က `branch` တွေကို `fetch` လုပ်လိုက်တိုင်း အပြောင်းအလဲ ရှိရင် ဒီ `config` ထဲမှာ `update` ဖြစ်နေပါလိမ့်မယ်။ ဒီလောက်ဆို `Git Basic Setup` ကို လုံလောက်ပြီလို့ ယူဆပါတယ်။

```
[core]
    repositoryformatversion = 0
    filemode = true
    bare = false
    logallrefupdates = true
    ignorecase = true
    precomposeunicode = true
[remote "origin"]
    url = git@github.com:kelvinkyaw/git\_tutorial.git
    fetch = +refs/heads/*:refs/remotes/origin/*
[branch "master"]
    remote = origin
    merge = refs/heads/master
```

ကျွန်တော်တို့ အသုံးများတဲ့ command တွေဘက် ဆက်ကြရအောင်။

အခန်း (၂) အသုံးအများဆုံး command များ

ရှေ့ဦးစွာ အခြားသူများရဲ့ opensource project ပဲဖြစ်ဖြစ်၊ ကိုယ် contribute လုပ်ရမယ့် private repo ဖြစ်ဖြစ် လက်ရှိ repo က ရှိနေပြီးသားဆိုရင်၊ ကိုယ့်ရဲ့ စက်ထဲကို clone လုပ်ရပါတယ်။ command က `git clone`။ သူ့နောက်ကလိုက်လာတာက repo ရဲ့ URL ပါ။

```
git clone git@github.com:kelvinkyaw/git\_tutorial.git
```

ဒါဆို `git_tutorial` ဆိုတဲ့ folder ထဲမှာ တစ်ခြား developers တွေ လုပ်နေတဲ့ default master branch က code တွေ ကိုယ့်စက်ထဲ ရောက်လာပါပြီ။

ကိုယ့်က develop လုပ်ဖို့ အဆင်သင့်ဖြစ်ပြီဆိုရင် branch အသစ်တစ်ခုယူရပါမယ်။ ဒီနေရာမှာ ကျွန်တော့်ရဲ့ rules က master branch ရယ်၊ develop branch ရယ်မှာ ဘယ်သူမှ တိုက်ရိုက် push လုပ်ခွင့်မရှိဘူးဆိုတဲ့ rules ကိုသတ်မှတ်ထားပါတယ်။ developers အချင်းချင်း သဘောတူညီချက်ယူ၍ဖြစ်စေ၊ repo ရဲ့ setting ထဲက rules သတ်မှတ်၍ ဖြစ်စေ ဆောက်ရွက်လို့ရပါတယ်။ အကောင်းဆုံးက rules နဲ့ သတ်မှတ်ထားရင် မှားပြီး push လုပ်မိတာမျိုးကို ရှောင်ရှားနိုင်ပါလိမ့်မယ်။ ဒီအကြောင်းကို နောက်ပိုင်း အခန်း (၄) မှာ ဘာကြောင့်လုပ်သင့်လဲ ရှင်းပြပေးပါမယ်။ (စကားချပ် - ဒါဟာ ကိုယ်လက်ရှိလုပ်နေတဲ့ company ရဲ့ rules နဲ့လည်း သက်ဆိုင်ပါတယ်။ ကျွန်တော်က ဒီလိုပြောလို့ လုပ်လိုက်ပါတယ်ဆိုတာတော့ မဖြစ်စေချင်ပါ။ ကောင်း၏ မကောင်း၏ အဝင်ခွင့်ကျ ဖြစ်မဖြစ် လေ့လာပြီးမှ လုပ်တာပိုကောင်းနိုင်ပါတယ်)

branch အသစ်ယူရင် သုံးရမယ့် command က

```
git checkout -b <branch_name>
```

ကိုယ်က folder လေးနဲ့ခွဲထားချင်တယ်။ ဥပမာ (feature, bugfix)။ ဒီလိုမျိုးဆို branch name က feature/ နဲ့စလို့ရပါတယ်။ developer အနေနဲ့ပါထပ်ခွဲထားချင်သေးတယ်ဆိုရင်လည်း feature/kevin/ ပြီးမှ ဘာအတွက်လုပ်တယ် ဆိုတဲ့ branch name။ ကျွန်တော်က လက်ရှိ issue tracker software တစ်ခုသုံးပါတယ်။ issue တစ်ခု create လုပ်လိုက်တိုင်း number တစ်ခု ထွက်ပါတယ်။ ဆိုပါစို့... issue တစ်ခုပေါ်လာပြီ၊ သူ့ရဲ့ issue က login form မှာ password က plain text နဲ့ encrypt မလုပ်ထားဘူးဆိုတဲ့ story ကို create လုပ်လိုက်မယ်၊ သူ့ရဲ့ number က 00001။ ဒါဆို branch name sample

က bugfix/kevin/00001-password-plain-text-in-login-form Command

အပြည့်အစုံဆိုရင်

```
git checkout -b bugfix/kevin/00001-password-plain-text-in-login-form
```

ဒါဆို နောက်ပိုင်းမှာ project management ပိုင်းက တာဝန်ရှိသူကလည်း 00001 ကိုကြည့်တာနဲ့

“ဪ story 00001 က fix လုပ်နေပြီ၊ Staging မှာ deploy လုပ်ပြီးပြီ၊ QA Test လုပ်နေပြီ”

စတဲ့ stage တွေကို tracking လုပ်ရတာ ပိုလွယ်သွားပါတယ်။

ကိုယ်ဘယ် files တွေ ပြောင်းလဲထားလဲ၊ အသစ်ထည့်ထားလဲ သိချင်ရင်

```
git status
```

git add ဆိုတဲ့ command ကတော့ repo ပေါ်တင်ချင်တဲ့ files တွေ folders တွေကို ကိုယ့် local ထဲမှာရှိတဲ့ cache (index) ထဲထည့်ထားတာပါ။ ဒီအချိန်မှာ cache ထဲမှာပဲရှိသေးတဲ့အတွက် အချိန်မရွေး

ပြန်လည်ပြင်ဆင်တာ လုပ်လို့ရပါသေးတယ်။ add ဆိုတဲ့ command နောက်က ထည့်ချင်တဲ့ file path or folder path ပါ။ ကိုယ့်က အကုန်ထည့်မှာ သေချာရင် git add . ဆိုပြီးလည်း ရပါသေးတယ်။ ဥပမာ

```
git add src/main.js
git add index.php
git add app/Controller
git add app/Model
git add tmp
```

cache ထဲမှာ ထည့်ထားတဲ့ အရာတွေကို ပြန်ဖြုတ်ချင်ရင် git reset သုံးပြီး ပြန်ဖြုတ်လို့ရပါသေးတယ်။

reset ရဲ့နောက်မှာ ထည့်ထားမိခဲ့တဲ့ files or folder ပါ။

```
git reset src/main.js
```

push မလုပ်ခင် commit message ထည့်ပေးရပါတယ်။ Command က

```
git commit -m "commit message"
```

ကျွန်တော် commit message ရေးတဲ့ပုံကို ကိုယ်ပိုင် format တစ်ခုထားပါတယ်။ ဒါမှသာ commit message ကိုကြည့်ပြီး story number နဲ့ track လုပ်လို့လဲ ရနိုင်မှာပါ။ ပုံစံက

```
git commit -m "[Login 00001] fixed plain text password"
```

ဒါဟာ ရှေ့မှာ ပြောခဲ့တဲ့ tracker နဲ့ link လိုသဘောထားပြီး ပေးသွားတဲ့ commit message ပါ။

Login ဆိုတာ controller ရဲ့ name ပါ။

00001 ဆိုတာက issue tracker ရဲ့ id ပါ။

နောက်ကတော့ ပေးချင်တဲ့ message ပေါ့။ တကယ်လို့ ကိုယ်က issue tracker မရှိဘူးဆိုလည်း developers တွေအားလုံးသဘောတူထားတဲ့ format တစ်ခုကို ပေးလည်း ရပါတယ်။

စိတ်ဝင်စားစရာနောက်တစ်ခုက commit message ရှေ့မှာ [WIP] ဆိုတာ မျိုးထည့်ထားရင် ဒါဟာ Work In Progress ဆိုတာကို သိပြီး Merge Request လုပ်မရအောင် လုပ်ထားလို့ရပါသေးတယ်။ ဒါကတော့ နောက်ပိုင်းမှာ အသုံးဝင်လာမယ့် အပိုင်းမို့ သေချာမှတ်ထားစေချင်ပါတယ်။

ပြီးတာနဲ့ ကိုယ့် local ထဲက changes ရှိတဲ့ code တွေကို GitHub, GitLab Server တွေပေါ်တင်တော့မယ်ဆို

```
git push <remote_name> <branch_name>
```

အခန်း (၁) မှာ git remote add origin ဆိုပြီးရိုက်ခဲ့တာကို မှတ်မိမယ်ထင်ပါတယ်။ origin လို့မပေးပဲ တခြားနာမည် ပေးထားခဲ့မယ်ဆိုရင် အဲ့ဒီ origin ရဲ့ နေရာမှာ replace လုပ်သွားရပါမယ်။ ကျွန်တော်က GitHub ကို origin လို့ပေးပြီး၊ GitLab ကို upstream လို့ ပေးခဲ့မယ်ဆိုရင်

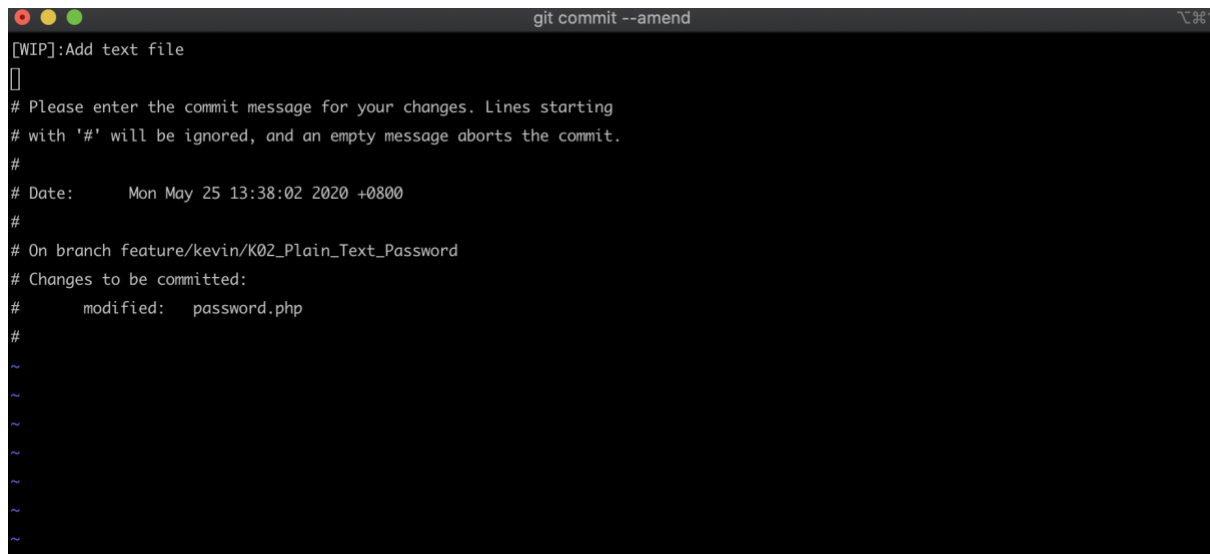
```
git remote add origin git@github.com:kelvinkyaw/git\_tutorial.git
git remote add upstream git@gitlab.com:kelvinkyaw/git\_tutorial.git
```

origin ရော upstream ရောမကြိုက်ဘူးဆိုလည်း မိမိနှစ်သက်ရာ ပေးလို့ရပါတယ်။ ဘယ်သူမှ သုံးကြတာမဟုတ်ပေမယ့် လုပ်လို့ရတယ်ဆိုတာကို ပြောပြတာပါ။

push မလုပ်ရသေးတဲ့ commit message ကို ပြင်ချင်တယ်ဆိုရင်

git push --amend ဆိုပြီးရိုက်လိုက်ရင် CLI (Command Line) ရဲ့ default editor (vim) တစ်ခုပေါ်လာပြီး ပြင်ချင်တဲ့ message ကို ဒီလိုပြင်လို့ရပါတယ်။ သတိထားရမှာက git မှာ push

လုပ်ပြီးသား message ကိုပြင်ပြီး commit ထပ်လုပ်ရင် နောက်ထပ် commit id တစ်ခု ထပ်တိုးတယ်ဆိုတာ သတိချပ်စေချင်ပါတယ်။ push လုပ်ပြီးသား message ကိုပြင်ရင် ကိုယ့်ရဲ့ server မှာ update ဖြစ်ဖို့အတွက် force push လုပ်ရတယ်ဆိုတာ သတိထားပါ။ push လုပ်ပြီးသား message ကိုပြင်ချင်ရင် လုပ်တဲ့ နည်းကို နောက်အပိုင်းတွေမှာ ပြောသွားပါမယ်။



```

git commit --amend
[WIP]:Add text file
[]
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Mon May 25 13:38:02 2020 +0800
#
# On branch feature/kevin/K02_Plain_Text_Password
# Changes to be committed:
#   modified:   password.php
#
~
~
~
~
~
~

```

နောက်တစ်ခုက branch တွေကို ဖျက်ပစ်ချင်ရင် သုံးတဲ့ command ပါ။

```

git branch -d <branch_name>
git branch -D <branch_name>

```

အကြီးအသေးကွာတာ အကြောင်းရှိပါတယ်။ -d က ကိုယ် push and merged လုပ်ပြီးသား ဆိုတဲ့အချိန်မှာ branch ကို ဖျက်ပစ်တာပါ။

-D ကတော့ delete with force ပါ။ ဘာမှ local ထဲမှာ မကျန်ပါဘူး။ အကုန် ဖျက်ပစ်ပါတယ်။ သတိထားပြီးသုံးသင့်တဲ့ option ပါ။ ပြီးတော့ remote က branch ဖျက်နည်းကျတော့ နောက်တစ်မျိုးပါ။

```

git push <remote_name> --delete <branch_name>

```

နောက်ထပ်အရေးကြီးတဲ့ basic command တစ်ခုက stash ဆိုတဲ့ command ပါ။

ဘယ်နေရာမှာအသုံးဝင်လဲဆိုရင် လုပ်နေရင်းတန်းလန်း commit လုပ်လို့လည်းမရသေးဘူး။ ရှေ့က လုပ်ထားတဲ့ တစ်ခြား branch ရဲ့ commit က file မှာ error တတ်လို့ fix အရင်လုပ်ရမယ်။ လုပ်လက်စကို commit လုပ်လိုက်ရင်လည်း မလိုအပ်ပဲ commit တွေ ထပ်နေအုံးမယ်။ ဒီ command ကိုရိုက်တာနဲ့ လက်ရှိ လုပ်နေတဲ့

အလုပ်တွေ အရင် original stage ပြောင်းသွားပါပြီ။ ပြီးတာနဲ့ ကိုယ်သွားချင်တဲ့ branch ကိုသွား၊ fix လုပ်။

Command က

```
git stash
```

သူ့နောက်မှာ option [list|show|drop|pop|apply|branch|clear] တွေရှိပါသေးတယ်။

အကုန်လုံးတော့ အသုံးမများပေမယ့် အနည်းဆုံးတော့ ဘာလဲသိထားရင် ကောင်းပါတယ်။

list|show ဆိုရင် ဘယ် files တွေ ဖျောက်ထားတာလဲကြည့်တာ

drop|clear ဆိုရင် ဖျောက်ပြီး ပြင်ထားတဲ့ files တွေကို cancel လုပ်လိုက်တာ။ checkout

သဘောပေါ့။

pop|apply ဆိုရင် ဖျောက်ထားတဲ့ files တွေကို ပြန် display ပြန်လုပ်တာ။

branch ဆိုတာကတော့ ဘယ် branch မှာ stash လုပ်ထားတာလဲဆိုတာပါ။ existing branch မဖြစ်ရပါဘူး။

branch အသစ်ထဲမှာ stash လုပ်တာနဲ့တူပါတယ်။ command အပြည့်အစုံက

```
git stash branch <branch_name>
```

```
git stash
```

နောက်အသုံးဝင်တဲ့ command က changes ထားပြီးသာ files ကို cancel လုပ်မယ်ဆို အကုန် original stage ကိုပြန်ရပါတယ်။

```
git checkout <file_name>
```

file ထဲမှာ ဘာတွေ ပြင်ထားလဲ သိချင်ရင်

```
git diff <file_name>
```

နောက်ဆုံး command ကတော့ merge ဆိုတဲ့ command ဖြစ်ပါတယ်။ ကျွန်တော့်အတွေးအကြံဆိုရင် ကိုယ့်ကိုလာပေါင်းတာလား၊ သူ့ကိုသွားပေါင်းတာလားဆိုတဲ့ ဒွိဟတွေ ရှိခဲ့ဘူးပါတယ်။ အဲ့အတွက် ကျွန်တော် ဥပမာတစ်ခုပြောပြပါမယ်။

ကိုယ့်မှာ ပိုက်ဆံ ၁၀ ရှိတယ်။ အောင်အောင်ဆီက ပိုက်ဆံ ၁၀ ချေးမယ်။ ကိုယ့်မှာ ၂၀ ဖြစ်သွားမယ်။ ကိုယ့်လက်ရှိ branch ထဲက ၁၀ ပါ။ အောင်အောင်က အချေးခံရမယ့် branch ထဲက ၁၀ ပါ။ ကိုယ်က develop branch မှာ active ဖြစ်နေတယ်။ feature_1 ဆိုတဲ့ branch ကို ကိုယ့်ဆီကို ပေါင်းထည့်ချင်မယ်။ Command က

```
git merge <branch_name>
```

ဒါဆို `feature_1` ဆိုတဲ့ branch ထဲက code တွေ `develop` ထဲရောက်လာပါပြီ။ ဒါပေမဲ့ ကျွန်တော် `merge` ကို ကိုယ့်ရဲ့ `local` ထဲမှာ ဘယ်တော့မှ မလုပ်ပါဘူး။ GitHub or GitLab ကနေပြီးတော့ `Merge Request` အရင်လုပ်ပြီးမှာ ပေါင်းပါတယ်။ ဒါကလည်း ကိုယ့် `local` ထဲမှာ `merge` လုပ်ပြီးရင် `remote branch` တွေကို `push` လုပ်ရတာကို ရှောင်ချင်လို့ပါ။ အဓိကက `develop` or `master branch` တွေကို `local` မှာ `merge` လုပ်တာထက် `remote` မှာ `merge` လုပ်တာ ပိုကောင်းတယ်လို့ ယူဆလို့ပါ။ ဘာကြောင့်လဲဆိုရင် ကျွန်တော် အခန်း(၄) မှာဆက်ရှင်းပြပါမယ်။

နောက်ထပ်အသုံးဝင်တဲ့ command တစ်ခုက

```
git fetch origin
```

ဆိုတာပဲဖြစ်ပါတယ်။ သူကတော့ ကိုယ့် လက်ရှိ code တွေ၊ branch တွေမှာ ပြောင်းလဲခြင်း မရှိပေမယ့် တစ်ခြား `developers` တွေ `push` လုပ်ထားတဲ့ code တွေကို branch အလိုက် ကိုယ့် `local` ထဲမှာ `update` ဖြစ်အောင် လုပ်တာပါ။ အလွယ်ဆုံးပြောရရင် A ဆိုတဲ့ developer က `feature_2` ဆိုတဲ့ branch ကို `push` လုပ်ထားမယ်။ ကိုယ်က သူ့ branch မှာ ဘာတွေ ပြောင်းထားလဲ ကိုယ့် `local` စက်ထဲမှာ ကြည့်ချင်တယ်ဆို `git checkout feature_2` ဆိုပြီးသွားရပါတယ်။ ကိုယ်က `git fetch origin` မခေါ်ထားရင် `remote` မှာရှိနေတဲ့ branch တွေကို `fetch` မလုပ်ထားတဲ့အတွက် `checkout` လုပ်လို့မရပါဘူး။

`Code Review` လုပ်တဲ့ အချိန်မှာ အသုံးဝင်ပါတယ်။ တဆက်တည်းမှာပဲ `code review` က `developer` တိုင်းအတွက် အရေးကြီးတယ်ဆိုတာ ပြောပြလိုပါတယ်။ သူများရေးထားတဲ့ code တွေကို အစိမ်းသက်သက် ဝင်ကြည့်ရတာထက် ကိုယ့်နဲ့အတူတူ အလုပ်လုပ်နေတဲ့ code တွေကိုကြည့်ရတာ `business logic` နားလည်ပြီးသား ဖြစ်နေလို့ `learn` လုပ်လို့လည်းရသလို `error` ပါဝင်မှုကိုလည်း လျော့ချပေးနိုင်ပါတယ်။

ဒီလောက်ဆို အသုံးများတဲ့ command တော်တော်များများ cover ဖြစ်နေပြီမို့ နောက်ထပ်အရေးကြီးတဲ့ `reset` ပိုင်းကို လှည့်ကြရအောင်။

အခန်း (၃) reset command အသုံးဝင်ပုံ

git reset ရဲ့ နောက်မှာ [--soft, --mixed, --hard, --merge, --keep] option ၅ ခုရှိပါတယ်။ အသုံးအများဆုံးကတော့ [--soft, --hard] ပါ။ ပြီးမှ ကိုယ် reset လုပ်ချင်တဲ့ target ရဲ့ commit။ အပြည့်အစုံ command က

```
git reset <--option> <commit>
```

အောက်က table လေးကိုကြည့်ပါ။

working	index	HEAD	target		working	index	HEAD
-----					-----		
A	B	C	D	--soft	A	B	D
				--mixed	A	D	D
				--hard	D	D	D

ဒီ table ကိုကြည့်မယ်ဆိုရင် ဒါလေးတွေ သိထားဖို့ လိုပါတယ်။

- working: လက်ရှိလုပ်၊ ပြင်နေဆဲ files တွေ
- index: commit မလုပ်ခင် git ရဲ့ cache ထဲမှာ git add ဆိုပြီး လုပ်ထားခဲ့တဲ့ files တွေနဲ့ လက်ရှိ changes ဖြစ်နေတဲ့ files တွေ
- HEAD: ဆိုတာက လက်ရှိကိုယ့်ရဲ့ branch နဲ့ active ဖြစ်နေတဲ့ <commit>
- target: ဆိုတာက ကိုယ်လက်ရှိ <commit> ကို target လုပ်မယ့် <commit> ရဲ့ value

[--soft] Option ဆို ဘာဖြစ်လဲ

ဒီ option က ကိုယ်လက်ရှိ လုပ်နေတဲ့ files တွေ၊ ပြောင်းထားတဲ့ code တွေ၊ index (cache)

ထဲမှာရှိနေတဲ့အရာတွေ မပျောက်သွားပဲ၊ ကိုယ့်ရဲ့လက်ရှိ <commit> က target <commit>

အဖြစ်ပြောင်းသွားပြီး၊ target <commit> မှာရှိတဲ့ changes တွေပါ ကိုယ့် local စက်ထဲရောက်လာမယ့် option ပါ။

ဥပမာ ကိုယ့် <commit> က <8846b1> အဲ့ဒီထဲမှာ index.php ဆိုတဲ့ file ကို ပြောင်းထားတယ်။

ကိုယ် reset လုပ်ချင်တဲ့ target <commit> က <593x82> နဲ့ သူ့ထဲမှာ faq.php ဆိုတဲ့ file

အသစ်ထည့်ထားတယ်။ <8846b1> ကို <593x82> နဲ့ soft reset လုပ်လိုက်ပြီဆို index.php

မှာ ပြောင်းထားတာတွေလည်း ရှိနေအုံးမယ်၊ `faq.php` ဆိုတဲ့ file အသစ်လည်း ရောက်လာမယ်။
command အပြည့်စုံက

```
git reset --soft 593x82
```

[--mixed] Option ဆို ဘာဖြစ်လဲ

ဒီ option ကလည်း လုံးဝကြီး History ပါမကျန်တော့အောင်ကို ဖြစ်သွားတဲ့ ရွေးချယ်မှုတစ်ခုမဟုတ်ပါဘူး။ လက်ရှိလုပ်နေတဲ့ working files တွေ၊ working directories တွေကျန်နေအုံးမှာပါ။ ဒါပေမဲ့ index ကတော့ override လုပ်လိုက်တဲ့ commit ရဲ့ index ဖြစ်သွားပြီး override လုပ်တဲ့ commit မှာ ရှိနေတဲ့ changes code တွေ ရောက်လာမှာဖြစ်ပါတယ်။ Soft နဲ့အလားတူပါပဲ။ Command က

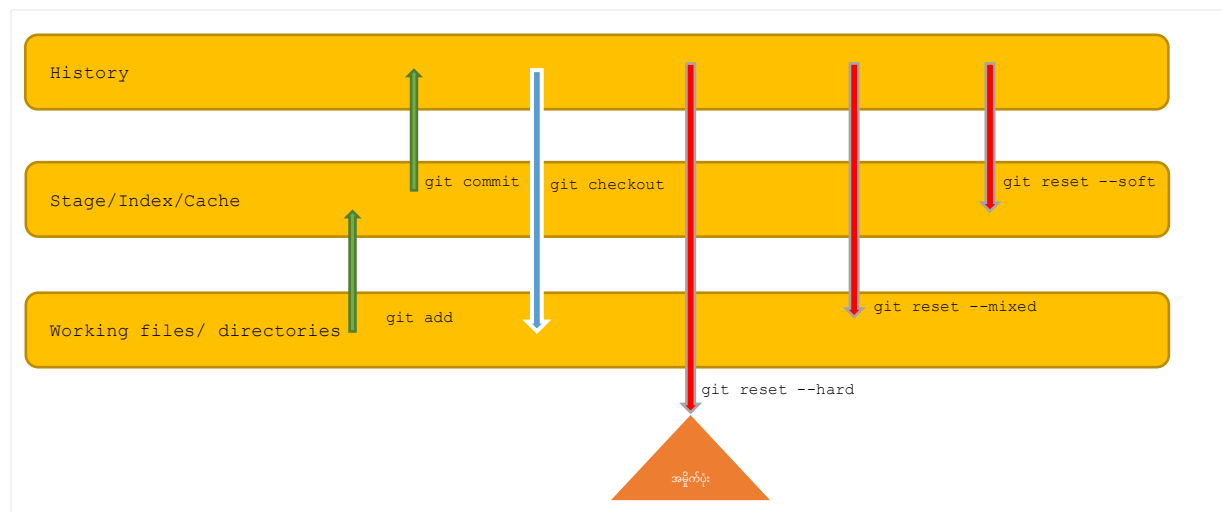
```
git reset --mixed 593x82
```

[--hard] Option ဆို ဘာဖြစ်လဲ

လုံးဝကိုယ်လုပ်ထားတဲ့ working files တွေ၊ directories တွေမကျန် go to trash ဖြစ်သွားပါပြီ။ ဘာနဲ့မှ recovery လုပ်လို့လည်းမရပါဘူး။ folder လိုက်ကြီး copy ကူးထားရင်တော့ တစ်မျိုးပေါ့။ ကျွန်တော်ဆို ဒီ command ကသေချာပြီဆိုမှ လုပ်ပါတယ်။ မသေချာရင် branch တစ်ခု အသစ်ယူပြီး commit လုပ်ထားတတ်ပါတယ်။ ဒါမှသာ ကိုယ့်ရဲ့ histories လေးတွေရှိနေမှာပါ။ လုပ်ခြင်း မလုပ်ခြင်းက ကိုယ့်အပေါ်မှာပဲမူတည်ပါတယ်။ soft ရဲ့ example အရဆို `<8846b1>` ဆိုတာမရှိတော့ပဲ `<593x82>` ပဲရှိနေတော့မှာပါ။ အန္တရာယ်ရှိတဲ့ command မို့ သတိထားပြီးသုံးသင့်ပါတယ်။ command အပြည့်အစုံက

```
git reset --hard 593x82
```

ပုံလေးကို နားလည်လွယ်အောင် တစ်ချက်ကြည့်လိုက်ရအောင်



[--keep] Option ဆို ဘယ်လို အလုပ်လုပ်သလဲ

ဒီ option အကြောင်း အသုံးမများလို့ မရေးပေးတော့ဘူးလို့ စဉ်းစားထားပေမယ့် တစ်ခါတစ်ရံအသုံးဝင်နိုင်လို့ ထည့်ရေးပေးထားပါတယ်။ ဘယ်လို scenario မှာသုံးတတ်လည်းဆိုရင် ကိုယ်ကလက်ရှိ branch1 မှာအလုပ်လုပ်နေတယ်။ ပြီးတော့ commit လုပ်ထားလိုက်တယ်။ နောက်ပြီး အဲ့ဒီ branch မှာပဲ ဆက်လုပ်နေမိတယ်။ အဲ့အချိန်မှာ အရေးကြီးတာ နောက်တစ်ခုပေါ်လာလို့ branch အသစ်ယူပြီး ဆက်လုပ်ရမယ်။ ကိုယ်က branch2 ကို switch လုပ်မိသွားတယ်။ ဒါပေမဲ့ branch2 က branch1 ကနေ ခွဲထွက်ရမှာ မဟုတ်ဘူး။ အမှန်ဆို develop ကနေခွဲထွက်ရမှာ။ ဒီလိုအချိန်မျိုးဆို အရမ်းအသုံးဝင်ပါတယ်။ ကိုယ်လက်ရှိခွဲထုတ်လိုက်တဲ့ branch က branch1 ကဖြစ်နေပေမယ့် ဒီ command ဆို ကိုယ်ခွဲထုတ်လိုက်ချင်တဲ့ target branch က commit ဖြစ်သွားပါပြီ။ command အပြည့်အစုံက

```
git reset --keep <commit>
```

အောက်က scenario လေးကို ကြည့်ပါ။

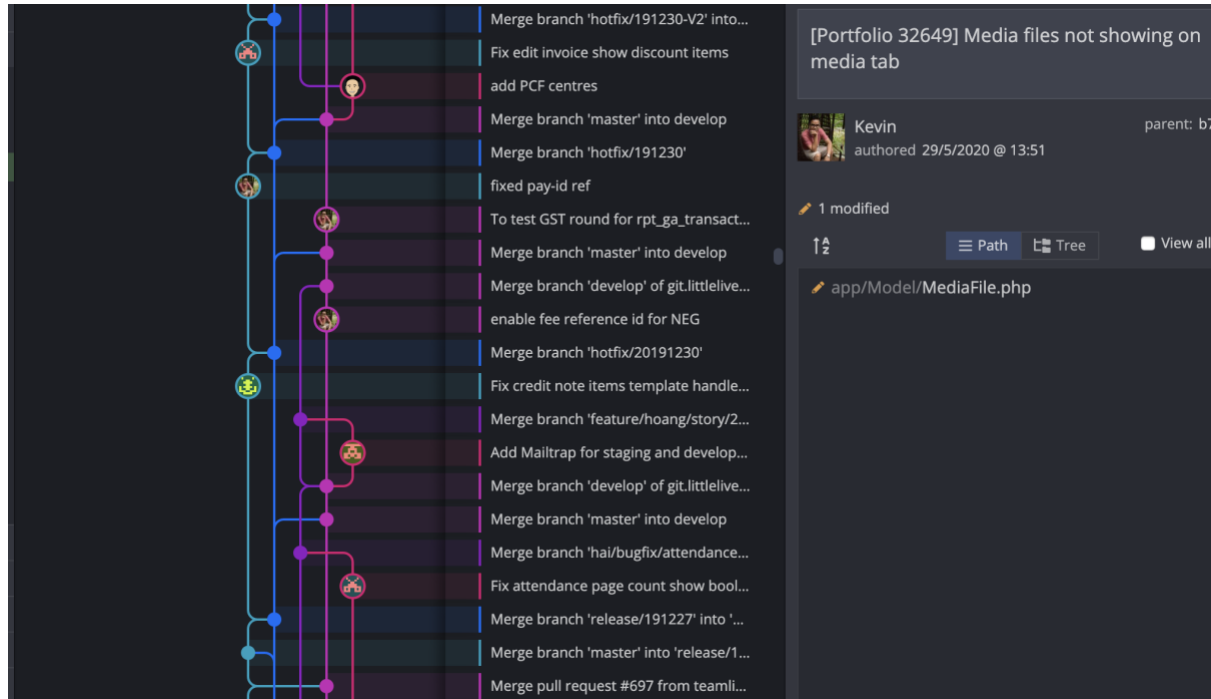
```
git checkout develop
git switch -c branch1
#do your stuff and changes
git commit -m "your commit message"
git switch -c branch2
git reset --keep <develop_commit>
```

ကြိုတုန်း နောက်ထပ် command လေးတစ်ခု bonus ပါ။ switch -c ဆိုတာ git checkout -b <new_branch_name> နဲ့အတူတူပါပဲ။ -c or -b မပါရင်တော့ branch တစ်ခုကနေ နောက်တစ်ခု ပြောင်းတာပါ။ git checkout <branch_name> နဲ့လည်း အတူတူပါပဲ။

[--merge] option ကိုတော့ ကျွန်တော် အထွေအထူး မပြောပြနေတော့ပါဘူး။ သုံးတာနည်းတာရယ်။ scenario က သိပ်ပြီး မထူးခြားတာရယ်။ သူထက်လွယ်တဲ့ နောက်ထပ်နည်းတွေ များတဲ့အတွက် ကျွန်တော်တော့ မသုံးပါဘူး။ သိချင်သေးတယ်ဆိုရင်တော့ research လုပ်ကြည့်ပြီး လာတိုင်ပင်လို့ရပါတယ်။ ဒီလောက်ဆို reset အကြောင်းကောင်းကောင်းသိပြီလို့ ယူဆပါတယ်။ ကျွန်တော်တို့ git flow ကိုဆက်ကြရအောင်။

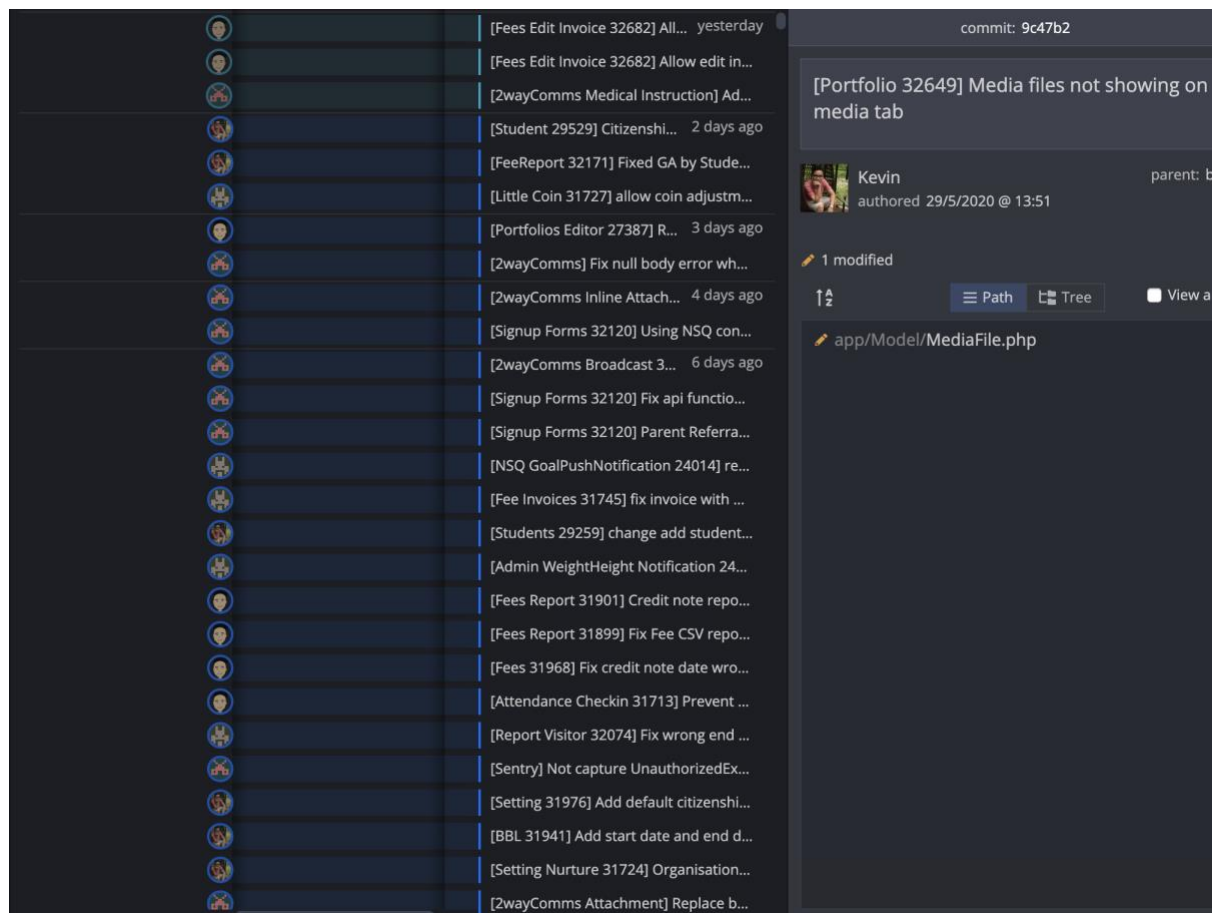
အခန်း (၄) git flow သို့မဟုတ် လှပတဲ့ လမ်းကြောင်းများ

ရှေ့ဦးစွာ ဘာမှ မပြောခင် အောက်က ပုံလေးကို တစ်ချက် ကြည့်ကြည့်ရအောင်။



လှပသေချာတဲ့ git flow ဆိုဒီလို ရှင်းမရအောင် ရှုပ်ယှက်ခက်မနေပါဘူး။ သက်ဆိုင်ရာ developer တိုင်းက feature အသစ်တွေ bug fix တွေဆို develop ကနေ branch အသစ်ယူကြတယ်။ ပြီးမှ ကိုယ် checkout လုပ်ထားတဲ့ branch ကနေပြီး develop ကို ပြန်ပြီး merge လုပ်ကြတယ်။ ပြီးတော့မှ master ကို ထပ်ပြီး merge ထပ်လုပ်ကြပြန်ပါတယ်။ ဒါတော်တော်များများနားလည်လက်ခံထားတဲ့ flow ပါ။ ဒီ flow တစ်ခုလုံးရဲ့ အားနည်းချက်ကို ပြပါဆိုရင် အပေါ်က ပုံကို ကြည့်ပါ။ ဒါ flow အနည်းငယ် ပြောင်းထားပြီးသားပုံပါ။ ကျွန်တော့်မှာ ပုံအဟောင်းမရှိတော့လို့ တကယ်ဆို အဲ့ထက်ပိုရှုပ်ပါတယ်။ အကယ်၍များ ကျွန်တော် အဲ့ထဲက commit လေးတစ်ခုပဲ release လုပ်ချင်တယ်ဆို သေအောင် တိုင်ပတ်ပါပြီ။

ဒါဆို ကျွန်တော်တို့ ရှင်းအောင်လုပ်လို့မရတော့ပဲ အရှုံးပေးရတော့မှာလား။ လုံးဝမဟုတ်ပါဘူး။ တစ်ယောက်တည်း project တစ်ခုလုံး လုပ်ထားတာလည်း မဟုတ်ဘူး။ လေးငါးခြောက်ယောက် ရေးထားတာကို စံတစ်ခုဖြစ်အောင် လုပ်မယ်ဆို စံတွေသတ်မှတ်ထားရပါမယ်။ ဘယ်လိုစံလဲ၊ တစ်လိုင်းတည်း ဖြစ်အောင် လုပ်လို့ရတဲ့ စံပါ။ ကျွန်တော် ရှေ့မှာတင်ပြခဲ့ဖူးသလို၊ ဘယ်သူမှ master နဲ့ develop ကို push တိုက်ရိုက်လုပ်ခြင်းမျိုး၊ local မှာ ဒီ branch တွေကို merge လုပ်ခြင်းမျိုး တွေ ရှောင်ပါတယ်။ နောက်ထပ်တစ်ပုံ ထပ်ကြည့်ရအောင်။



ကျွန်တော်တို့ လက်ရှိ developer ဖိ ယောက်နဲ့ core product မှာရေးနေကြတာပါ။ release မလုပ်ချင်တဲ့ commit ကို ချန်ခဲ့မလား၊ commit လေးတစ်ခုပဲ release လုပ်မလား ကြိုက်တဲ့ပုံစံလုပ်ပြီး release လုပ်လို့ရပါတယ်။ ဒီ flow ကိုမရခင် ကျွန်တော် git ကို အခြေခံလောက်ပဲ နားလည်ခဲ့တာပါ ရိုးသားစွာဝန်ခံပါတယ်။

Standard ဖြစ်ဖို့အတွက် checklist

- မည်သူမှ master or develop ကိုတိုက်ရိုက် push လုပ်ခွင့်မပြု - (Exception case ရှိ)
- branch အသစ်အားလုံး develop ကနေပဲ checkout လုပ်ရမယ်။
- develop ကအသစ်မယူခင် origin က update ကို `git fetch origin && git pull --rebase` command သုံးပြီး update အမြဲယူထားရမယ်။
- `git pull origin <develop> or <master>` လုံးဝမသုံးရပါဘူး။
- မိမိ local စက်ထဲမှာ develop or master နဲ့ ကိုယ့် branch ဘယ်တော့မှ merge လုပ်ခွင့်မရှိ

- Develop လုပ်ပြီးသာ code တွေကို branch အသစ်နဲ့ GitHub or GitLab ပေါ်တင်ထားပြီးပြီးဆို Merge Request လုပ်ပြီး သက်ဆိုင်ရာ teammate ကို review လုပ်ခိုင်း၊ approve ဖြစ်မှ GitHub or GitLab မှာပဲ merge လုပ်
- အမြဲတမ်း fast-forward merge strategy ကိုပဲသုံးရမယ်။

ထူးဆန်းတာတစ်ခု မြင်မိမယ်ထင်ပါတယ်။ fast-forward merge ဆိုတဲ့ merge လုပ်တဲ့ ပုံစံတစ်ခုပါ။

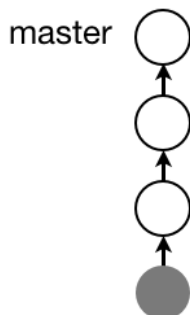
ပုံမှန် git မှာ merge လုပ်တဲ့အခါ fast-forward ကိုပဲသုံးကြပါတယ်။ non-fast-forward (no-ff)

ဆိုရင်ကို default configuration မှာသတ်မှတ်ထားရင်သော်လည်းကောင်း၊ -no-ff

လို့ထည့်ရိုက်ရင်သော်လည်းကောင်း execute လုပ်ပါတယ်။ ဘာကွာလဲဆိုရင် histories ကွာသွားပါတယ်။

အောက်က ပုံလေးကို ကြည့်ပါ။

```
git fetch ariya
git merge ariya/speedup
```



```
git fetch ariya
git merge -no-ff ariya/speedup
```

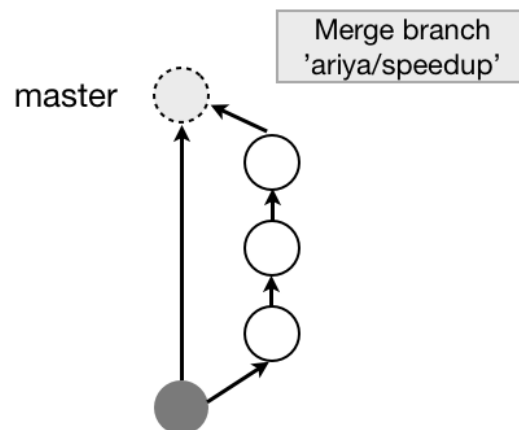
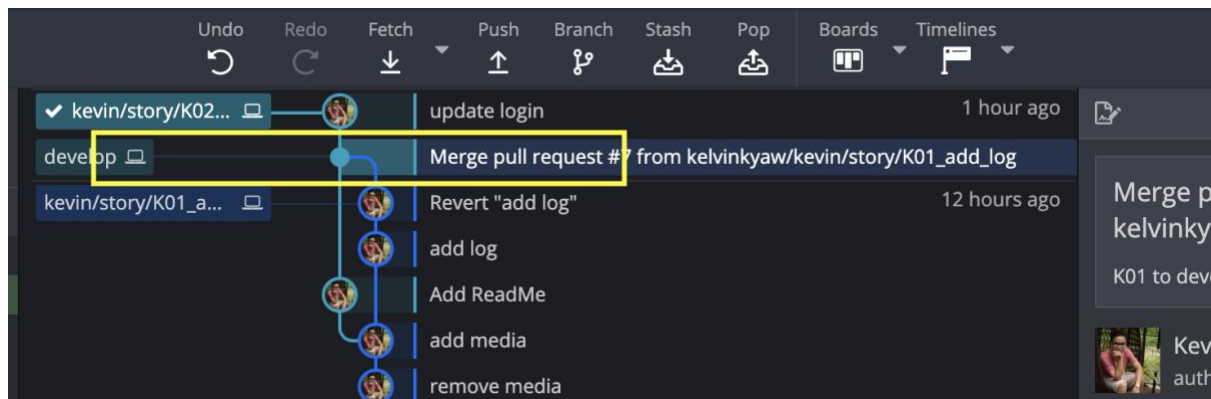


Figure 1 Source: ariya.io

fast-forward နဲ့ merge လုပ်ရင် တစ်လိုင်းတည်း ဖြစ်သွားတဲ့အပြင် dotted line နဲ့ပြထားတဲ့ ညာဘက်ကပုံလိုလည်း commit နောက်တစ်ခု အပိုမထွက်နေတော့ပါဘူး။ ကျွန်တော့်ရဲ့ပုံလေးကို ကြည့်ပါ။



-no-ff ကိုသုံးပြီး merge လုပ်ထားပုံပါ။ ဒီလောက်ဆို နောက်ဆုံး checklist ကိုသိလောက်ပါပြီ။ ဒီလို checklist ကို သတ်မှတ်ထားပြီးပြီဆို ကျွန်တော်တို့ လိုင်းတစ်လိုင်းတည်း ဖြစ်ဖို့က လမ်းတစ်ဝက် ရောက်လာပါပြီ။ နောက်ထပ်တစ်ဝက်ရဲ့အခြေခံက squash, cherry-pick and rebase ပါ။ ဒီသုံးခု အကျယ်နောက်အပိုင်းတွေမှာ တစ်ပိုင်းချင်းစီ ပြောပြသွားပါမယ်။

ဒီအခန်း တစ်ခုလုံးရဲ့ရည်ရွယ်ချက်က ရှုပ်ယှက်ခက်နေတဲ့ လိုင်းတွေ အများကြီး ရှိမနေပဲ တစ်လိုင်းတည်းဖြစ်အောင်လုပ်မယ်။ ပြီးမှ commit လေးတွေပဲ ရွေးပြီး release လုပ်ချင်တာတို့၊ branch တစ်ခုတည်းမှာ မလိုအပ်ပဲ ထပ်နေတဲ့ commit တွေဖယ်ရှားပစ်ချင်တာတို့စတဲ့ အလုပ်တွေအပြင် ကြည့်လိုက်တာနဲ့ “အော် ဒါက ဒီလိုရှိပါလား၊ wowwww” ဖြစ်သွားရအောင်ပါ။ တစ်ယောက်တည်းလုပ်ပြီး၊ လုပ်သမျှ commit တိုင်းလည်း develop မှာကြီးပဲ push လုပ်လဲ one line ဖြစ်ပါတယ်။ ၇

အဲ့တော့ squash ဘက်ကို လှည့်ကြရအောင်

အခန်း (၅) squash အကြောင်း သိကောင်းစရာ

Squash ဆိုပြီး သူ့ကို တိုက်ရိုက် run လို့ command တော့ မရှိတာကလည်း git ရဲ့ မျက်လှည့်ဆန်တဲ့ ပညာလို့ပဲ တင်စားချင်ပါတယ်။ branch တစ်ခုထဲမှာ commit တွေ အခါခါလုပ်မိနေပြီ။ ပြီးတော့ commit တိုင်းကလည်း feature တစ်ခုတည်းနဲ့ပဲ သက်ဆိုင်နေတယ်ဆိုရင် squash ဆိုတာကို သုံးသင့်ပါတယ်။ squash ဆိုတာကို အခုနောက်ပိုင်း GitLab မှာ Merge Request (MR) လုပ်တိုင်း checkbox option လေးနဲ့ squash commit လုပ်မှာလားဆိုပြီး မေးတတ်ပါတယ်။

အရှင်းဆုံးက အလွယ်နည်းနဲ့ ပြောပြပါဆိုရင် squash ဆိုတာ commit 3 ခုရှိတာကို တစ်ခုတည်း ဖြစ်အောင် ပေါင်းလိုက်တာပါ။ အများကိန်းကနေ အနည်းကိန်းရအောင် လုပ်တဲ့ပုံစံပါပဲ။ နောက်ပြီး squash မှာ ဘယ်ဟာကို အရင်ယူရမယ်ဆိုတဲ့ စံမရှိပါဘူး။ နောက်ဆုံးထွက်လာတဲ့ result သည် commit အားလုံးမှာ လုပ်ထားသမျှ changes အားလုံး တစ်စုတစ်စည်းတည်းဖြစ်ပြီး commit တစ်ခုတည်းအနေနဲ့ ပေါင်းသွားတာပါပဲ။ အခြေခံသဘောတရားက အများကနေ အနည်းလုပ်လို့ရတယ်ဆိုပေမဲ့ commit တိုင်းကိုတော့ squash လုပ်ဖို့မသင့်ပါဘူး။ ကျွန်တော်အပေါ်မှာပြခဲ့တဲ့ တစ်လိုင်းတည်းကို squash လုပ်လိုက်ရင် commit တစ်ခုတည်းမှာ အားလုံးပေါင်းသွားပြီး၊ နောက်ပိုင်း ပြဿနာရှိလို့ ခွဲထုတ်ချင်ရင် မရတော့ပါဘူး။ ဒါကြောင့် squash လုပ်မယ်ဆိုရင် bug fix တစ်ခုတည်း (သို့) feature တစ်ခုတည်းမှာ multiple commit တွေကလွဲ၍ bug မတူတဲ့ feature မတူတဲ့ commit ဆို squash လုံးဝမလုပ်သင့်ပါဘူး။

နားမလည်သေးဘူးထင်ရင် ဒီလို ဥပမာကို ဖတ်ကြည့်ပါ။ branch ၂ ခုရှိနေပါမယ်။ တစ်ခုက password က plain text ဖြစ်နေတာကို fix လုပ်တဲ့ issue။ နောက်တစ်ခုက users တွေကို password reset လုပ်လို့ရအောင် ခွင့်ပြုတဲ့ feature branch ပါ။ သူတို့နှစ်ခုက ရည်ရွယ်ချက်လည်း မတူသလို logic တွေကလည်း ဆက်စပ်မှု မရှိ၊ ရေးရမယ့် controller တွေကလည်း တစ်ခုစီ။ ဒါကို squash လုပ်လိုက်လို့ management က “Hey we’re going to release password issue by today” ဆို သေပြီလေ။ commit ၂ ခုကို တစ်ခုတည်း ပေါင်းလိုက်တဲ့ အပြစ်။ ဒါဆို ၂ ခုကို ပေါင်းလို့ရတယ်ဆို၊ အခုကျတော့ရော။ ကျွန်တော်ပြောတဲ့ ပေါင်းတယ်ဆိုတာက branch တစ်ခုတည်းမှာပဲ commit တွေက တစ်ခုထက်ပိုပြီး ရှိနေတာ။ (ဥပမာ၊ အောက်က branch တစ်ခုစီမှာ commit ၁၀ ကြိမ်လုပ်ထားမိတယ်၊ ၅ခုစီပေါ့။ branch မတူတာကို ပေါင်းခိုင်းတာမဟုတ်ပါဘူး။ တူညီတဲ့ ရည်ရွယ်ချက်ရှိတဲ့ code တွေကို commit အများကြီး လုပ်ထားမိတယ်ဆိုမှ ပေါင်းခိုင်းတာပါ။

```
bugfix/kevin/00001-password-plain-text-in-login-form
feature/kevin/00002-allow-users-to-reset-their-password
```

သုံးရမယ့် command က

```
git rebase -i HEAD~n
```

ဒီ `n` ရဲ့ value က commit ဘယ်နှခုကို squash လုပ်ချင်လည်း ဆိုတာပါ။ ၅ ခုရှိနေရင် 5 လို့ရိုက်ပေါ့။ အဲ့ဒီလို ရိုက်လိုက်တာနဲ့ အောက်ကလို vim editor နဲ့ပေါ်လာပါလိမ့်မယ်။ ကိုယ်က IDE သုံးတဲ့သူဆို ပေါင်းချင်တဲ့ commit တွေကို select မှတ်၊ right click နှိပ်ပြီး `squash commit` ကိုရွေးလိုက်ရင် ရပါပြီ။ (IDE ပေါ်မူတည်ပြီး ပြုလုပ်ပုံ ကွာနိုင်ပါတယ်။ ကျွန်တော်က GitKraken သုံးပါတယ်)

သူ့ရဲ့ editor မှာ default value က `<commit>` ရဲ့ ရှေ့မှာ `pick` ဆိုတာလေးပြနေပါတယ်။ အောက်က `<commit>` 2 ခုကို ပေါင်းချင်တယ်ဆို `pick` အစား `squash` ဒါမှမဟုတ် `s` ဆိုတဲ့ keyword နဲ့ ပြောင်းပေးရပါတယ်။ အောက်မှာလုပ်လို့ရတဲ့ options တွေများစွာရှိနေပါတယ်။ နောက်အပိုင်းတွေမှာပြောပြသွားပါမယ်။ အရေးအကြီးဆုံးက commit တစ်ခုကို မှားဖျက်မိသွားရင် ပျောက်သွားနိုင်ပါတယ်။ Command သုံးမယ်ဆို သတိထားပါ။

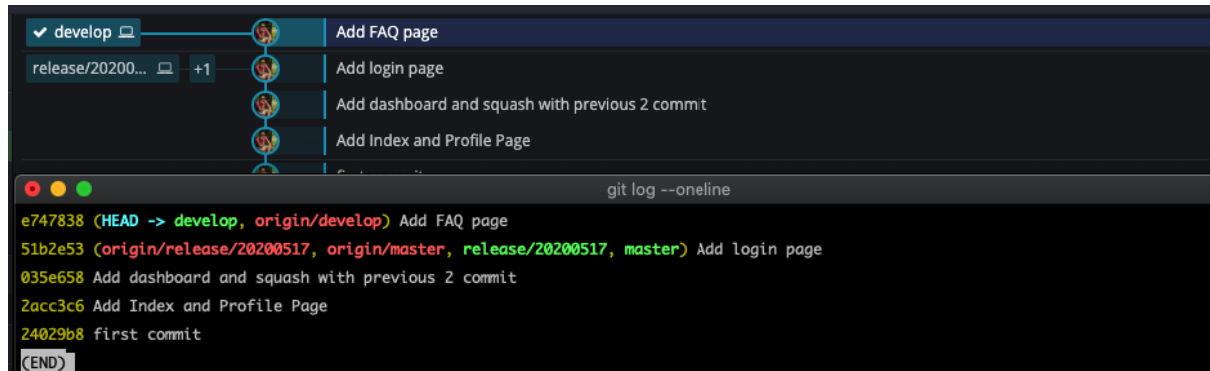
```
git rebase -i HEAD~2
pick 8f312eb Add FAQ page
pick 60d1dbc Add user file

# Rebase 51b2e53..60d1dbc onto 51b2e53 (2 commands)
#
# Commands:
# p, pick <commit> = use commit
# r, reword <commit> = use commit, but edit the commit message
# e, edit <commit> = use commit, but stop for amending
# s, squash <commit> = use commit, but meld into previous commit
# f, fixup <commit> = like "squash", but discard this commit's log message
# x, exec <command> = run command (the rest of the line) using shell
# b, break = stop here (continue rebase later with 'git rebase --continue')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .      create a merge commit using the original merge commit's
# .      message (or the oneline, if no original merge commit was
# .      specified). Use -c <commit> to reword the commit message.
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
```

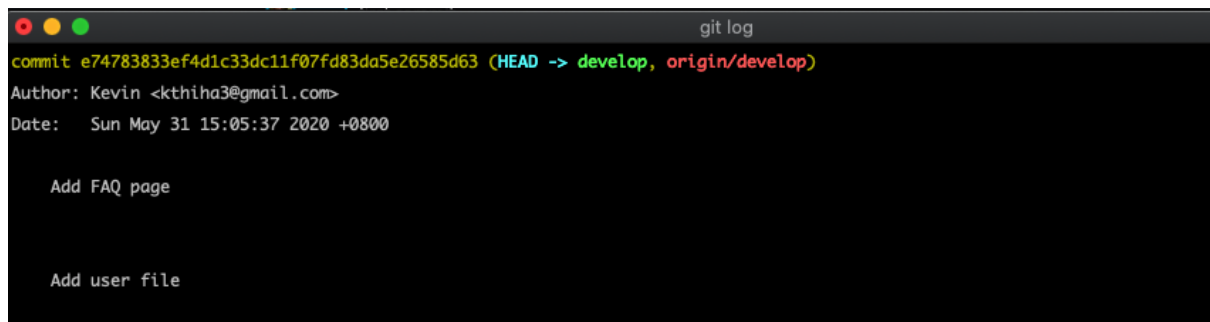
Keyword လည်း ပြောင်းပြီးသွားပြီဆို လက်ရှိ editor မှာပဲ ပြောင်းသွားပြီး commit message ပြောင်းမလား၊ မပြောင်းတော့ဘူးလားဆိုပြီး မေးပါလိမ့်မယ်။ မပြောင်းဘူးဆိုလည်း commit message တွေက တစ်လိုင်းပြီးတစ်လိုင်း အပေါ်ကနေ အောက်အထိ ပြနေပြီး၊ ပြောင်းမယ်ဆို message အသစ်ပြန်ရိုက်၊ ပြီးတာနဲ့ save လုပ်ပြီးသွားပြီးဆို ကိုယ့်ရဲ့ index ထဲမှာ squash လုပ်ပြီးသွားပါပြီ။ ကျွန်တော် squash လုပ်ပြီးသားပုံစံကို

ကြည့်ကြည့်ပါ။ ပြီးတာနဲ့ ကျွန်တော်တို့က remote origin ကိုလည်း update လုပ်ပေးရပါတယ်။ ဒီနေရာမှာ update လုပ်တာက GitHub or GitLab မှာရှိနေတဲ့ histories နဲ့ ကျွန်တော်တို့ local မှာရှိတဲ့ histories ကမတူတော့တဲ့အတွက် force push လုပ်ပေးမှ ရပါတယ်။ Command အပြည့်အစုံက

```
git push origin --force <branch_name>
```



ခုနက <commit> ၂ ခုနေရာမှာ <commit> နောက်တစ်ခုရောက်သွားပါပြီ။ message ကို တစ်ချက်ကြည့် ရအောင်။



Commit message ၂ ခုကနေ တစ်ခုတည်း ပေါင်းသွားတာကို သတိထားမိပါလိမ့်မယ်။ သတိထားရမှာ တစ်ခုရှိပါတယ်။ commit သုံးခုရှိလို့ ပေါင်းချင်ရင် အပေါ်ဆုံး commit နှစ်ခုက squash လုပ်ရင် သုံးခုမြောက် commit မှာ ပေါင်းသွားပါတယ်။ ဒါကြောင့် သုံးခုကို squash လုပ်မယ်ဆို ထိပ်ဆုံးနှစ်ခုမှာ squash option ဆိုတာနဲ့ သူ့အောက် commit မှာလာပေါင်းပါတယ်။ သုံးခုစလုံး squash keyword လုပ်စရာမလိုပါဘူး။ သတိထားသင့်ပါတယ်။

ဒီလောက်ဆို squash အကြောင်းနားလည်မယ်လို့ ယူဆပါပြီ။ ကျွန်တော်တို့ cherry-pick အကြောင်း ဆက်ရအောင်။

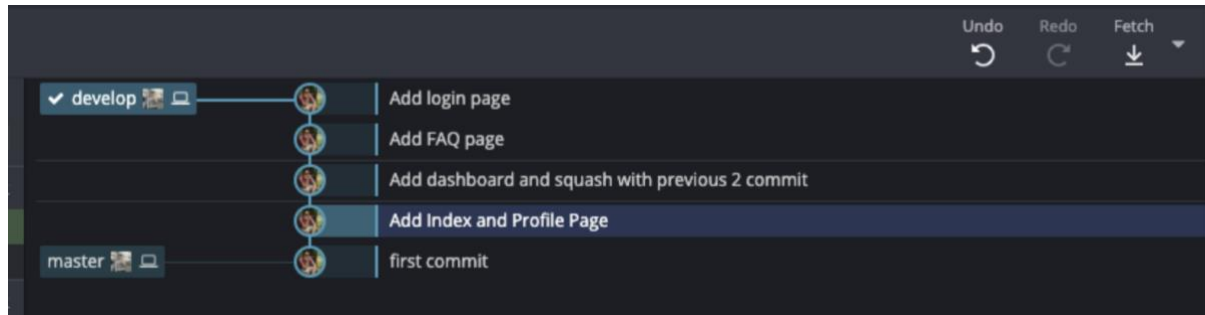
အခန်း (၆) cherry-pick ဆိုတာဘာလဲ

Cherry သီးခူးတာလို ကျွန်တော့် blog လေးမှာ တင်ထားပေးဖူးပါတယ်။ တကယ်လည်း ဆင်တူပါတယ်။ လိုချင်တဲ့ အပိုင်းလေးပဲ ယူပြီး မိမိလုပ်လိုတဲ့ action ကိုလုပ်ဆောင်တာပါ။ လုပ်လိုတဲ့ action က release လုပ်ချင်တာ၊ ဒါမှမဟုတ် rebase လုပ်ဖို့ ခက်သွားလို့ cherry-pick လုပ်တာ စတဲ့ရည်ရွယ်ချက် မျိုးစုံရှိနိုင်ပါတယ်။

ကိုယ့်ရဲ့လက်ရှိ branch မှာ commit သုံးခု ရှိနေပြီးတော့ develop ကိုလည်း merge လုပ်ပြီးသွားပြီဆိုပါစို့။ develop မှာ ကိုယ့်ပေါင်းထည့်လိုက်တဲ့ feature တွေအပြင်၊ တစ်ခြား developers တွေရေးထားတဲ့ feature တွေ၊ bug fix တွေလည်းရှိနေနိုင်ပါတယ်။ ဒီအချိန်မှာ QA team ကနေ စစ်ပြီး၊ ကိုယ့်ရဲ့ feature တစ်ခုပဲ pass ဖြစ်မယ်၊ ကျန်တာတွေ အကုန် failed ဖြစ်ပြီဆို၊ သူတို့တွေရဲ့ failed ဖြစ်သွားတဲ့ code တွေ fixed လုပ်ပြီးချိန်အထိ စောင့်မလား၊ ကိုယ် feature ကို အရင် release လုပ်မလား ဆိုတဲ့ မေးခွန်းပေါ်လာပါပြီ။ ကဲ release လုပ်မယ်ဆိုရင် အဲ့ဒီ commit သုံးခုကို release branch ထဲ ပေါင်းထည့်ရပါတော့မယ်။ ဒီလို scenario ဆို cherry-pick ကအသုံးဝင်ပါပြီ။

cherry-pick လုပ်မယ်ဆိုရင် ဘယ်တော့မှ အပေါ်ဆုံး commit ကနေမလုပ်ရပါဘူး။ လုပ်လိုက်တာနဲ့ commit လုပ်ထားတဲ့ order တွေပြောင်းတဲ့အပြင် ကိုယ့် code တွေပါ အခန့်မသင့်ရင် ကျန်ခဲ့နိုင်ပါတယ်။ ဒါကြောင့် ကျွန်တော် ရှေ့ပိုင်းမှာ squash ဆိုတဲ့ command အကြောင်းပြောခဲ့တာပါ။ တစ်ခုပဲလုပ်ရရင် အမှားပါနိုင်ချေနည်းပြီးတော့ commit တစ်ခုထက်ပိုသွားရင် အမှားပါနိုင်ချေများပါတယ်။ အခုတော့ squash ကိုခဏမေ့ပြီး cherry-pick လုပ်ဆောင်ပုံ ဆက်ကြည့်ရအောင်။

ကိုယ့်မှာ commit သုံးခုဆိုရင် first commit, ပြီးရင် second commit, နောက်ဆုံးမှ third commit အစဉ်လိုက် လုပ်သွားရပါမယ်။ အောက်က ဥပမာလေးကို ကြည့်ကြည့်ရအောင်။ ဒီလို order အစဉ်လိုက်က ကျွန်တော် develop branch တစ်ခုထဲသုံးပြီး လုပ်ထားတာမဟုတ်ပါ။ feature 3 ကို ကျွန်တော့် ရှေ့ပိုင်းတွေမှာ ပြောခဲ့သလို သေချာမှန်ကန်တဲ့ ပုံစံတစ်ခုကို ချထားပြီးမှ ရလာတဲ့ ပုံစံဆိုတာ သိထားစေလိုပါတယ်။



ကျွန်တော်က အခု release branch အသစ်တစ်ခုယူပြီး Add FAQ Page ဆိုတဲ့ commit ကလွဲပြီးကျန်တဲ့ commit တွေကို release branch ထဲ ထည့်ချင်ပါတယ်။ cherry-pick လုပ်ကြည့်ပါမယ်။ command အပြည့်အစုံက

```
git cherry-pick <commit> or <branch_name>
```

ကျွန်တော်ကတော့ branch 2 ခုပဲကျန်တော့တဲ့အတွက် <commit> နဲ့ပဲ cherry-pick လုပ်ပါမယ်။ တစ်ခုခု conflict ဖြစ်သွားလို့ မသေချာရင် abort လုပ်လို့ရပါတယ်။ Abort လုပ်တဲ့ command က

```
git cherry-pick --abort
```

ကျွန်တော်အခု master branch ကနေပြီး release branch တစ်ခုယူလိုက်ပါမယ်။

```
git checkout master
git checkout -b release/20200517
git cherry-pick <71886c6d> #first commit
git cherry-pick <37cb26d2> #second commit
git cherry-pick <35be910e> #third commit
git push origin release/20200517
```

ဒါဆို ကျွန်တော်တို့ cherry-pick ကပြီးသွားပါပြီ။ နောက်ဆုံး commit သည် third commit ဖြစ်ပြီး နောက်ဆုံးမှ လုပ်ပါတယ် (လိုဝ် order မှားလို့မရပါ) ။ ပြီးသွားတာနဲ့ ကျွန်တော်တို့က GitHub ကဆို Pull Request (PR) နဲ့ GitLab ကဆို Merge Request (MR) လုပ်ပြီး master နဲ့ ပေါင်းပါမယ်။ ဒါဆို ကျွန်တော်တို့ release လုပ်လို့ရပါပြီ။ remote origin မှာတော့ master branch က update ဖြစ်သွားပါပြီ။ ဒါပေမဲ့ ကျွန်တော်တို့ local ထဲက master ကတော့ sync မဖြစ်သေးပါဘူး။ ဒါကြောင့် sync ဖြစ်အောင် ကျွန်တော်ပြောခဲ့ဖူးတဲ့ command နှစ်ခု ပြန် run ပေးဖို့ လိုပါသေးတယ်။

```
git fetch origin
git pull --rebase
```

ဒါဆို local ထဲက master နဲ့ရော၊ remote origin က master နဲ့ရော sync ဖြစ်သွားပြီး update ဖြစ်သွားပါပြီ။ ဒါပေမဲ့ ကျွန်တော်တို့ရဲ့ flow အရ master branch က develop branch ရဲ့ရှေ့ကို ရောက်နေပါပြီ။ ပုံမှန်အားဖြင့် develop ဆိုတာက master branch ရဲ့ရှေ့မှာပဲ အမြဲရှိနေသင့်ပါတယ်။ ဒါကြောင့် rebase လုပ်ပေးရပါတယ်။ ဒါမှ master branch ရဲ့နောက်မှာ develop branch မကျန်နေခဲ့ပဲ၊ develop ရဲ့နောက်မှာပဲ master ရှိနေမှာဖြစ်ပါတယ်။ ကဲ cherry-pick ကလည်းလုံလောက်ပြီဆိုတော့ နောက်ဆုံးအရေးကြီးတဲ့ rebase ကိုဆက်ကြည့်ရအောင်။

အခန်း (၇) rebase အသုံးပြုပုံ

အသုံးပြုပုံကို မပြောခင် rebase ဆိုတာဘာလဲ အရင်နားလည်ဖို့ လိုမယ်လို့ယူဆပါတယ်။ ကျွန်တော့်ရဲ့ အဓိပ္ပါယ်ဖွင့်ဆိုချက်ကို ပြောပြပါဆိုရင် ကိုယ့်ရဲ့ လက်ရှိ branch ကို target branch ရဲ့ရှေ့ကိုရောက်အောင် လုပ်ပြီး ကိုယ့်မှာ မရှိသေးတဲ့ files တွေ၊ folder တွေ၊ code update တွေကို ကိုယ့်ရဲ့ လက်ရှိ branch ထဲမှာ update ဖြစ်သွားအောင် လုပ်ပေးတဲ့ command တစ်ခုပါ။

ထင်ရှားတဲ့ ဥပမာကို ရေးပြရမယ်ဆိုရင် ကျွန်တော့်မှာ branch နှစ်ခုရှိနေမယ်ဆိုပါစို့။ ပြီးတော့ သူတို့ နှစ်ခုစလုံးကလည်း develop ဆိုတဲ့ branch ကနေပဲ ခွဲထုတ်ထားတယ်လို့ ယူဆထားပါ။ branch name ကို ကျွန်တော် အောက်ကပုံစံပေးလိုက်ပါမယ်။

```
Git checkout develop
Git checkout -b feature/kevin-story-00003-file-upload-features
//Do some commit
Git checkout develop && git checkout -b feature/kevin-story-00004-add-check-in-api
//do some commit
```

ဒီလိုပြီးသွားပြီဆို ကျွန်တော့်ရဲ့ လက်ရှိ branch က feature/kevin-story-00004-add-check-in-api ဖြစ်ပြီးတော့ ဒီ feature ကို ၁ ပတ်လောက် ရေးလိုက်ရတယ်လို့ သတ်မှတ်ထားပါ။ အချိန် ၁ ပတ်အတောအတွင်း ကိုယ့်ရဲ့ရှေ့မှာ release တစ်ခုလုပ်ပြီးသွားပြီး၊ ပြီးတော့ တစ်ခြား developer တွေရဲ့ code အသစ်တွေ develop ထဲမှာ အသစ်ထပ်ရောက်နေပြီ။ ဒီအချိန်မှာ ကျွန်တော့်ရဲ့ လက်ရှိ develop branch ကတော်တော် အိုပြီး update ဖြစ်မနေတော့ပါဘူး။ ကျွန်တော့်ရဲ့ လက်ရှိ branch ကို develop ရဲ့ရှေ့ရောက်အောင် လုပ်ချင်ပါတယ်။ နှစ်နည်းလုပ်လို့ရပါတယ်။ တစ်နည်းက rebase ဆိုတာကို ဘယ်လိုလုပ်ရမှန်းမသိတဲ့ လူတွေ လုပ်လေ့ရှိတာဖြစ်ပြီး၊ နောက်တစ်နည်းကတော့ အခု topic ရဲ့ rebase ပါ။ နောက်ဆုံးနည်းကို အရင်လုပ်ကြည့်ရအောင်။ ကိုယ်က လက်ရှိ feature/kevin-story-00004-add-check-in-api မှာ။ ဒါဆို command က

```
git rebase <branch_name>
```

နောက်က ကိုယ်ဘယ် branch ရဲ့ရှေ့ကိုရောက်ချင်တာလဲဆိုတဲ့ target branch ပါ။ တစ်ခါတစ်ရံမှာ rebase လုပ်နေရင်းနဲ့ conflict တွေရှိတတ်ပါတယ်။ ဒီလိုအချိန်ကျရင် conflict တွေကို ဖြေရှင်းပေးရပါတယ်။ အဲ့ဒီလို ဖြေရှင်းပြီးသွားရင် နောက်ထပ် command တစ်ခု ထပ် run ရပါတယ်။ ထပ် run ရမယ့် command က

```
git rebase --continue
```

ဆိုတာပါ။ ဒါဆိုကိုယ့်ရဲ့ rebase process ကအောင်မြင်စွာနဲ့ ပြီးဆုံးသွားပါပြီ။ ဒါပေမဲ့တစ်ခါတစ်ခါ ကိုယ့်က rebase process ကြီးတစ်ခုလုံးက မသေချာတော့ဘူးဆိုရင်တော့ abort လုပ်လို့ရပါတယ်။ command က

```
git rebase --abort
```

ဒီလို run လိုက်ပြီဆို rebase တစ်ခုလုံး cancel ဖြစ်ပြီး ဘာမှ ပြောင်းလဲမှုမရှိတော့ပါဘူး။ မှတ်ထားရမှာက ကျွန်တော်တို့ interactive mode နဲ့ rebase လုပ်မယ်ဆိုလည်း ရပါသေးတယ်။ ဘာလို့ interactive mode ကိုရွေးကြလည်းဆိုရင် သူက process တစ်ခုလုံးကို ရှင်းရှင်းလင်းလင်း တစ်ခုပြီးမှ နောက်တစ်ခုကို process လုပ်တဲ့အတွက် -i (interactive) ဆိုရင်တော့ commit message ကိုပြင်မလား၊ squash လုပ်ချင်သေးလား၊ commit တွေကို drop လုပ်ချင်သေးလား စတာတွေကို လုပ်လို့ရပါသေးတယ်။ ကျွန်တော် ပြီးခဲ့တဲ့အခန်းမှာ ပြောခဲ့တဲ့ squash ကလည်း interactive rebase ကိုသုံးပြီး လုပ်ထားတာပါ။

ကဲ rebase မသုံးပဲ နောက်ထပ်နည်းတစ်နည်းကျန်သေးတယ်လို့ ပြောခဲ့ဖူးပါတယ်။ ဒါဟာ နည်းလမ်းဆိုပေမဲ့ လုပ်လို့ရတယ်လို့ပဲ ပြောတာပါ။ ကျွန်တော် rebase ကို သေချာမသိခင်က လုပ်ခဲ့ဖူးတဲ့နည်းပါ။ နောက်ဆုံး end result ကတော့ တူပေမယ့် တစ်လိုင်းတည်းဖြစ်တော့မှာ မဟုတ်တဲ့ ပြဿနာရှိပါတယ်။ လုပ်တဲ့နည်းက ဒီလိုပါ။ ပထမဆုံး develop branch ကနေ branch အသစ်ထပ်ယူ၊ ပြီးမှ အသစ်ယူတဲ့ branch နဲ့ ဟောင်းနေတဲ့ branch နဲ့ merge ပြန်လုပ်တဲ့နည်းပါ။ လုပ်လို့ရတဲ့ နည်းပေမယ့် ကျွန်တော်တော့ မသုံးတော့ပါ။

အခု rebase process ကတော့ ကျွန်တော့်တို့ရဲ့ local စက်ထဲက branch မှာတော့ ပြီးသွားပေမယ့် remote origin မှာရှိတဲ့ သက်ဆိုင်ရာ branch မှာတော့ sync မဖြစ်သေးပါဘူး။ ဒါကြောင့် ကျွန်တော်တို့ sync ဖြစ်အောင် push လုပ်ပေးရပါတယ်။ ရိုးရိုး push နဲ့ မရပါဘူး။ force push လုပ်ပေးရပါတယ်။

```
git push origin --force <branch_name>
```

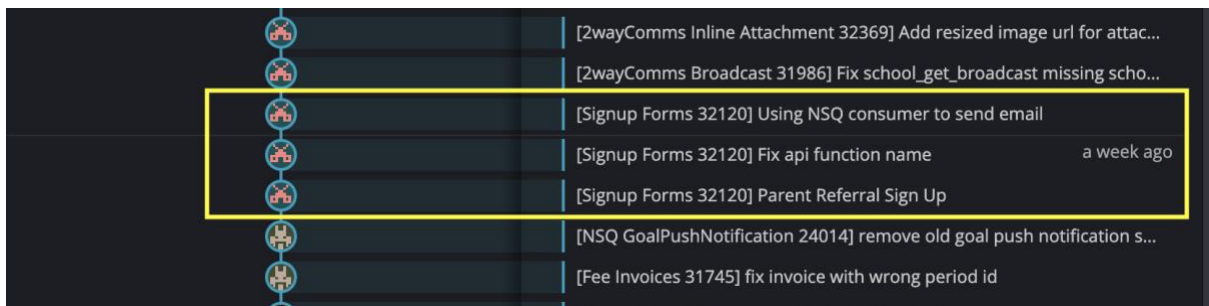
အခုလောက်ဆို rebase အကြောင်း ပုံတွေ လုံးဝမပါပေမယ့် နားလည်လောက်ပြီလို့ ယူဆပါတယ်။ ဒါကြောင့် နောက်ထပ် commit တွေကို ဘယ်လို drop လုပ်လည်းဆိုတဲ့ အကြောင်း နောက်တစ်ခန်းမှာ ဆက်ကြည့်ကြရအောင်။

အခန်း (၈) မလိုချင်တဲ့ commit တွေကို ဘယ်လို drop လုပ်မလဲ နှင့် commit တွေကို ပြန်စီမံမယ်

Drop commit အကြောင်း မစဉ်းစားခင် commit တွေကို စိစစ်ရာလိုလား ဆိုတဲ့ မေးခွန်း စာဖတ်သူတို့ စဉ်းစားမိကောင်း စဉ်းစားမိနိုင်ပါတယ်။ ပုံမှန်အချိန်ဆိုရင်တော့ စီတယ်ဆိုတာ မလိုအပ်ပေမယ့် တူညီတဲ့ feature တစ်ခုတည်းကို commit နှစ်ခု ရှေ့ဆင့်နောက်ဆင့် ရှိမနေဘူး။ ကိုယ်က တူနေတဲ့အတွက် squash လုပ်ချင်တယ်ဆို ရှေ့ဆင့်နောက်ဆင့် commit ဖြစ်မှသာ လုပ်၍ အဆင်ပြေတဲ့အတွက် ကျွန်တော်တို့က စီရတာဖြစ်ပါတယ်။ အောက်က ပုံလေးကို ကြည့်ကြည့်ရအောင်။

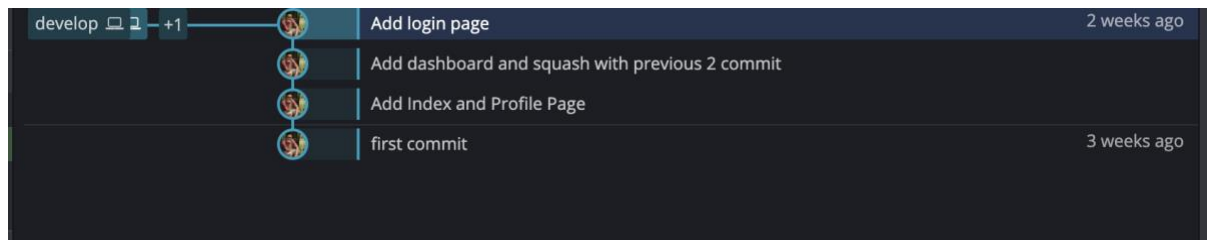


ဒီ commit သုံးခုဟာ feature တစ်ခုတည်းအတွက် commit လုပ်ထားတာပါ။ ဘာကြောင့် ဒီလိုဖြစ်နေလဲဆိုတော့ တစ်ခါတစ်ရံမှာ PR Review လုပ်တဲ့သူက code ကို သေချာမစစ်ပဲ merge လုပ်လိုက်မယ်။ ပြီးတော့ တစ်ခြား branch တစ်ခုကိုလည်း merge ထပ်လုပ်မယ်။ ပြီးတော့ QA process မှာ failed ဖြစ်သွားပြီး၊ fixed လုပ်ဖို့အတွက် branch အသစ်ထပ်ယူ၊ fixed လုပ်၊ ပြီးတော့ PR ပြန်လုပ်၊ ပြန် merge လုပ်ပြီးသွားခဲ့ရင် ရှေ့ဆင့်နောက်ဆင့်မဟုတ်တော့ပဲ ကြားထဲမှာ တစ်ခြား commit တွေရှိနေနိုင်ပါတယ်။ တကယ်ဆို တူတဲ့ commit တွေကို squash လုပ်ချင်လို့ ရှေ့ဆင့်နောက်ဆင့်ဖြစ်အောင် စီလိုက်ချင်တာပါ။ စီပြီးသွားရင် squash လုပ်လို့လွယ်သွားပါပြီ။



ရှေ့ပြီးသွားရင် အပေါ်က result လိုဖြစ်သွားပြီး ကျွန်တော်တို့ squash လုပ်လိုရသွားပါပြီ။ ဒါကလည်း မိမိပေါ်မှာသာ မှုတည်ပါတယ်။ ပုံမှန် git ကိုကြောက်တတ်တဲ့သူတွေကတော့ မလုပ်ရဲကြပါဘူး။ တစ်ချို့ကျတော့ လုပ်လိုရမှန်း မသိတာ ဖြစ်နေနိုင်ပါတယ်။ “ရပါတယ်ကွာ code ပါရင် ပြီးရောဆိုရော” ဆိုတဲ့လူတွေ အတွက်ကတော့ ဒါဟာ ဟာသတစ်ခုလို ဖြစ်နေနိုင်ပါတယ်။ ကိုယ့်ရဲ့ code တွေတောင် ကျစ်ကျစ်လစ်လစ် ဖြစ်အောင် ရေးဖို့ကြိုးစားကြသေးရင် git ကိုလည်း ရှင်းလင်းစွာ commit လုပ်ထားနိုင်ရင်တော့ အတိုင်းထက်အလွန်ပေါ့ဗျာ။

ကဲသုံးသွားတဲ့ command ကတော့ အပေါ်မှာ တင်ပြထားတဲ့ interactive rebase နဲ့အတူတူပါပဲ။ အောက်ကပုံလေးကို ကြည့်ပြီး ပြောင်းရအောင်။ Add login Page ဆိုတဲ့ commit ကို add dashboard ဆိုတဲ့ အောက်ကို ပို့ကြည့်မယ်။ command ကတော့ သိတဲ့အတိုင်း `git rebase -i HEAD~n<3>`

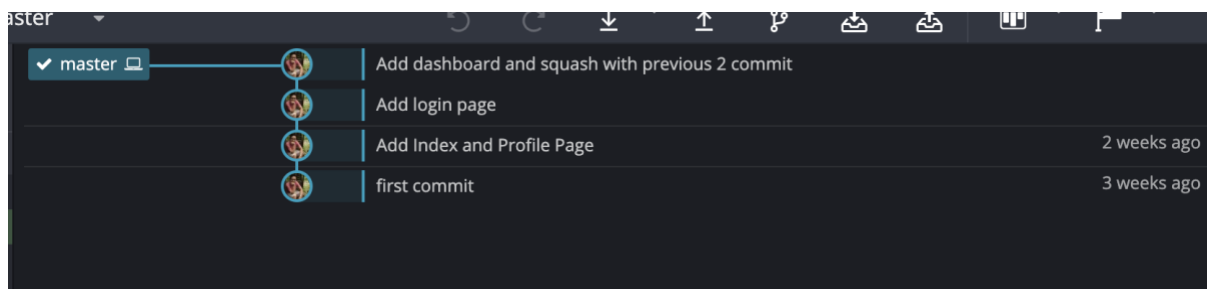


n ရဲ့ value ရှေ့မှာပြောခဲ့ပြီးပါပြီနော်။ commit သုံးခုက ဒီလိုရှိနေပါတယ်။

```
pick 2acc3c6 Add Index and Profile Page
pick 035e658 Add dashboard and squash with previous 2 commit
pick 51b2e53 Add login page
```

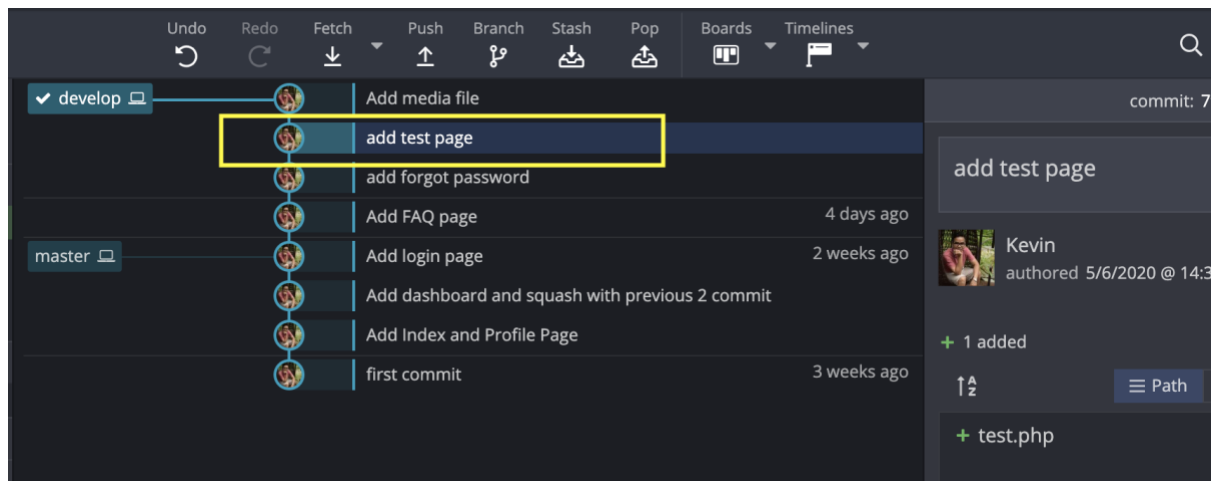
အဲ့ဒါကို ဒီလိုပြောင်းလိုက်ပါမယ်။

```
pick 2acc3c6 Add Index and Profile Page
pick 51b2e53 Add login page
pick 035e658 Add dashboard and squash with previous 2 commit
```



ပြီးသွားတာနဲ့ force push နဲ့ update လုပ်လိုက်ပါမယ်။ ကျွန်တော်အခု ပြင်လိုက်တာ master branch မှာ အရင် စမ်းပြလိုက်တာဖြစ်ပါတယ်။ တကယ်တော့ master မှာ ဒါမျိုးမလုပ်ပါဘူးနော်။ ကျွန်တော်က ဘာမှအရေးမကြီးလို့ စမ်းပြရုံသက်သက်ပါနော်။ force push command ကတော့ အပေါ်မှာ ပြောပြခဲ့ဖူးတဲ့ command နဲ့အတူတူပါပဲ။ နောက်ထပ်အရေးအကြီးဆုံး တစ်ချက်က စီတုံးနေရာမှာလည်း sorting order မှားခဲ့မယ်ဆိုရင် conflict ဖြစ်နိုင်တယ်ဆိုတာ မှတ်ထားသင့်ပါတယ်။ ဥပမာ login.php မှာ ပထမပြင်တယ်။ ပြီးတော့ commit လုပ်တယ်။ နောက်ထပ် commit တစ်ခုကလည်း login.php မှာပဲ ထပ်ပြင်တယ်။ sort လုပ်လိုက်လို့ ဒုတိယ အကြိမ် commit ကို အောက်မှာထားပြီး၊ ပထမ commit ကို အပေါ်တင်လိုက်ရင်တော့ သေချာတာ conflict ဖြစ်ပါပြီ။ သတိထားပါ။

စီပြီးသွားပြီ ဆိုတော့ drop လုပ်ကြည့်ရအောင်။ ဒီတစ်ခေါက် develop branch မှာလုပ်ကြည့်ရအောင်။ drop လုပ်တဲ့ command သည်လည်း interactive rebase command ပုံစံပါပဲ။ သတိထားရမှာ တစ်ခုပဲဖြစ်ပါတယ်။ drop လုပ်လိုက်ရင် ကိုယ့်ရဲ့လက်ရှိ branch ထဲမှာ မရှိနိုင်တော့ပါဘူး။ ကိုယ့်က သက်ဆိုင်ရာ feature branch ကို မဖျက်ပြစ်ပဲထားမယ်ဆိုရင်တော့ backup ရနိုင်ပေမယ့် မထားဘူးဆိုရင် backup မရှိနိုင်တော့ပါဘူး။ ဒါကြောင့် drop မလုပ်ခင် သေချာစဉ်းစားဖို့ လိုပြီး လိုအပ်မှသာ drop လုပ်သင့်ပါတယ်။ ကျွန်တော်တို့ အဝါရောင်ပြထားတဲ့ commit ကို drop လုပ်ကြည့်ရအောင်။



ကျန် `git log --oneline` command ကို run လိုက်ရင် master အထက်မှာ commit လေးခုရှိနေကြောင်း အောက်မှာပြထားတဲ့ ပုံလိုပြပါတယ်။ ဒါဆိုကျွန်တော်တို့ လေးခုကို interactive rebase လုပ်ရမယ်ဆိုတာ သိသွားပါပြီ။ ဒါကြောင့် command က

```
git rebase -i HEAD~4
```

လိုရိုက်ပြီးတာနဲ့ ဖျက်ချင်တဲ့ (drop လုပ်ချင်တဲ့) commit ကို vim editor ကနေတစ်ဆင့် ဖျက်ထုတ်လိုက်ပါမယ်။

```
→ git_sheet_cheat git:(develop) git log --oneline

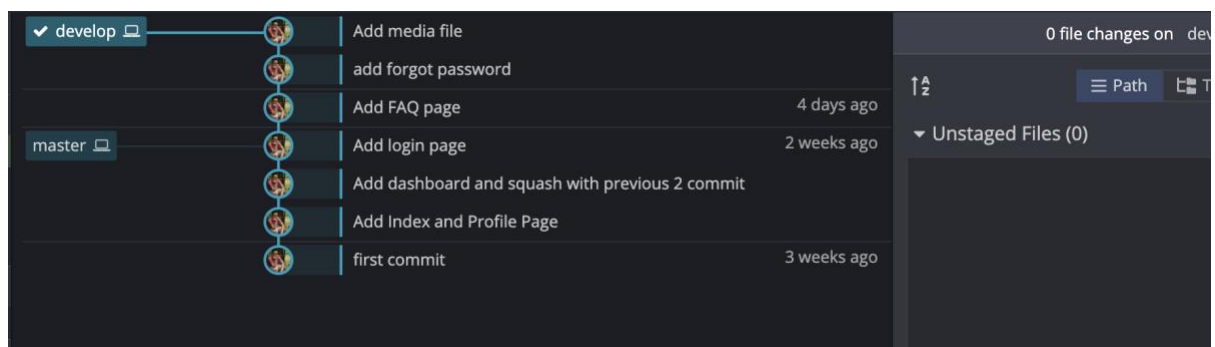
→ git_sheet_cheat git:(develop)
964124c (HEAD -> develop, origin/develop) Add media file
7fef437 add test page
efb771f add forgot password
e747838 Add FAQ page
51b2e53 (origin/release/20200517, origin/master, release/20200517, master) Add login page
035e658 Add dashboard and squash with previous 2 commit
2acc3c6 Add Index and Profile Page
24029b8 first commit
(END)
```

```
pick e747838 Add FAQ page
pick efb771f add forgot password
pick 7fef437 add test page
pick 964124c Add media file
```

ကနေ အောက်က ပုံစံပြောင်းပြီး သိမ်းလိုက်တာနဲ့ drop commit process ကပြီးသွားပါပြီ။

```
pick e747838 Add FAQ page
pick efb771f add forgot password
pick 964124c Add media file
```

ရလာတဲ့ result ကအောက်ကပုံစံဖြစ်ပါတယ်။ ပြီးသွားတာနဲ့ remote origin မှာ sync ဖြစ်ဖို့အတွက် force push command ကိုထိုးစံအတိုင်း run ပေးရပါမယ်။ ဒါဆိုကျွန်တော်တို့ ရဲ့ drop commit လုပ်တာ အောင်မြင်စွာ ပြီးဆုံးသွားပါပြီ။



သတိထားစရာ တစ်ခုကျန်ပါသေးတယ်။ commit တွေကို drop လုပ်ရင် အပေါ်ဆုံးကနေ အောက်ကိုသွားတဲ့ ပုံစံကို သုံးပေးရပါမယ်။ သူက cherry-pick နဲ့ ပြောင်းပြန်ပါ။ ဒါကြောင့် recap အနေနဲ့ cherry-pick ဆို

အောက်ကနေ အပေါ် drop လုပ်ရင် အပေါ်ကနေ အောက်ဆိုတာ မှတ်ထားပေးပါ။ ကျွန်တော်တို့ နောက်ထပ် အသုံးဝင်မယ့် interactive rebase လုပ်ရင် တွေ့ရတဲ့ reword ဆိုတာရင် edit ဆိုတာရယ်ကို ဆက်ပြီးကြည့်ရအောင်။ သူတို့ နှစ်ခုက ခပ်ဆင်ဆင်ဆိုပေမဲ့ တကယ်တော့ ကွာပါတယ်။ reword ဆိုတာက commit message ကိုပဲ ပြောင်းလဲတာဖြစ်ပြီးတော့ သူ့ရဲ့ contents တွေကို လုံးဝ ပြောင်းလဲလို့ မရပါဘူး။ ဒါပေမဲ့ edit ဆိုတဲ့ option ကတော့ commit message ရော၊ commit contents ရောကို ပြောင်းလဲခွင့်ရရှိပါတယ်။ ကျွန်တော်တို့ စမ်းကြည့်ရအောင်။ command က git rebase -i HEAD~n

```
pick b5d3a31 add forgot password feature
pick 1b2ce1d Add media file
```

ကနေပြီးတော့ အောက်က reword keyword သုံးပြီး commit ကိုပြောင်းလဲလိုက်ပါမယ်။ ပြောင်းချင်တဲ့ message က တိုက်ရိုက် ရိုက်လို့မရသေးပါဘူး။ နောက်တစ်ဆင့်မှာမှ ပြောင်းလဲချင်တဲ့ message ကို vim editor သုံးပြီး ရိုက်ထည့်ပေးရမှာ ဖြစ်ပါတယ်။

```
reword b5d3a31 add forgot password feature
pick 1b2ce1d Add media file
```

ရိုက်ပြီးသွားပြီဆို vim ကနေ သိမ်းပြီးထွက်လိုက်တာနဲ့ ကျွန်တော်တို့ရဲ့ reword process ကပြီးဆုံးသွားပါပြီ။ git push --amend နဲ့မတူတဲ့အချက်က reword သည် remote origin မှာ push လုပ်ပြီးသား အချိန်မှာ သုံးလို့ရပါတယ်။ ဒါပေမဲ့ push amend ဆိုရင်တော့ remote origin ကို push မလုပ်ရသေးခင် အချိန်မှာ သုံးသင့်တဲ့ command ဆိုတာ ခွဲခြားသိမြင်သင့်ပါတယ်။

edit process ကတော့ edit လို့ ရိုက်ထည့်ပြီး vim မှာသိမ်းပြီးထွက်လိုက်ရင် နောက် process မှာ commit message ပြင်မှာလား၊ rebase continue လုပ်မှာလားဆိုပြီး မေးပါလိမ့်မယ်။ အောက်က ပုံကို ကြည့်ရအောင်။

You can amend the commit now, with

```
git commit --amend
```

Once you are satisfied with your changes, run

```
git rebase --continue
```

ကိုယ်က message ပဲပြင်ချင်တယ်ဆို git commit --amend လို့ရိုက်လိုက်ရင် reword process အတိုင်းဆက်လုပ်သွားရုံပါပဲ။ code တွေပြင်ချင် ဖျက်ချင်တာတွေလုပ်ချင်ရင်တော့ ကိုယ့်ရဲ့သက်ဆိုင်ရာ IDE

သုံးပြီးပြင်ဆင်လိုရပါပြီ။ ပြင်ပြီးတာနဲ့ `git add` တွေ့တာတွေ မလိုတော့ပါဘူး။ `git rebase --continue` ဆိုတာက ရိုက်ထည့်လိုက်တာနဲ့ ကျွန်တော်တို့ရဲ့ `edit process` က ပြီးဆုံးသွားပါပြီ။ နောက်ဆုံး `command` ကတော့ မပါမဖြစ် `force push` ပဲဖြစ်ပါတယ်။ ဒီလောက်ဆို `reword` နဲ့ `edit` ရဲ့ခြားနားချက်ကို နားလည်ပြီလို့ ယူဆပါတယ်။ ကျန်တဲ့ `option` တွေက အသုံးနည်းတာရယ်၊ ကျွန်တော်လည်း သေသေချာချာ မသုံးမိသေးတာရယ် ကြောင့် ကျွန်တော်ထည့်မရေးပေးတော့ပါဘူး။ အကယ်၍ စမ်းချင်တယ်ဆိုရင်တော့ စမ်းပြီး ကျွန်တော်နဲ့ လာတိုင်ပင်လိုရပါတယ် ခင်ဗျာ။ ကဲ ဒီလောက်ဆို `squash` ရယ်၊ `cherry-pick` ရယ်၊ `rebase` ရယ်သုံးပြီး ဖြောင့်တန်းတဲ့ လိုင်းတွေရအောင် စီစဉ်နိုင်မယ်လို့ မျှော်လင့်ပါတယ်။ ဒါတွေဟာ အခြေခံကျတဲ့ `command` တွေဖြစ်ပေမယ့် အသုံးနဲ့တာရယ်၊ သုံးရတဲ့ `user` ကစိုးရိမ်ထိတ်လန့်တတ်မှုတွေရယ်ကြောင့် `low profile` ဖြစ်နေတဲ့ `command` လေးတွေပေါ့ဗျာ။ `hot` ဖြစ်အောင် လုပ်ပေးဖို့ စာဖတ်သူတို့ တာဝန်ရှိပါတယ်။ ကျွန်တော်တို့ `revert commit` အကြောင်း ဆက်ကြည့်ရအောင်။

အခန်း (၉) git revert အကြောင်း သိကောင်းစရာ

ကျွန်တော့်ကို git revert အကြောင်း အဓိပ္ပါယ်ဖွင့်ဆိုပြပါဆိုရင် ကိုယ် commit လုပ်လိုက်မိတာရဲ့ ဆန့်ကျင်ဘက်ကို ပြောင်းလဲခြင်းလို့ သိအိုရီကျကျ ဖွင့်ဆိုပါတယ်။ ဥပမာ ကိုယ်က file တစ်ဖိုင်အသစ်ထပ်ထည့်မိတယ်။ revert လုပ်လိုက်ရင် အသစ်ထည့်ထားတဲ့ file ပြန်ပြီး remove လုပ်လိုက်သလိုဖြစ်သွားမယ်။ code lines အသစ်တွေတိုးလိုက်မယ်။ Revert လုပ်လိုက်ရင် အသစ်ထည့်ထားတဲ့ code တွေ ပြန်ဖျက်သွားမယ်။ ဒါဟာ revert ရဲ့ပုံစံဖြစ်ပါတယ်။

သုံးရမယ့် command ပုံစံတွေကို တစ်ချက်ကြည့်လိုက်ရအောင်။ CLI မှာ git revert ဆိုပြီးရိုက်လိုက်တာနဲ့ သုံးလိုရတဲ့ option တွေအများကြီး ထွက်လာပါလိမ့်မယ်။ အဲ့ဒီထဲကမှ အသုံးများတာတွေကို ကြည့်ရအောင်။

`git revert -e <commit>` ကတော့ commit message ကိုပြင်မယ်လို့ ပြောတာပါ။ ပုံမှန်အားဖြင့်ကတော့ Revert keyword နဲ့ commit message မှာ Prefix အနေနဲ့ default ပါပြီးသားပါ။ တကယ်လို့ ကိုယ်က မကြိုက်ဘူး၊ ပြင်ချင်တယ်ဆိုရင်တော့ `-e subcommand` ကိုသုံးပြီးပြင်ရပါတယ်။ ထူးခြားမှုက `-e` လို့တစ်ခုတည်း သုံးတာနဲ့ git commit ပါ run သွားပြီး commit နောက်ထပ်တစ်ခုပါသွားပါတယ်။ ကိုယ်က commit တစ်ခုအနေနဲ့ မထွက်သွားချင်ဘူးဆိုရင်တော့ နောက်ထပ် `-n` ဆိုတဲ့ option ထပ်ထည့်ပေးရပါတယ်။ ဒါဆို command အသစ်က `git revert -e -n <commit>` ဖြစ်သွားပြီး commit နောက်တစ်ခုထပ်မတိုးလာပဲ ကိုယ်လုပ်ခဲ့တဲ့ action တွေရဲ့ ဆန့်ကျင်ဘက်ကို ပြောင်းလဲလို့ရပါပြီ။ commit တိုက်ရိုက်မလုပ်တဲ့အတွက် ကိုယ်တိုင်တော့ commit လုပ်ပေးဖို့ လိုအပ်မှာ ဖြစ်ပါတယ်။

နောက်ထပ် option တစ်ခုက `-s` ဆိုတဲ့ option ပါ။ ဒါကိုထည့်လိုက်ရင် ဒီ revert ကိုချိန်းတာ သိပါတယ်ဆိုတဲ့ acknowledge သဘော subcommand တစ်ခုပါလာပါတယ်။ `-n` မှာ မသိသာပေမယ့် `-e -s` ဆိုရင် နောက်ထပ် Sign Off အသစ်တစ်လှိုင်းဝင်လာပြီးတော့ ကိုယ့်ရဲ့ Git account နဲ့ email default ပါဝင်နေမှာဖြစ်ပါတယ်။ command ကိုကြည့်ကြည့်ပါ။ `git revert -e -s <commit>`

```
Revert "remove media"

This reverts commit 3fe03fab0b4196a1e1290a62027baba0721d46ae.

Signed-off-by: Kevin <kthiha3@gmail.com>
```

ဒါဆို Signed-off အနေနဲ့ ဒီလိုပြောင်းတာကို သူသိပါတယ်ဆိုတဲ့ ack တစ်ခုရသွားပါတယ်။ နောက်ထပ် တစ်ခုကတော့ `-m` subcommand အကြောင်းဖြစ်ပါတယ်။ ပုံမှန် `commit` တွေကို `revert` လုပ်တာ ပြဿနာမရှိပေမယ့် `branch` နှစ်ခုကို `merge` လုပ်ပြီးသား `commit` တစ်ခုကို `revert` လုပ်မယ်ဆိုရင်တော့ `-m` က လိုလာပါတယ်။ `-m` ရဲ့သဘောက ဘယ် `parent` ကို ယူမလဲဆိုတာကို ပြောရတာပါ။ ကျွန်တော်တို့ ဥပမာလေး ကြည့်ရအောင်။ အကယ်၍ ကျွန်တော်တို့က `branch` နှစ်ခုကို ပေါင်းလိုက်မိတယ်ဆိုရင်၊ အောက်က ပုံလေးအတိုင်း `git log HEAD~1` ရိုက်လိုက်ရင် မြင်တွေ့ရပါမယ်။ ဘာရိုက်ခဲ့တာလည်း ရှင်းပြပါမယ်။ `git log` ဆိုတာက `log` ကိုပြန်ကြည့်တာဖြစ်ပြီး `HEAD` ဆိုတာ လက်ရှိ `active` ဖြစ်နေတဲ့ `commit` ကို ခေါ်ဆိုပြီး `~n` ဆိုတာကတော့ `HEAD` ကနေ `commit` ဘယ်နှစ်ခုမြောက်ကို `log` လုပ်ချင်တာလည်း ခေါ်ဆိုလိုက်တာပါ။ ကြည့်ချင်တဲ့ ကိန်းပေါ်မူတည်ပြီး သုံးခုမြောက်ဆို 3 လို့ရိုက်လို့ရနိုင်ပါတယ်။ နောက်ဆုံး -1 ကတော့ `limit` လုပ်လိုက်တာပါ။ ကြည့်ချင်တဲ့ `limit` ပေါ်မူတည်ပြီး ပြောင်းလဲခွင့်ရှိပါတယ်။ အောက်မှာမြင်ရတဲ့ ပုံအတိုင်း ကျွန်တော့်မှာ `commit` နှစ်ခုကို ပေါင်းထားလိုက်ပါပြီ။ တကယ်လို့ ဒီလိုအချိန်မှာ `revert` လုပ်မယ်ဆို `mainline` ကိုရွေးပေးရပါတယ်။

```

commit 7ecb6867a307013162755355275c9803e70ac205 (origin/develop, develop)
Merge: b98b757 a52041a
Author: Kev <kthiha3@gmail.com>
Date: Sun Jun 7 12:44:16 2020 +0800

    Merge pull request #7 from kelvinkyaw/kevin/story/K01_add_log

    K01 to develop
(END)

```

```
git revert HEAD~1 --no-edit --mainline 1
```

လို့ရိုက်လိုက်ရင် `revert` လုပ်သွားမယ့် `commit` က `<b98b757>` ဖြစ်သွားပြီးတော့

```
git revert HEAD~1 --no-edit --mainline 2
```

ဆိုရင်တော့ `<a52041a>` ကို `revert` လုပ်သွားမှာဖြစ်ပါတယ်။

`-m` ရဲ့ `command` နောက်မှာ အမြဲတမ်း `args` တစ်ခု မဖြစ်မနေလိုက်ရပါတယ်။ `zero` ထက်ကြီးရတဲ့ `number value` ပါ။ `revert` ရဲ့ `subcommand` တွေထဲမှာ အရှုပ်ဆုံးဆိုတဲ့ `-m` အကြောင်းနားလည်မယ်လို့ယူဆပါတယ်။

အခုပြောခဲ့တာတွေက revert မှာပါတဲ့ subcommand တွေအကြောင်းဖြစ်ပြီး option တွေအကြောင်း ကျန်ပါသေးတယ်။ ရှေ့ပိုင်းက အခန်းတွေကို ပိုင်နိုင်ခဲ့ရင် option တွေကလည်း အေးဆေးဖြစ်သွားမှာပါ။ ဘာ option တွေရှိလည်း ကြည့်ရအောင်။

[quit, continue, abort, skip, cleanup] ဆိုပြီး ငါးခုရှိပါတယ်။ continue နဲ့ abort ကတော့ ရင်းနှီးပြီးသားလို့ ယူဆပါတယ်။ quit ဆိုတဲ့ option ကတော့ လက်ရှိရောက်နေတဲ့ နေရာမှာပဲ ရပ်လိုက်တာကို ဆိုလိုပါတယ်။ တစ်ဝက်တစ်ပျက်ပဲ လုပ်မယ်ဆိုရင် `git revert --quit` ဆိုပြီးရိုက်လိုက်ရုံပါပဲ။ သူက abort နဲ့မတူပါဘူး။ abort က process တစ်ခုလုံး ဘာမှပြောင်းလဲခြင်းမရှိပါဘူး။ `git revert --skip` ကတော့ လက်ရှိ commit ကိုကျော်လိုက်ပြီး နောက်တစ်ခု ဆက်လုပ်ပါမယ်ဆိုတဲ့ သဘောပါ။ နောက်ဆုံးကျန်တဲ့ `git revert --cleanup` ကတော့ commit message တွေမှာ extra space တွေပါနေရင် trim လုပ်တဲ့ သဘောနဲ့တူသလို Program Language တွေမှာပါတဲ့ space တွေကို string replace လုပ်တဲ့ပုံစံနဲ့ ဆင်တူပါတယ်။ နောက်ထပ်အသုံးနည်းတဲ့ option တွေလည်း ရှိပါသေးတယ်။ `git revert` လို့ရိုက်ကြည့်ပြီး စမ်းကြည့်နိုင်ပါတယ်။ ဒီလောက်ဆို `git revert` မှာ လုပ်လို့ရတဲ့ feature တွေနဲ့ `git revert` အကြောင်း သိသွား လောက်ပြီးလို့ ယူဆပါတယ်။ ကျွန်တော်တို့ နောက်ဆုံးခန်း ဖြစ်တဲ့ `git log` နဲ့ tag အကြောင်း ဆက်ကြည့်ရအောင်။

အခန်း (၁၀) git log နှင့် git tagging

ပုံမှန် IDE မသုံးပဲ CLI မှာပဲ သုံးတဲ့သူတွေအတွက်ကတော့ git log က မရှိမဖြစ် အရေးပါလှပါတယ်။ repo တစ်ခုလုံးမှာ ရှိတဲ့ changes တွေ၊ ဘာ action တွေလုပ်သွားလဲ (ကျွန်တော်အပေါ်က တင်ပြသွားတဲ့ command အစုံအထိ ပြန်ကြည့်လို့တောင်ရပါတယ်)၊ လိုင်းတွေဘယ်လို ဖြစ်နေလဲကအစ ကြည့်လို့ရပါတယ်။ အောက်မှာ ဆက်ကြည့်ရအောင်

တစ်လိုင်းတည်းကြည့်ချင်ရင်	<code>git log --oneline</code>
ရှိသမျှ limit မထားပဲ ကြည့်ချင်ရင်	<code>git log</code>
Limit လေးနဲ့ကြည့်ချင်ရင် (n ကတော့ ဘယ်နှစ်ခု limit လုပ်ချင်လည်းဆိုတာပါ)	<code>git log -n</code>
Log နဲ့အတူ ဘာတွေပြောင်းလဲထားလဲ သိချင်ရင်တော့ diff အနေနဲ့ပါ ပြပါတယ်။	<code>git log -p</code>
ပြောင်းထားတဲ့ file တွေပဲသိချင်ရင်တော့	<code>git log --stat</code>
Author အနေနဲ့ ရှာချင်ရင်တော့	<code>git log --author="name"</code>
Commit တွေကို commit message နှင့်အတူ pattern ပေါ်မူတည်ပြီး ရှာချင်ရင်တော့	<code>git log --grep="pattern"</code>
သတ်မှတ်ထားတဲ့ range အတွင်းရှာကြည့်ချင်ရင် (<commit_id>, HEAD, <branch_name> or တစ်ခြား revision reference တွေအကုန်)	<code>git log <since> <until></code>
သက်ဆိုင်ရာ file လေးပေါ်မှာပဲ ကြည့်မယ်ဆိုရင်တော့	<code>git log -- <filename></code>
Graph ပုံစံ လုပ်ထားတဲ့ ပုံစံနဲ့ကြည့်မယ်ဆိုရင်တော့	<code>git log --graph --decorate</code>
ဒီတစ်ခုကတော့ အရမ်းအသုံးဝင်တဲ့ တစ်ခုပါ။ log ပုံစံလို့ အတိအကျ မဆိုသာပေမယ့် ဘာတွေ လုပ်ခဲ့လဲဆိုတဲ့ histories အတိအကျ ပြန်ကြည့်လို့ရတဲ့ command ဖြစ်ပါတယ်။	<code>git reflog</code>

git ရဲ့ ပုံမှန် commit id တွေက ရှည်လျားလွန်းပါတယ်။ ရှည်လွန်းရင်လည်း ကြည့်ရတာ ရှုပ်တော့ short form ပုံစံလည်း သုံးလို့ရပါသေးတယ်။ ဒီတိုင်းသုံးမရလို့ plug-in အတွက် command သက်သက် ထပ်ထည့်ပေးရပါတယ်။ ကျွန်တော် ဒီအကြောင်းကို blog က အပိုင်းသုံးမှာ ဖော်ပြထားပြီး ဖြစ်ပါတယ်။ သုံးတဲ့ command ကတော့ git lol ဆိုတာပါ။ ချက်ချင်းရိုက်မရသေးပါဘူး။ Plug-in လုပ်နည်း အရင် ကြည့်ပါအုံးနော်။

ပုံလေးကို အရင် ကြည့်ကြည့်ပါအုံး။ graph လေးနဲ့ရော short form လေးပါ ပြထားလို့ ကျွန်တော်ကတော့ သဘောကျပါတယ်။

```
* aec88c3 (HEAD -> develop, origin/develop) Kevin/story/k01 add log (#6)
* b98b757 Add ReadMe
| * a52041a (origin/kevin/story/K01_add_log, kevin/story/K01_add_log) Revert "add log"
| * 25ae7ae add log
|/
* fdeb99f add media
* 3fe03fa remove media
* 069cf59 Revert "Revert "Add media file""
* f212d9a Revert "Add media file"
* 8a8b1d7 Add media file
* 3dd9ba8 add forgot password feature
* e747838 Add FAQ page
| * ccbb4dc (origin/feature/kevin/K02_Plain_Text_Password_Stash) WIP password
| | * 0180d37 (origin/feature/kevin/K01_Enhancement_Login_UI) [Login] Add main.js
| | /
| * 8f312eb Add FAQ page
| /
* 51b2e53 (origin/release/20200517, origin/master, release/20200517, master) Add login page
* 035e658 Add dashboard and squash with previous 2 commit
* 2acc3c6 Add Index and Profile Page
* 24029b8 first commit
(END)
```

အရင်ဆုံး ရိုက်ရမယ့် command က

```
git config --global alias.hist "log --pretty=format:'%h %ad | %s%d [%an]' --graph --date=short"
git config --global alias.lol "log --graph --decorate --pretty=oneline --abbrev-commit --all"
git config --global alias.mylog "log --pretty=format:'%h %s [%an]' --graph"
```

ပြီးတာနဲ့ configuration မှန်မမှန် ပြန်စစ်ရပါတယ်။ command က git config --list မှန်သွားပြီဆိုတာနဲ့ အောက်က result မြင်ရပါလိမ့်မယ်။

```
alias.hist=log --pretty=format:"%h %ad | %s%d [%an]" --graph --date=short
alias.lol=log --graph --decorate --pretty=oneline --abbrev-commit --all
alias.mylog=log --pretty=format:'%h %s [%an]' --graph
```

ဒီလို ပေါ်လာပြီဆို ကျွန်တော်တို့ `git lol` ရိုက်ကြည့်လို့ရပါပြီ။ ဒီလောက်ဆို `git log` က ကုန်သလောက်နီးပါးဖြစ်သွားပါပြီ။ ကျွန်တော်တို့ `tagging` ဘက်လှည့်ရအောင်။

ကျွန်တော်အရင် အလုပ်တွေမှာတုန်းကတော့ `release` လုပ်ပြီးတိုင်း `tag` လုပ်ရမယ်လို့ မတွေးခဲ့ဖူးပါဘူး။ တကယ်တော့ `tag` တွဲတာဟာလည်း ကောင်းတဲ့ အလေ့အကျင့်တစ်ခုလို့တောင် ပြောလို့ရပါတယ်။ `release` လုပ်ပြီးတိုင်း `tag` လေးတွဲထားရင် ဘယ်နေ့က ဘာတွေ `release` လုပ်ခဲ့လည်း ဆိုတာ ကြည့်လို့ရမှာ ဖြစ်သလို `version histories` အတွက်လည်း အထောက်အကူပြုမှာ ဖြစ်ပါတယ်။ ကိုယ့်က `git tag` လို့ရိုက်လိုက်တာနဲ့ ကိုယ့်မှာ `tag` လုပ်ထားပြီးသား တွေကို မြင်ရမှာဖြစ်ပါတယ်။ လုပ်ပုံလုပ်နည်းလေးကို ကြည့်ရအောင်။

```
git tag -a <tag_name> -m "<tag message>"
```

`-a` ရဲ့နောက်မှာ သုံးမယ့် `tag name` ဖြစ်ပြီးတော့ `-m` ရဲ့နောက်မှာ ပေးချင်တဲ့ `message` ဖြစ်ပါတယ်။ ကျွန်တော့်ရဲ့ `naming convention` ကတော့ `release` အတွက်ဆို `release/YYYYMMDD` ပါ။ `Hotfix` အတွက်ဆိုရင်တော့ `Hotfix/YYYYMMDD`၊ တစ်ချို့ကျတော့ `version number` သုံးပါတယ်။ ကိုယ်နှစ်သက်ရာကို သုံးလို့ရပါတယ်။ ကိုယ့်က `tag` ပေါ်မှာမူတည်ပြီး ဘာတွေ ပြောင်းလဲမှုရှိလဲဆိုတာသိချင်ရင်တော့

```
git show <tag_name>
```

လို့ရိုက်ထည့်တာနဲ့ `tag` မှာရှိထားတဲ့ `commit` တွေ၊ `diff` တွေ၊ `author`၊ `date`၊ `<commit_id>` စတဲ့ ပြည့်စုံတဲ့ `info` အကုန်တွေရမှာဖြစ်ပါတယ်။ နောက်ထပ်စိတ်ဝင်စားစရာကောင်းတဲ့ `tag type` တစ်မျိုးရှိပါသေးတယ်။ `lightweight tag` လို့ခေါ်ပါတယ်။ ပေါ့ပါးပါးပါး ဖြစ်တဲ့အတွက် ဘာ `option` မှာရိုက်စရာမလိုပဲ

```
git tag <tag>
```

လို့ရိုက်ထည့်ရုံနဲ့ `tag` လုပ်ပေးပါတယ်။ တစ်ခုပါပဲ။ ကိုယ့်အနေနဲ့ `lightweight tag` မှန်း သိအောင်လို့တော့ ခွဲခြားထားသင့်ပါတယ်။ ဥပမာ `tag name` ကို `<tag_lw>` ပေးထားသင့်ပါတယ်။ နှစ်သက်သလို ပေးလို့ရပါတယ်။ အခုတင်ပြခဲ့တာတွေကို `local` မှာပဲ `tag` ပါသေးတယ်။ `remote origin` ကိုထုံးစံအတိုင်း `sync` ဖြစ်ဖို့ `push` လုပ်ပေးရပါမယ်။ `Command` က

```
git push origin <tag_name>
# if you have more than one tags that haven't push yet
git push origin --tags
```

ဆိုပြီးနှစ်ခုရိုက်လို့ရပါတယ်။ တကယ်လို့ tag ကိုဖျက်ချင်တယ်ဆိုရင် -d ဆိုတဲ့အတိုင်း သုံးလို့ရပါတယ်။ -d ကထုံးစံအတိုင်း local မှာဖျက်တာဖြစ်ပြီး remote origin မှာလည်း ဖျက်ပေးဖို့ လိုအပ်တတ်ပါတယ်။
command အပြည့်အစုံက

```
Git tag -d <tag_name>
Git push origin :refs/tags/<tag_name>
```

နောက်တစ်နည်းကတော့ ပိုလွယ်တဲ့ ပုံစံပါ။

```
git push origin --delete <tag_name>
```

နောက်တစ်ခုကတော့ tag ကို rename လုပ်တာပါ။ command အပြည့်အစုံက

```
git tag <new_tag_name> <old_tag_name>
git tag -d <old_tag_name>
git push origin :refs/tags/<old_tag_name>
git push --tags
```

ဟုတ်ကဲ့ ဒါဆို tagging မှာပါတဲ့ အကြောင်း cover ဖြစ်သွားပြီ ဖြစ်တဲ့အတွက် ကျွန်တော့်ရဲ့ git for geeks by Kevin ဆိုတဲ့ စာအုပ်ကို ဒီမှာပဲအဆုံးသတ်လိုက်ပါတယ်နော်။

စာအုပ်နဲ့ပက်သက်ပြီး မေးလို့တဲ့ အကြောင်းအရာတွေ၊ feedbacks တွေကို kthiha3@gmail.com သို့ ပေးပို့နိုင်ပါတယ်။ ကျေးဇူးအထူးတင်ရှိပါတယ်။