협업 중심의  GitHub

양호준 6/26

-  자소설닷컴 iOS 개발자

- 앱스쿨 3기 멘토
  (1기 멘토로도 참여)

목차

- Git은 어떻게 생기게 되었을까?
- 간단한 Git 사용법
  - git add / git commit / .gitignore / git restore / git reset / git revert
    git merge / git rebase / git stash
  - 브랜치 전략
- GitHub 활용하기
  - git fetch / git pull
  - Pull Request / 코드리뷰 / 이슈 관리
  - 포트폴리오로 GitHub 사용하기

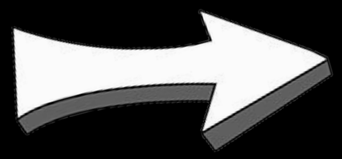
Git은 어떻게 생기게 됐을까?

Git 이란 무엇일까?

버전을 편리하게 관리할 수 있는 도구
(Version Control System 중 하나)

그렇다면 버전은 왜 관리해야 하는 것일까?

파일로 관리하게 되는 경우



```
commit 8ff84ba8e2a5864f567400f5029455d08a0d206d (HEAD -> develop, origin/develop, origin/HEAD)
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Fri Nov 18 08:37:01 2022 +0900

    fix: 리스트에 아무것도 없는 경우 다시 돌아올 때 강제종료되는 문제 해결

commit acf61c348dbeb48b0a6815715859a8c012079774
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Fri Nov 4 21:07:11 2022 +0900

    refactor: 배포도 개발 서버를 사용하도록 수정

commit 71f9d408a3e485fbbf4136682a4c7cae65b79b4a
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Fri Nov 4 20:58:29 2022 +0900

    fix: 왔어요/안왔어요 API 파라미터 수정

    - userID에 내 id가 아닌 해당 셀의 id를 보내도록 수정

commit bca009ea97a0bf4078a7bacd8025609c1350474a
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Thu Nov 3 23:38:40 2022 +0900

    chore: swiftlint 규칙 수정
```

Git을 통해 관리하는 경우

협업을 하다보면...

- 동시에 다른 작업을 병렬적으로 진행해야 하고
- 코드에 문제가 발생하여 이전 코드로 돌아가야 할 수도 있고
- 다른 사람들이 어떻게 코드를 작성했는지 히스토리를 봐야할 수도 있다

<디자이너의 흔한 최종 파일>

0912_아이콘_최종.png
0912_아이콘_최종_수정.png
0913_아이콘_최종_수정_2차.png
0916_아이콘_파이널.png
0916_아이콘_진짜_파이널.png
0917_아이콘_확정.png

0919_아이콘_NEW_시안_5종.jpg

그런데 이렇게 파일로 관리하게 되면 어떻게 될까?

파일로 협업을 하는 경우

- 병렬로 작업을 하고 합치려면 변경 사항들을 전부 파악해서 이를 잘 병합해야 함
- 이전으로는 파일 수정 날짜만 잘 기록되어 있다면 돌아갈 수 있음
  하지만 큰 작업의 경우 이런 파일들이 어마어마하게 많이 생길 것
- 어떤 부분에서 변경 사항이 있었는지 한 눈에 알 수 없음

협업에 아주 불리하며, 규모가 커질수록 이런 점이 더욱 두드러짐!

Git 등장!!

Centralized Version Control

서버를 통해 버전을 관리

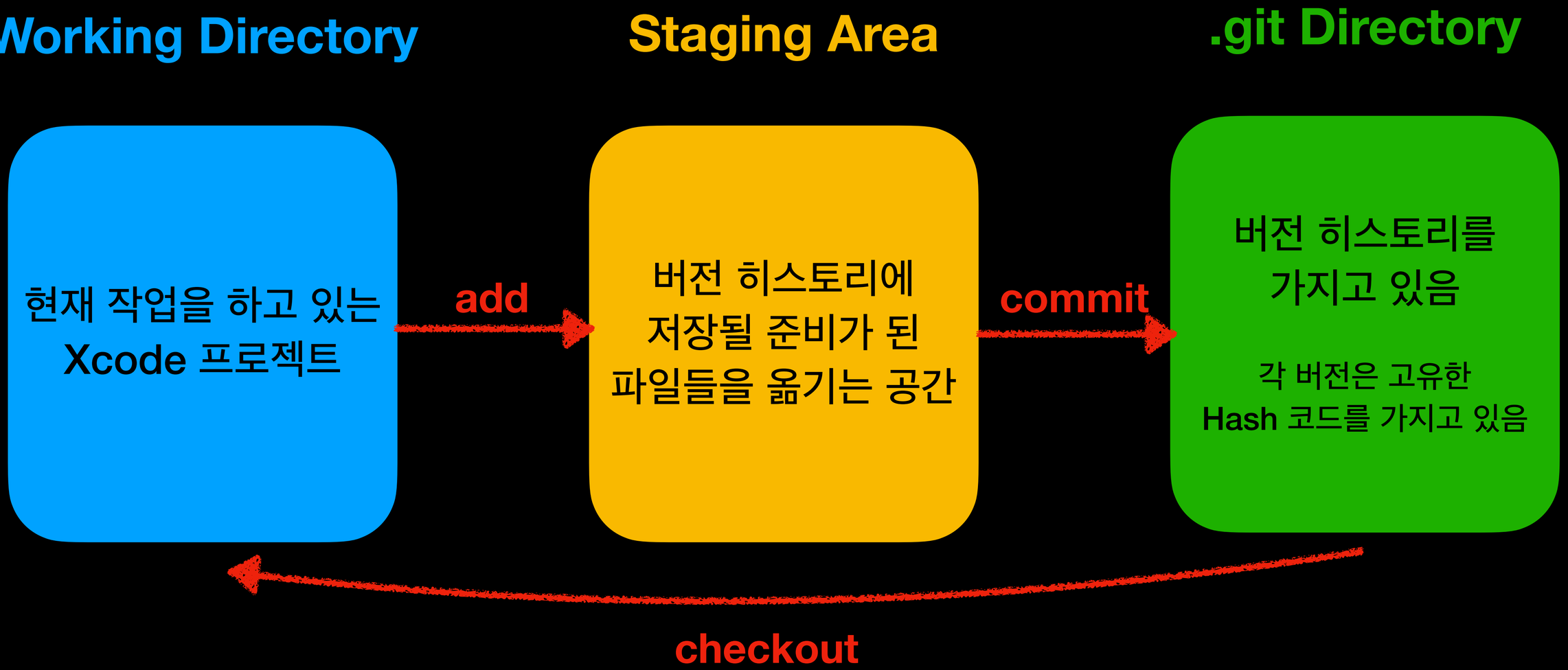
- 서버를 사용하기 때문에 오프라인에선 사용이 불가
- 서버에 문제가 생기는 경우 버전 관리를 통한 업무 불가

Distributed Version Control

서버 및 개발자 각각 버전에 대한 내용을 가지고 있음

- 서버나 특정 개발자가 가진 버전에 문제가 생기더라도 다른 버전을 들고와서 문제를 해결하기 쉬움

Git의 WorkFlow (Local)



간단한 Git 사용법

간단한 Git 사용 방법

git add

Working Directory에 있는 file의 변경사항을 Staging Area로 옮기는 역할을 함

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        practice1.txt
        practice2.txt
```

git add . 를 사용하면 Untracked files 전체를 Staging Area로 옮길 수 있음

git restore --staged {파일명} 를 사용하면 다시 Untracked files로 옮길 수 있음

간단한 Git 사용 방법

git commit

Staging Area에 있는 파일들의 변경 히스토리를 .git Directory로 옮기는 역할

```
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitignore
        renamed:    practice.txt -> practice1.txt
        new file:   practice2.txt
```

아직 Remote (Github)에 push를 하지 않았다면 **git commit --amend**를 사용해 커밋 수정 가능

이미 올려버렸다면 강제 푸시 (**git push -f**)가 불가피함

Commit 템플릿 생성하기

1. .gitmessage.txt (이름은 자유) 파일을 생성한다 (vi .gitmessage.txt) → 단일 프로젝트에 적용

1-1. .gitmessage.txt (이름은 자유) 파일을 생성한다 (vi ~/.gitmessage.txt) → 전체 프로젝트에 적용

2. i로 수정 모드로 변경 후 원하는 템플릿을 작성한다
   (템플릿은 찾아보면 많이 나옵니다)

3. 작성 후 :wq + enter를 눌러 작성한 템플릿을 파일에 저장한다

4. git config commit.template .gitmessage.txt 명령어를 통해 템플릿으로 등록한다
   → 단일 프로젝트에 적용

4-1. git config --global commit.template ~/.gitmessage.txt 명령어를 통해 템플릿으로 등록한다
   → 전체 프로젝트에 적용

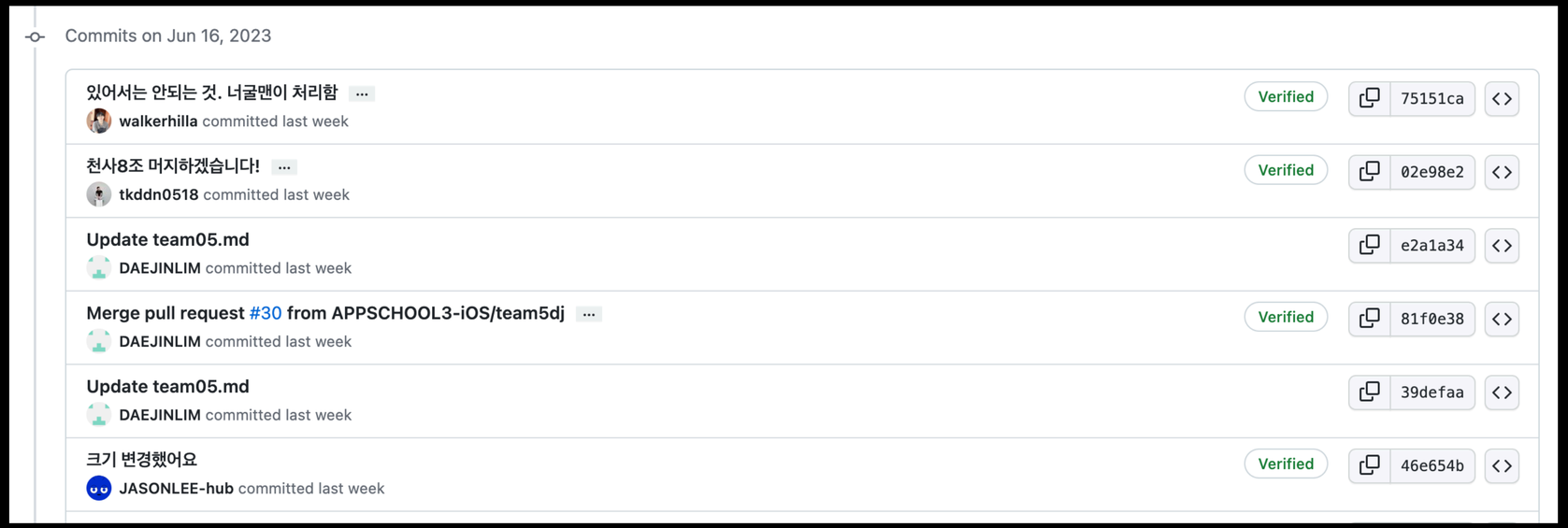
5. git config --list를 통해 템플릿이 잘 등록되었는지 확인

6. 커밋 템플릿을 제거하려면 git config --unset --global commit.template

왜 템플릿을 설정하는 것이 좋을까?

개발자는 협업을 할 때 커밋 메세지를 통해 작업 내용을 파악하게 됨

따라서 일관된 커밋 메세지를 작성할 때 다른 개발자들도 본인의 작업 내용을 빠르게 파악할 수 있다



작업하는데는 전혀 문제 없지만 나중에 해당 커밋이 어떤 작업인지 정확히 알 수 있을까?

👍 참고할 만한 커밋 메세지 규칙

Karma 방식 (http://karma-runner.github.io/6.4/dev/git-commit-msg.html)

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

type으로 들어가는 것은 feat / fix / docs / test 등이 있으며 이는 프로젝트를 시작할 때 정해보고 시작하는 것을 권장

Commit은 어떻게 쪼갤까?

Commit은 어떻게 쪼갤까?

사실 이는 **답이 없는 문제**…

하지만 커밋의 단위가 너무 크면 추후 커밋 히스토리를 확인하거나 되돌릴 때 어려움

그렇다고 너무 작게 쪼개면 무의미한 커밋들이 생길 수 있음

1. 기능 단위로 하나의 기능만 하나의 커밋에 넣자
2. 나중에 되돌려도 작업에 큰 영향이 없도록 쪼개자

Pull Request도 동일하게 고려해야 함

이 부분은 안 올리고 싶다면

부수적인 내용이거나 외부에 노출되면 안되는 중요한 내용인 경우 이를 어떻게 관리하면 좋을까?

.gitignore

Git 버전 관리에서 제외할 파일 목록을 지정하는 파일

vi .gitignore 명령어를 통해 파일을 생성하고 바로 내용을 작성해주면 됨
(물론 touch .gitignore로 생성 후 작업도 가능)

그럼 여기서 어떤 것들을 넣으면 좋을까?

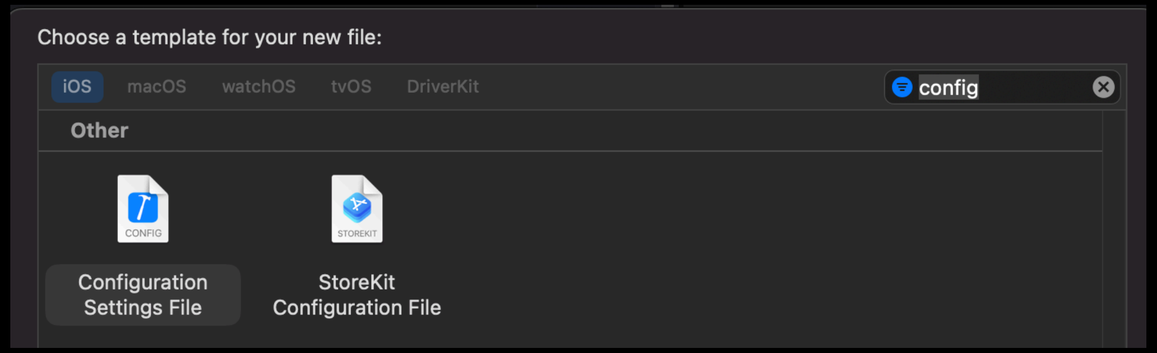
https://www.toptal.com/developers/gitignore

요 사이트에서 키워드를 넣고 생성하면 어느정도 알아서 생성해줌

그럼 보안 관련 중요한 파일들은 어떻게 관리할까?

이 또한 일단 .gitignore 파일에 해당 파일을 등록해서 Staging Area로 해당 파일이 올라가지 않도록 관리함

1. Configuration Settings File을 생성



2. 이런 식으로 해당 파일에 관리해야 할 키 값 등을 정의해둠

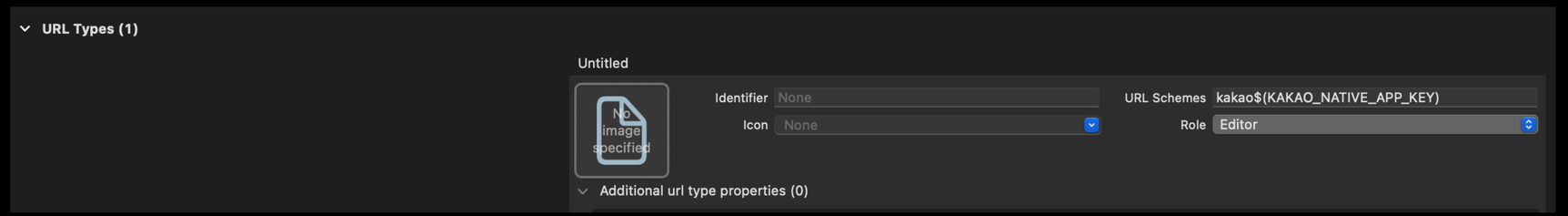
```
11
12  KAKAO_NATIVE_APP_KEY = [redacted]
13  BASE_URL = [redacted]
14
```

그럼 보안 관련 중요한 파일들은 어떻게 관리할까?

3. info.plist에 프로퍼티 등록

4. 사용할 때에는 아래 이미지처럼 사용

```
var baseURL: String {
    guard let baseURL = Bundle.main.infoDictionary?["BaseURL"] as? String else {
        return ""
    }

    return baseURL
}
```



5. 마지막으로 config 파일을 .gitignore에 올려서 해당 파일은 올라가지 않도록 구현

*.xcconfig를 등록해놓으면 config 파일은 전부 Staging Area에 올라가지 않게 됨

그럼 보안 관련 중요한 파일들은 어떻게 관리할까?

물론 이런 방식만 존재하는 것은 아니며 회사마다 정책이 다를 수 있음

단순하게 Property List 추가해서


```
guard let value = plist?.object(forKey: "") as? String else {
    fatalError("Couldn't find key '' in ''")
}
return value
```

이런 식으로 사용하고 해당 파일을 .gitignore에 등록하는 식으로 구현도 가능

분명 .gitignore에 등록했는데 변경사항으로 잡히지...?

이미 Git에 해당 변경사항이 잡힌 경우 .gitignore에 등록해도 적용이 안 됨

따라서 Working Directory에선 파일을 유지한 채 Git에 올라간 파일을 제거하려면

git rm --cached {파일 명}

--cached를 붙이지 않으면 WorkingDirectory와 Git 모두에서 파일이 삭제됨
--cached를 붙여야 Git에서만 해당 파일이 삭제됨

실수를 어떻게 해결할 수 있을까?

기본적으로 커밋 단위를 너무 크게 하지 않는 것이 좋다.
다시 되돌려야 하는 경우 살려도 되는 부분도 날아갈수도...

- 수정사항을 전부 반영하지 못했다...
- 커밋 메세지가 마음에 들지 않는다
- 커밋을 쪼개고 싶거나 혹은 합치고 싶거나
- 커밋한 내용에서 버그를 발견해서 이를 되돌리고 싶다

커밋하지 않은 내용들을 취소하고 싶을 때

git restore

이전에는 git checkout 명령어를 통해 해당 작업을 했으나 checkout이 가지는 기능들이 많아지면서
명령어들이 새롭게 추가되었다 (Git 2.23부터)

브랜치를 변경하는 명령어: git checkout {브랜치} → git switch {브랜치}

특정 커밋으로 이동하는 경우 : git checkout {커밋 해시코드} → 요건 그대로 사용

Staging Area로 올리지 않은 내용을 취소할 때 : git checkout - - {파일명} → git restore {파일명}
Staging Area로 올린 내용을 취소할 때 : git restore --staged {파일명}

이때 새롭게 추가한 파일의 경우 지워지지 않음으로 git clean -fd를 통해 지워야 함
(아니면 아예 reset - -hard HEAD로 날려버려도 됨)

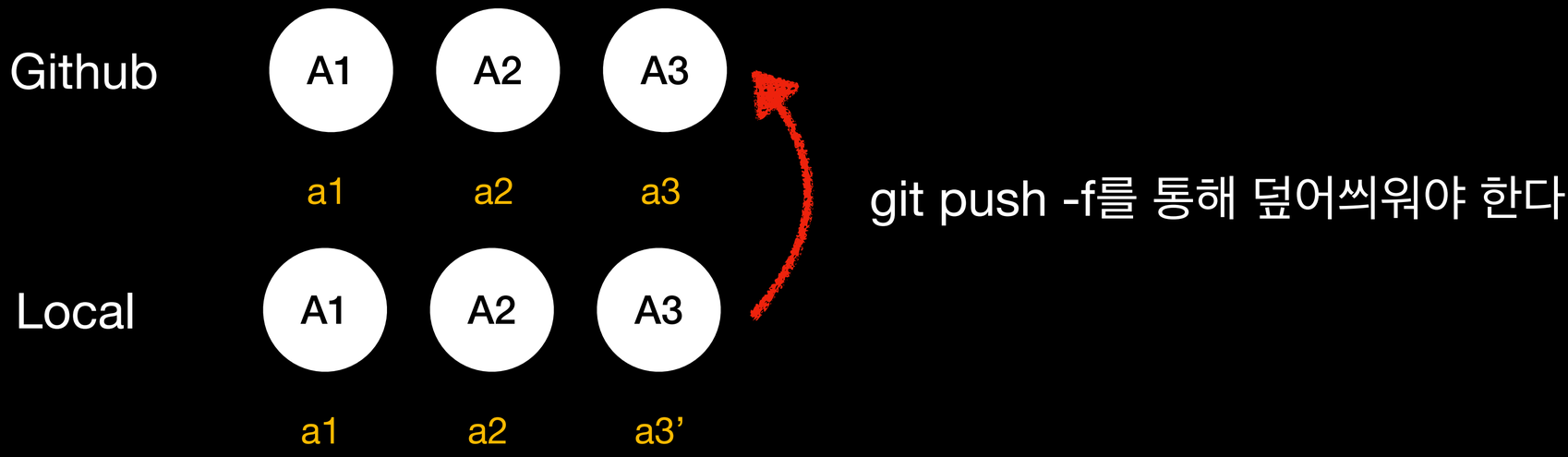
커밋한 내용을 수정해야 할 때

git commit --amend

단순히 Commit 메세지를 수정할 수도 있고, 수정 내용을 추가해서 다시 커밋을 올릴 수 있다

이때 커밋 순서는 변경되지 않지만 커밋의 해시 코드는 바뀌게 된다

이미 Remote(Github)에 커밋한 내용을 push한 경우에는 git push를 통해 해당 커밋을 다시 올릴 수 없다



왜 강제 푸시를 하면 위험할까?

a5 해시 값을 갖도록 변경해버리면…

Github

A1 A2 A3 A4

a1 a2 a3 a4

Branch를 파서 열심히 작업 중

B1 B2 B3

아예 이전 커밋 기록이 덮어 씌워져서 다른 코드를 기반으로 작업했던 개발자들에게 혼란을 줌

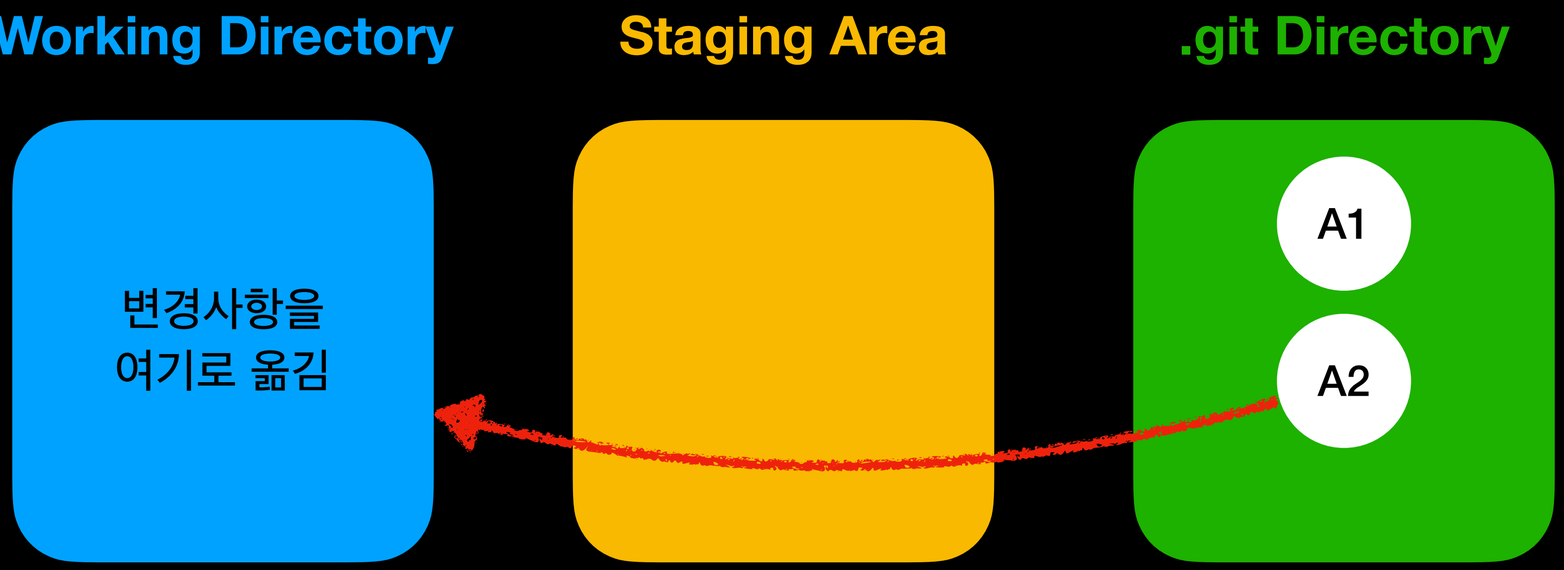
Conflict가 발생할 가능성이 높아짐

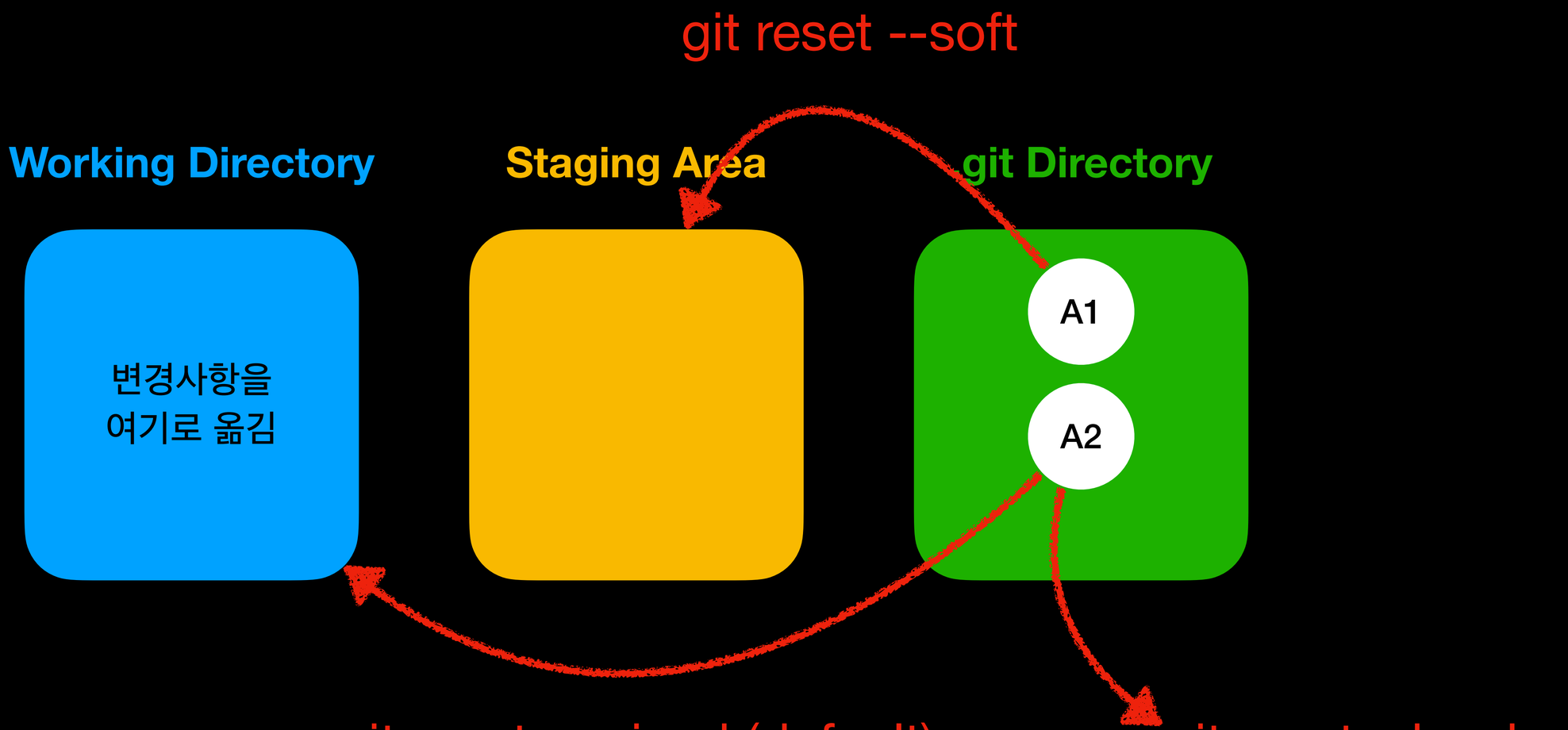
아예 이전 커밋으로 돌아가고 싶다면

git reset

현재 Head 값을 기준으로 되돌릴 수도 있고, → git reset HEAD ~2
커밋 해시값을 이용해 되돌릴 수도 있다 → git reset {해시값}

이때 git reset만 사용하게 되면 git reset --mixed와 동일한 의미




git reset --soft
Working Directory
Staging Area
git Directory
변경사항을
여기로 옮김
A1
A2
git reset --mixed (default)
git reset --hard
변경 사항을 아예 날려버림

그런데 Reset을 다시 되돌릴 수 없을까...

git reset / git reflog의 조합

git reflog를 사용하게 되면 지금까지 사용한 Git 명령어들의 history를 볼 수 있다

```
78db99b (HEAD -> main, origin/main) HEAD@{0}: pull origin main (finish): returning to refs/heads/main
78db99b (HEAD -> main, origin/main) HEAD@{1}: pull origin main (pick): feat: practice 파일  생성
59100ef HEAD@{2}: pull origin main (start): checkout 59100ef5cfda4c8d7756ca44f51fad25224ed6b0
7fd8987 HEAD@{3}: commit (initial): feat: practice 파일  생성
```

여기 있는 해시값을 활용해 돌아가려면...!

git reset --hard {해시값} 명령어를 이용하면 된다

그러면 reset을 하기 전 상태로 돌아갈 수 있다

기록은 남긴 채로 다시 이전 커밋으로 돌아갈래

git revert

git reset의 경우 다시 이전 커밋으로 돌아갔다는 기록이 전혀 남지 않는다
만약 이를 기록으로 남기고 싶다면...?

```
commit 6d25228994905c4fe757f050e6c57e1da71f3781 (HEAD -> main)
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Tue Jun 20 21:48:03 2023 +0900

    Revert "Initial commit"

    This reverts commit 59100ef5cfda4c8d7756ca44f51fad25224ed6b0.

commit 78db99b7c0926a81584ff2b297457a8aec8cab34 (origin/main)
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Mon Jun 19 22:22:58 2023 +0900

    feat: practice 파일 생성

commit 59100ef5cfda4c8d7756ca44f51fad25224ed6b0
Author: 양호준 <yanghojoon8487@gmail.com>
Date:   Mon Jun 19 22:21:28 2023 +0900

    Initial commit
```

git revert를 사용하면 된다

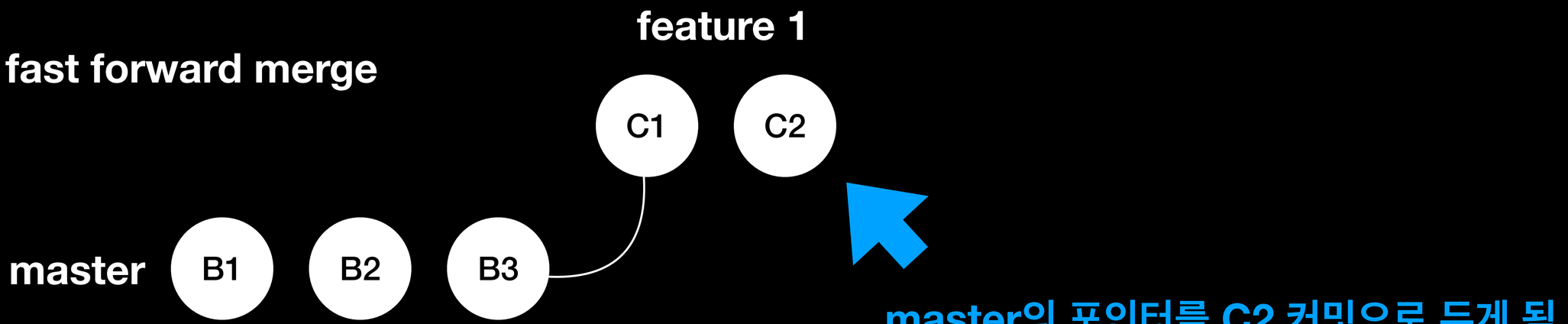
그러면 다시 이전 커밋으로 돌아갔다는 커밋이 찍힌채로 되돌아가게 된다

물론 git revert --no-commit 명령어를 이용해 커밋을 남기지 않고 돌아갈 수도 있다.

브랜치 병합하기

git merge

fast forward merge

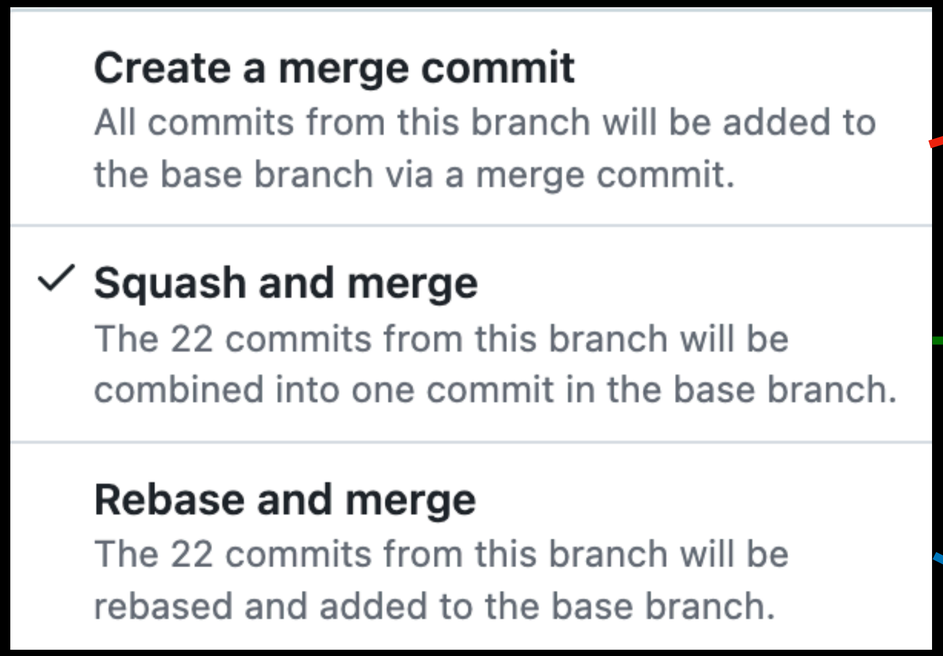


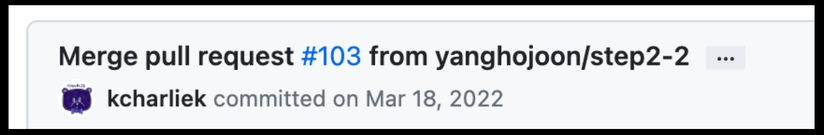
단 이 때 Merge가 됐다는 커밋은 따로 남지 않게 됨

Github에서의 Merge

하지만 현재 저는 PR을 올리고 리뷰를 받은 후 Github을 통해 Merge를 하는 방식을 주로 사용

Github은 3가지 방식의 Merge를 제공함





지금까지 커밋이 쌓이고 마지막에 위 이미지처럼 머지 커밋이 남게 됨

해당 PR에 쌓은 커밋을 하나의 커밋으로 정리해서 Merge가 됨

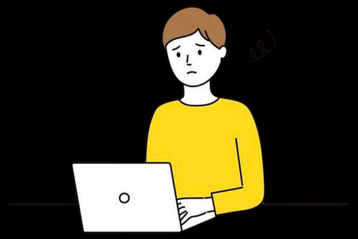
지금까지 작성한 커밋이 그대로 머지됨

fast forwad merge와 유사

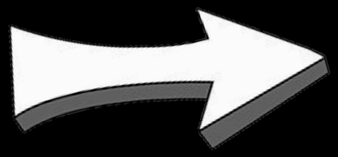
협업 시 가장 중요!!

git rebase 협업 시에 가장 **많이 / 중요**하게 사용하는 명령어

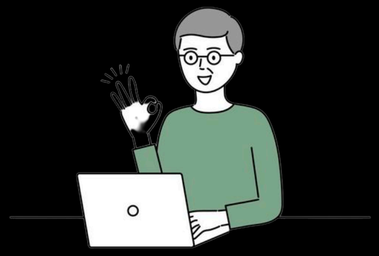
A: 설정화면 리뉴얼 작업



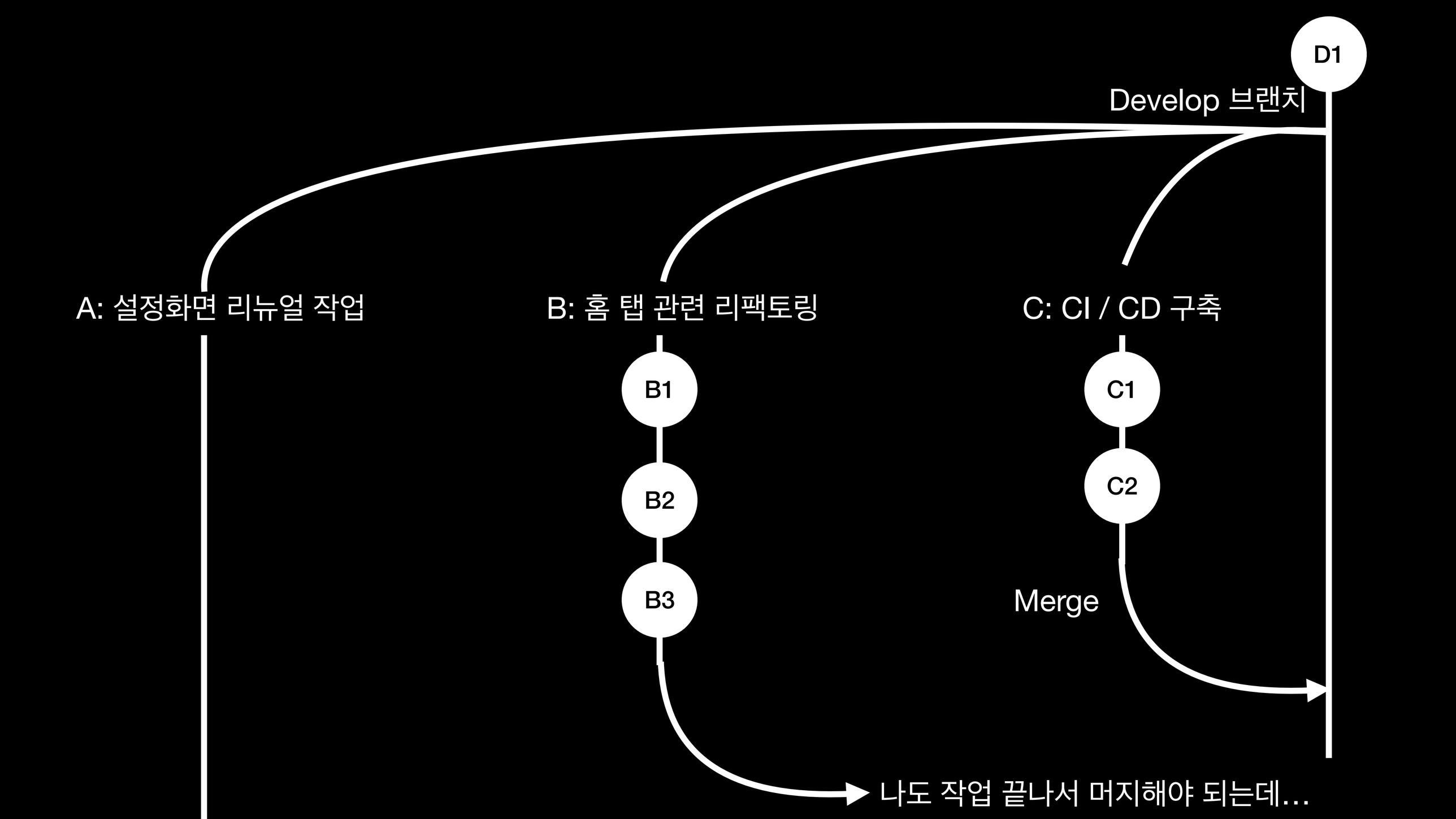
B: 홈 탭 관련 리팩토링



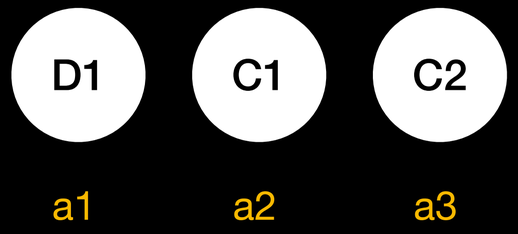
대부분 모든 작업들이
동시에 수행됨
(비동기)



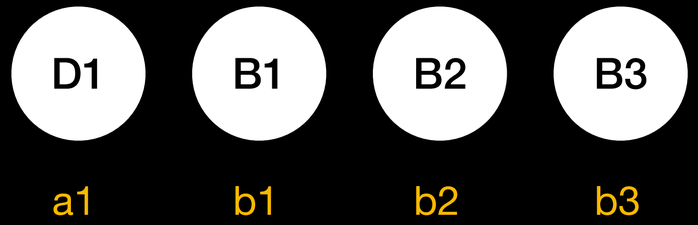
C: CI / CD 구축


D1
Develop 브랜치
A: 설정화면 리뉴얼 작업
B: 홈 탭 관련 리팩토링
C: CI / CD 구축
B1
B2
B3
C1
C2
Merge
나도 작업 끝나서 머지해야 되는데…

현재 Develop 브랜치를 git log로 찍어보면



B의 Branch를 git log로 찍어보면

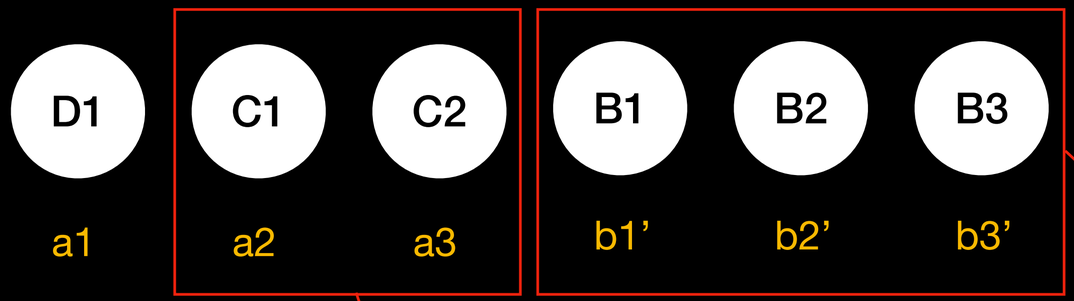



```
commit 8ff84ba8e2a5864f567400f5029455d08a0d206d (HEAD -> develop, origin/develop, origin/HEAD)
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Fri Nov 18 08:37:01 2022 +0900

    fix: 리스트에 아무것도 없는 경우 다시 돌아올 때 강제종료되는 문제 해결

commit acf61c348dbeb48b0a6815715859a8c012079774
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Fri Nov 4 21:07:11 2022 +0900

    refactor: 배포도 개발 서버를 사용하도록 수정

commit 71f9d408a3e485fbbf4136682a4c7cae65b79b4a
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Fri Nov 4 20:58:29 2022 +0900

    fix: 왔어요/안왔어요 API 파라미터 수정

    - userID에 내 id가 아닌 해당 셀의 id를 보내도록 수정

commit bca009ea97a0bf4078a7bacd8025609c1350474a
Author: yanghojoon <yanghojoon8487@gmail.com>
Date:   Thu Nov 3 23:38:40 2022 +0900

    chore: swiftlint 규칙 수정
```

원래 해시값은 훨씬 복잡하지만 설명의 편의를 위해 간단하게 표현

B의 브랜치를 바로 Develop에 붙여버리게 되면 이런 상황에서 Conflict이 발생 가능

그래서 필요한 것이 Rebase



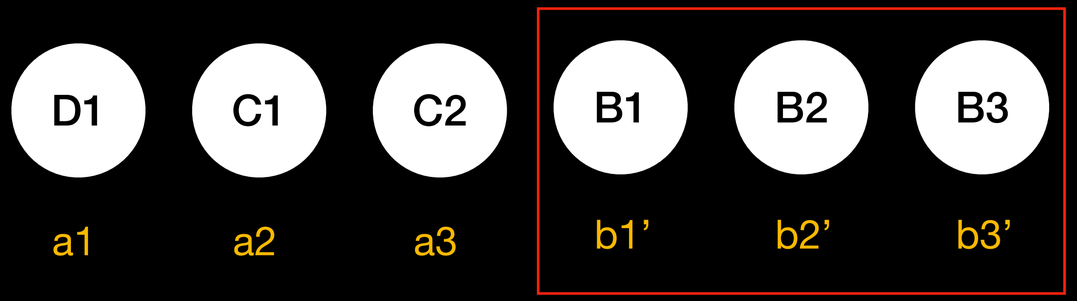
C가 작업한 내용도 유지하면서

B의 작업도 뒤로 붙여야 함

git rebase {베이스를 맞춰야 하는 브랜치}

현재는 develop 브랜치를 기반으로 베이스를 맞춰줘야 함

B의 Branch를 git log로 찍어보면



여기서 주의깊게 봐야할 부분은 B의 커밋들의 해시값

내용들은 동일하나 해시값이 변경되었다?!

Remote에 올린 커밋 로그와 해시값이 다르기 때문에 강제 푸쉬가 필요함

git push -f

예전에 올린 커밋을 수정하고 싶다면...

신나게 개발을 하고 있었는데

앗... ... 이전에 올린 커밋 메세지에 오타를 발견했다...

예전에 올린 커밋을 수정하고 싶다면...

git rebase -i (interactive)

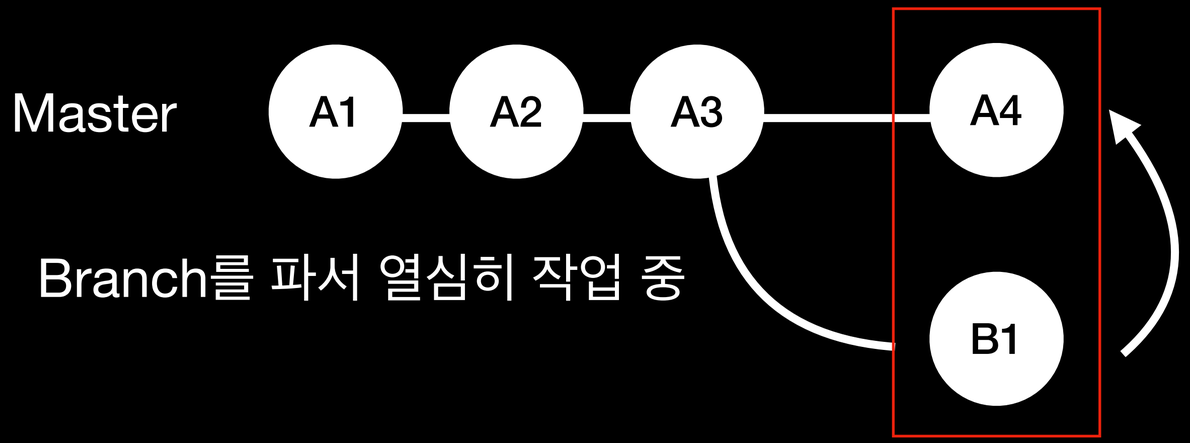
```
pick 78db99b feat: practice 파일 생성
pick 6d25228 Revert "Initial commit"

# Rebase 59100ef..6d25228 onto 59100ef (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 [-C | -c] <commit> = like "squash" but keep only the previous
#                    commit's log message, unless -C is used, in which case
#                    keep only this commit's message; -c is same as -C but
#                    opens the editor
# 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
# u, update-ref <ref> = track a placeholder for the <ref> to be updated
#                       to this position in the new commits. The <ref> is
#                       updated at the end of the rebase
#
```

친절하게 설명이 다 나와 있다

우리는 커밋 메세지를 수정하려 하니 pick을 reword로 변경한 후 :wq를 통해 저장해주면 된다

Conflict은 왜 날까?

Conflict의 경우 보통 같은 파일 내에서 수정을 한 이후 Merge나 Rebase를 하려 하면 발생한다



Merge를 하려 한다면…

Git 입장 : 아니 여기서도 변경되고 저기서도 변경됐는데 어떤 걸 기준으로 합쳐야 해… 몰라…

Conflict 발생!!!!

Conflict은 어떻게 해결하면 좋을까?

1. 두 곳에서 동일한 파일을 수정하거나 추가해서 Conflict이 발생함

```
interactive rebase in progress; onto 89ddb19
Last command done (1 command done):
   pick 470f8dd feat: 기본 달력 버튼 액션 추가
No commands remaining.
You are currently rebasing branch 'feature/basic-calendar' on '89ddb19'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both modified:   JTAppleCalendarExample.xcodeproj/project.pbxproj
        both added:      JTAppleCalendarExample/BasicCalendarViewController.swift
        both modified:   JTAppleCalendarExample/ViewController.swift

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    JTAppleCalendarExample.xcodeproj/project.xcworkspace/xcshareddata/swiftpm/Package.resolved

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

하나의 파일에 2곳에서 수정 / 추가가 발생하여 문제가 발생함

Conflict이 난 부분은 git status를 통해 확인이 가능함

2. open -a Xcode {Conflict이 발생한 파일 경로}를 통해 Conflict이 발생한 파일을 연다

```
63  ¬
64  <<<<<<< HEAD¬                                          Source control conflict ma
65  ····navigationController?.pushViewController(basicCalendarViewController, animated: true)¬
66  =======¬
67  ····self.navigationController?.pushViewController(basicCalendarViewController, animated: true)
68  >>>>>>> 470f8dd (feat: 기본 달력 버튼 액션 추가)¬
69  ····}¬
```

현재 HEAD에 구현된 코드

Conflict이 발생한 커밋에 구현된 코드

3. Conflict이 발생한 커밋에 구현된 코드와 HEAD에 구현된 코드의 내용 중 남길 코드를 선택한다
   (둘 다 남겨놓는 것도 가능)

4. Conflict이 발생한 부분을 위 방법으로 전부 수정했으면 **git add .**를 통해 수정 사항을 Staging Area로 이동

5. git rebase --continue를 통해 리베이스를 계속 진행시킴

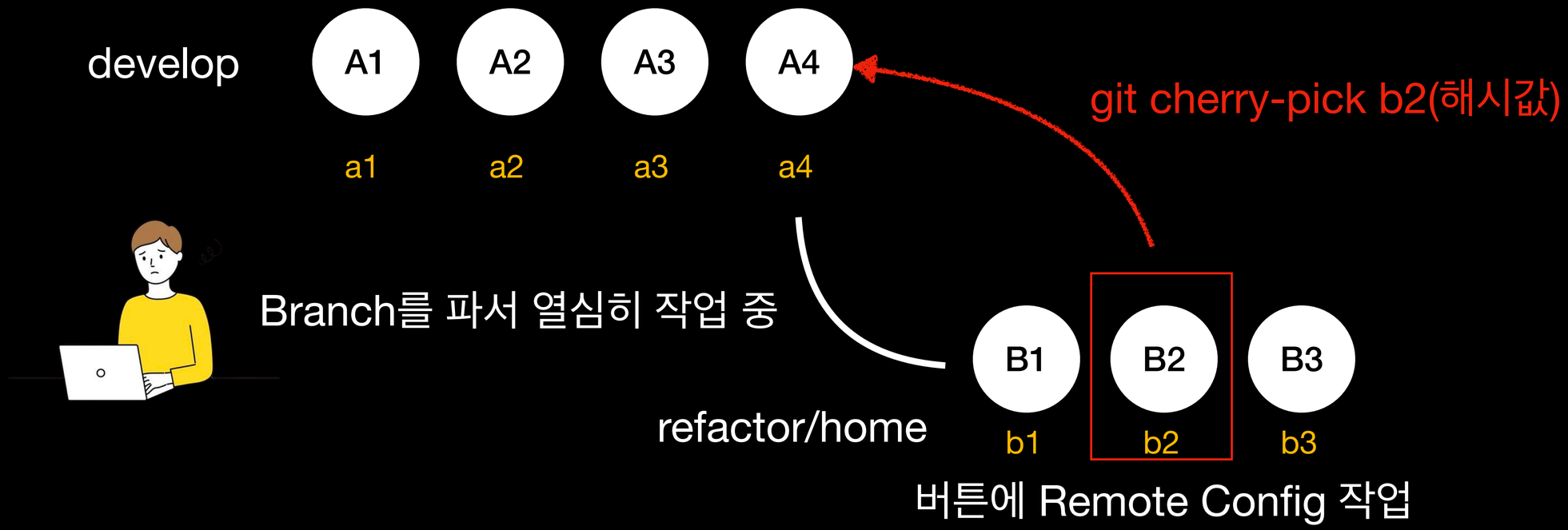
6. 정상적으로 rebase가 마무리될 때까지 해당 작업 반복

```
> git rebase --continue
[detached HEAD 3a90748] feat: 기본 달력 버튼 액션 추가
 1 file changed, 1 insertion(+), 1 deletion(-)
Successfully rebased and updated refs/heads/feature/basic-calendar.
```

특정 커밋만 빼오기

git cherry-pick

개발을 하다보면 작업은 취소됐으나 특정 기능만 빼올 수 있다



커밋을 하지 않고 변경 사항을 저장하고 싶다면

git stash

열심히 작업을 하다가 다른 브랜치로 이동할 일이 생긴다면?

```
error: Your local changes to the following files would be overwritten by checkout:
        jasoseol/Chat/Message/MessageCell/MessageReport.swift
Please commit your changes or stash them before you switch branches.
Aborting
```

변경사항을 그대로 둔 채로 **git switch**를 활용해 브랜치 이동을 하면 위와 같은 에러가 발생함

그런데 에러 문구를 잘 보면 변경사항을 커밋하거나 stash하라고 한다

(Stash를 하게되면 별도의 저장 공간에 변경 사항을 저장할 수 있음!!)

Stash를 하면 아래처럼 저장을 잘 했다고 나온다

```
> git stash
Saved working directory and index state WIP on
```

저장할 내용을 확인하고 싶다면 **git stash list** 명령어를 통해 확인이 가능하다

```
stash@{0}:
stash@{1}:
```

스택 구조로 가장 나중에 들어간 것부터 꺼낼 수 있다

가장 최근에 저장한 내용을 다시 불러오려면

git stash pop : Stash List에서 가장 최근 저장 내용을 제거하면서 내용을 불러옴

git stash apply : Stash List를 그대로 두고 가장 최근 저장 내용을 불러옴

저장된 내용을 지우려면

git stash drop : Stash List에서 가장 최근 저장 내용을 제거

git stash clear : Stash List를 전부 제거

브랜치 전략

프로젝트가 커지면서, 참여하는 개발자가 많아지게 되면
이에 따라 Branch도 많아지게 된다

따라서 브랜치 전략을 어떻게 가져가야 할 지 정하는 것이 필요하다

회사마다 가져가는 브랜치 전략은 정말 다양할 수 있으나

GitFlow

Trunk-based

Git 명령어 단축키 만들기

1. open ~/.gitconfig 를 통해 Config 파일을 연다

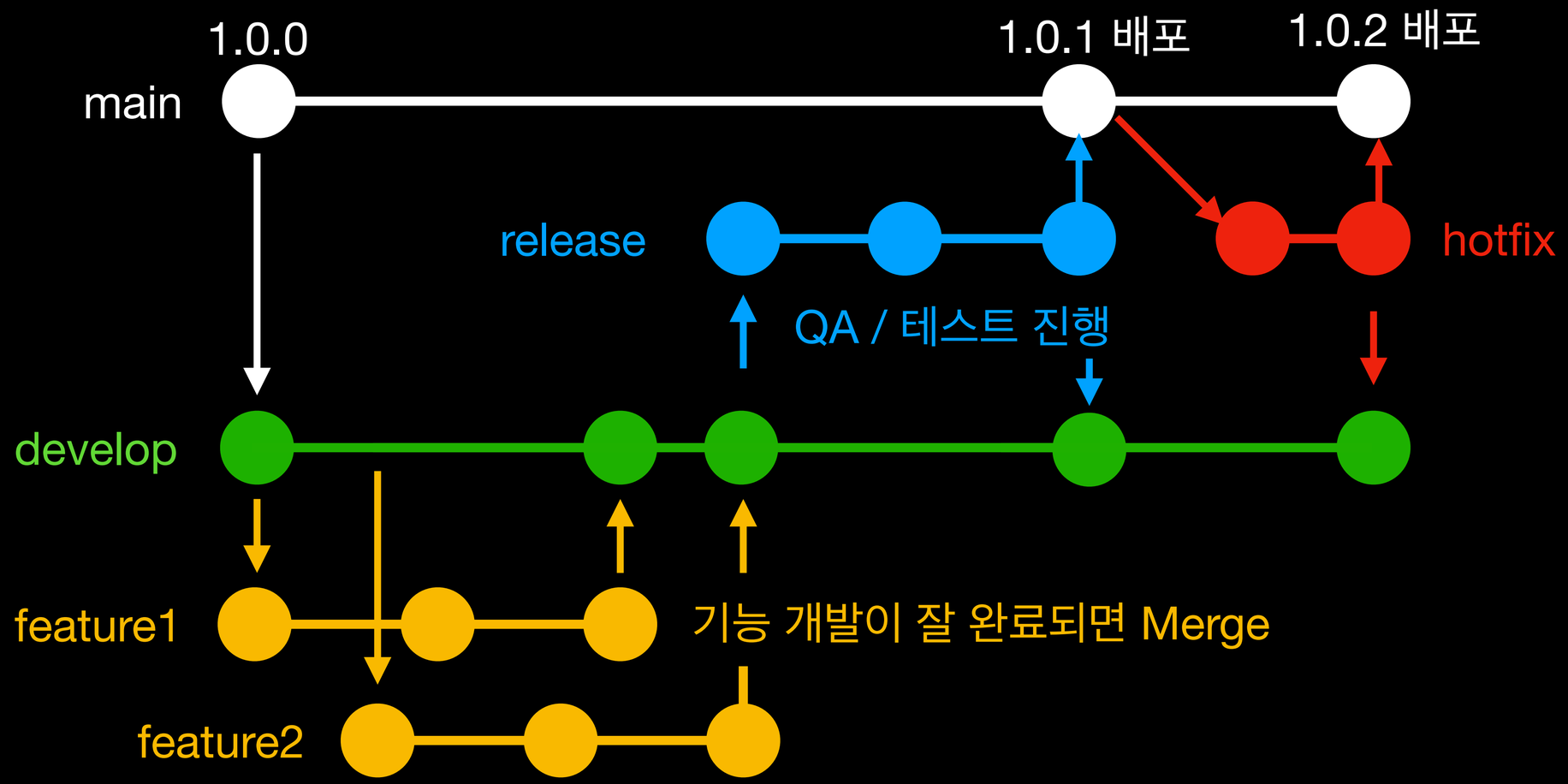
2. alias 쪽에 원하는 단축키를 작성하고 해당 명령어가 어떤 명령어인지 작성한다

```
[alias]
    hist = log --graph --all --pretty=format:'%C(yellow)[%ad]%C(reset) %C(green)[%h]%C(reset) |
%C(white)%s %C(bold red){{%an}}%C(reset) %C(blue)%d%C(reset)' --date=short
    s = status
```

3. git을 붙이고 작성한 단축키를 작성하면 단축키로도 명령어 호출이 가능하다

브랜치 전략 - GitFlow

1. main (master)
2. develop
3. feature
4. release
5. hotfix



물론 필요에 따라 생략되거나 추가되는 브랜치도 존재할 수 있음

브랜치 전략 - GitFlow

장점

1. 테스트와 리뷰를 할 수 있는 상황이 많아 코드 품질을 보장하고
   버그를 조기 제거하는데 도움이 됨
2. 코드 스타일을 일정하게 하는데 도움이 됨

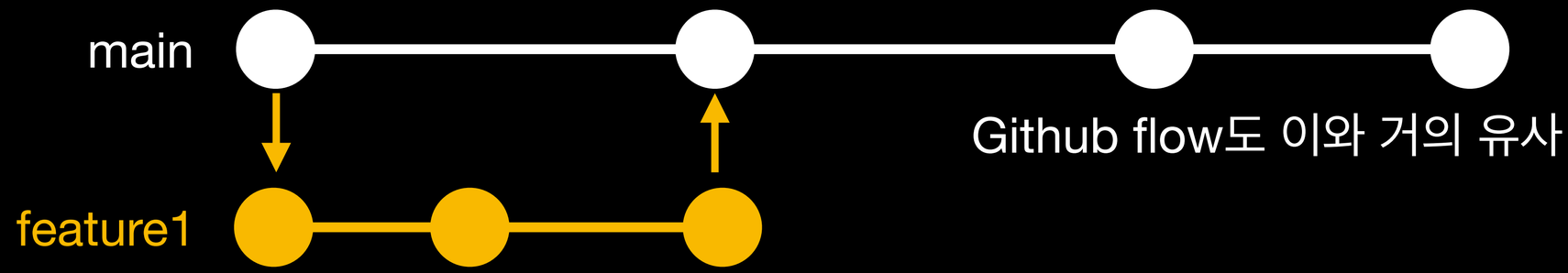
단점

1. 개발 속도가 늦어질 수 있음
2. 리뷰 과정에서 특정 개발자의 Micro Managing이 발생할 수 있음

이미 프로덕트가 어느정도 존재하고 주니어 개발자가 많은 경우 유용

브랜치 전략 - Trunk-based

하나의 브랜치를 잘 관리하자!!



필요할 때 브랜치를 파서 개발 후 바로 merge

브랜치 전략 - Trunk-based

장점

1. 빠른 개발 속도를 가져갈 수 있음
2. 포괄적인 자동화 테스트 가능 (CI / CD 도입이 용이)

단점

1. Feature 브랜치에서 많은 테스트가 필요함 → 코드 품질 저하로 이어질 수 있음

숙련된 시니어 개발자가 주로 작업을 하거나 빠른 작업이 필요할 때

GitHub 활용하기

그렇다면 Github은 왜 필요할까?

지금까지는 Git의 기본적인 명령어들에 대해 살펴봤음

그런데 이렇게 작업을 하다가...

- 내 맥북이 고장이 나버렸다면
- 다른 곳에서 다른 컴퓨터로 작업을 해야 한다면
- 다른 개발자들과 협업이 필요하다면

작업 내용을 서버에 올려야 함 → Github

Github에 있는 내용 로컬로 가져오기

git fetch VS git pull

둘 다 로컬로 서버에 있는 버전 히스토리를 가져오는 것은 동일

git fetch

로컬에 있는 **HEAD**는 그대로 두고 서버에 있는 히스토리를 가져옴

로컬의 작업에는 영향을 미치지 않음

git pull

git fetch를 수행한 후 로컬 브랜치도 이에 맞게 업데이트하는 작업을 수행함 **(git fetch + git merge)**

로컬의 작업 트리와 커밋 히스토리가 자동으로 업데이트

Pull Request

리모트에서 Merge 전, 소스코드의 변경사항을 다른 개발자들에게 공유하고 리뷰를 받을 수 있도록 함

- PR을 통해 다른 개발자들이 작성한 코드를 확인할 수 있음
- 상호 간에 피드백을 주고 받으며 성장할 수 있음
- 변경 사항을 구현하면서 새롭게 알게 된 지식을 공유할 수 있음
- 변경 사항에서 구현한 기능에 대한 히스토리로서의 역할을 할 수 있음
- 미처 찾지 못한 버그를 찾아내고 수정할 수 있음

기록으로서의 PR

작업을 관리하고 이에 대한 히스토리를 남길 수 있는 툴은 다양하다

하지만 이 툴들을 3~5년 뒤에도 계속 사용한다는 보장이 있을까?

하지만 GitHub은…?

PR을 어떻게 작성하는 것이 좋을까?

변경된 소스코드는 적게, 변경 사항에 대한 설명은 자세하게

동료와 작업을 하는데 1,000줄 이상의 변경사항이 있는 PR을 리뷰 요청한다...!

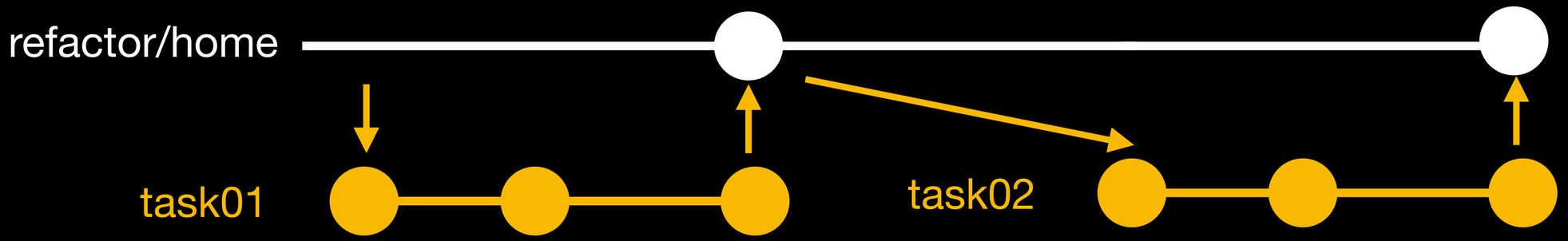
동료도 다른 작업을 진행하고 있는데 이렇게 많은 양의 코드를 리뷰 요청한다면

- 리뷰의 퀄리티도 낮아지고
- 동료의 시간도 많이 잡아먹게 된다

따라서 PR은 최대한 하나의 기능만 들어가도록 쪼개서 보내는 것이 좋다

~~물론 현업에서 업무를 하다보면 불가피하게 PR이 엄청 길어지는 경우도 생기긴... 함...~~

Feature 브랜치를 파서 작업 계획을 짜다가 너무 길어질 것 같다면 브랜치를 더 쪼개자



변경 사항은 최대한 자세하게 작성해주는 것이 좋다

변경 사항을 작성하게된 배경과 작업 내용, 테스트 방법, 스크린샷들을 포함하여 작성

기능 요약

네이버 영화 API를 통해 검색어에 해당하는 영화 목록을 불러옵니다.
각 Cell에 있는 별표를 누르면 즐겨찾기 목록에 등록되며, 다른 곳을 검색하고 오더라도 별표는 유지됩니다.

작업 내용

1. 네이버 API 등록 및 네트워킹 코드 추상화

네이버 API의 경우 Bundle ID와 함께 앱을 등록해야 API를 사용할 수 있었기에 등록을 진행했습니다. 등록과 함께 나오는 ID와 Secret의 경우 URLRequest를 해줄 때 Header로 추가를 해줘야 했습니다.

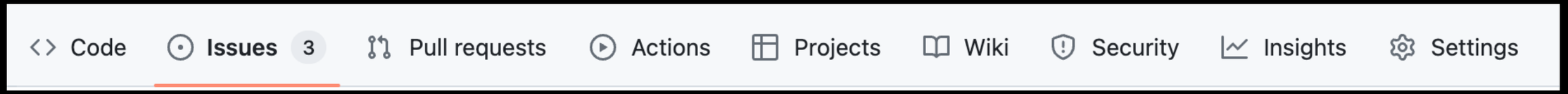
따라서 URLRequest의 Extension에서 init을 구현하여, 여기서 header를 설정해줄 수 있도록 구현했습니다.

```
extension URLRequest {
    init?(api: APIProtocol) {
        let clientID = "{ID를 넣었습니다}"
        let clientSecret = "{Secret 값을 넣었습니다}"

        guard let url = api.url else {
            return nil
        }

        self.init(url: url)
        self.httpMethod = "\(api.method)"
        self.setValue(clientID, forHTTPHeaderField: "X-Naver-Client-Id")
        self.setValue(clientSecret, forHTTPHeaderField: "X-Naver-Client-Secret")
    }
}
```

확장성을 고려해 요청할 API를 `APIProtocol`로 추상화해서 만약 API가 추가되더라도 프로토콜만 채택하면 되도록 했습니다. 다만 `HttpMethod`의 경우 아직 Get만 사용하면 되기 때문에 case를 `get`만 뒀습니다.

URLSession의 경우 `MockURLSession` 추가를 위해 이를 한 단계 추상화하고, URLSession이 이를 채택하도록 했습니다.

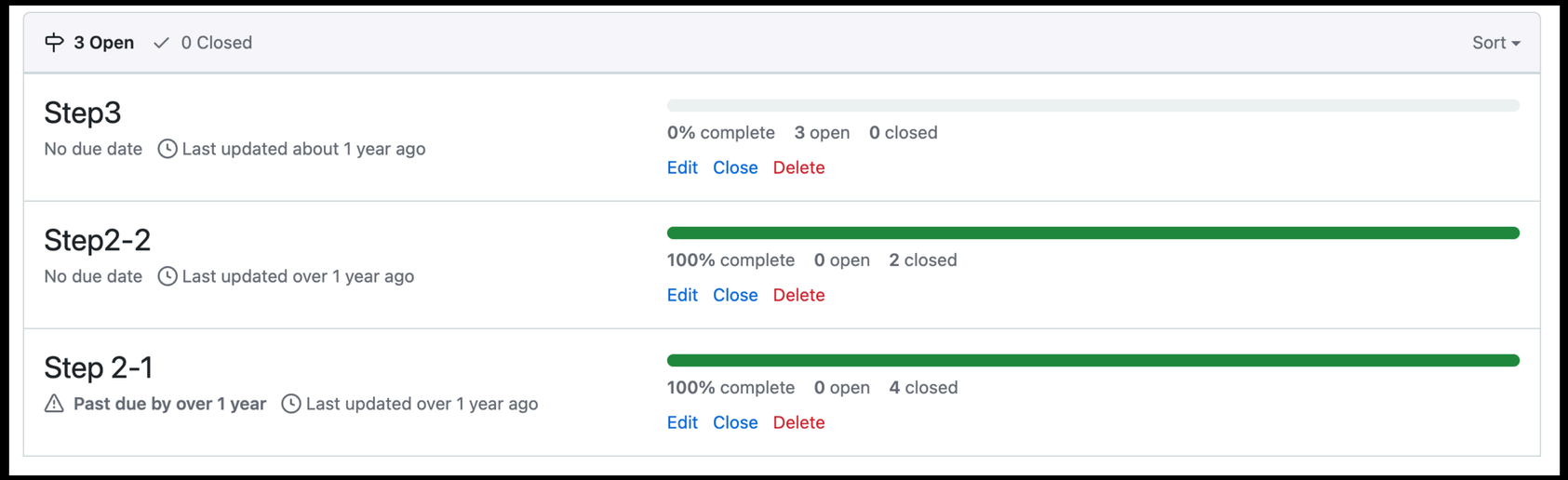
이슈를 만들고 연동하기



GitHub을 보면 Issues 탭을 확인할 수 있다
프로젝트에 필요한 작업이 있다면 미리 Issue에 추가해 놓는다면 특정 작업을 누락하는 확률이 줄어든다

이슈 별로 구분하기 좋도록 Labels와 Milestones를 각 이슈별로 적용할 수 있다.

- Labels : Commit Message 앞에 붙이는 타입 역할로 보면 됨
- Milestones : 이슈들을 큰 틀로 묶을 수 있음. 작업 진척도를 확인할 수 있음



이런 이슈들을 PR과 연동할 경우 보다 시너지를 낼 수 있다

PR과 마찬가지고 Repository 별 Issue도 고유한 번호를 가지게 된다

따라서 아래 이미지처럼 PR Description에 close #{이슈번호}를 적어놓는다면 PR과 이슈가 자동 연동된다

그리고 PR이 머지되는 경우 이슈는 자동으로 Closed된다

close #960

변경 사항을 자세히 PR에 기록하게 되면 이것 자체로 하나의 히스토리가 된다

새롭게 알게 된 내용을 공유해서 팀원 전체의 성장을 유도할 수 있다

리뷰어가 빠르게 맥락을 파악하고 적절한 리뷰를 할 수 있다

PR에서 바로 해당 커밋 히스토리를 확인할 수 있기 때문에 커밋 메세지도 잘 써야 한다!!

추후 과제전형을 할 때에도 이렇게 문서화를 잘해놓으면 플러스 요인이 될 수도...

그럼 이렇게 잘 쓴 PR을 받았으면 리뷰는 어떻게...?

~~난 다른 사람들에 비해 실력도 떨어지고 내 피드백이 도움이나 될까...?~~

코드에 대한 생각은 사람마다 다르고 정답이 없기 때문에 당연히 도움이 됨

그리고 단순히 질문이더라도 리뷰어 입장에선 배울 수 있고, 리뷰를 받는 사람도 설명할 수 있는 기회가 생김

1. Convention에 대한 피드백

소스코드의 Convention 유지를 통해 코드 전체적인 퀄리티를 올릴 수 있음

물론 SwiftLint 등의 툴 도입을 통해 이런 피드백이 달리지 않도록 신경을 사전에 쓰는 것이 중요

스타일쉐어 Swift 스타일 가이드: https://github.com/StyleShare/swift-style-guide

2. 로직에 대한 피드백

강제 종료, 메모리 누수 등이 발생할 수 있는 코드인 경우 이에 대한 피드백을 통해 버그를 줄일 수 있음

메모리를 덜 사용하거나 더 빠른 로직이 있다면 이에 대한 제안도 좋음

3. 코드에 대한 질문

PR을 통해 다른 동료개발자들에게 소스코드를 공유하는 의미도 있는 만큼 질문을 통해 코드 이해도를 올리는 것도 좋음

포트폴리오로 GitHub 사용하기

강의를 듣고 계신 분들은 현재 취업을 목표로 공부를 하고 있다

Repository를 각각 하나의 포트폴리오로 활용하는 것이 중요함

- ReadMe를 꼼꼼히 작성하자

  PR과 동일하게 프로젝트의 배경과 Trouble Shooting 등의 내용을 꼼꼼히 미루지 말고 적어두자
  포폴로도 활용 가능하지만, 이후 자기소개서를 작성할 때에도 도움이 될 것

- Star를 잘 찍어두자

  Star를 통해 이 개발자가 어떤 것에 관심이 있는지를 확인할 수 있다

- 커밋 메세지와 PR은 항상 잘 작성하자

  이 2개를 통해 협업이 이뤄지는 만큼 이 부분은 항상 신경쓰자