개요

- **Working Directory 작업 단계**
  - Working Directory에서 수정한 파일 내용을 이전 커밋 상태로 되돌리기
  - `git restore`
- **Staging Area 작업 단계**
  - Staging Area에 반영된 파일을 Working Directory로 되돌리기
  - `git rm --cached`
  - `git restore --staged`
- **Repository 작업 단계**
  - 커밋을 완료한 파일을 Staging Area로 되돌리기
  - `git commit --amend`

0.1 Working Directory

깃으로 관리하고 있는 로컬에서의 폴더에서 트랙킹이 되고 있는 상태를 의미

git add . 하기 전

0.1 Staging Area

git add . 이후 단계로 working directory 이후 단계

만약 이미 프로토타입이 올라가 있는 상태라면

레포지토리에 올라가 있는 것과의 차이점만 add 하게 된다.

0.1 Repository

깃 랩, 깃 허브에 올린 상태가 아님

레포지토리 단계는 깃에게 등록한 것으로

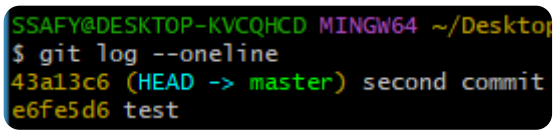
git commit 단계를 의미한다.

0.0.1 명령어

- $ rm -rf {{파일명}} : 삭제
- HEAD는 나의 branch에서 최신 영역을 의미한다.?
    - 그래서 root-commit 이 없을 경우 HEAD는 없을 수 밖에 없다.
- mkdir : 폴더 만들기

Working Directory 작업 단계 되돌리기

1. git init
2. touch a.md (마크다운타입 a 파일 생성)
3. git add .
4. git commit
5. git log --oneline (로그 확인)
6. vim a.md (a.md 파일을 vim로 열기)
7. Insert or i 누르기
8. 작성후
9. esc 누르고 :wq 를 입력하면 나가진다.
10. 다시 git add, git commit 을 하고
11. git log --oneline 으로 확인하면

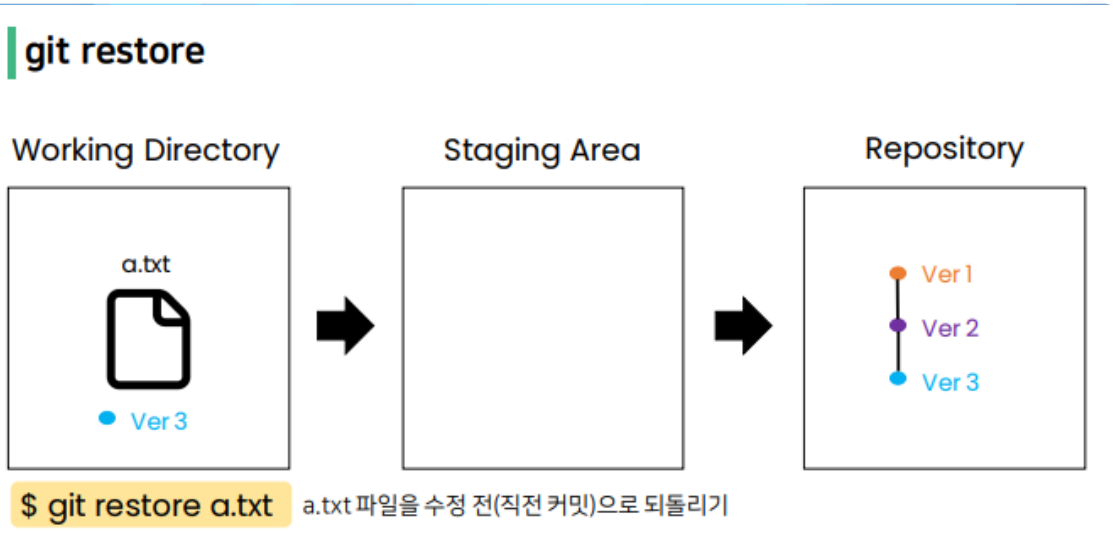


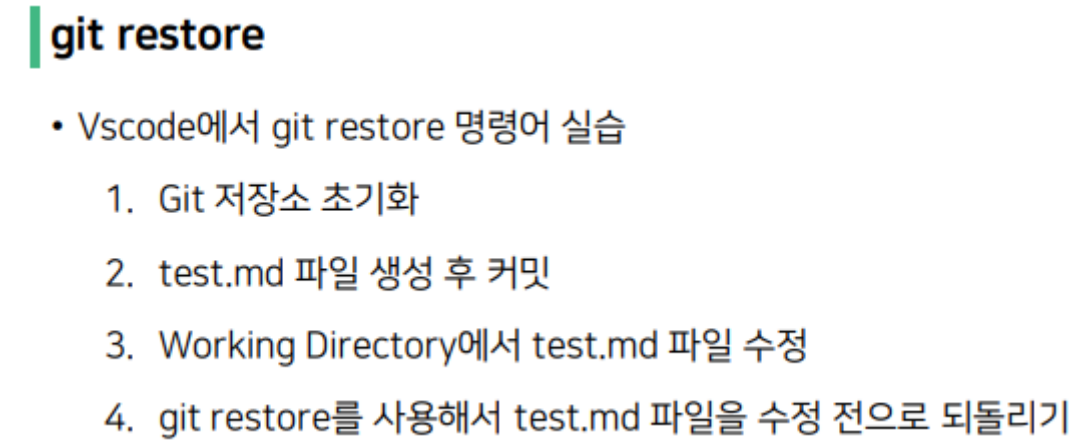
12. 다시 수정 후
13. git status 를 입력하면 차이점이 있다.
14. 최신 버전으로 되돌리는 방법(즉. 수정 이전)
    1. git restore {{파일명}}

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$  git restore a.md

SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git status
On branch master
nothing to commit, working tree clean
```

git restore 를 조심해야하는 경우

해당 작업은 로컬에서의 작업을 되돌리는 것이기에 조심해야한다(즉, 백업 후 하는게 좋을듯)





Staging Area 작업 단계 되돌리기

root commit 이란 저장소에 커밋을 한번도 하지 않은 경우(비교 대상이 없을 경우)

- Staging Area에 반영된 파일을 Working Directory로 되돌리기 (== Unstage)
- root-commit 여부에 따라 두 가지 명령어로 나뉨
  - root-commit이 없는 경우 : `git rm --cached`
  - root-commit이 있는 경우 : `git restore --staged`

0.1 git rm --cached

git rm --cached {{파일 명}} 는 add 이후 저장을 취소할 때 사용한다.

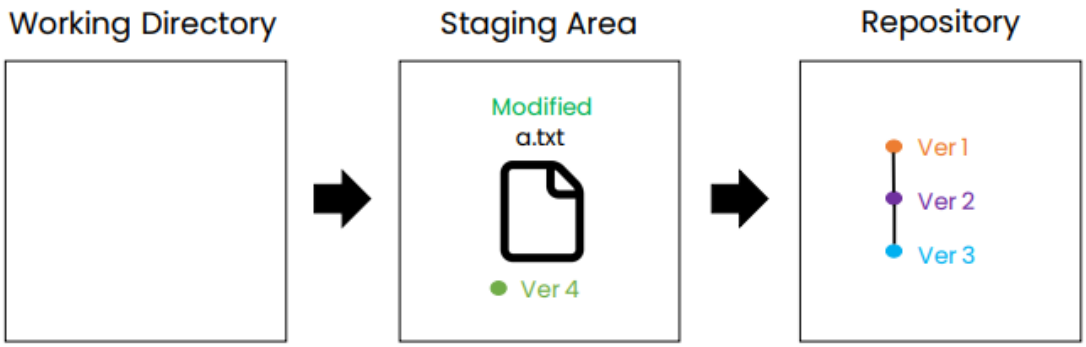
root-commit 이 없을 경우 사용

git rm --cached

- **"to unstage and remove paths only from the staging area"**
- root-commit이 없는 경우 사용(Git 저장소가 만들어지고 한 번도 커밋을 안 한 경우)
- `git rm --cached {파일 이름}`

rm으로 삭제하는 이유는 비교대상이 없기 때문에 그냥 staging Area에 올렸던 캐싱을 지우는 것. (즉 add 한 것을 지우는 행위를 의미한다.)

git restore --staged

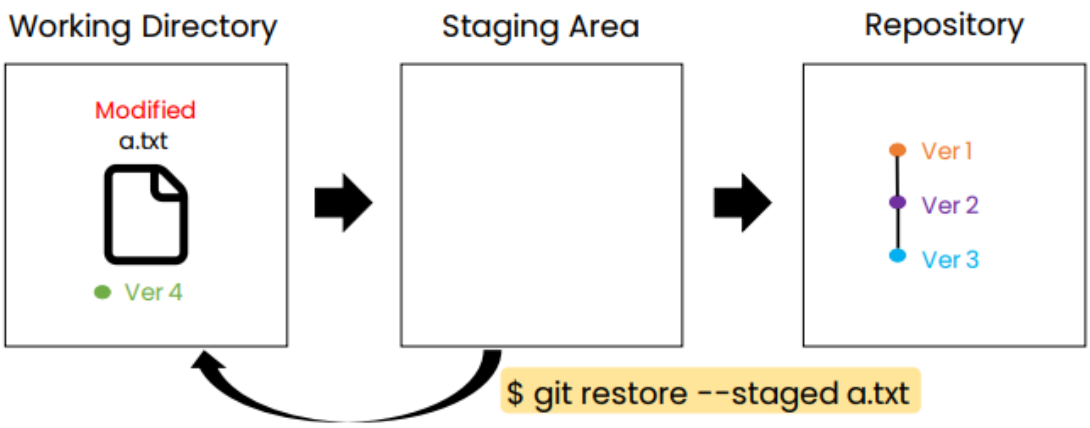


0.1 git restore --staged

git restore --staged {{파일명}} : commit 을 한 적이 있는 경우 초기 상태로 돌아가도록 하는 명령문

즉 root-commit 이 있을 경우 사용

git restore --staged



git restore --staged

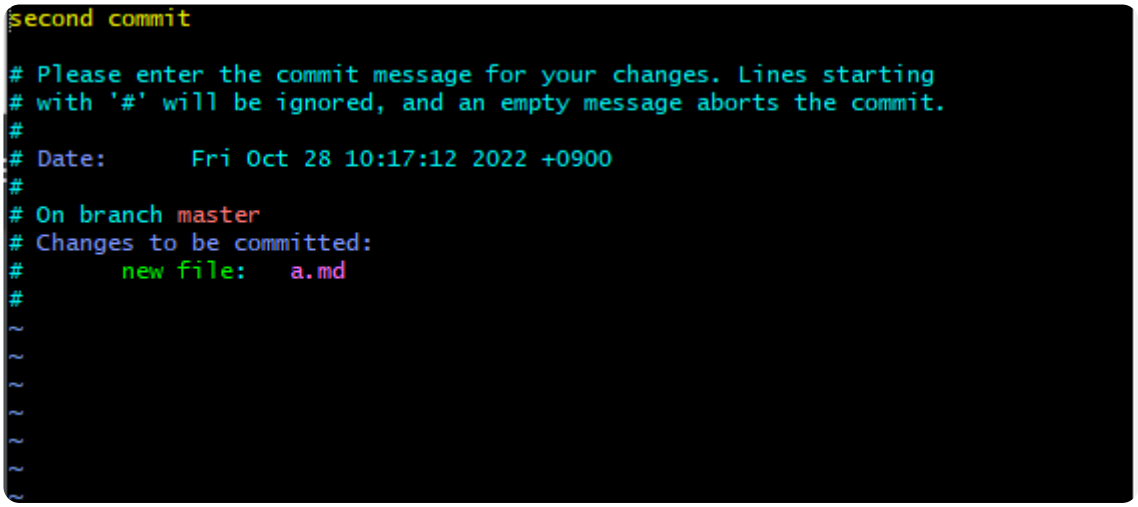
- **"the contents are restored from HEAD"**
- root-commit이 있는 경우 사용(Git 저장소에 한 개 이상의 커밋이 있는 경우)
- git restore --staged {파일 이름}
- [참고] git 2.23.0 버전 이전에는 git reset HEAD {파일 이름}

git restore --staged

- Vscode에서 git restore --staged 명령어 실습
  1. Git 저장소 초기화
  2. test.md 파일 생성 후 커밋
  3. test.md 파일 수정 후 add
  4. git restore --staged를 사용해서 Staging Area에 반영된 파일을 되돌리기

Repository 작업 단계 되돌리기

git commit --amend 를 입력하면 vim 이 열리게 된다.



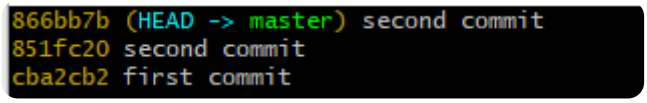
그래서 insert 혹은 i 를 입력 후 수정

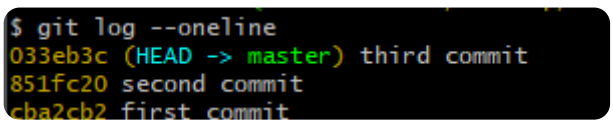
그리고 esc -> : -> wq 하게 되면 저장 후 종료하게 된다.

위와 같이 하게 되면 commit 의 명칭을 변경하게 된다.

그런데 이렇게 수정하게 되면 *커밋 명*만 바꾸는 것이 아닌

커밋 자체를 새로 하게 되는 것.





git commit --amend

- 커밋을 완료한 파일을 Staging Area로 되돌리기
- 상황 별로 두 가지 기능으로 나뉨
  - Staging Area에 새로 올라온 내용이 없다면, 직전 커밋의 메시지만 수정
  - Staging Area에 새로 올라온 내용이 있다면, 직전 커밋을 덮어쓰기
- 이전 커밋을 완전히 고쳐서 새 커밋으로 변경하므로,
  이전 커밋은 일어나지 않은 일이 되며 히스토리에도 남지 않음을 주의할 것!

`여기서 commit 을 한 이후 다른 파일을 추가했을 경우`

새로 커밋을 하게 되면 버전관리가 안되기에

이때도 git commit --amend 를 실행 후 다른 입력없이

바로 종료하게 되면 기존의 커밋 데이터와 이후 추가 데이터가 합쳐지게 된다.

---

git reset

시게를 마치 과거로돌리는 듯한 행위로, 프로젝트를 특정 커밋 상태로 되돌림

특정 커밋으로 되돌아 갔을 때, 해당 커밋 이후로 쌓았던 커밋들은 전부 사라짐

git reset [옵션] {커밋 ID}

옵션은 soft, mixed, hard 중 하나를 작성

커밋 ID는 되돌아가고 싶은 시점의 커밋 ID를 작성

즉 버전관리가 잘 안되었을 때 사용하는 것.

0.1 soft

```
$ git log --oneline
20d320d (HEAD -> master) third
1eb059e second
6baf32f first
```

위의 사진처럼 같은 .git 으로 관리되고 있는 것이라면 ID가 같기 때문에

다른 유저가 해당 파일을 가지고만 있다면 복구가 가능하다.

그래서 현재는 third commit 을 가리키고 있는데

이것을 first 로 이동하기

```
$ git reset --soft 6baf32f
```

위오 같이 입력하게 되면

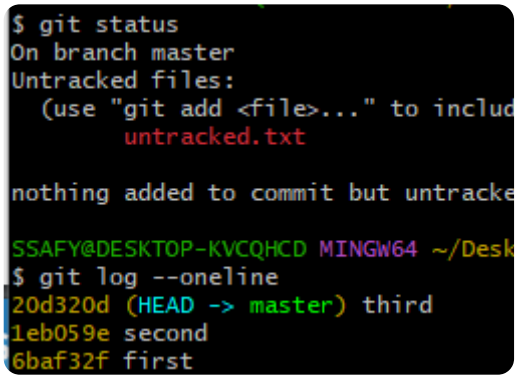
```
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   2.txt
        new file:   3.txt

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

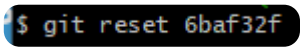
이렇게 이전 commit 으로 돌아가게 된다.

로컬의 파일도 아무것도 달라진것은 없지만

커밋에 대한 데이터만 변경되어 원하는 지점의 commit으로 돌아가게 된다.

0.1 mixed





저렇게만 적는이유는 mixed는 reset 의 기본 명령어기 때문이고

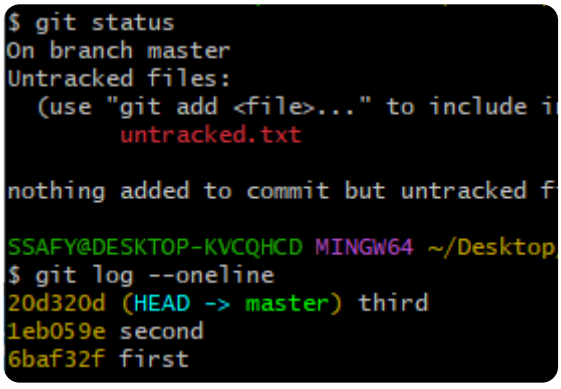
add 조차 되지 않은 상태로 만들어 주기에

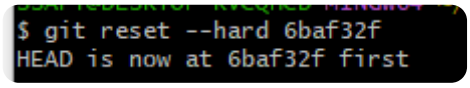
원하지 않는 내용이 들어가지 않도록 add 완료 이전으로 돌아갈 수 있다.

그래서 soft, mixed reset의 경우는 로컬의 파일 데이터에 대해선 건들이지 않고

commit 자체만 변경하게 된다.

0.1 hard

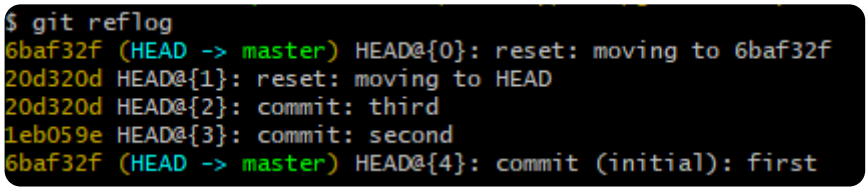




이렇게 하면 해당 commit 지점의 모든 데이터를 삭제

여기서 실수로 잘못 지웠을 경우

git reflog 를 하게 되면



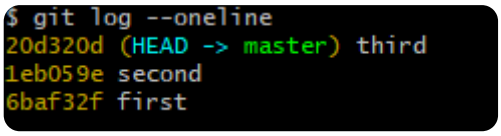
Git revert

과거를 없었던 일로 만드는 행위로, 이전 커밋을 취소한다는 새로운 커밋을 생성함

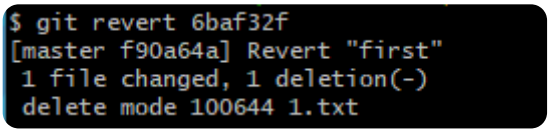
git revert {커밋 ID}

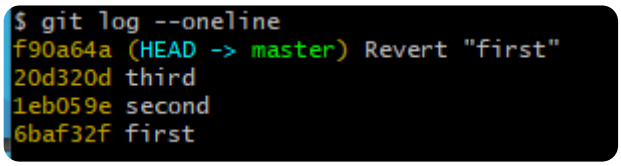
커밋 ID는 취소하고 싶은 커밋 ID를 작성

revert 를 하는 이유는 이전의 커밋을 제거한 상황을 다시 최신 commit 으로 흔적을 남긴다



git revert {{커밋 ID}} 이후 나가게 되면 아래와 같고





위와 같은 사진처럼 커밋이 지워지지 않고 해당 버전이 가장 최신 HEAD가 된다.

그래서 위의 작업은 first 커밋을 없었던 일로 만든다는 것을 의미한다.

즉 revert 를 하게 되면 지정 커밋ID 에 대한 커밋 버전을 없애버린다는 것.

여기서 한번에 없애버리는 방법은

```
$ git revert 1eb059e 20d320d
[master 6a18a2b] Revert "second"
 1 file changed, 1 deletion(-)
 delete mode 100644 2.txt
[master d166606] Revert "third"
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 3.txt
```

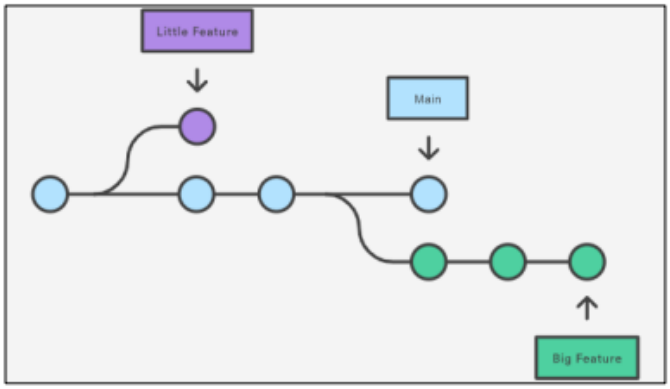
여기서 commit을 자동으로 하지 않도록 하는 방법은

git revert --no-commit {{커밋ID}}

Git branch

위에서 우리가 revert를 하고 나니 (master)에서 (master|REVERTING)로 변경이 되었다.

- 브랜치(Branch)는 나뭇가지라는 뜻으로, 여러 갈래로 작업 공간을 나누어 독립적으로 작업할 수 있도록 도와주는 Git의 도구



지금껏 교수님과 lectures 로 했다.

0.0.1 branch

- 버전관리를 용이하게 하기 위해서
- 독립 공간을 형성하기 때문에 원본에 대해 안전하다

장점

1. 브랜치는 독립 공간을 형성하기 때문에 원본(master)에 대해 안전함
2. 하나의 작업은 하나의 브랜치로 나누어 진행되므로 체계적인 개발이 가능함
3. Git은 브랜치를 만드는 속도가 굉장히 빠르고, 적은 용량을 소모함

우리가 주로 사용할 명령어로는

git branch 밖에 없긴하다.

git branch

- 브랜치의 조회, 생성, 삭제와 관련된 Git 명령어
- **조회**
  - `git branch` # 로컬 저장소의 브랜치 목록 확인
  - `git branch -r` # 원격 저장소의 브랜치 목록 확인
- **생성**
  - `git branch {브랜치 이름}` # 새로운 브랜치 생성
  - `git branch {브랜치 이름} {커밋 ID}` # 특정 커밋 기준으로 브랜치 생성
- **삭제**
  - `git branch -d {브랜치 이름}` # 병합된 브랜치만 삭제 가능
  - `git branch -D {브랜치 이름}` # 강제 삭제

git switch

- 현재 브랜치에서 다른 브랜치로 이동하는 명령어
- `git switch {브랜치 이름}` # 다른 브랜치로 이동
- `git switch -c {브랜치 이름}` # 브랜치를 새로 생성 및 이동
- `git switch -c {브랜치 이름} {커밋 ID}` # 특정 커밋 기준으로 브랜치 생성 및 이동
- switch하기 전에, 해당 브랜치의 변경 사항을 반드시 커밋 해야함을 주의 할 것!
  - 다른 브랜치에서 파일을 만들고 커밋 하지 않은 상태에서 switch를 하면 브랜치를 이동했음에도 불구하고 해당 파일이 그대로 남아있게 됨

1. 브랜치 조회

```
$ git branch -r
```

2. 브랜치 생성(커밋이 있어야 해당 작업 공간으로 부터 분화가 생길 수 있다.)

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git add .

SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git commit -m "first commit"
[master (root-commit) f1ca872] first commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 a.txt

SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git branch KHminor
```

3. 새롭게 파일을 생성 후 커밋을 하게 되면 HEAD는 master을 가리킨다.

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git commit -m "create b"
[master 4a711c4] create b
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 b.txt

SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git log --oneline
4a711c4 (HEAD -> master) create b
f1ca872 (KHminor) first commit
```

4. 다시 한번 조회

```
$ git branch
  KHminor
* master
```

5. 이후 작업공간을 변경하고 싶을 경우

```
@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
t switch KHminor
ched to branch 'KHminor'

@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
```

(master) -> (KHminor) 로 변경되었다.

그리고 로그를 찍어보면 master에서 변경된 사항이 나타아지 않는다.(b.txt)

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git log --oneline
f1ca872 (HEAD -> KHminor) first commit
```

6. 이후 KHminor에서 파일을 생성후 커밋을 하고

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git log --oneline
1228493 (HEAD -> KHminor) create c
f1ca872 first commit
```

7. switch 하게 되면

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git log --oneline
4a711c4 (HEAD -> master) create b
f1ca872 first commit
```

KHminor에서 커밋한거는 log가 찍히지 않는다.

만약 브랜치를 바꾸지 않고 master에서 작업을 했다면 커밋하기 전 상태로 돌려놓고 브랜치를 바꾸면 된다.

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git log --oneline
1c14d6b (HEAD -> KHminor) create d
1228493 create c
f1ca872 first commit
```

8. 이후 master 작업과 KHminor의 작업을 합치고 싶을 경우(대상으로 switch) master은 병합이 되어진 것이고 KHminor은 그냥 그대로이다.

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git merge KHminor
Merge made by the 'ort' strategy.
 c.txt | 0
 d.txt | 0
 2 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 c.txt
 create mode 100644 d.txt
```

위와 같이 하게 되면

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git log --oneline
f02b791 (HEAD -> master) Merge branch 'KHminor'
1c14d6b (KHminor) create d
1228493 create c
4a711c4 create b
f1ca872 first commit
```

9. 해당 과정을 그래프로 본다면

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git log --oneline --graph
*   f02b791 (HEAD -> master) Merge branch 'KHminor'
|\
| * 1c14d6b (KHminor) create d
| * 1228493 create c
* | 4a711c4 create b
|/
* f1ca872 first commit
```

Git merge

git merge

- 분기된 브랜치(Branch)들을 하나로 합치는 명령어
- master 브랜치가 상용이므로, 주로 master 브랜치에 병합
- git merge {합칠 브랜치 이름}
  - 병합하기 전에 브랜치를 합치려고 하는, 즉 메인 브랜치로 switch 해야함
  - 병합에는 세 종류가 존재
    1. Fast-Forward
    2. 3-way Merge
    3. Merge Conflict

---

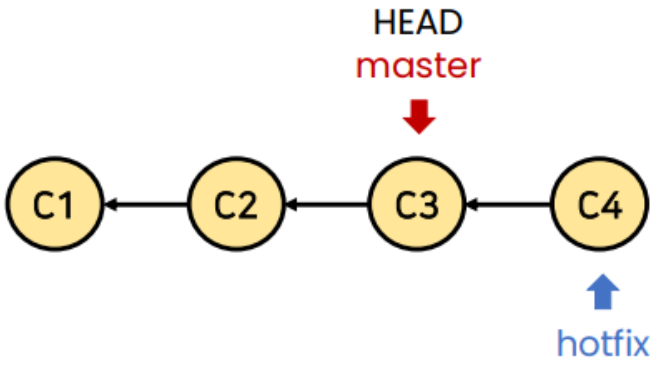
1. Fast Forward

가져와서 그냥 냅다 합쳐주는 것?

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git merge KHminor
Updating 64b19ac..9a763ed
Fast-forward
 a.txt | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

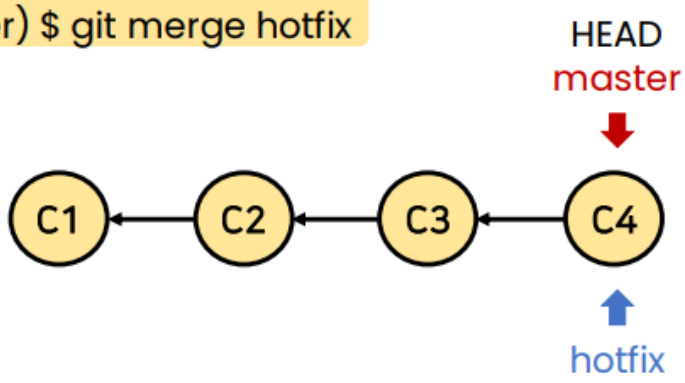
1. Fast-Forward (1/3)

- 마치 빨리감기처럼 브랜치가 가리키는 커밋을 앞으로 이동시키는 방법



1. Fast-Forward (2/3)

- 마치 빨리감기처럼 브랜치가 가리키는 커밋을 앞으로 이동시키는 방법
- (master) $ git merge hotfix



1. Fast-Forward (3/3)

- 마치 빨리감기처럼 브랜치가 가리키는 커밋을 앞으로 이동시키는 방법
- (master) $ git merge hotfix

HEAD
master

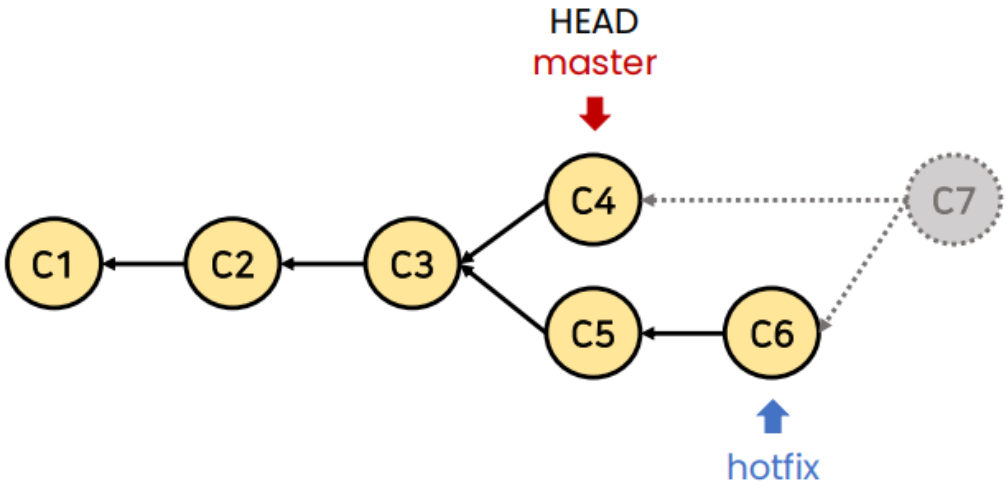
```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ git merge hotfix
Updating f9bd642..e2edd64
Fast-forward
 a.txt | 3 ++-
 1 file changed, 2 insertions(+), 1 deletion(-)
```

hotfix

3. 1.1 Merge Conflict

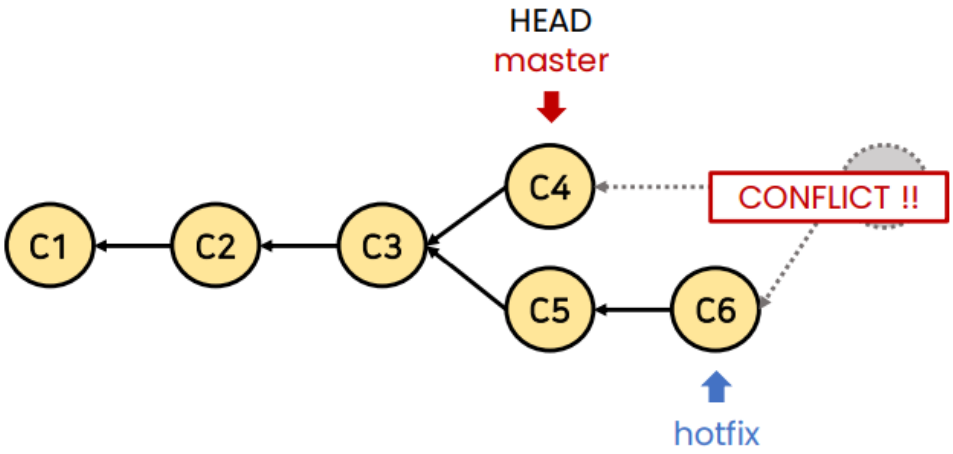
1.
   - 두 브랜치에서 같은 부분을 수정한 경우,
     Git이 어느 브랜치의 내용으로 작성해야 하는지 판단하지 못하여
     충돌(Conflict)이 발생했을 때 이를 해결하며 병합하는 방법
   - 보통 같은 파일의 같은 부분을 수정했을 때 자주 발생함

- (master) $ git merge hotfix



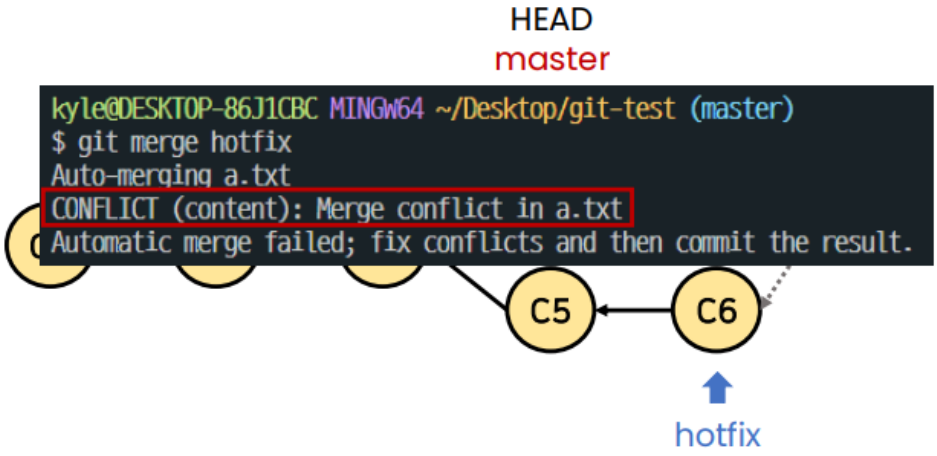
3. Merge Conflict (3/7)

- master 브랜치와 hotfix 브랜치에서 같은 파일의 같은 부분을 수정하여 충돌 발생



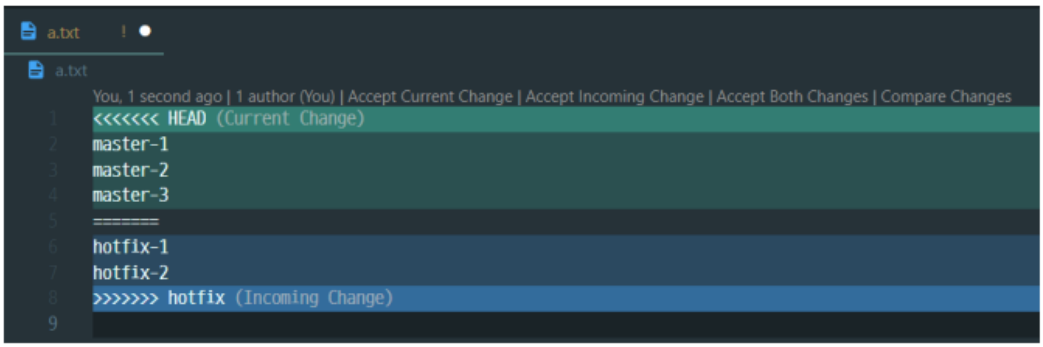
3. Merge Conflict (4/7)

- master 브랜치와 hotfix 브랜치에서 같은 파일의 같은 부분을 수정하여 충돌 발생



3. Merge Conflict (5/7)

- 충돌이 발생한 부분은 작성자가 직접 해결 해야함



1. 마스터에서 데이터를 수정후 커밋

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git commit -m "im master"
[master 64b19ac] im master
 1 file changed, 1 insertion(+)

SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (master)
$ git log --oneline --graph
* 64b19ac (HEAD -> master) im master
*   f02b791 Merge branch 'KHminor'
|\
| * 1c14d6b (KHminor) create d
| * 1228493 create c
* | 4a711c4 create b
|/
* f1ca872 first commit
```

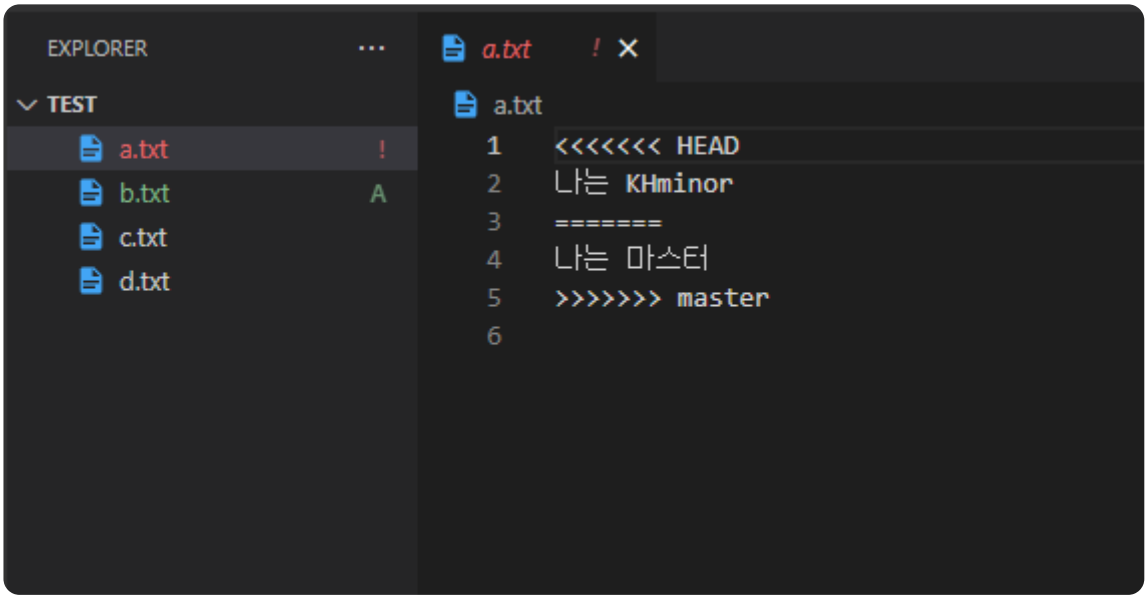
2. 이후 스위치 하여 다른 브랜치에서 같은 파일을 수정 후 커밋

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git log --oneline
446079f (HEAD -> KHminor) im KHminor
1c14d6b create d
1228493 create c
f1ca872 first commit
```

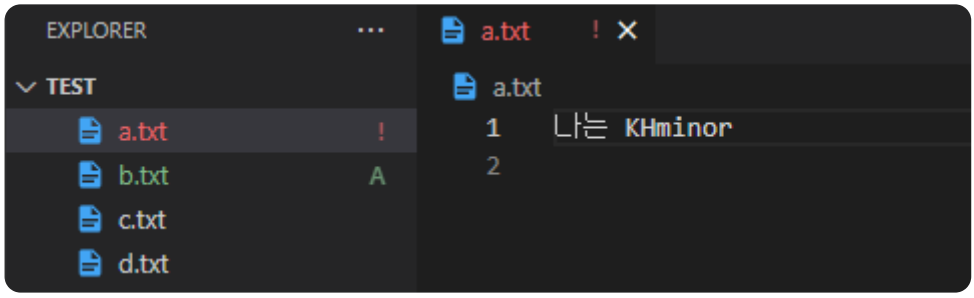
3. 이후 KHminor을 master과 병합을 하려고 하는데

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git merge master
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
```

4. 해당 파일인 a.txt 를 텍스트 에디터로 열어보게 되면



5. 현재 KHmior 의 데스크이기에 위에 텍스트로 먼저 나오게 되고
   merge 할 master은 아래에 나오게 된다.
   그래서 수정하고 나면 문제가 없이 해결이 된다.



6. status를 찍으면

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor|MERGING)
$ git status
On branch KHminor
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   b.txt

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   a.txt
```

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git log --oneline
9a763ed (HEAD -> KHminor) fix conflict
446079f im KHminor
64b19ac (master) im master
f02b791 Merge branch 'KHminor'
1c14d6b create d
1228493 create c
4a711c4 create b
f1ca872 first commit
```

```
SSAFY@DESKTOP-KVCQHCD MINGW64 ~/Desktop/test (KHminor)
$ git log --oneline --graph
*   9a763ed (HEAD -> KHminor) fix conflict
|\
| * 64b19ac (master) im master
| *   f02b791 Merge branch 'KHminor'
| |\
| * | 4a711c4 create b
* | | 446079f im KHminor
| |/
|/|
* | 1c14d6b create d
* | 1228493 create c
|/
* f1ca872 first commit
```

그래프를 보면 호출한 기준으로 부터 일직선으로 작업이 되기에 3줄이 되었다.

일반적으로 merge 하는 시점은 해당 과정이 마쳤을때 하게 된다??

merge 순서(대략적 순서?):

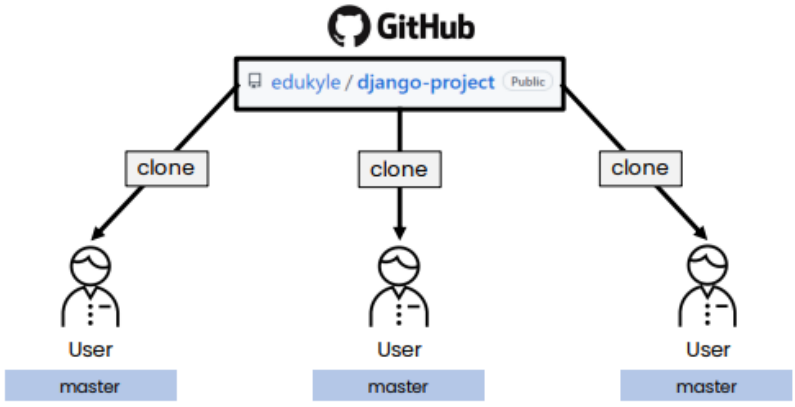
1. master repo 에서 branch 를 생성 후
2. 커밋 , 풀 완료 후
3. 깃 웹 사이트에서 등록 한 후
4. master repo에서 다 승인을 하여 당겨온 후? (create merge request)
5. master 입장에서 pull 한 다음
6. branch에서 git merge master 하게 되면 모두 가져오게 된다.

Shared repository model

- 원격 저장소가 자신의 소유이거나 Collaborator로 등록되어 있는 경우
- master 브랜치에 직접 개발하는 것이 아니라, 기능별로 브랜치를 따로 만들어 개발
- Pull Request를 사용하여 팀원 간 변경 내용에 대한 소통 진행

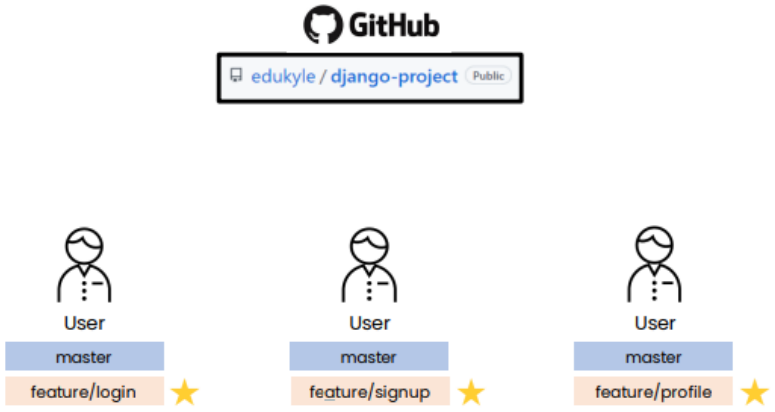
따라하기 (1/10)

- 소유권이 있는 원격 저장소를 로컬 저장소로 clone 받기



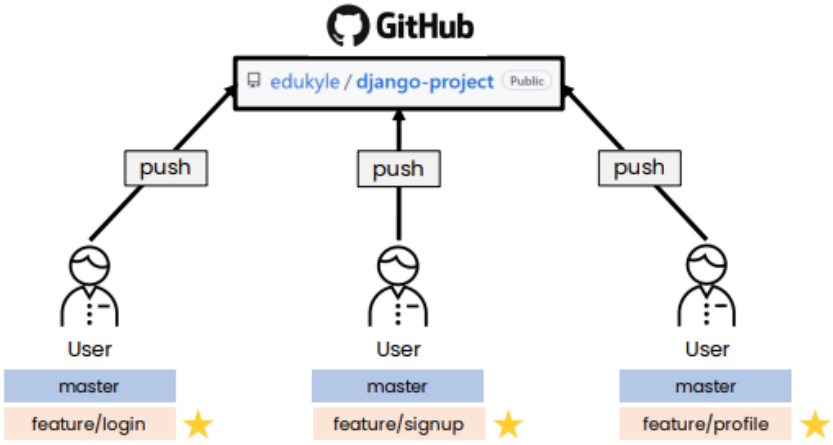
따라하기 (2/10)

- 사용자는 자신이 작업할 기능에 대한 브랜치를 생성하고, 그 안에서 기능을 구현



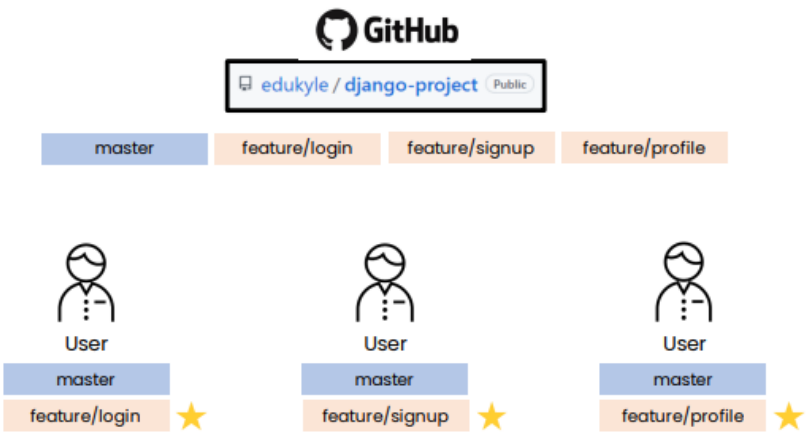
따라하기 (3/10)

- 기능 구현이 완료되면, 원격 저장소에 해당 브랜치를 Push



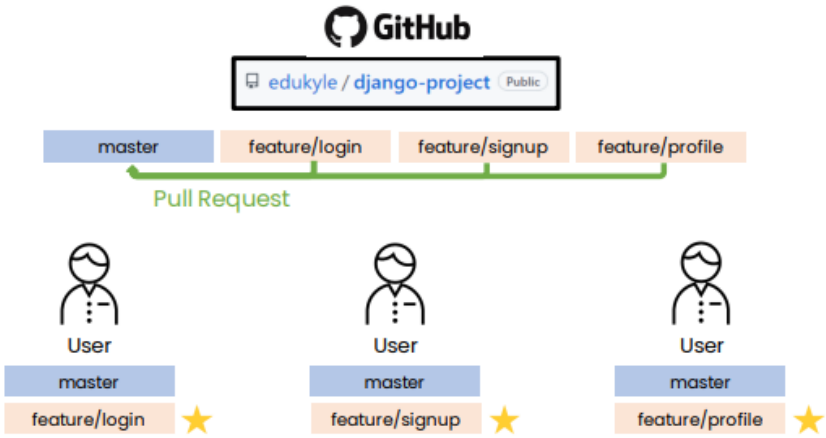
따라하기 (4/10)

- 원격 저장소에 각 기능의 브랜치가 반영됨



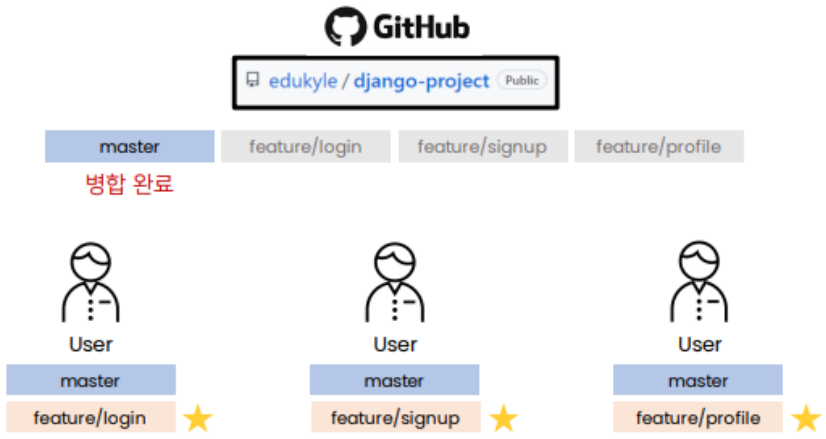
따라하기 (5/10)

- Pull Request를 통해 브랜치를 master에 반영해달라는 요청을 보냄



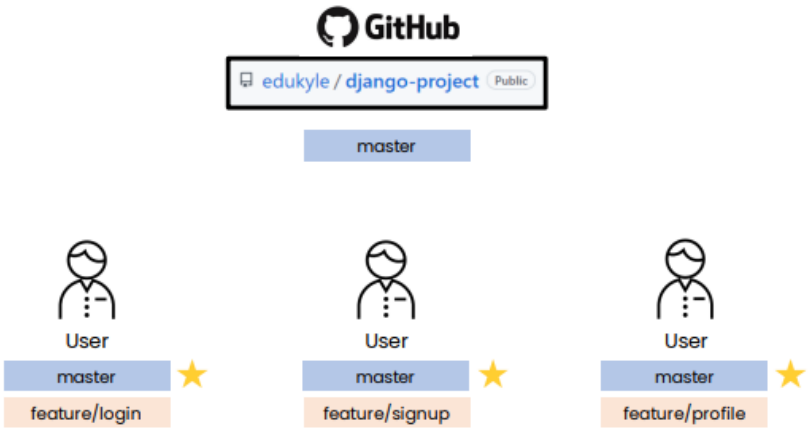
따라하기 (6/10)

- 병합이 완료된 브랜치는 불필요하므로 원격 저장소에서 삭제



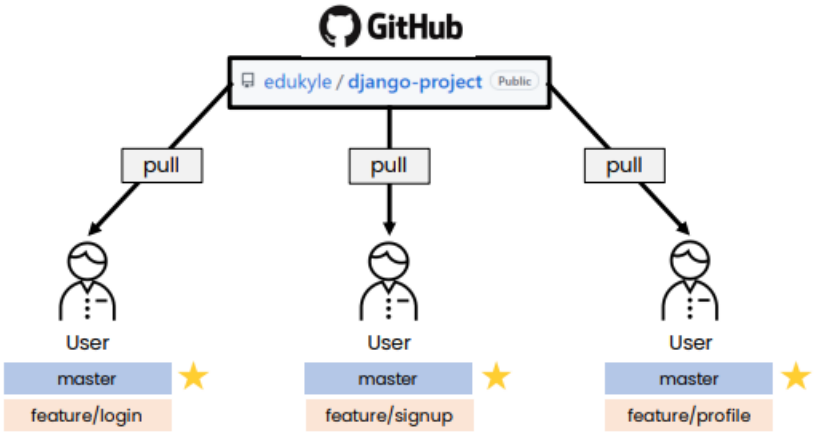
따라하기 (7/10)

- 원격 저장소에서 병합이 완료되면, 사용자는 로컬에서 master 브랜치로 switch



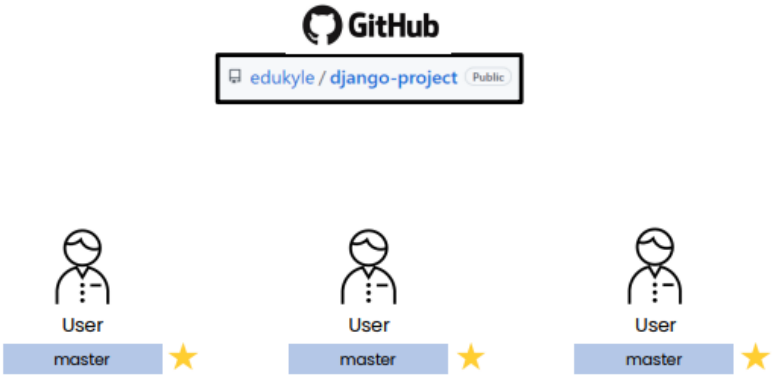
따라하기 (8/10)

- 병합으로 인해 변경된 원격 저장소의 master 내용을 로컬에 Pull



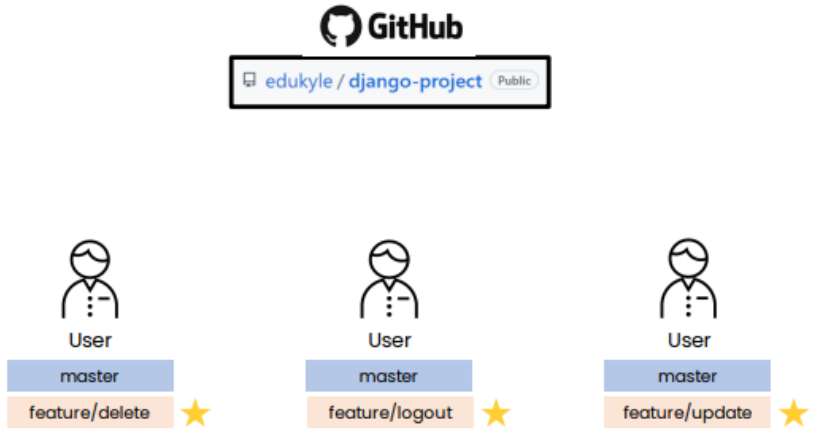
따라하기 (9/10)

- 원격 저장소 master의 내용을 받았으므로, 기존 로컬 브랜치 삭제 (한 사이클 종료)



따라하기 (10/10)

- 새 기능 추가를 위해 새로운 브랜치를 생성하며 지금까지의 과정을 반복



Fork & Pull model (깃헙에서 실습했음)

Shared repository model 이거와 거의 유사.

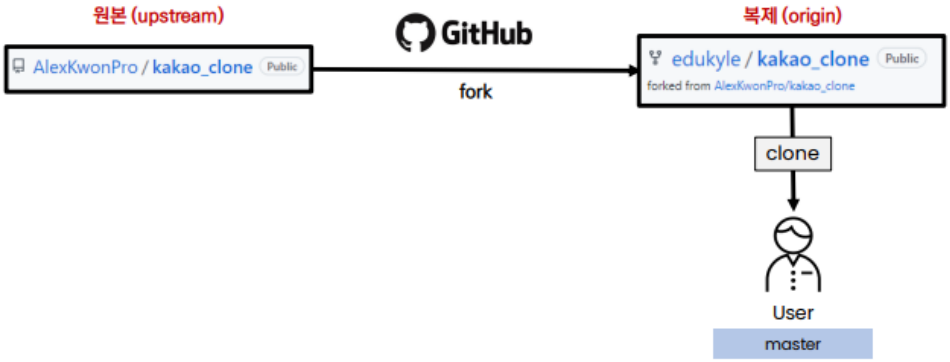
github 에서 Fork 만 해주면 됨

따라하기 (1/12)

- 소유권이 없는 원격 저장소를 fork를 통해 내 원격 저장소로 복제

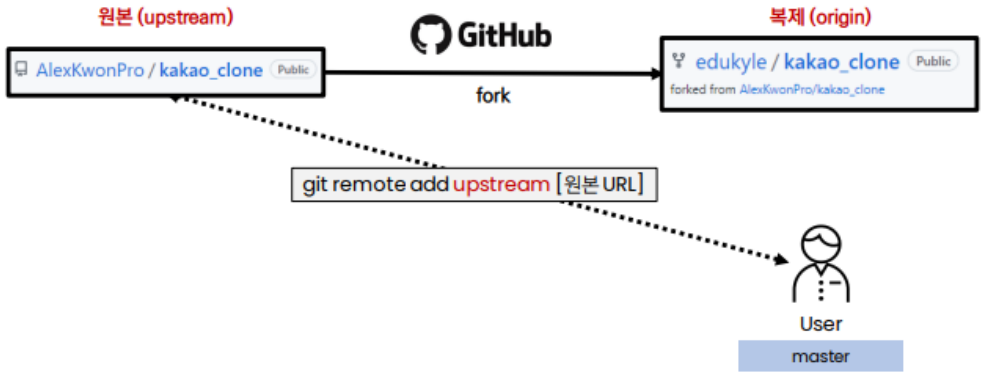
따라하기 (2/12)

- fork 이후, 복제된 내 원격 저장소를 로컬 저장소에 clone



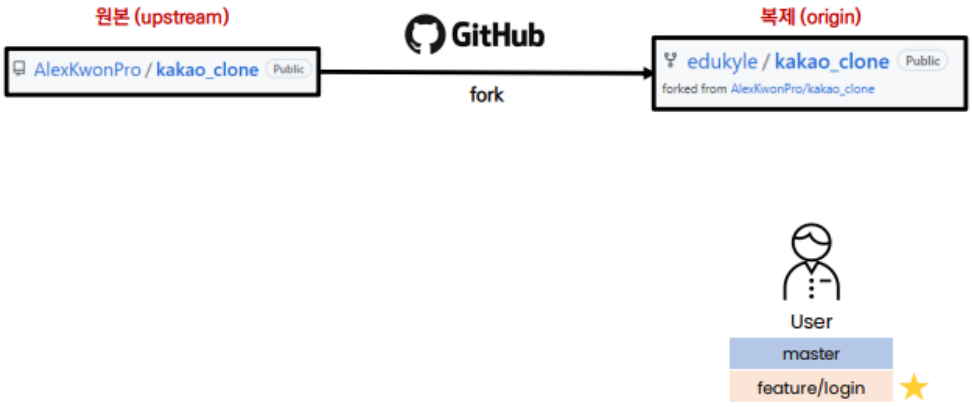
따라하기 (3/12)

- 이후에 로컬 저장소와 원본 원격 저장소를 동기화 하기 위해 연결



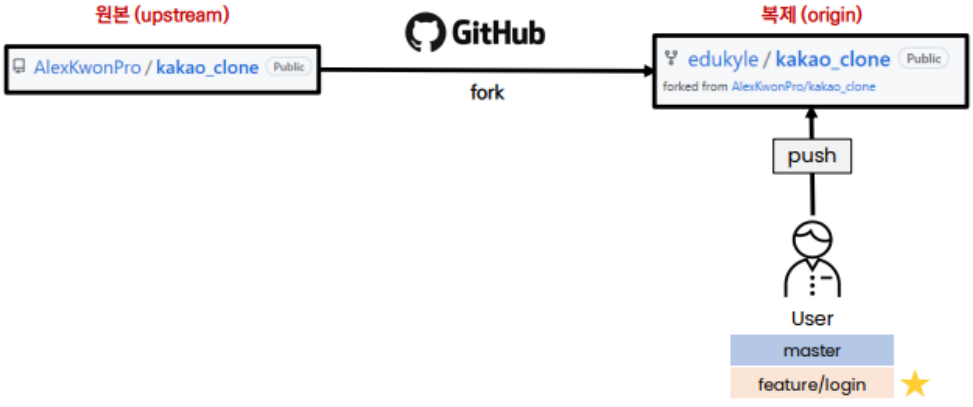
따라하기 (4/12)

- 사용자는 자신이 작업할 기능에 대한 브랜치를 생성하고, 그 안에서 기능을 구현



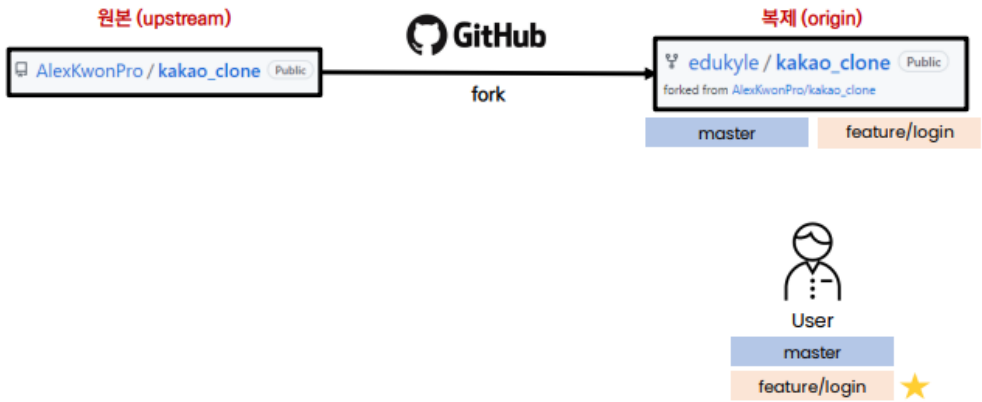
따라하기 (5/12)

- 기능 구현이 완료되면, 복제 원격 저장소(origin)에 해당 브랜치를 Push



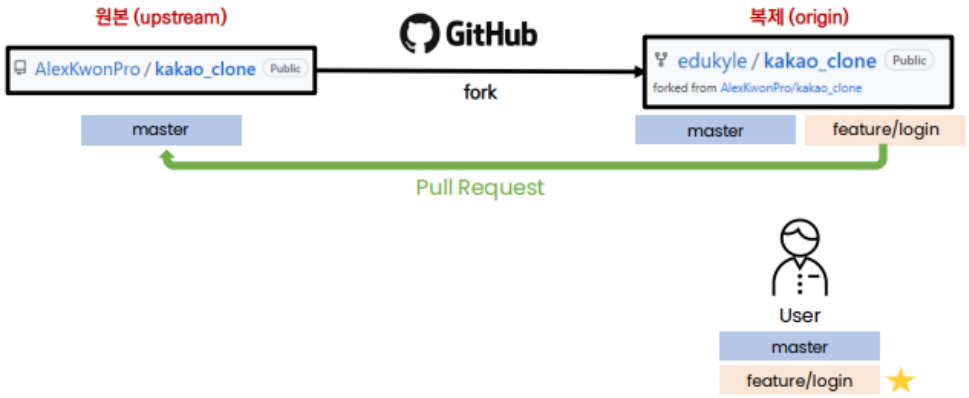
따라하기 (6/12)

- 복제 원격 저장소(origin)에 브랜치가 반영됨



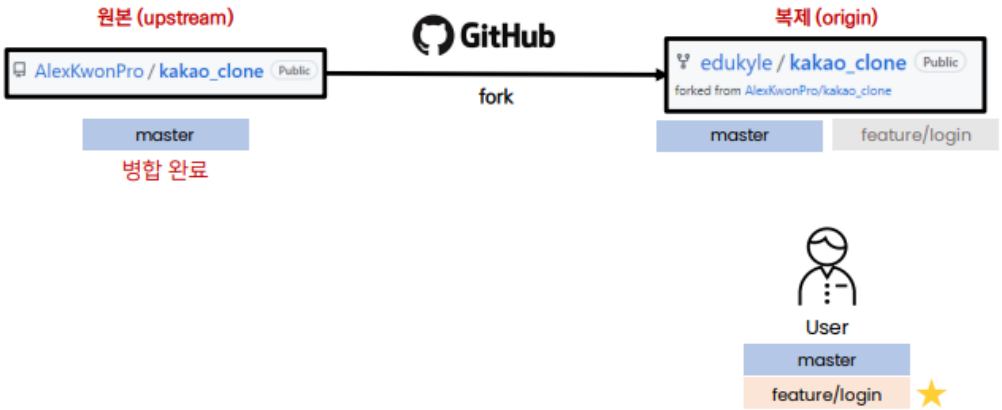
따라하기 (7/12)

- Pull Request를 통해 origin의 브랜치를 upstream에 반영해달라는 요청을 보냄



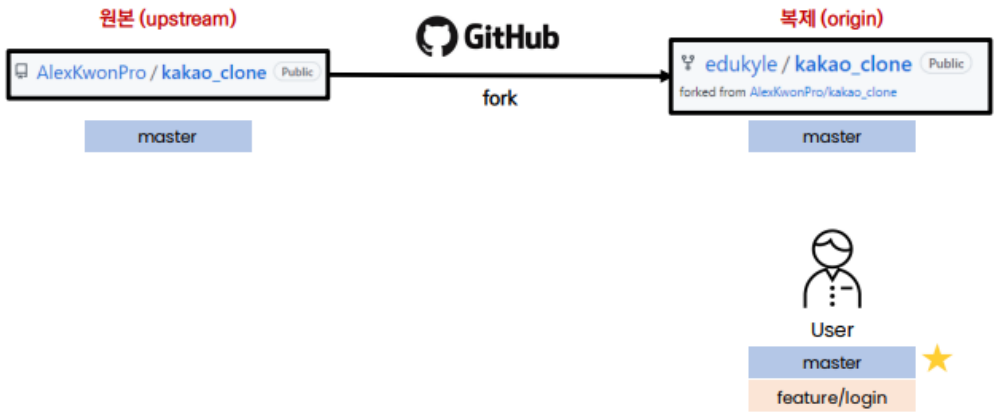
따라하기 (8/12)

- upstream에 브랜치가 병합되면 origin의 브랜치는 삭제



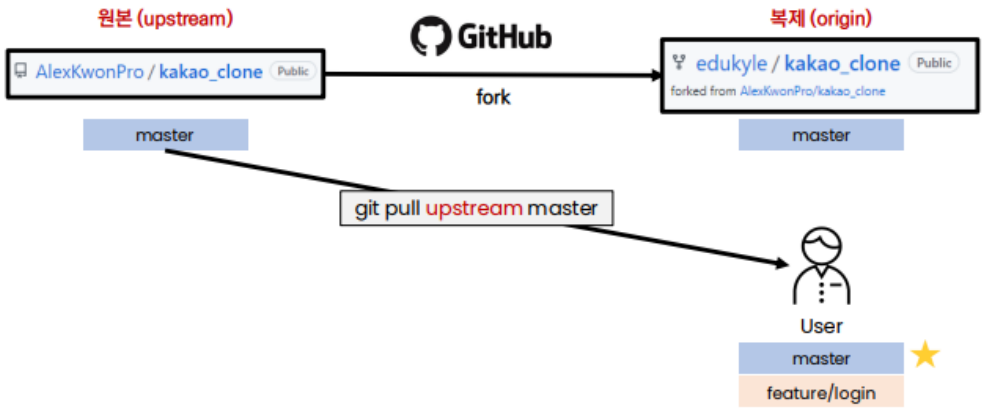
따라하기 (9/12)

- 이후 사용자는 로컬에서 master 브랜치로 switch



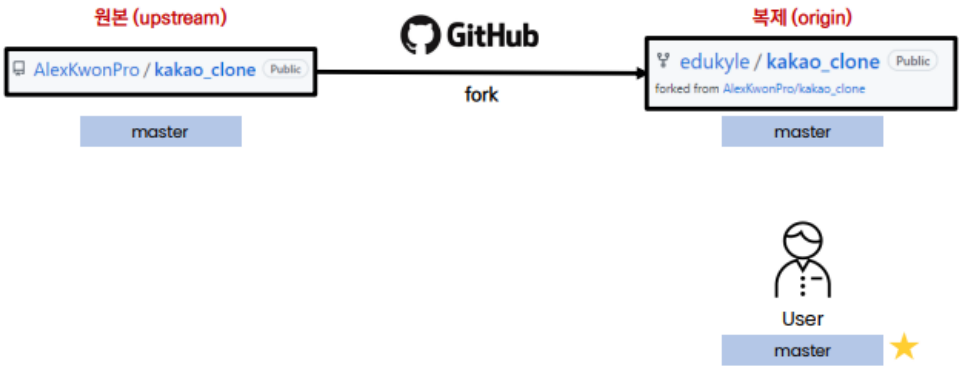
따라하기 (10/12)

- 병합으로 인해 변경된 upstream의 master 내용을 로컬에 Pull



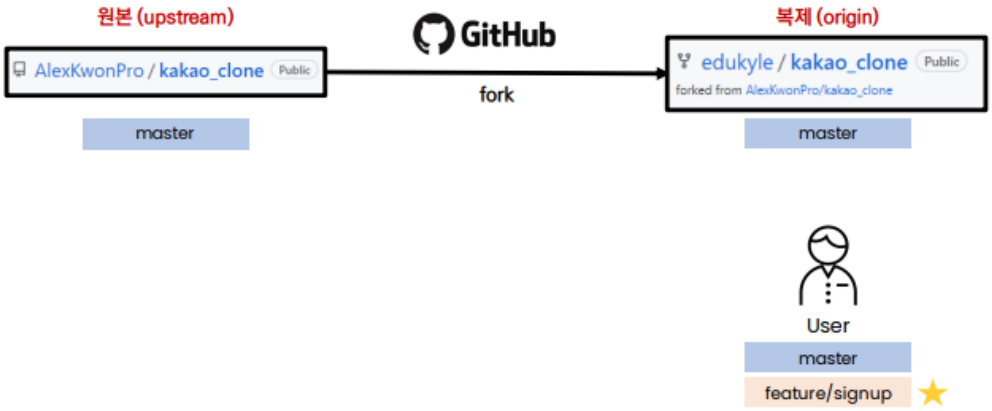
따라하기 (11/12)

- upstream의 master 내용을 받았으므로, 기존 로컬 브랜치 삭제 (한 사이클 종료)



따라하기 (12/12)

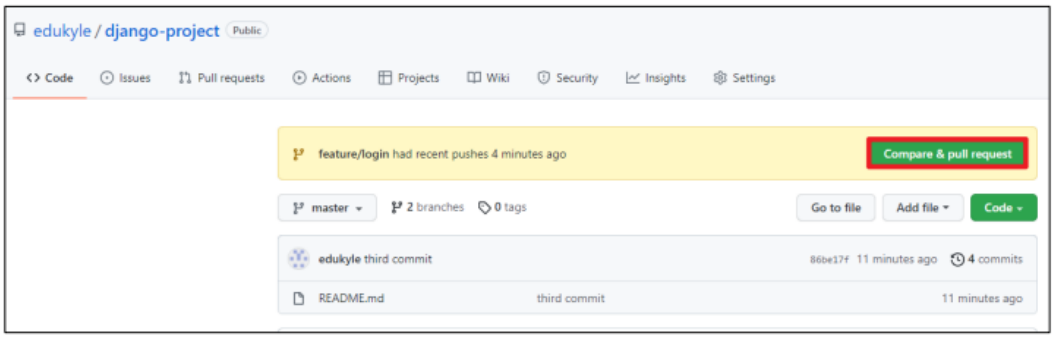
- 새로운 기능 추가를 위해 새로운 브랜치를 생성하며 위 과정을 반복



PR (Pull Reqeust)

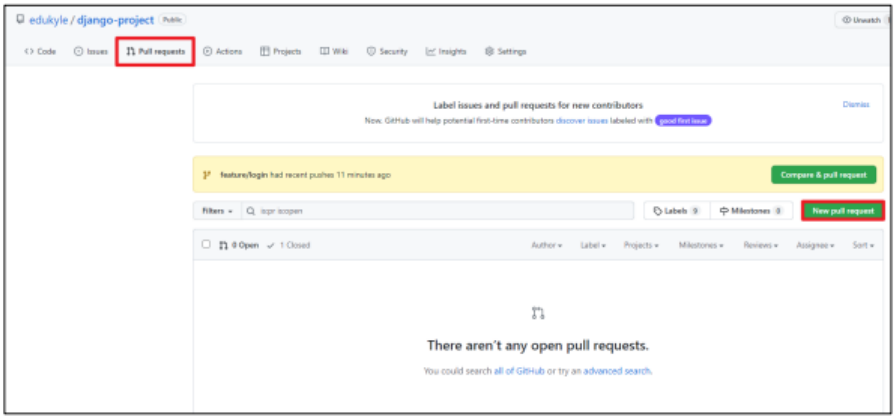
Github에서 Pull Request 보내기 (1/10)

- 브랜치를 Push 했을 때 나타나는 Compare & pull request 버튼을 클릭



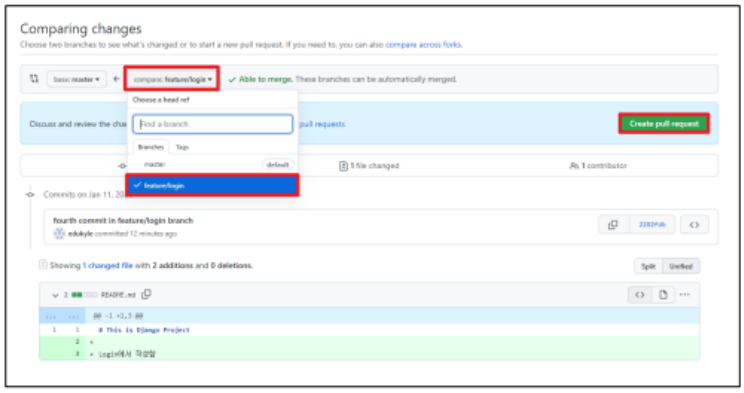
Github에서 Pull Request 보내기 (2/10)

- 혹은 상단 바의 Pull requests → New pull request을 통해서도 가능



Github에서 Pull Request 보내기 (3/10)

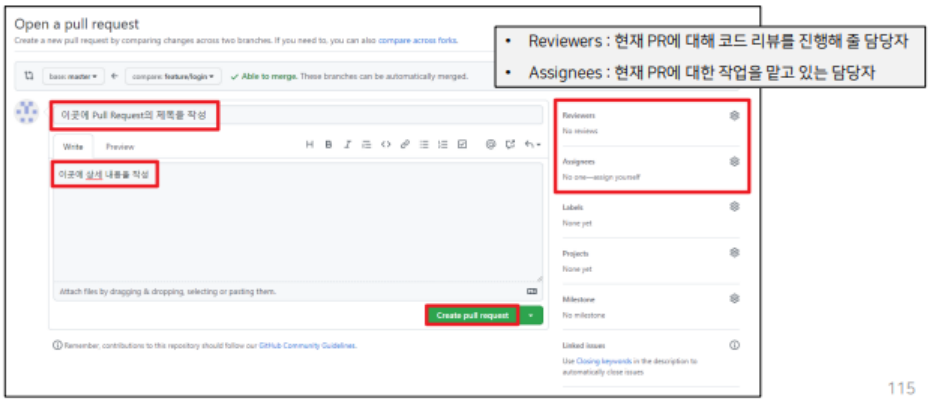
- 병합될 대상인 base는 master 브랜치로 설정
- 병합할 대상인 compare는 feature/login 브랜치로 설정



114

Github에서 Pull Request 보내기 (4/10)

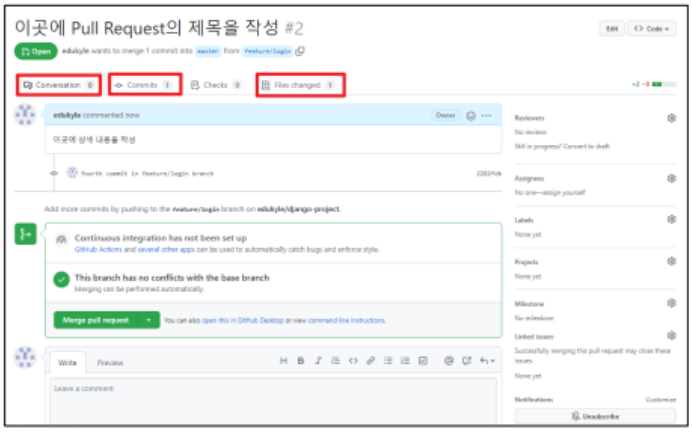
- Pull Request에 대한 제목과 내용, 각종 담당자를 지정하는 페이지
- 모두 작성했다면 Create pull request를 눌러서 PR을 생성



115

Github에서 Pull Request 보내기 (5/10)

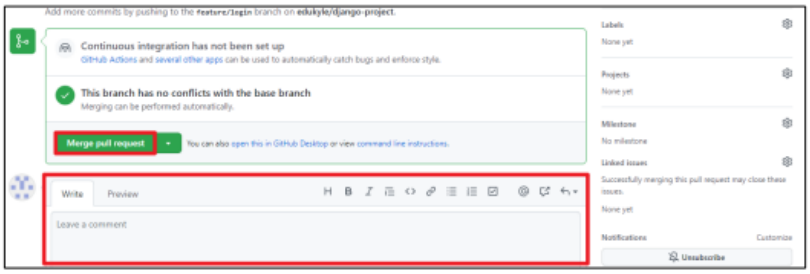
- PR이 생성되면 Conversation, Commits, Files changed 화면을 확인 가능



116

Github에서 Pull Request 보내기 (5/10)

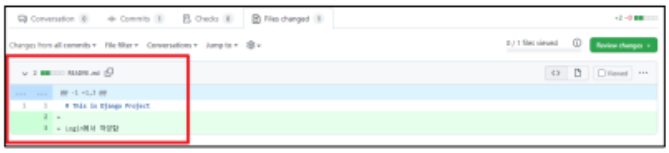
1. Conversation
   - 아래 Write 부분에서 별도로 comment를 작성할 수 있음
   - Merge pull request 버튼을 누르면 병합 시작
   - 충돌(conflict) 상황에서는 충돌을 해결하라고 나타남



Github에서 Pull Request 보내기 (5/10)

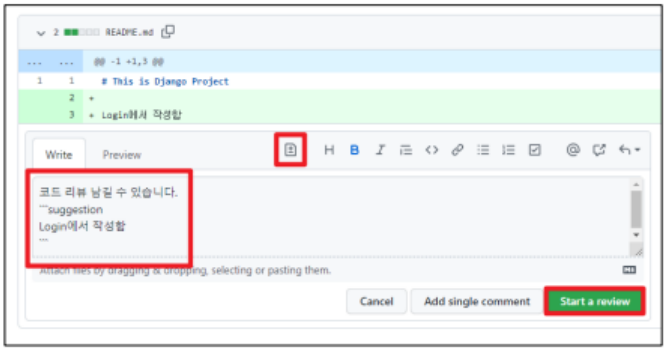
2. Commits
   - PR을 통해 반영될 커밋들을 볼 수 있음

3. Files changed
   - 파일의 변화 내역들을 볼 수 있음



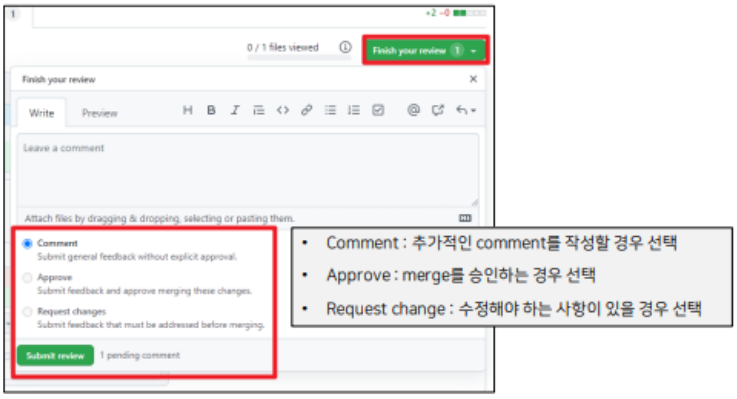
Github에서 Pull Request 보내기 (6/10)

- 코드리뷰를 원하는 라인에서 + 를 눌러서 해당 라인에 리뷰를 남길 수 있음
- 빨간 사각형으로 표시된 작은 아이콘을 클릭하면, suggestion 기능(코드를 이렇게 바꾸라고 추천하는 기능)을 넣을 수도 있음



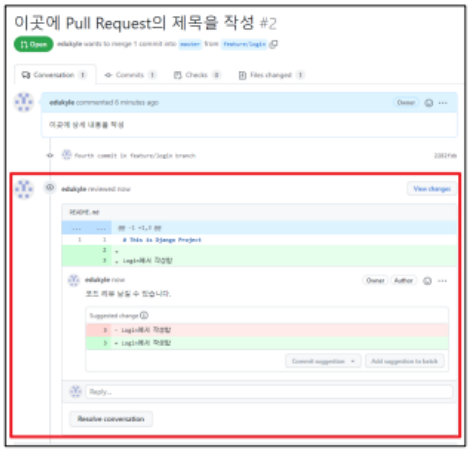
Github에서 Pull Request 보내기 (7/10)

- 코드 리뷰를 끝내려면 Finish your review 버튼을 클릭
- 그리고 옵션을 선택한 후 Submit review를 클릭



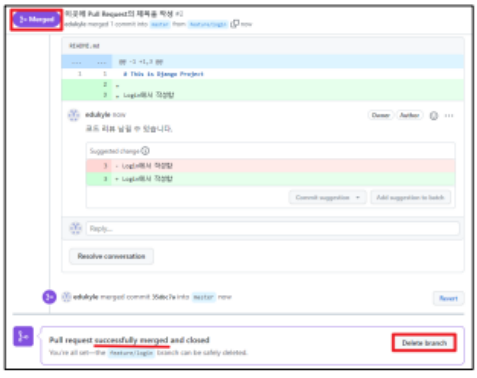
Github에서 Pull Request 보내기 (8/10)

- 다시 conversation으로 가보면 진행했던 리뷰가 나타난 것을 확인 가능



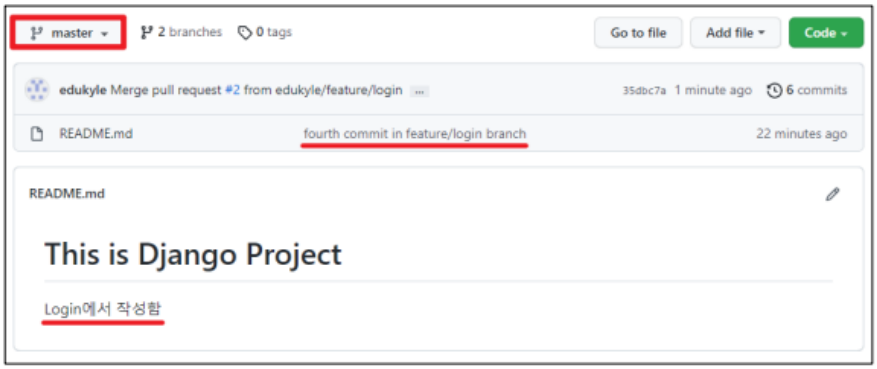
Github에서 Pull Request 보내기 (9/10)

- 병합을 하게 되면 아래와 같이 보라색으로 병합이 완료되었다고 나오면 성공
- Delete branch 버튼을 통해 병합된 feature/login 브랜치 삭제 가능 (원격 저장소에서만 지워짐)



Github에서 Pull Request 보내기 (10/10)

- master 브랜치를 선택하여 feature/login의 내용이 master에 병합된 결과를 확인
- 이후 로컬 저장소의 master 브랜치에서 git pull을 이용해 로컬과 원격을 동기화 해야함



Git 브랜치 전략

Git 브랜치 전략

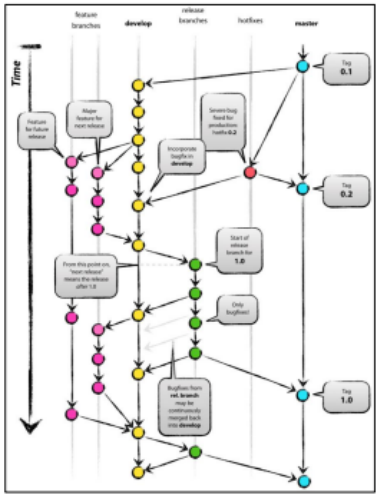
개요

- "the strategy that software development teams adopt when writing, merging and deploying code when using a version control system."
- 여러 개발자가 하나의 레포지토리를 사용하는 환경에서 변경 내용의 충돌을 줄이고 협업을 효율적으로 하고자 만들어진 브랜치 생성 규칙 혹은 방법론
- 대표적인 예시로는 git-flow, github-flow, gitlab-flow가 있음

Git 브랜치 전략

git-flow

- 2010년 Vincent Driessen이 제안한 git 브랜치 전략
- 아래와 같이 5개의 브랜치로 나누어 소스코드를 관리
  - master : 제품으로 출시될 수 있는 브랜치
  - develop : 다음 출시 버전을 개발하는 브랜치
  - feature : 기능을 개발하는 브랜치
  - release : 이번 출시 버전을 준비하는 브랜치
  - hotfix : 출시 버전에서 발생한 버그를 수정 하는 브랜치
- 대규모 프로젝트에 적합한 브랜치 전략



개발 과정에서는 develop 에서 작업하기