

Git Advanced

INDEX

- Git undoing
- Git reset & revert
- Git branch & merge
- Git workflow

Git undoing

| 개요

- Git 작업 되돌리기(Undoing)
- Git에서 되돌리기는 작업 상태에 따라 크게 세 가지로 분류
 - Working Directory 작업 단계
 - Staging Area 작업 단계
 - Repository 작업 단계

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`

Working Directory 작업 단계 되돌리기

Working Directory 작업 단계 되돌리기

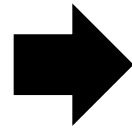
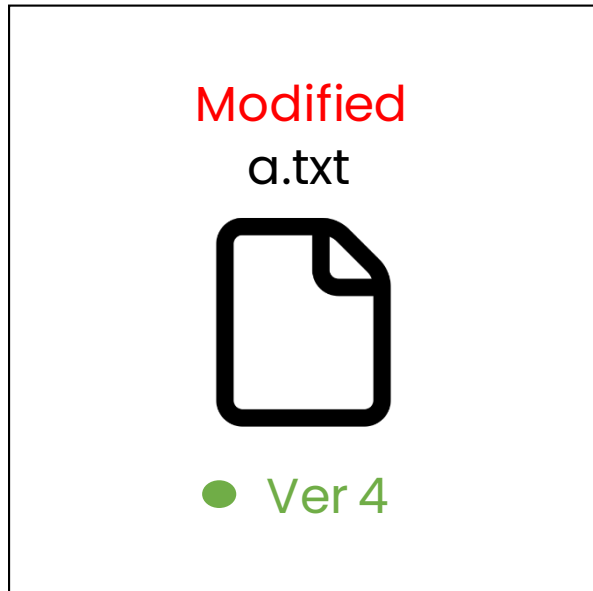
| git restore

- Working Directory에서 수정한 파일을 수정 전(직전 커밋)으로 되돌리기
- 이미 버전 관리가 되고 있는 파일만 되돌리기 가능
- git restore를 통해 되돌리면, 해당 내용을 복원할 수 없으니 주의할 것!
- git restore {파일 이름}
- [참고] git 2.23.0 버전 이전에는 git checkout -- {파일 이름}

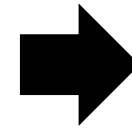
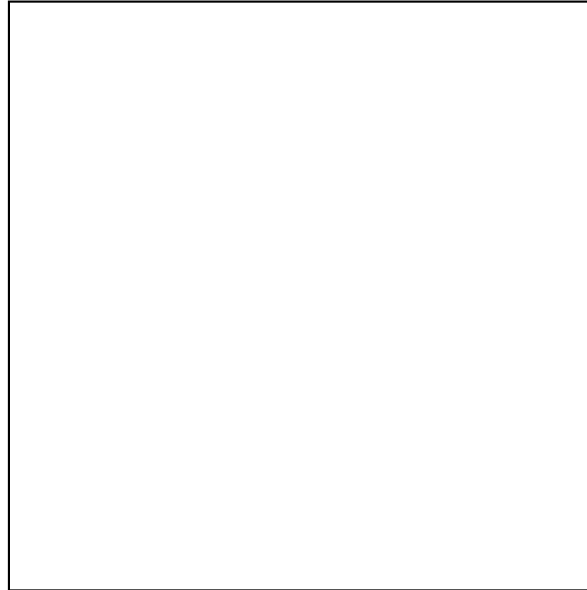
Working Directory 작업 단계 되돌리기

| git restore

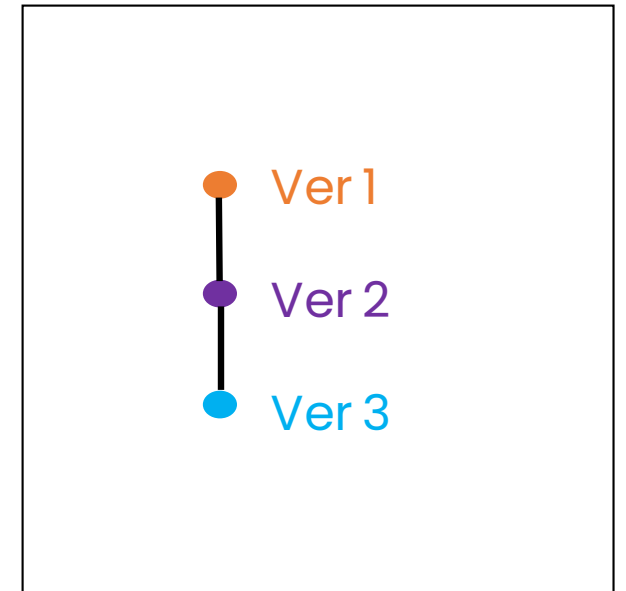
Working Directory



Staging Area



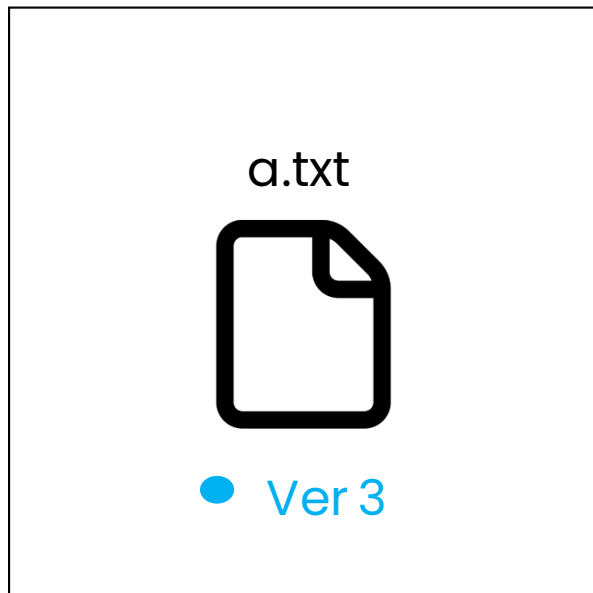
Repository



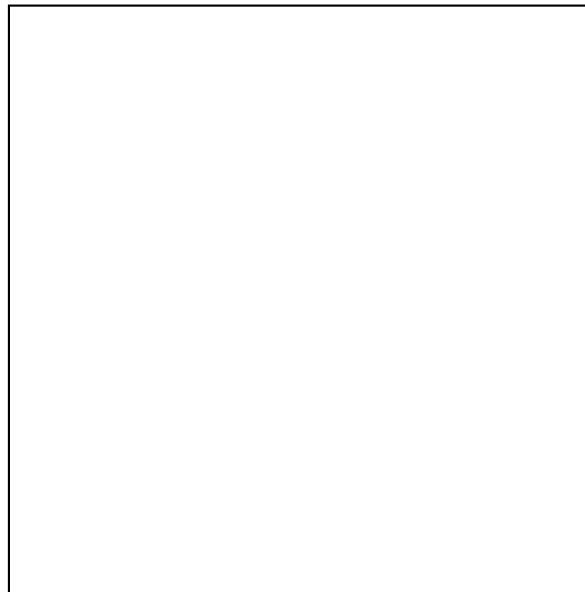
Working Directory 작업 단계 되돌리기

git restore

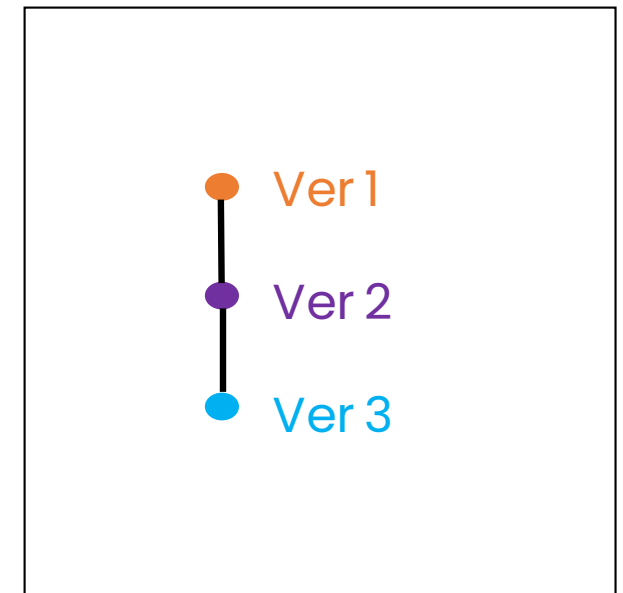
Working Directory



Staging Area



Repository



```
$ git restore a.txt
```

a.txt 파일을 수정 전(직전 커밋)으로 되돌리기

Working Directory 작업 단계 되돌리기

| git restore

- Vscode에서 git restore 명령어 실습
 1. Git 저장소 초기화
 2. test.md 파일 생성 후 커밋
 3. Working Directory에서 test.md 파일 수정
 4. git restore를 사용해서 test.md 파일을 수정 전으로 되돌리기

Staging Area

작업 단계 되돌리기

Staging Area 작업 단계 되돌리기

| 개요

- 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`

Staging Area 작업 단계 되돌리기

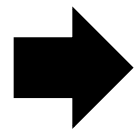
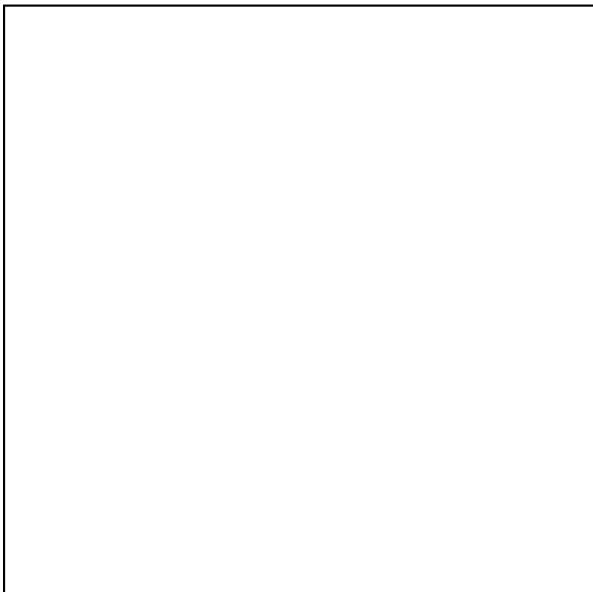
| git rm --cached

- **“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}`

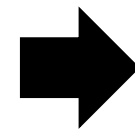
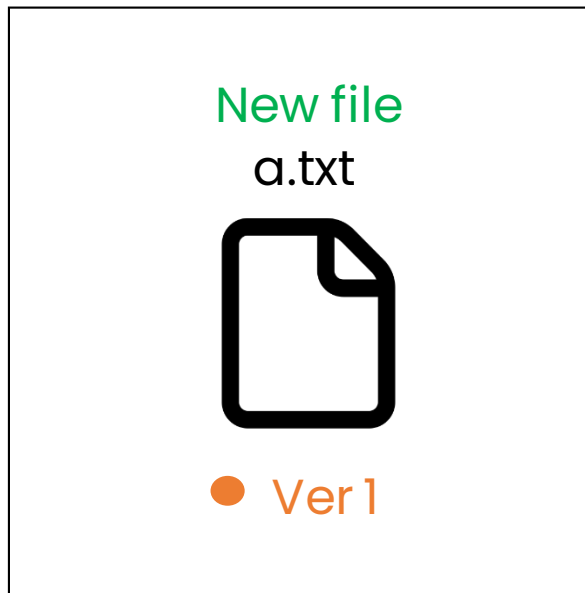
Staging Area 작업 단계 되돌리기

| `git rm --cached`

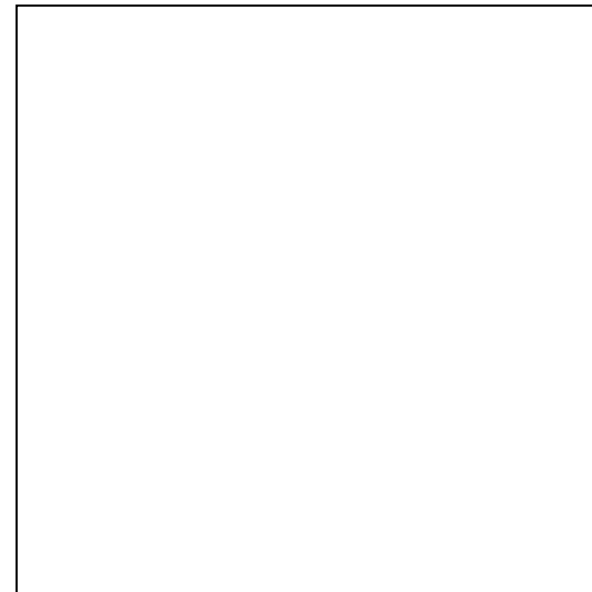
Working Directory



Staging Area



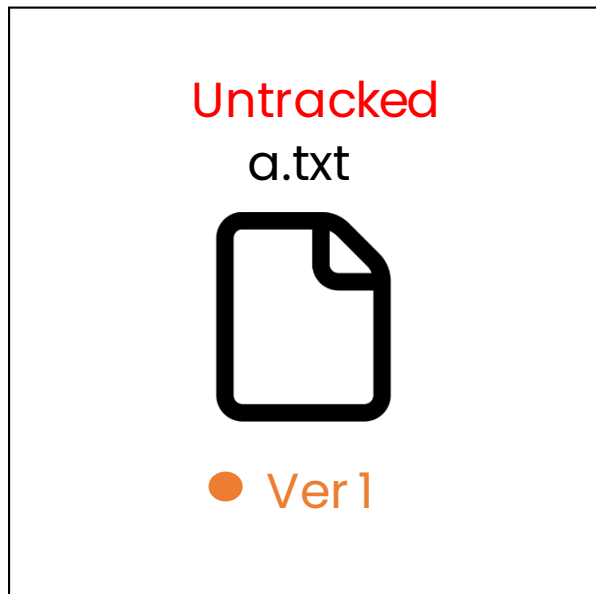
Repository



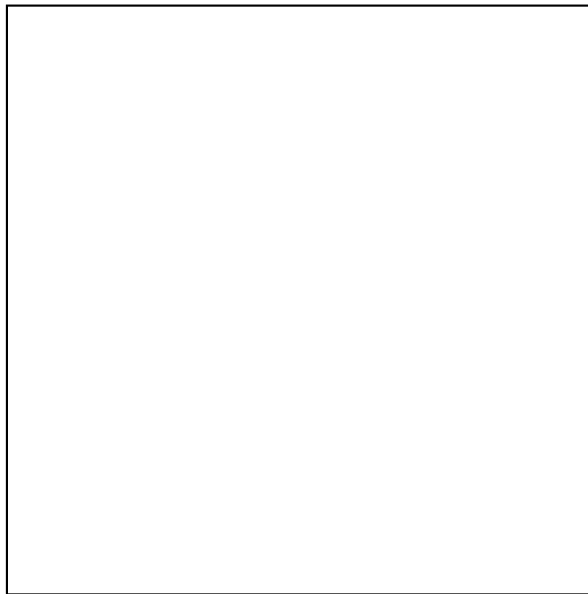
Staging Area 작업 단계 되돌리기

git rm --cached

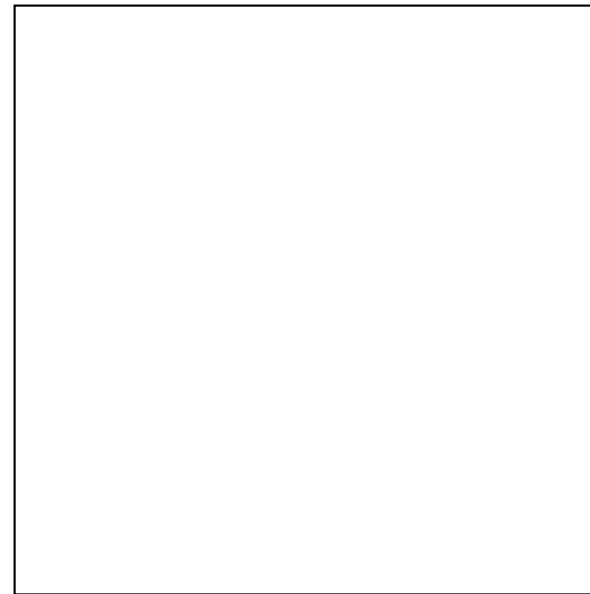
Working Directory



Staging Area



Repository



\$ git rm --cached a.txt

Staging Area 작업 단계 되돌리기

| git rm --cached

- Vscode에서 git rm --cached 명령어 실습
 1. Git 저장소 초기화
 2. test.md 파일 생성 후 add
 3. git rm --cached를 사용해서 Staging Area에 반영된 파일을 되돌리기

Staging Area 작업 단계 되돌리기

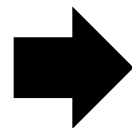
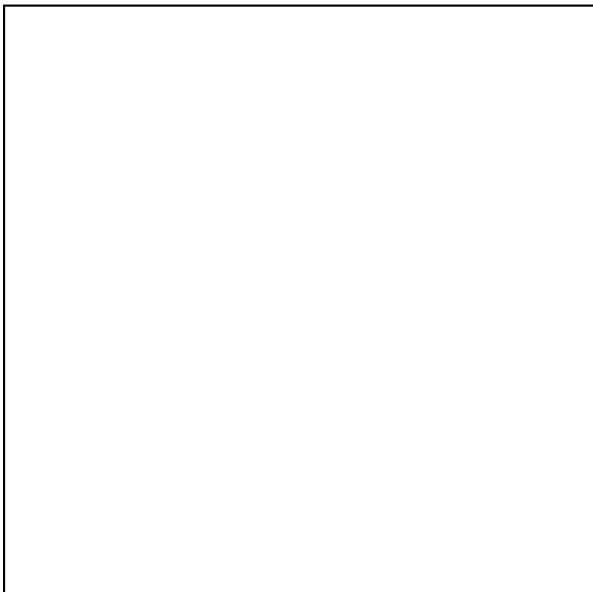
| git restore --staged

- **“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}`

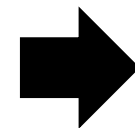
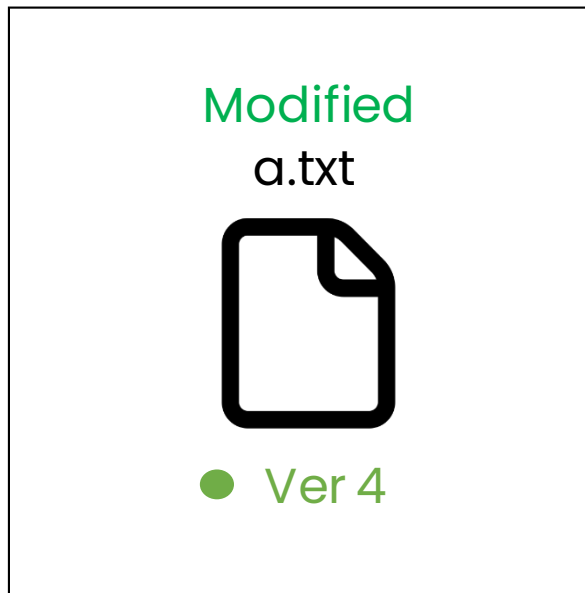
Staging Area 작업 단계 되돌리기

| git restore --staged

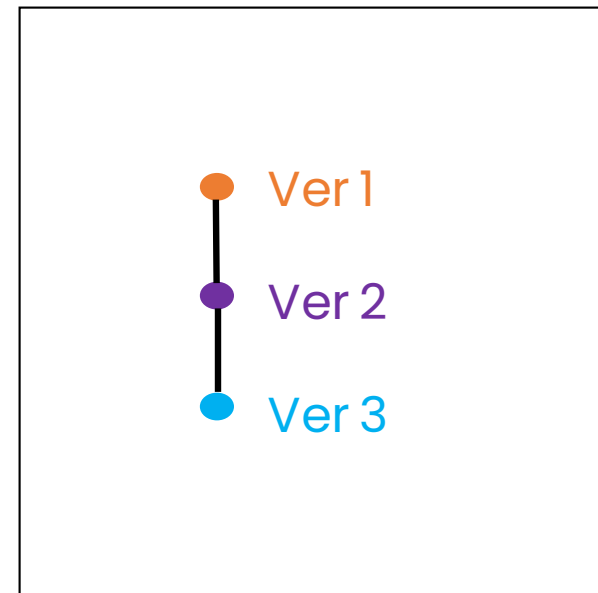
Working Directory



Staging Area



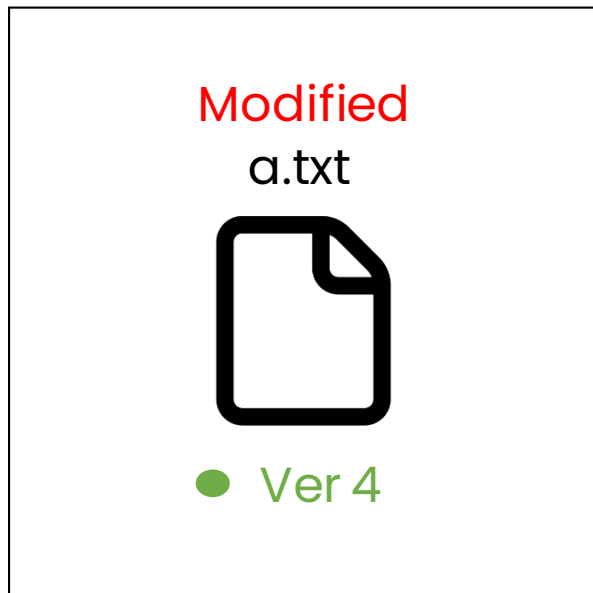
Repository



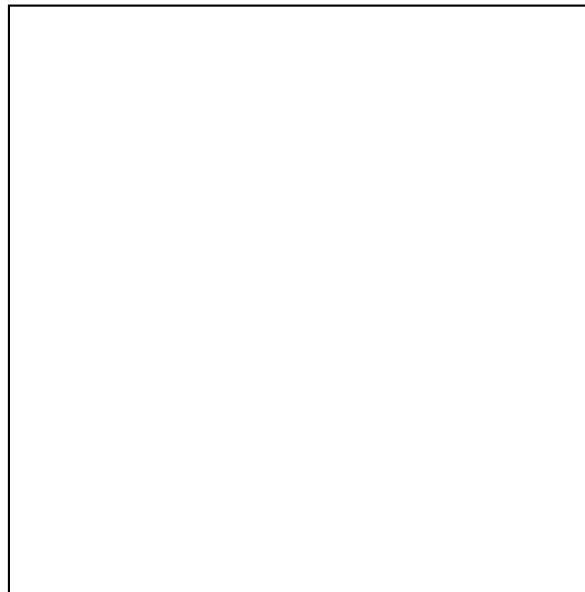
Staging Area 작업 단계 되돌리기

git restore --staged

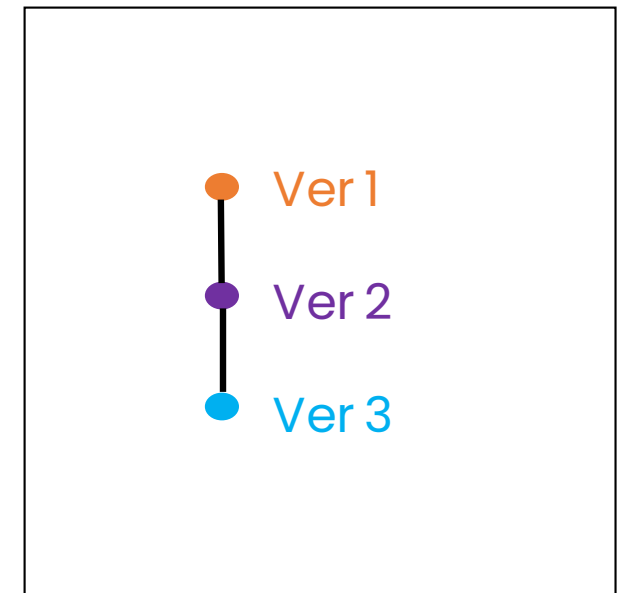
Working Directory



Staging Area



Repository



`$ git restore --staged a.txt`

Staging Area 작업 단계 되돌리기

| 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

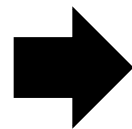
- 커밋을 완료한 파일을 Staging Area로 되돌리기
- 상황 별로 두 가지 기능으로 나뉨
 - Staging Area에 새로 올라온 내용이 없다면, 직전 커밋의 메시지만 수정
 - Staging Area에 새로 올라온 내용이 있다면, 직전 커밋을 덮어쓰기
- 이전 커밋을 완전히 고쳐서 새 커밋으로 변경하므로,
이전 커밋은 일어나지 않은 일이 되며 히스토리에도 남지 않음을 주의할 것!

Repository 작업 단계 되돌리기

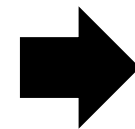
첫 번째 상황 (1/5)

- Staging Area에 새로 올라온 내용이 없다면, 직전 커밋의 메시지만 수정

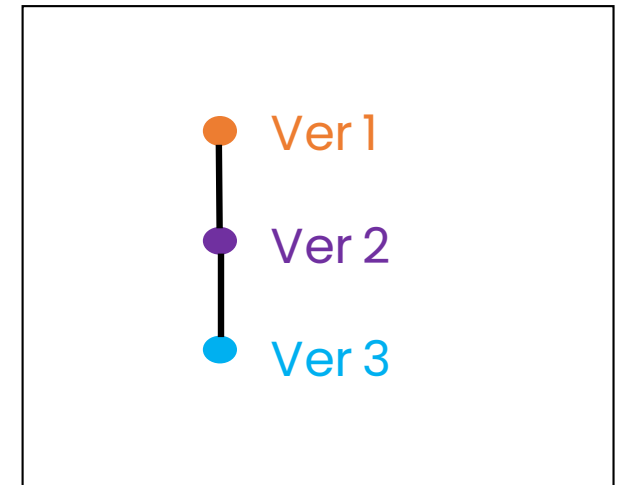
Working Directory



Staging Area



Repository

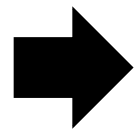


Repository 작업 단계 되돌리기

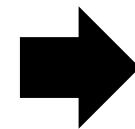
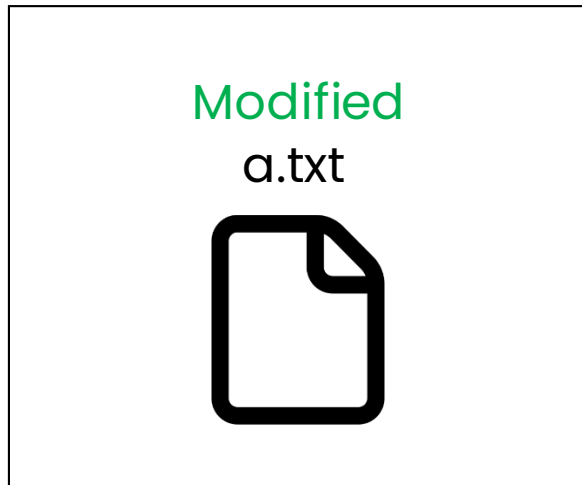
첫 번째 상황 (2/5)

- Staging Area에 새로 올라온 내용이 없다면, 직전 커밋의 메시지만 수정

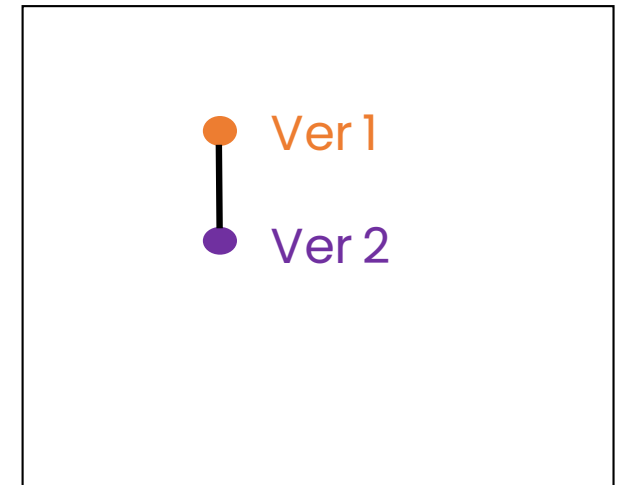
Working Directory



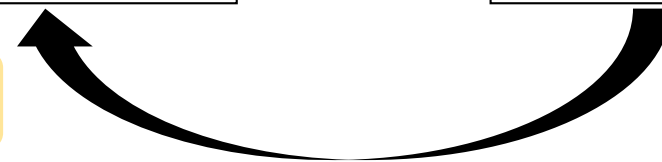
Staging Area



Repository



`$ git commit --amend`



Repository 작업 단계 되돌리기

첫 번째 상황 (3/5)

- Staging Area에 새로 올라온 내용이 없다면, 직전 커밋의 메시지만 수정

Work

```
05cc305 (HEAD -> master) second commit-amend를 통해 커밋 메시지
second commit-amend를 통해 커밋 메시지 수정

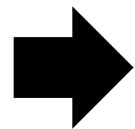
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit
.
#
# Date:      Sun Feb 6 17:30:59 2022 +0900
#
# On branch master
# Changes to be committed:
#       modified:   a.txt
#
```

Repository 작업 단계 되돌리기

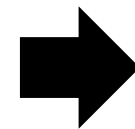
첫 번째 상황 (4/5)

- Staging Area에 새로 올라온 내용이 없다면, 직전 커밋의 메시지만 수정

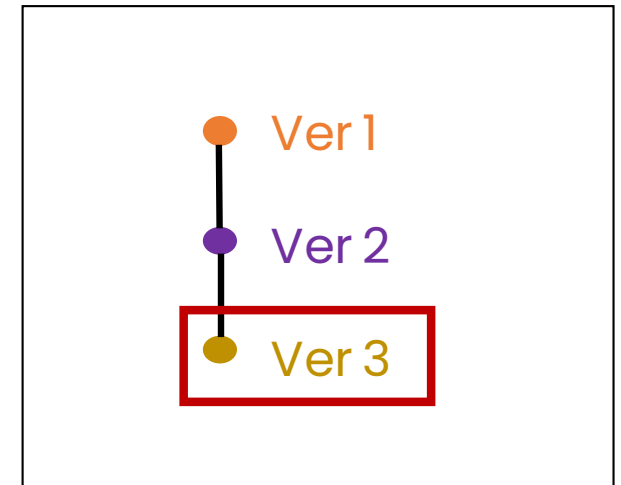
Working Directory



Staging Area



Repository



커밋 메시지가 변경되며 직전 커밋은 삭제되고 새로운 커밋이 생성됨

첫 번째 상황 (5/5)

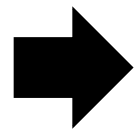
- Vscode에서 `git commit --amend` 첫 번째 상황 실습
 1. Git 저장소 초기화
 2. `test.md` 파일 생성 후 커밋
 3. `git commit --amend`를 사용해서 직전 커밋의 메시지만 수정하기
- [참고] Vim 간단 사용법
 - 입력 모드(i) : 문서 편집 가능
 - 명령 모드(esc)
 - 저장 및 종료(:wq)
 - 강제 종료(:q!)

Repository 작업 단계 되돌리기

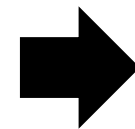
두 번째 상황 (1/5)

- Staging Area에 새로 올라온 내용이 있다면, 직전 커밋을 덮어쓰기

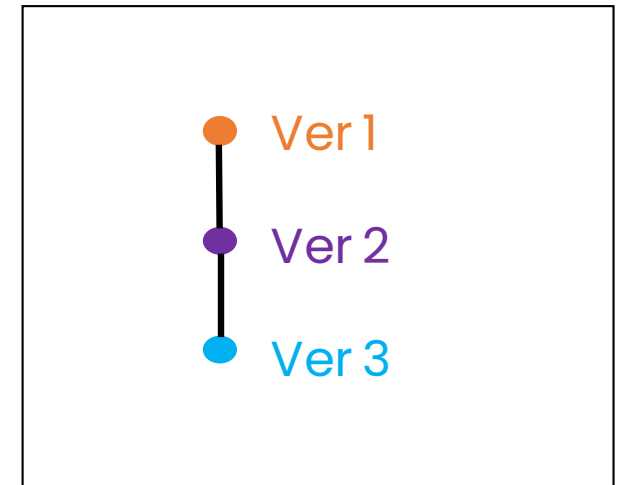
Working Directory



Staging Area



Repository

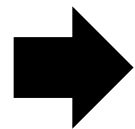


Repository 작업 단계 되돌리기

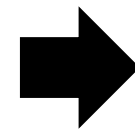
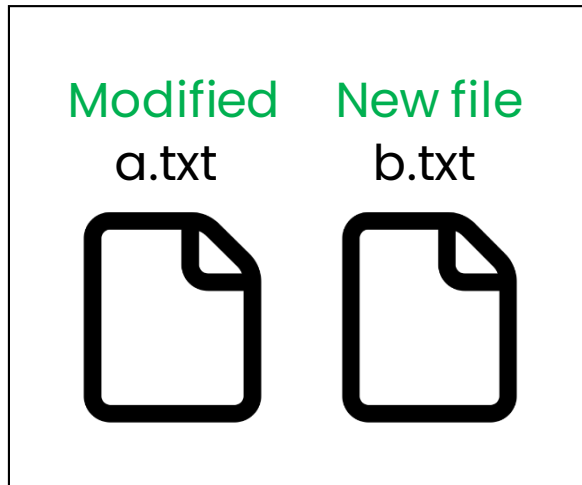
두 번째 상황 (2/5)

- Staging Area에 새로 올라온 내용이 있다면, 직전 커밋을 덮어쓰기

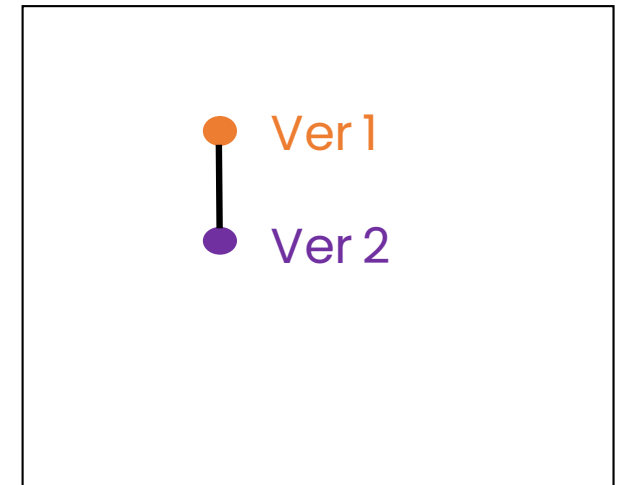
Working Directory



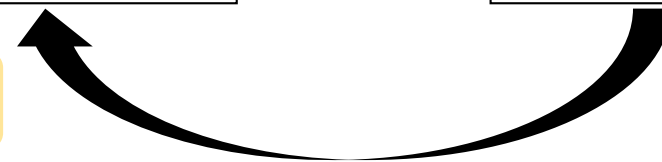
Staging Area



Repository



`$ git commit --amend`



Repository 작업 단계 되돌리기

두 번째 상황 (3/5)

- Staging Area에 새로 올라온 내용이 있다면, 직전 커밋을 덮어쓰기

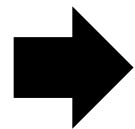
```
W 2nd commit-add title
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# Date:      Mon Feb 7 11:06:30 2022 +0900
#
# On branch master
# Changes to be committed:
#   modified:   a.txt
#   new file:   b.txt
#
~
```

Repository 작업 단계 되돌리기

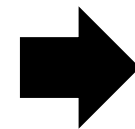
두 번째 상황 (4/5)

- Staging Area에 새로 올라온 내용이 있다면, 직전 커밋을 덮어쓰기

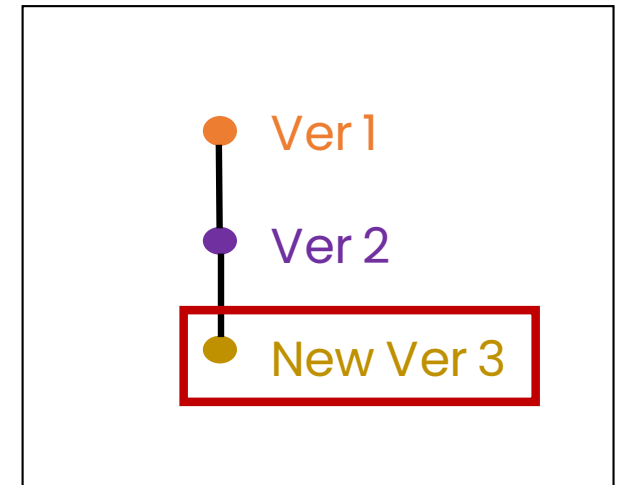
Working Directory



Staging Area



Repository



직전 커밋은 삭제되고 b.txt가 새로 생성된 내용까지 포함한 새로운 커밋이 생성됨

| 두 번째 상황 (5/5)

- Vscode에서 git commit --amend 두 번째 상황 실습
 1. Git 저장소 초기화
 2. test.md 파일 생성 후 커밋
 3. 직전 커밋에 누락된 파일을 Staging Area에 반영하기
 4. git commit --amend를 사용해서 직전 커밋을 덮어쓰기

이어서...

삼성 청년 SW 아카데미

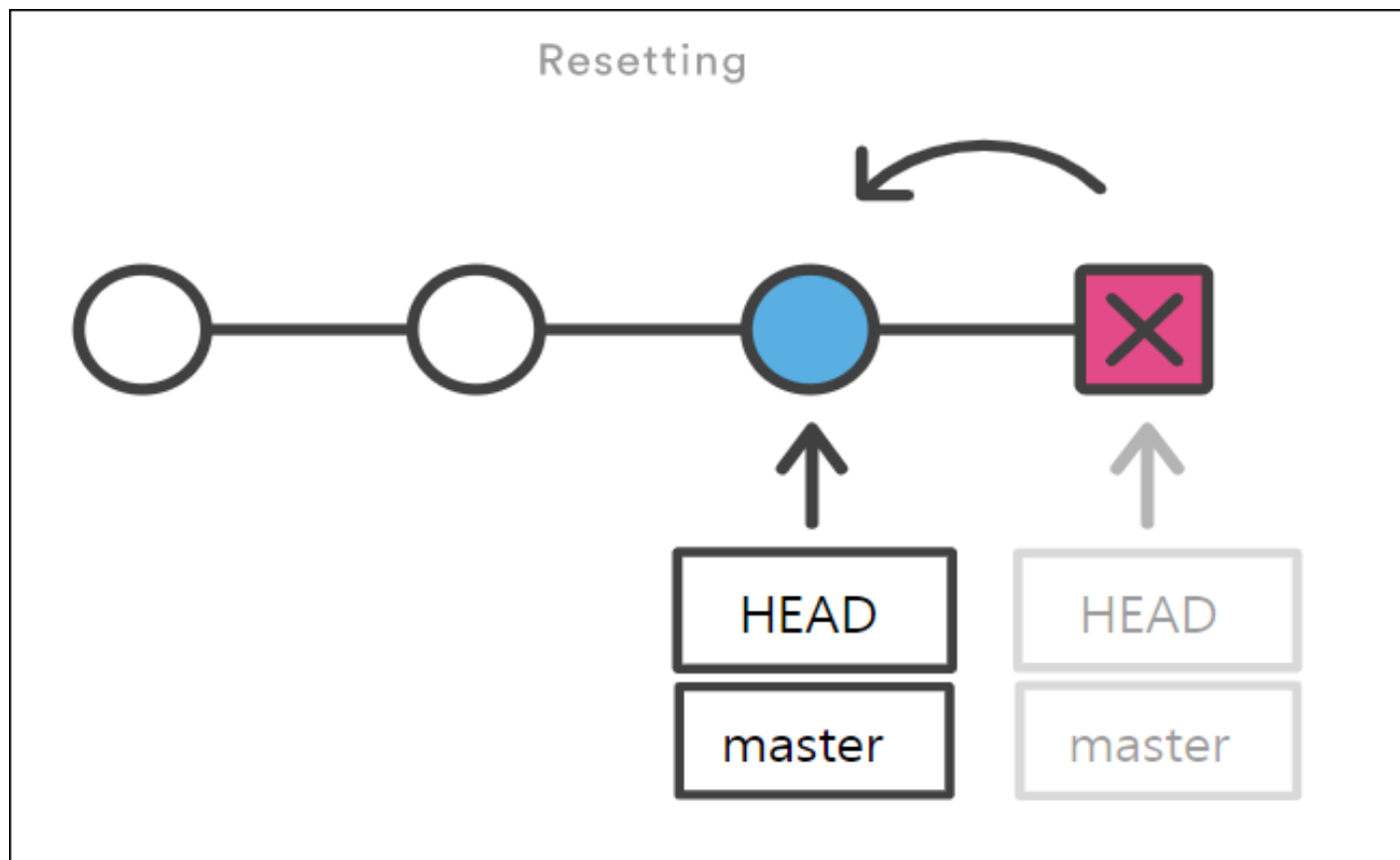
Git reset & revert

Git reset

| 개요

- 시계를 마치 과거로 돌리는 듯한 행위로, 프로젝트를 특정 커밋(버전) 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

| 개요



git reset의 세 가지 옵션

- `--soft`
 - 해당 커밋으로 되돌아가고
 - 되돌아간 커밋 이후의 파일들은 Staging Area로 돌려놓음
- `--mixed`
 - 해당 커밋으로 되돌아가고
 - 되돌아간 커밋 이후의 파일들은 Working Directory로 돌려놓음
 - git reset 옵션의 기본값
- `--hard`
 - 해당 커밋으로 되돌아가고
 - 되돌아간 커밋 이후의 파일들은 모두 Working Directory에서 삭제 → **따라서 사용 시 주의할 것!**
 - 기존의 Untracked 파일은 사라지지 않고 Untracked로 남아있음

git reset의 세 가지 옵션

특정 커밋으로 reset 했을 때
특정 커밋 이후에 커밋 되었던 파일들의 상태

옵션	working directory	staging area	repository
--soft		V	HEAD가 특정 커밋을 가리킴
--mixed	V		HEAD가 특정 커밋을 가리킴
--hard			HEAD가 특정 커밋을 가리킴

* 단, --hard 옵션 사용 시 기존의 Untracked 파일은 여전히 Untracked로 남음

[참고] git reflog

- git reset의 hard 옵션은 Working Directory 내용까지 삭제하므로 위험할 수 있음
- git reflog 명령어를 이용하면 reset 하기 전의 과거 커밋 내역을 모두 조회 가능

```
$ git reflog
1a410ef HEAD@{0}: reset: moving to
1a410ef ab1afef HEAD@{1}: commit: modified repo.rb a bit
484a592 HEAD@{2}: commit: added repo.rb
```

- 이후 해당 커밋으로 reset 하면 hard 옵션으로 삭제된 파일도 복구 가능

| 따라하기

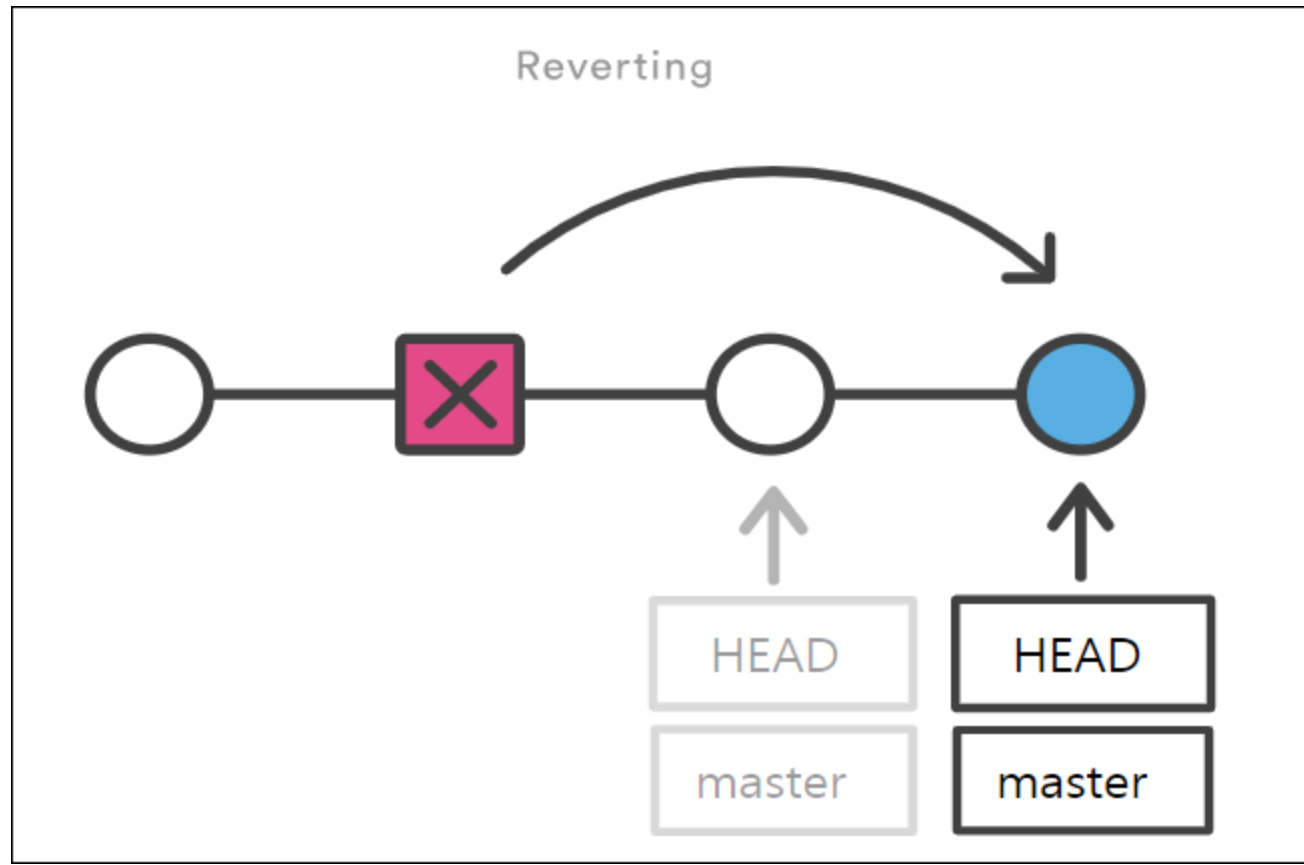
- Vscode에서 git reset 실습
 1. 실습 전 환경 세팅 - 제공되는 zip 파일 다운로드 및 압축 풀기
 2. soft 폴더에서 git reset --soft 실습
 3. mixed 폴더에서 git reset --mixed 실습
 4. hard 폴더에서 git reset --hard 실습

Git revert

| 개요

- 과거를 없었던 일로 만드는 행위로, 이전 커밋을 취소한다는 새로운 커밋을 생성함
- `git revert {커밋 ID}`
 - 커밋 ID는 취소하고 싶은 커밋 ID를 작성

| 개요



| git reset과의 차이점

- 개념적 차이

- reset은 커밋 내역을 삭제하는 반면, revert는 새로운 커밋을 생성함
- revert는 Github를 이용해 협업할 때, 커밋 내역의 차이로 인한 충돌 방지 가능

- 문법적 차이

- `git reset 5sd2f42`라고 작성하면 5sd2f42라는 커밋으로 되돌린다는 뜻
- `git revert 5sd2f42`라고 작성하면 5sd2f42라는 커밋 한 개를 취소한다는 뜻
(5sd2f42라는 커밋이 취소되었다는 내용의 새로운 커밋을 생성함)

| 따라하기

- Vscode에서 git revert 실습
 1. 실습 전 환경 세팅 - 제공되는 zip 파일 다운로드 및 압축 풀기
 2. git revert 실습

이어서...

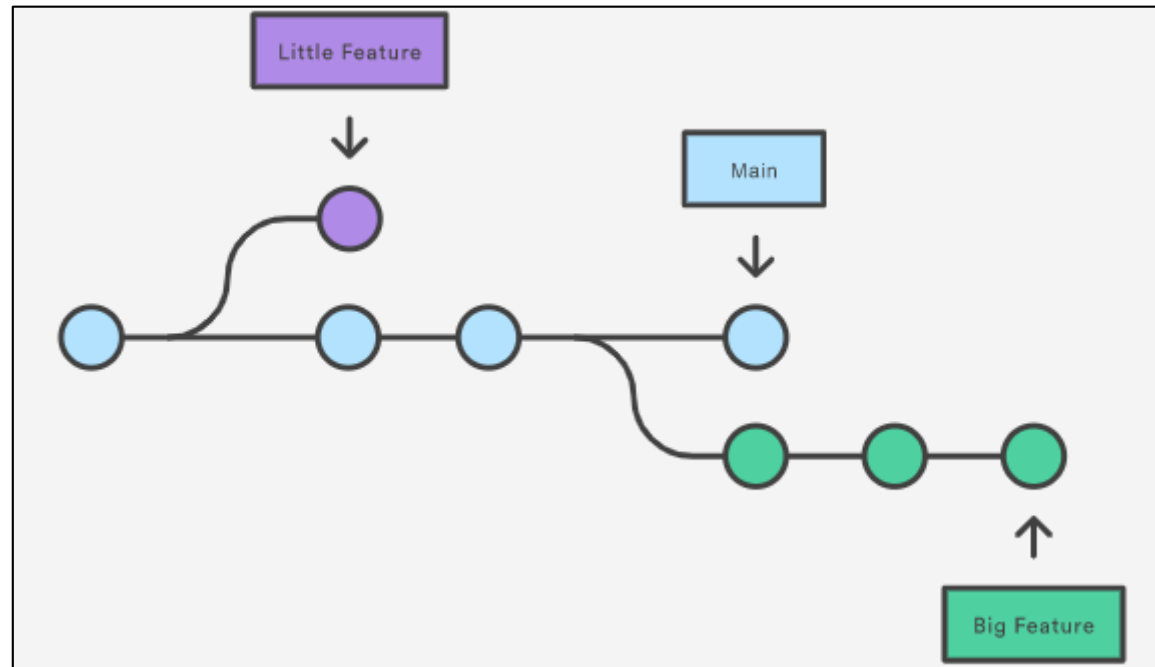
삼성 청년 SW 아카데미

Git branch & merge

Git branch

개요

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



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

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

[참고] HEAD

- “This is a pointer to the local branch you’re currently on.”
- HEAD는 현재 브랜치를 가리키고, 각 브랜치는 자신의 최신 커밋을 가리키므로 결국 HEAD가 현재 브랜치의 최신 커밋을 가리킨다고 할 수 있음
- `git log` 혹은 `cat .git/HEAD`를 통해서 현재 HEAD가 어떤 브랜치를 가리키는지 알 수 있음

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ git log --oneline
af8692c (HEAD -> master) c
cc3650a (hotfix) b
b582ee9 a
```

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ cat .git/HEAD
ref: refs/heads/master
```

[참고] HEAD

- 결국 git switch는 현재 브랜치에서 다른 브랜치로 HEAD를 이동시키는 명령어

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ git log --oneline
af8692c (HEAD -> master) c
cc3650a (hotfix) b
b582ee9 a
```

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ cat .git/HEAD
ref: refs/heads/master
```

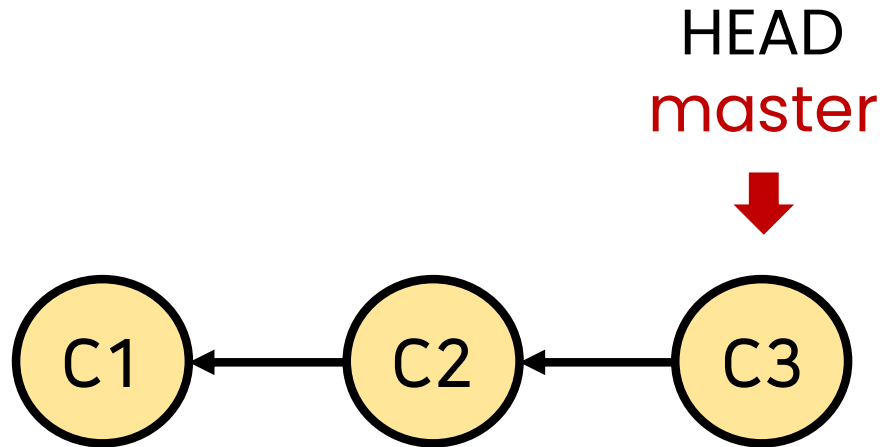
```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ git switch hotfix
Switched to branch 'hotfix'
```

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (hotfix)
$ git log --oneline
cc3650a (HEAD -> hotfix) b
b582ee9 a
```

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (hotfix)
$ cat .git/HEAD
ref: refs/heads/hotfix
```

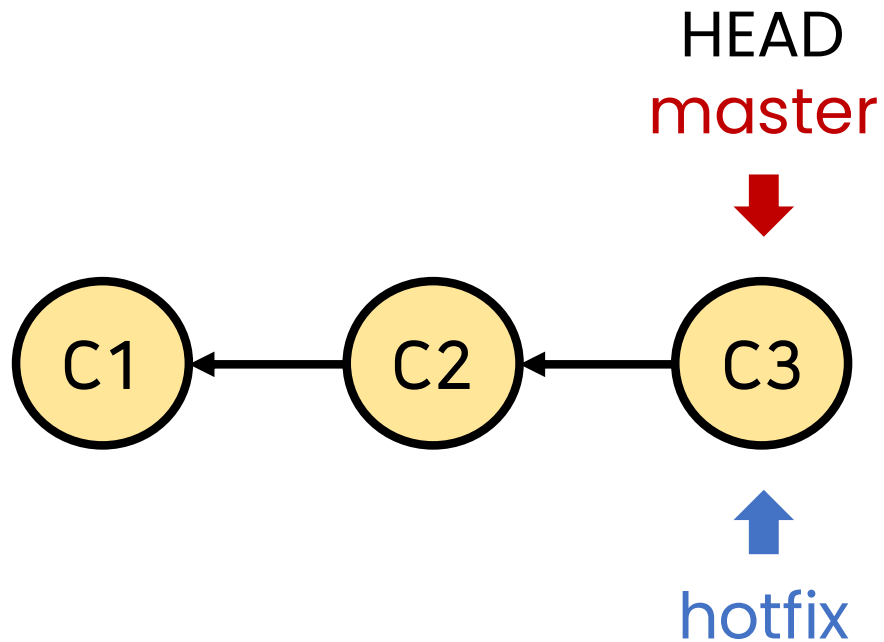
[참고] branch와 switch 그림으로 파악하기 (1/6)

- 현재 HEAD는 master를 가리키고, master의 최신 커밋은 C3인 상태



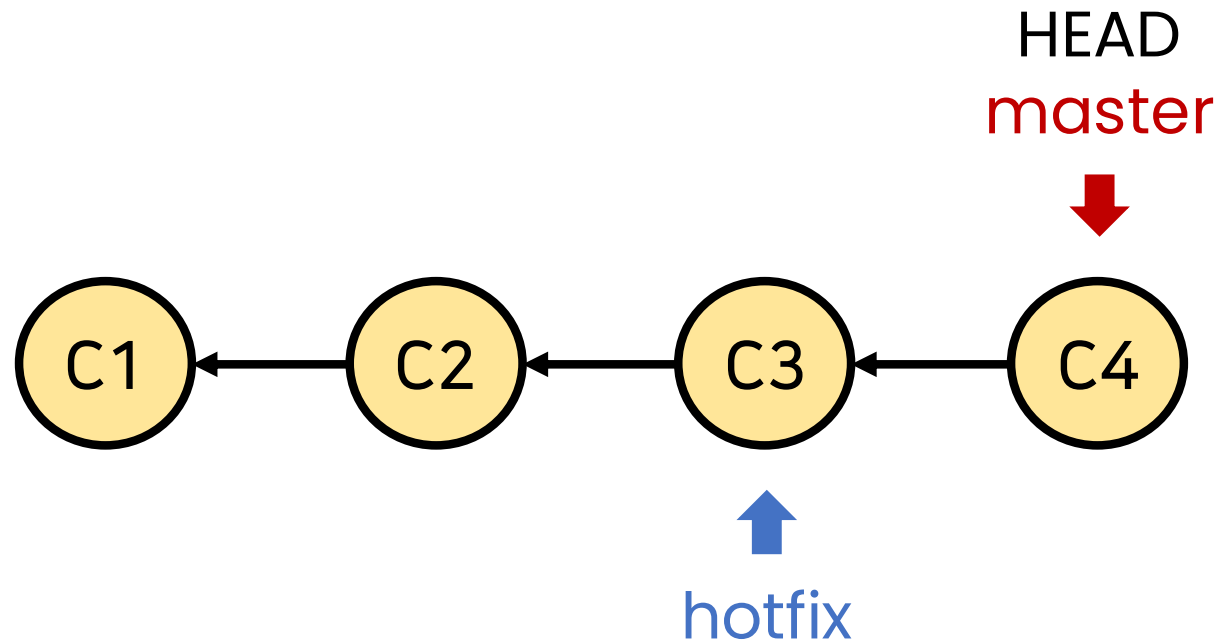
[참고] branch와 switch 그림으로 파악하기 (2/6)

- (master) \$ git branch hotfix



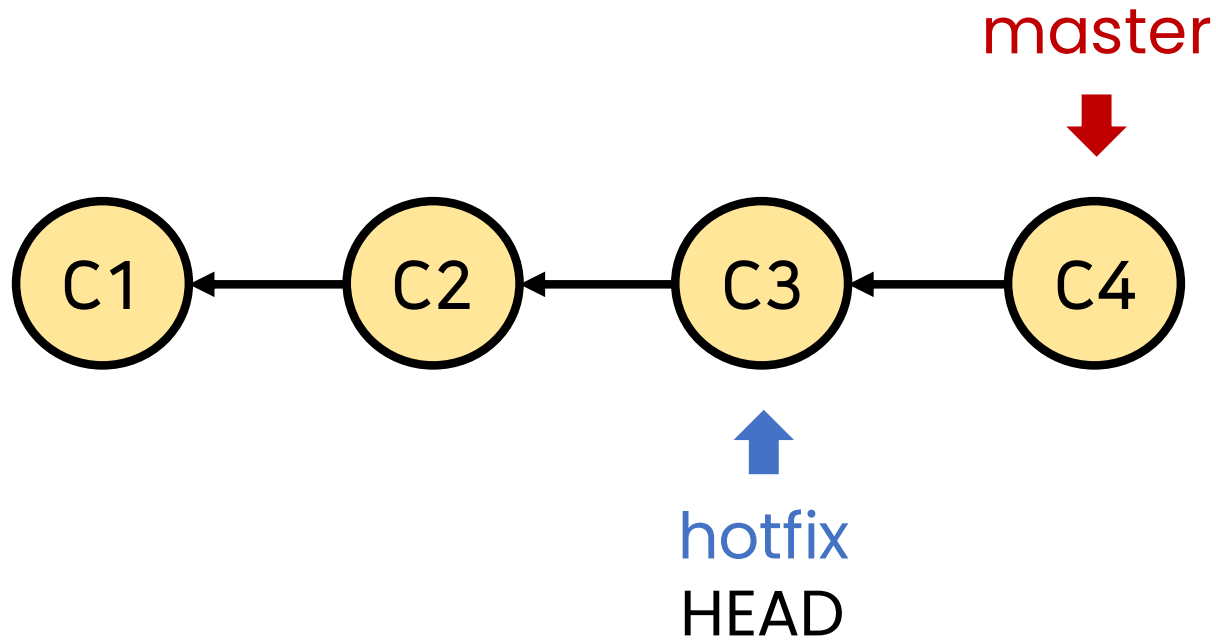
[참고] branch와 switch 그림으로 파악하기 (3/6)

- (master) \$ git commit -m "C4"



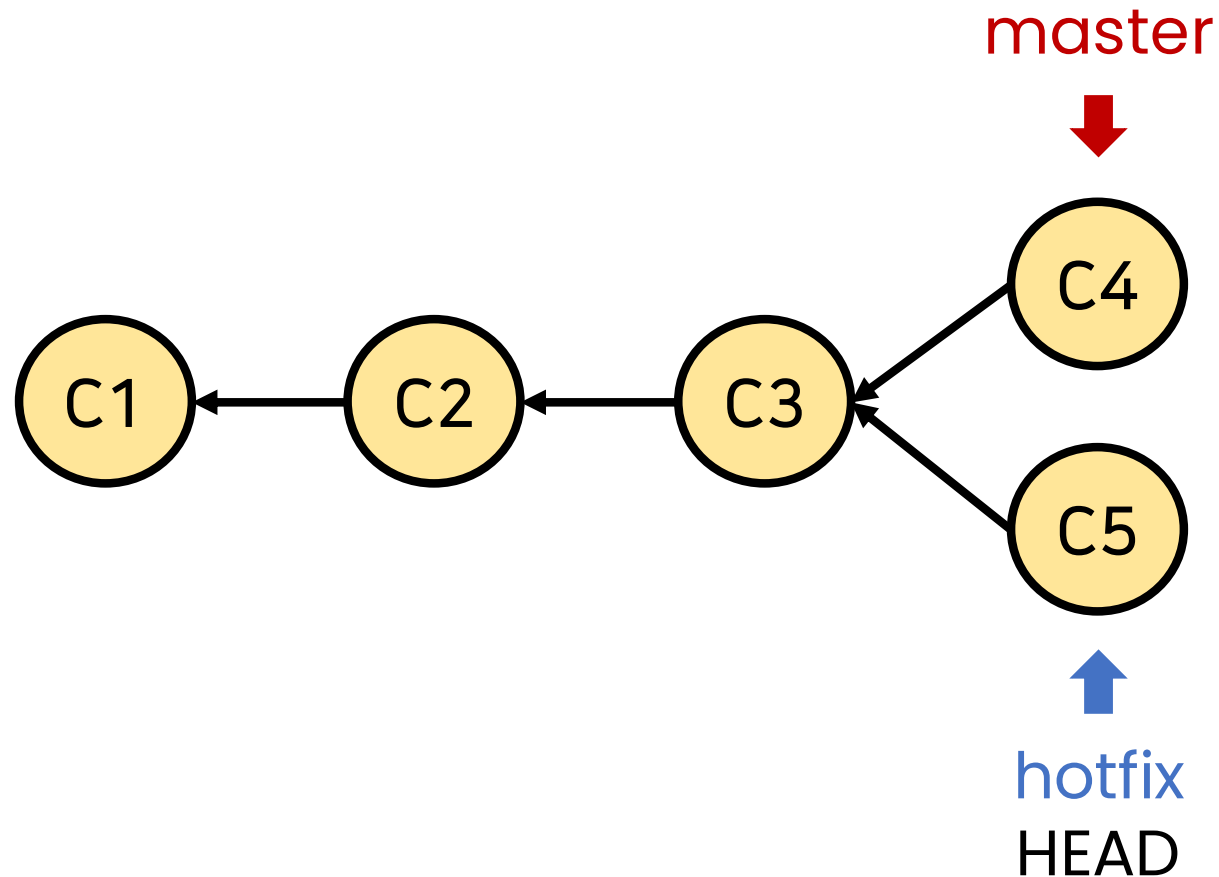
[참고] branch와 switch 그림으로 파악하기 (4/6)

- (master) \$ git switch hotfix



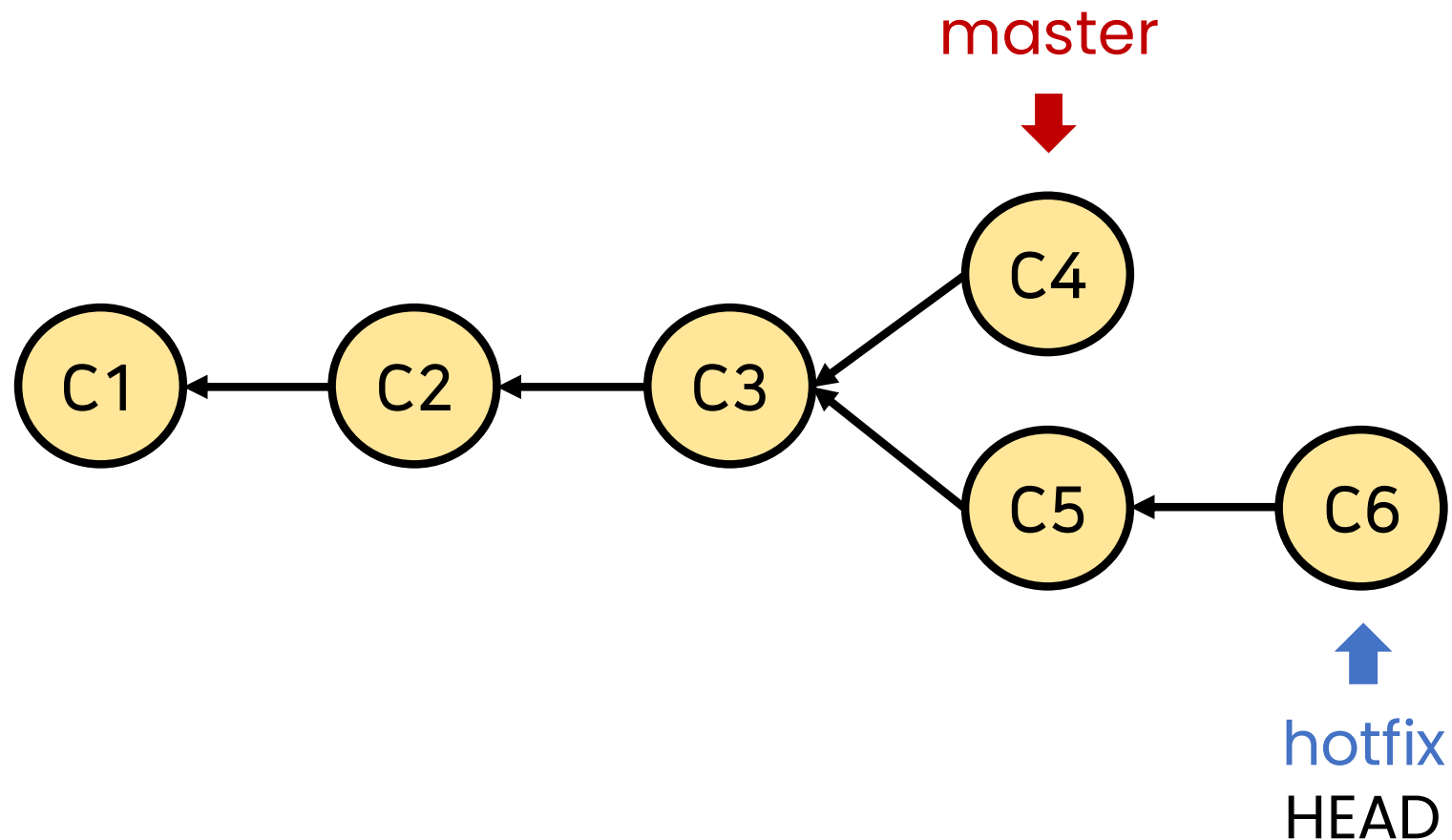
[참고] branch와 switch 그림으로 파악하기 (5/6)

- (hotfix) \$ git commit -m "C5"



[참고] branch와 switch 그림으로 파악하기 (6/6)

- (hotfix) \$ git commit -m "C6"



따라하기

- Vscode에서 git branch & switch 실습
 1. 실습 전 환경 세팅
 2. git branch 실습
 3. git switch 실습
 4. 분기하는 두 브랜치 만들기 실습

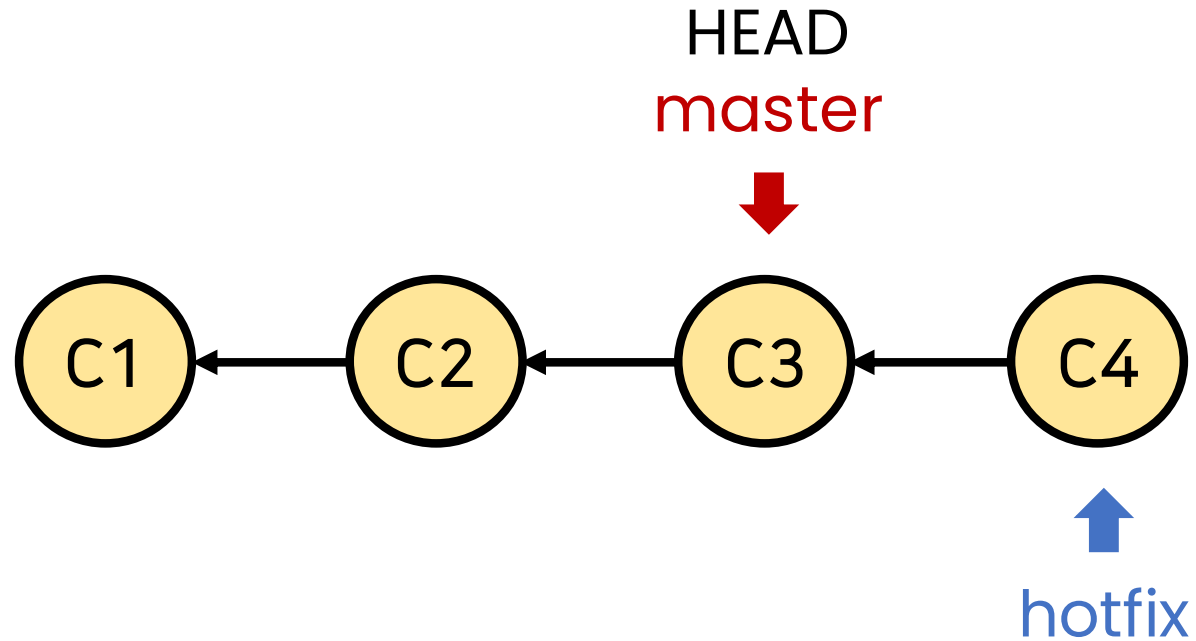
Git merge

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

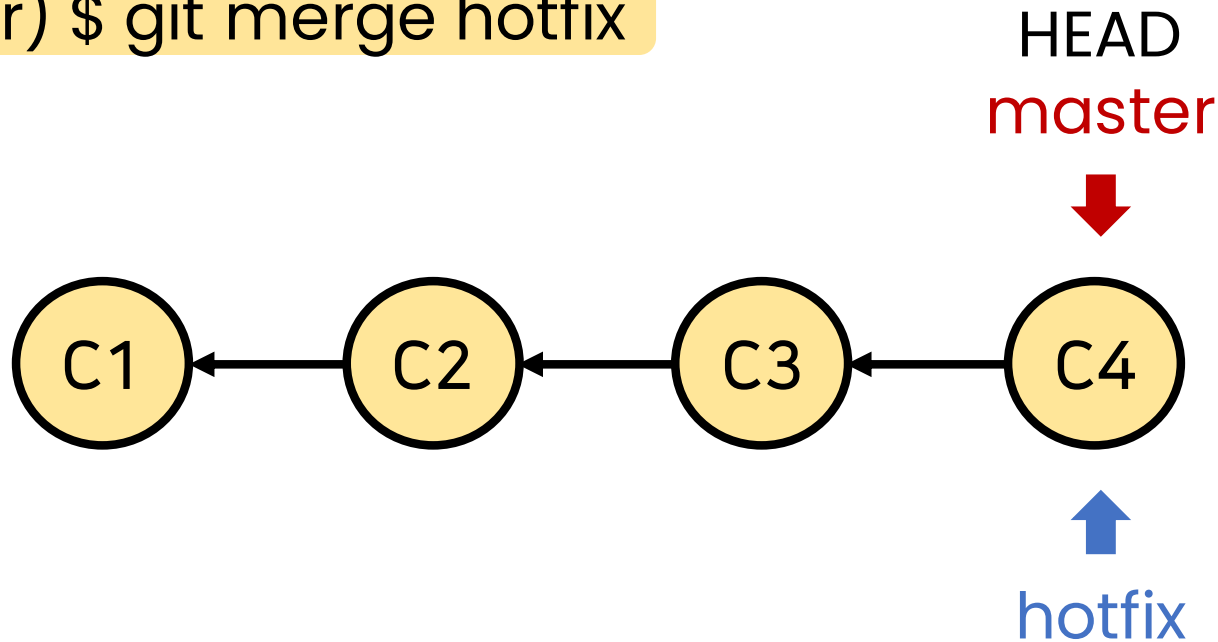
1. Fast-Forward (1/3)

- 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

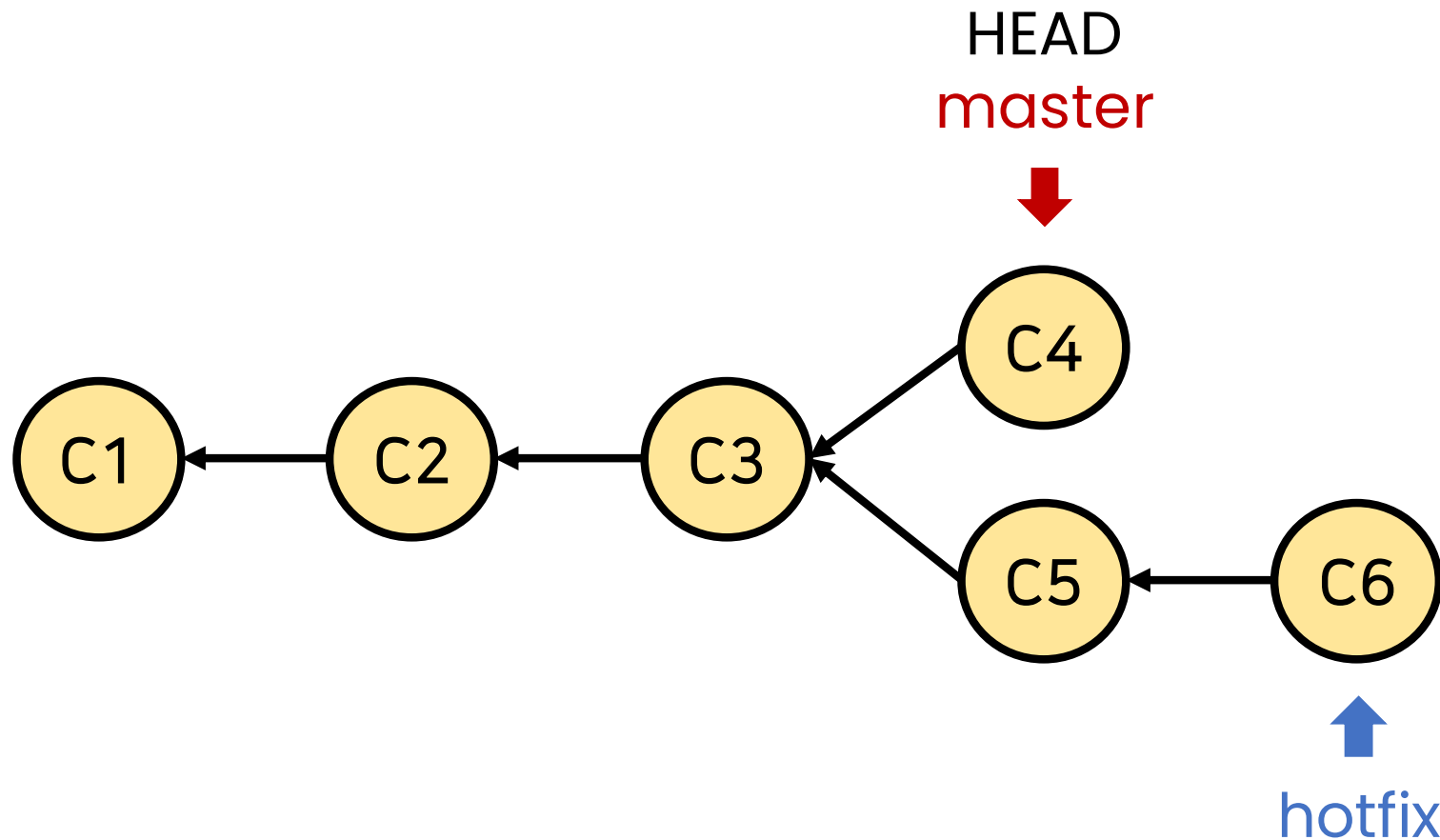
HEAD
master

```
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

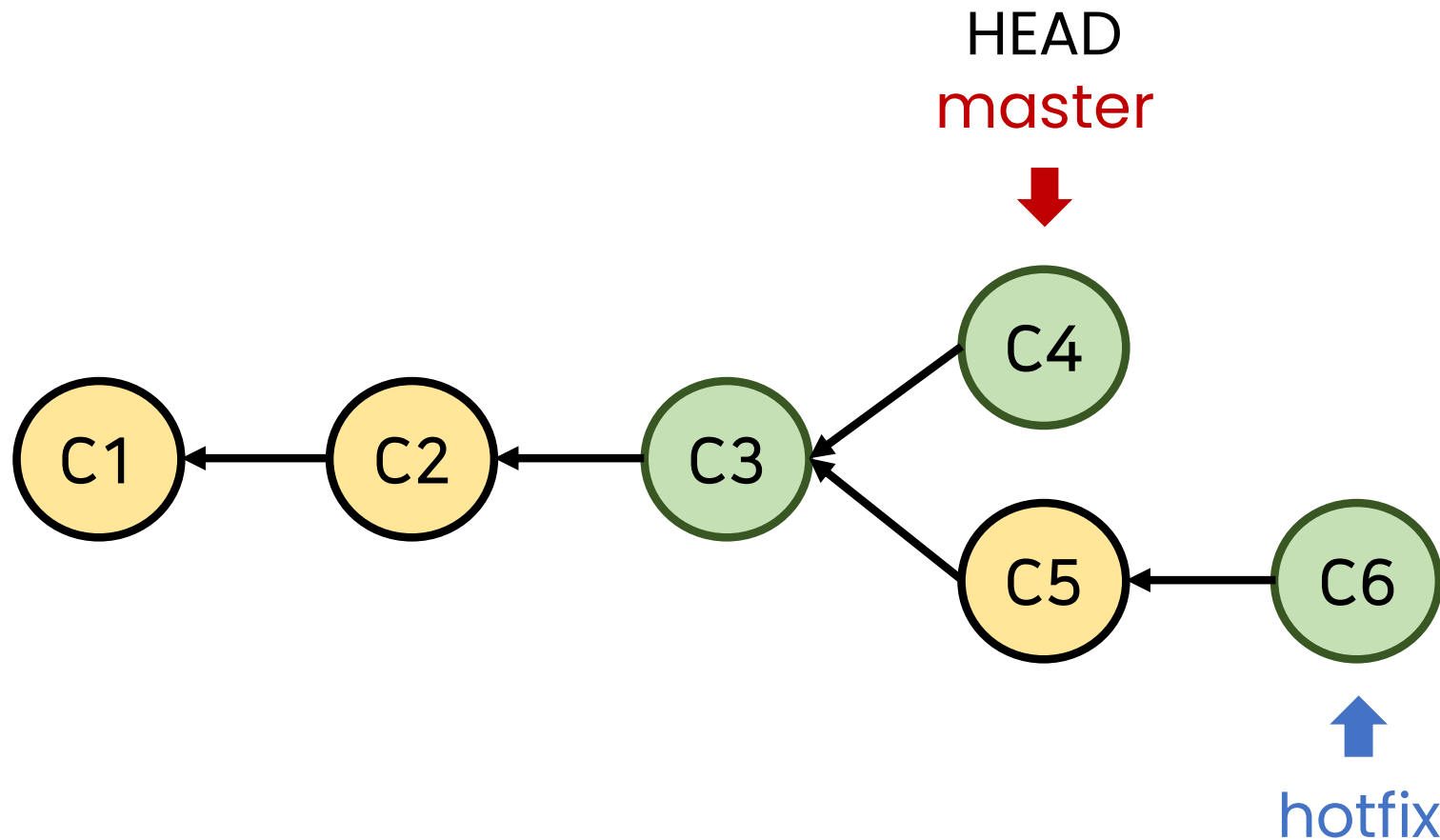
2. 3-way Merge (1/4)

- 각 브랜치의 커밋 두 개와 공통 조상 하나를 사용하여 병합하는 방법



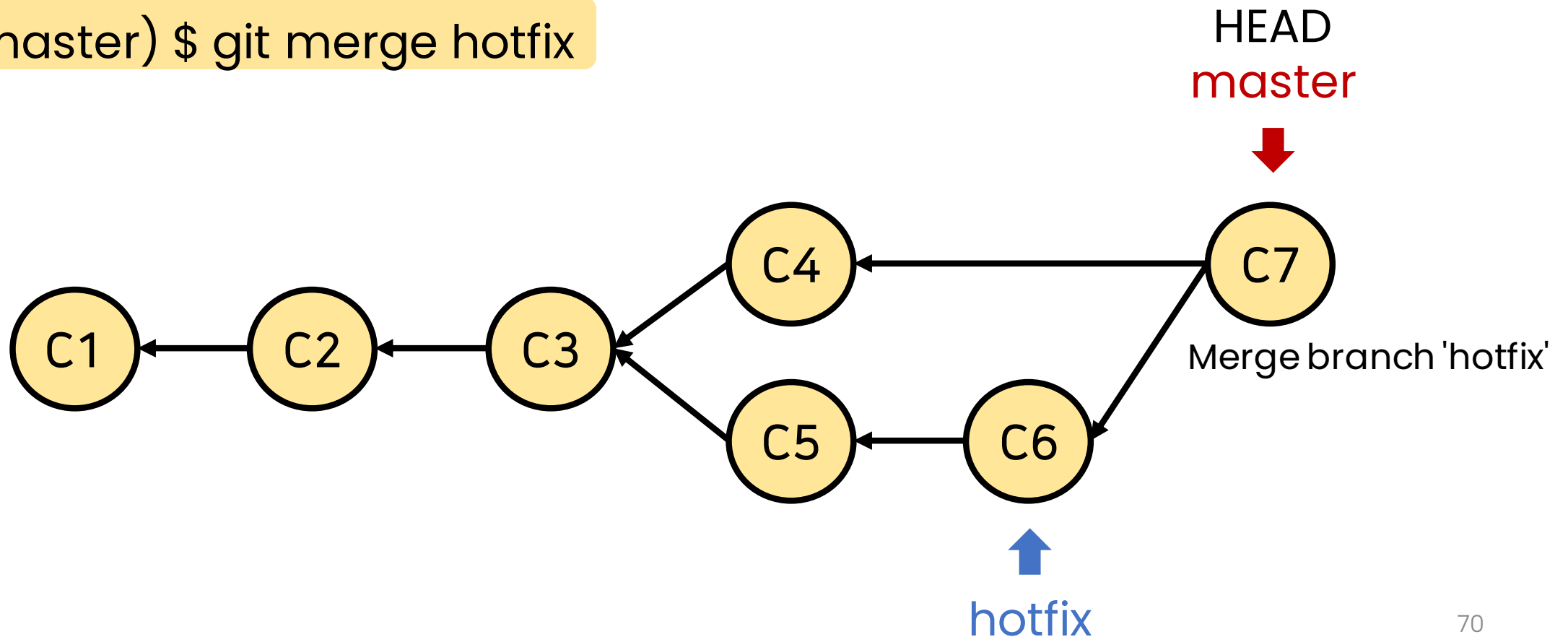
2. 3-way Merge (2/4)

- 각 브랜치의 커밋 두 개와 공통 조상 하나를 사용하여 병합하는 방법



2. 3-way Merge (3/4)

- 각 브랜치의 커밋 두 개와 공통 조상 하나를 사용하여 병합하는 방법
- (master) \$ git merge hotfix



2. 3-way Merge (4/4)

- 각 브랜치의 커밋 두 개와 공통 조상 하나를 사용하여 병합하는 방법
- (master) \$ git merge hotfix

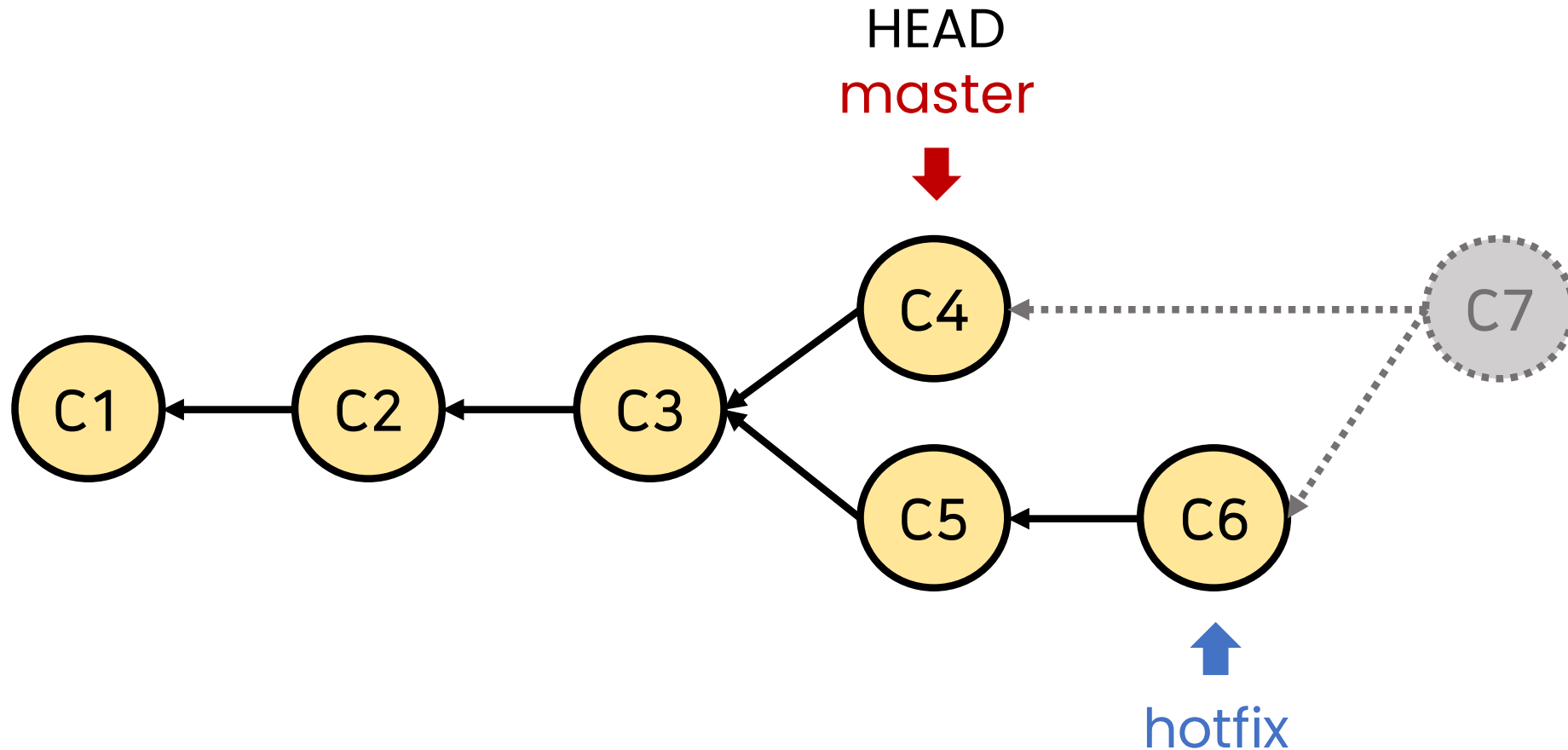


3. Merge Conflict (1/7)

- 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

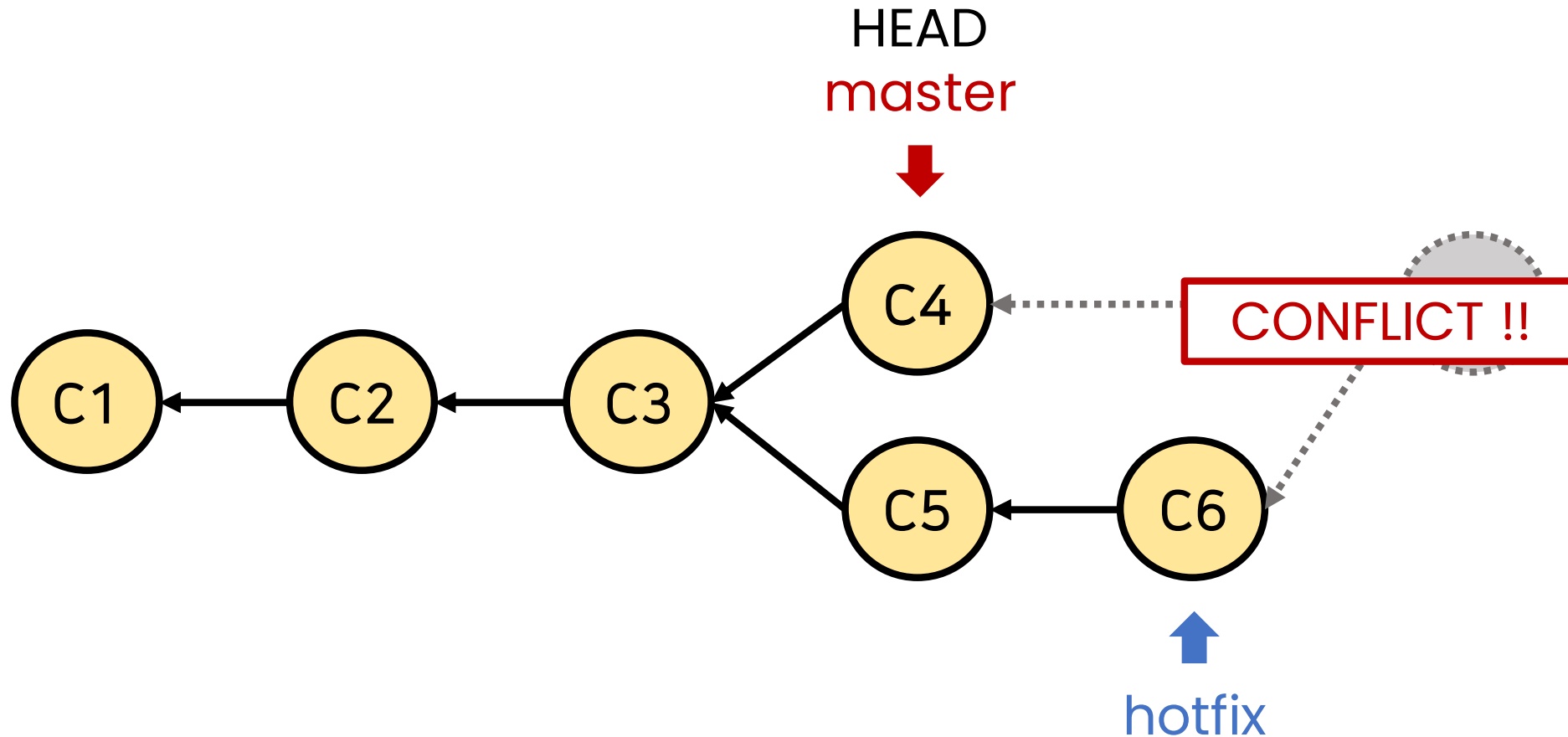
3. Merge Conflict (2/7)

- (master) \$ git merge hotfix



3. Merge Conflict (3/7)

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

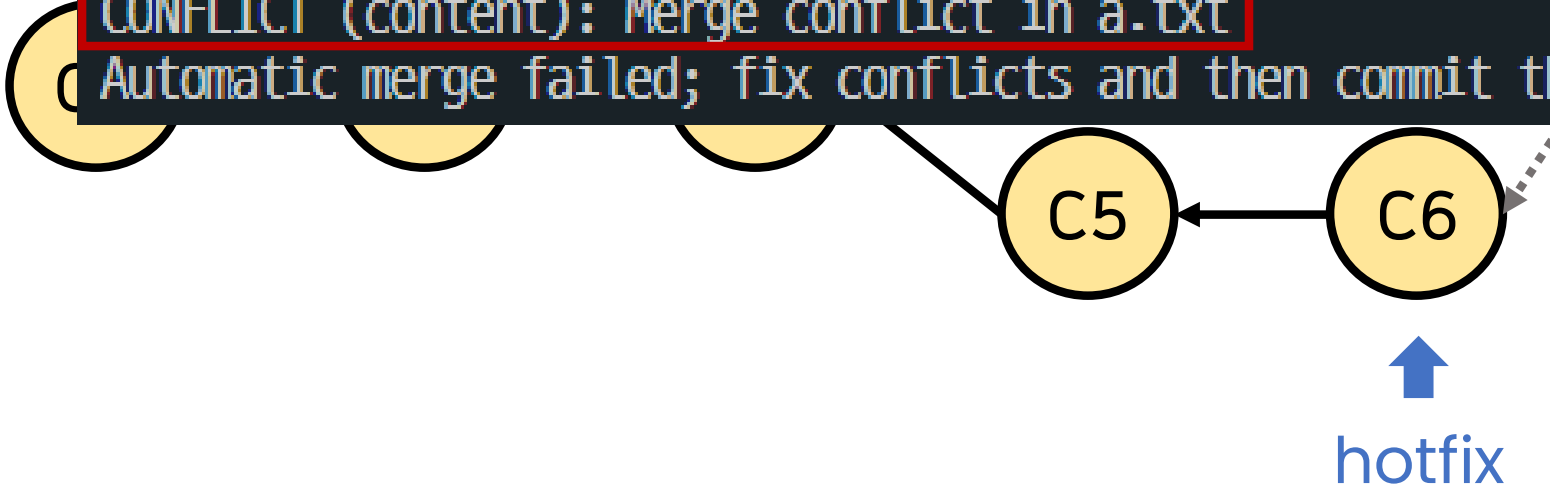


3. Merge Conflict (4/7)

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

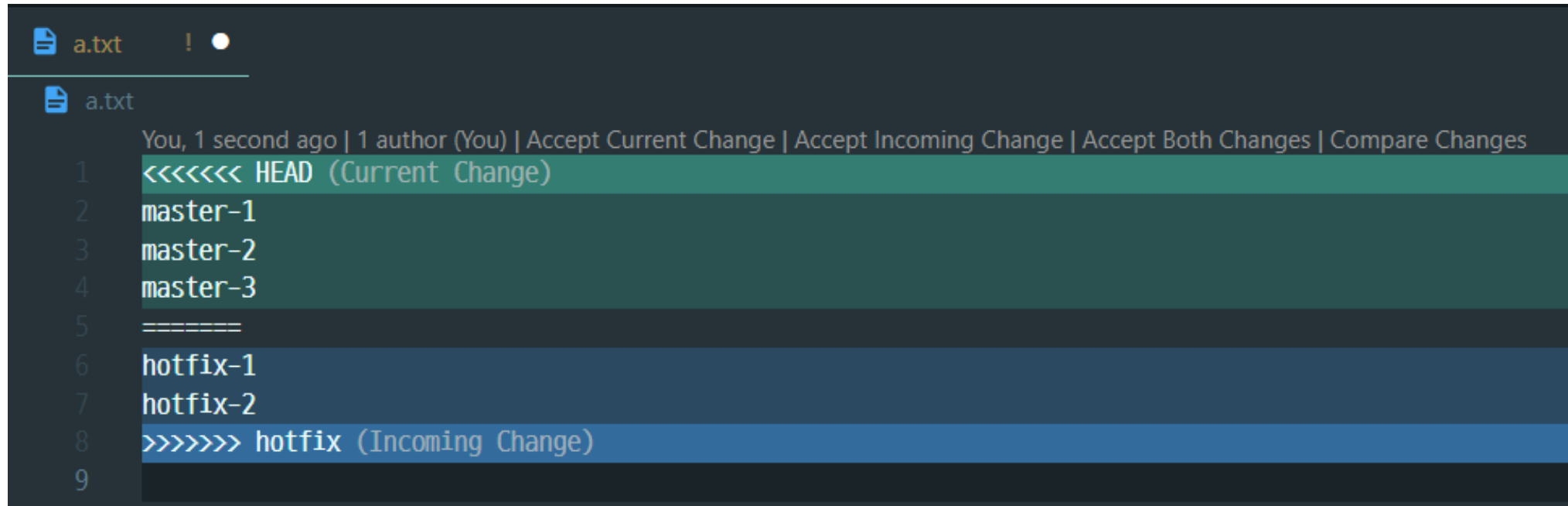
HEAD
master

```
kyle@DESKTOP-86J1CBC MINGW64 ~/Desktop/git-test (master)
$ git merge hotfix
Auto-merging a.txt
CONFLICT (content): Merge conflict in a.txt
Automatic merge failed; fix conflicts and then commit the result.
```



3. Merge Conflict (5/7)

- 충돌이 발생한 부분은 작성자가 직접 해결 해야함



```
a.txt
You, 1 second ago | 1 author (You) | Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<<< HEAD (Current Change)
2 master-1
3 master-2
4 master-3
5 =====
6 hotfix-1
7 hotfix-2
8 >>>>>> hotfix (Incoming Change)
9
```

3. Merge Conflict (6/7)

- 충돌 해결 후, 병합된 내용을 기록한 Merge Commit 생성

```
Merge branch 'hotfix'
```

```
# Conflicts:
```

```
#   a.txt
```

```
#
```

```
# It looks like you may be committing a merge.
```

```
# If this is not correct, please run
```

```
#   git update-ref -d MERGE_HEAD
```

```
# and try again.
```

```
# Please enter the commit message for your changes. Lines starting
```

```
# with '#' will be ignored, and an empty message aborts the commit.
```

```
#
```

```
# On branch master
```

```
# All conflicts fixed but you are still merging.
```

```
#
```

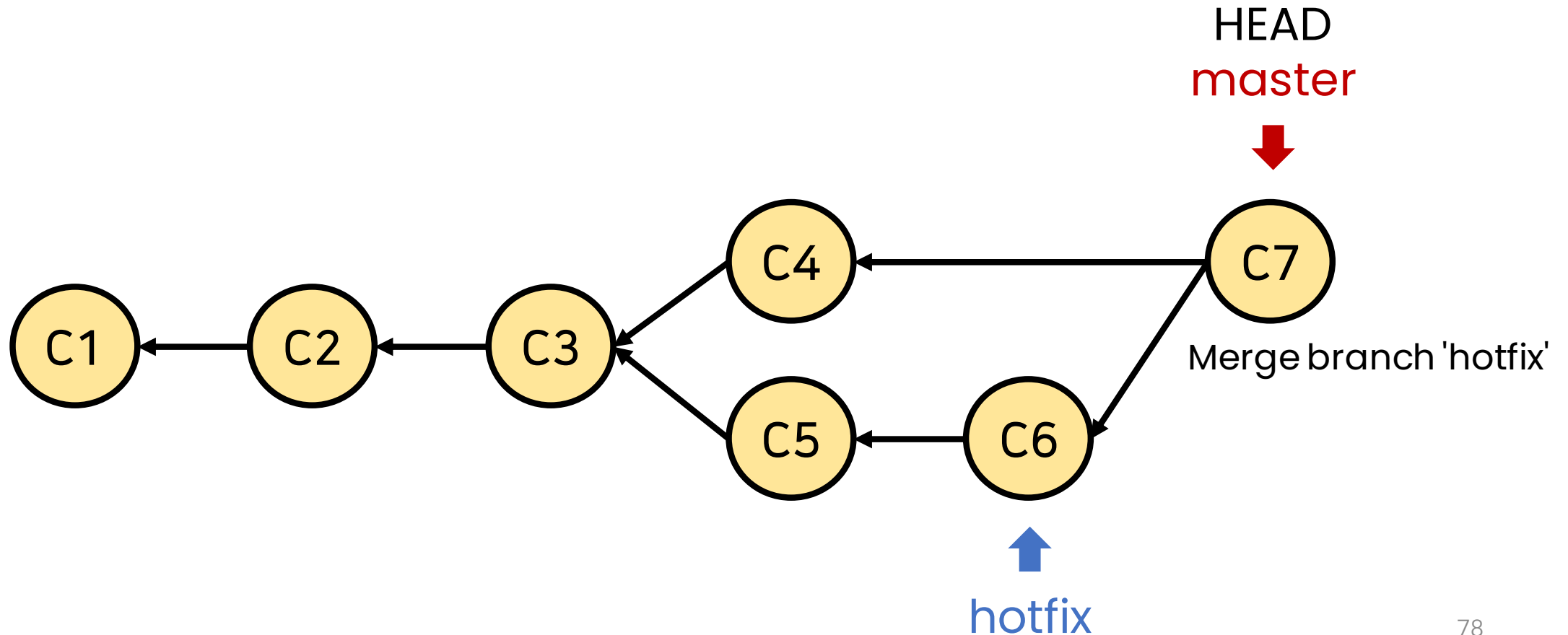
```
# Changes to be committed:
```

```
#   modified:   a.txt
```

```
#
```

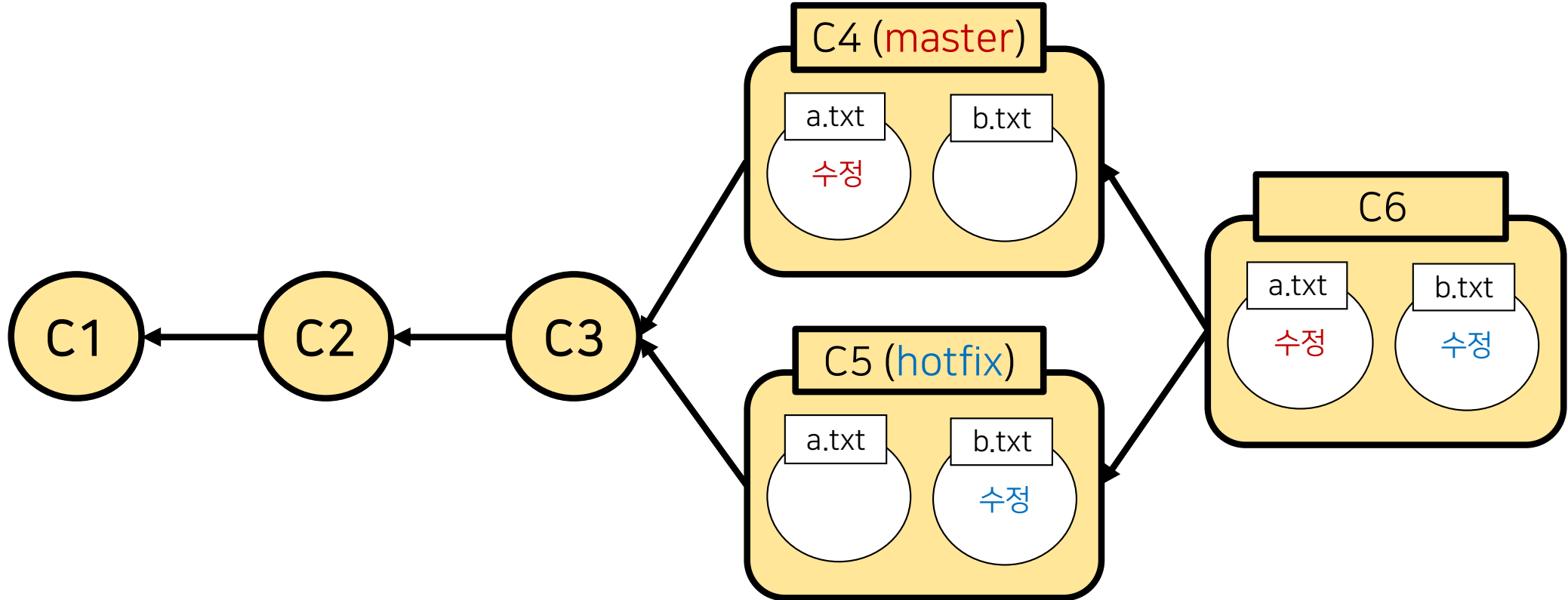
3. Merge Conflict (7/7)

- 충돌 해결 후, 병합된 내용을 기록한 Merge Commit 생성



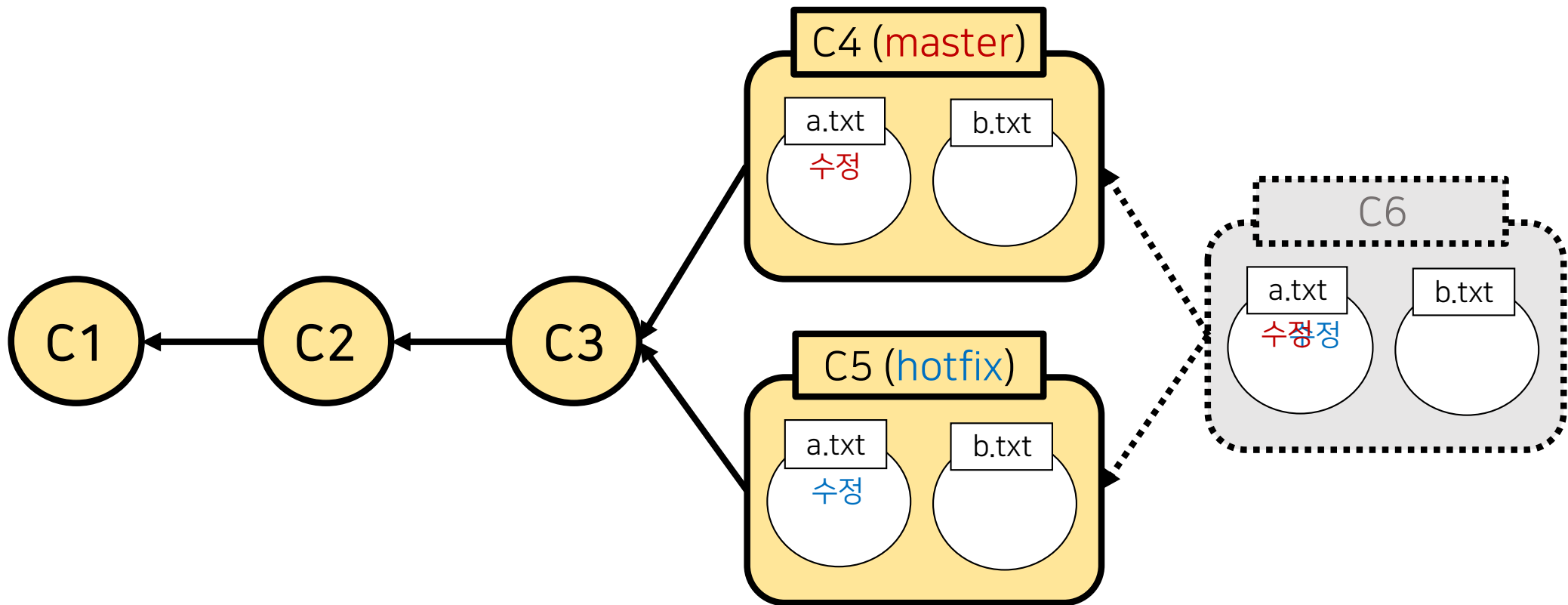
[참고] 충돌은 언제 일어나는가?

- 두 브랜치에서 서로 다른 파일을 수정 후 병합하는 경우 → 자연스럽게 Merge



[참고] 충돌은 언제 일어나는가?

- 두 브랜치에서 **같은 파일의 같은 부분**을 수정 후 병합하는 경우 → 충돌(Conflict)



| 따라하기

- Vscode에서 git merge 실습
 1. Fast-Forward 상황 실습
 2. 다른 파일을 수정 후 병합하여 3-way Merge 상황 실습
 3. 같은 파일의 같은 부분을 수정 후 병합하여 Merge Conflict 상황 실습

이어서...

삼성 청년 SW 아카데미

Git workflow

| 개요

- Branch와 원격 저장소를 이용해 협업을 하는 두 가지 방법
 - 원격 저장소 소유권이 있는 경우 → Shared repository model
 - 원격 저장소 소유권이 없는 경우 → Fork & Pull model

Shared repository model

Shared repository model

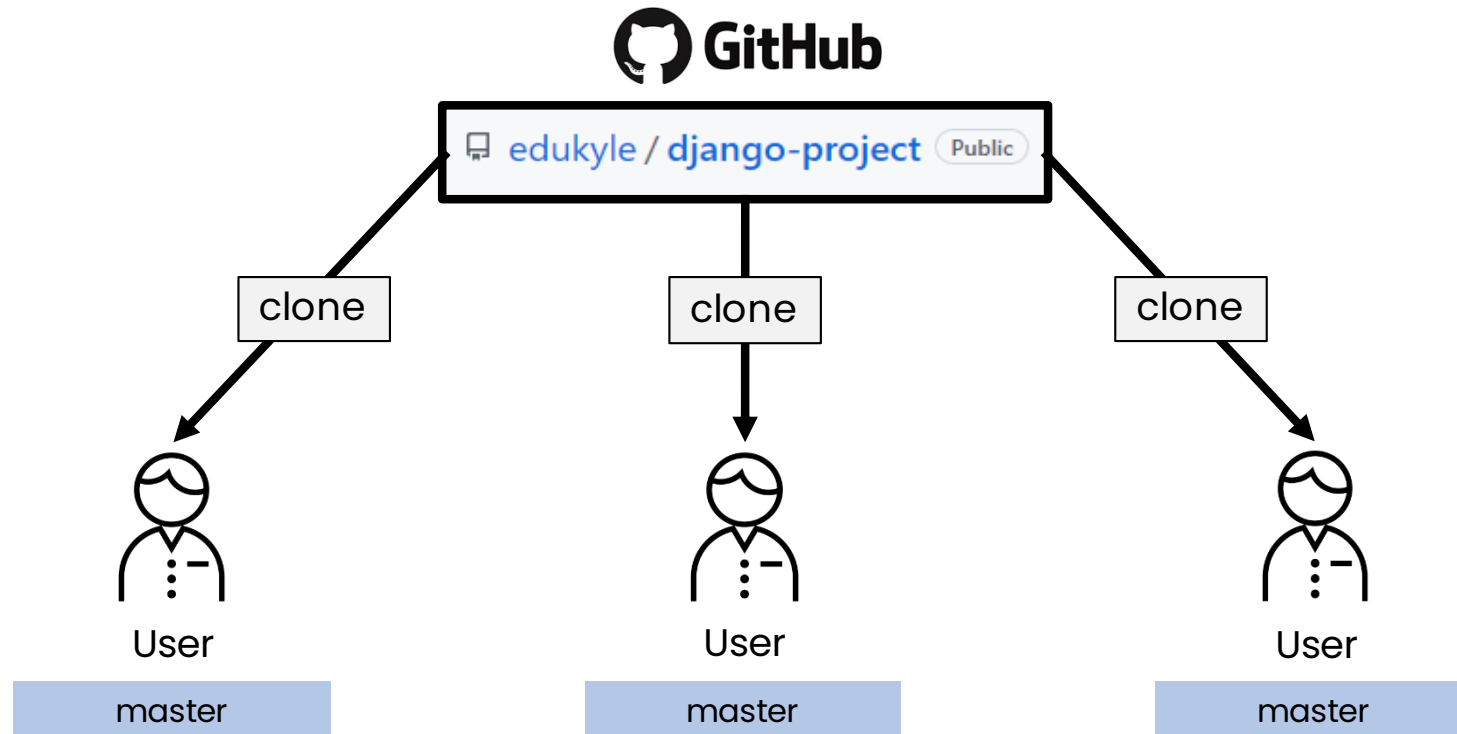
| 개요

- 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

Shared repository model

따라하기 (1/10)

- 소유권이 있는 원격 저장소를 로컬 저장소로 clone 받기



Shared repository model

따라하기 (2/10)

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



edukyle / django-project Public



User

master

feature/login



User

master

feature/signup



User

master

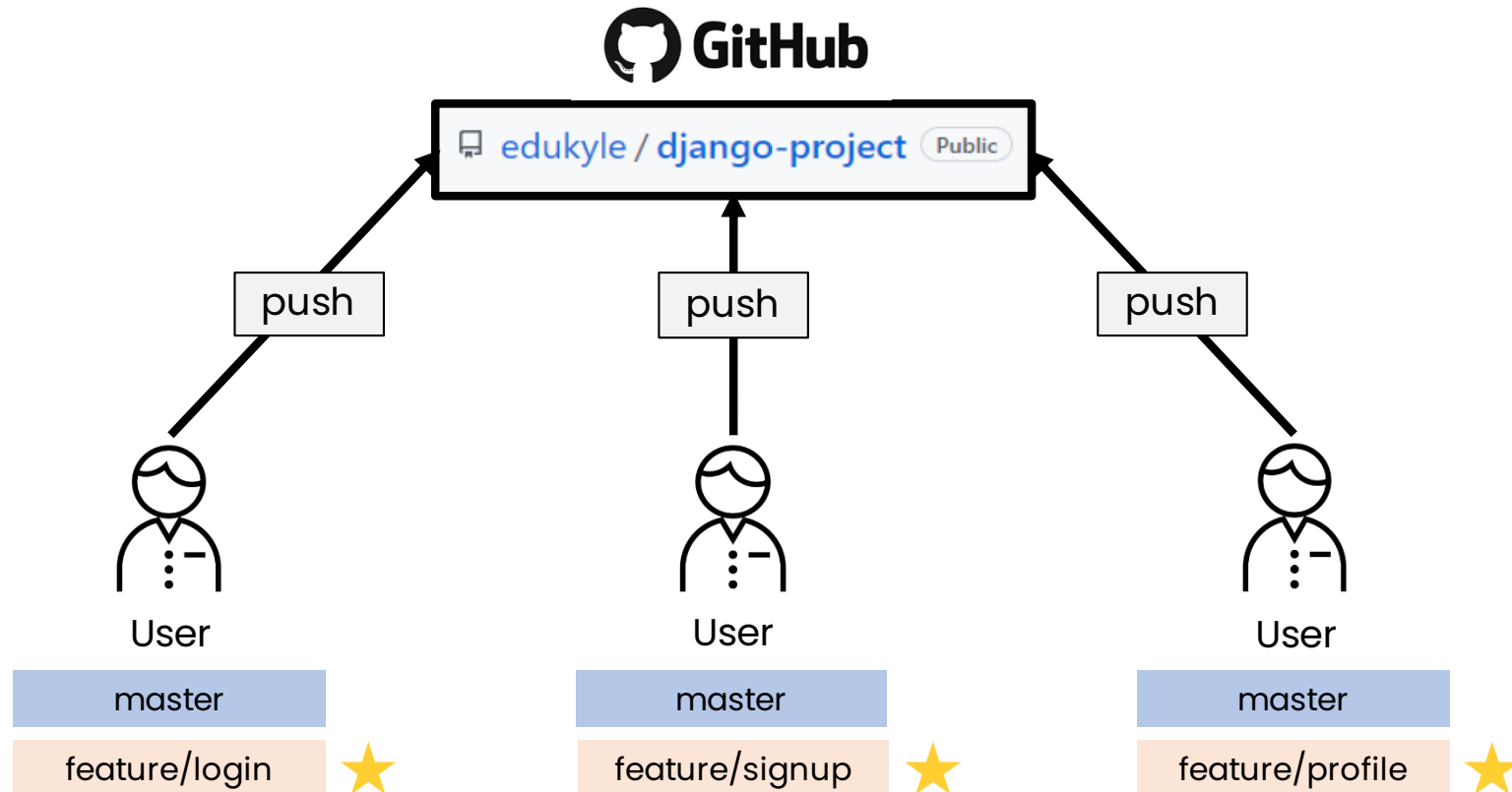
feature/profile



Shared repository model

따라하기 (3/10)

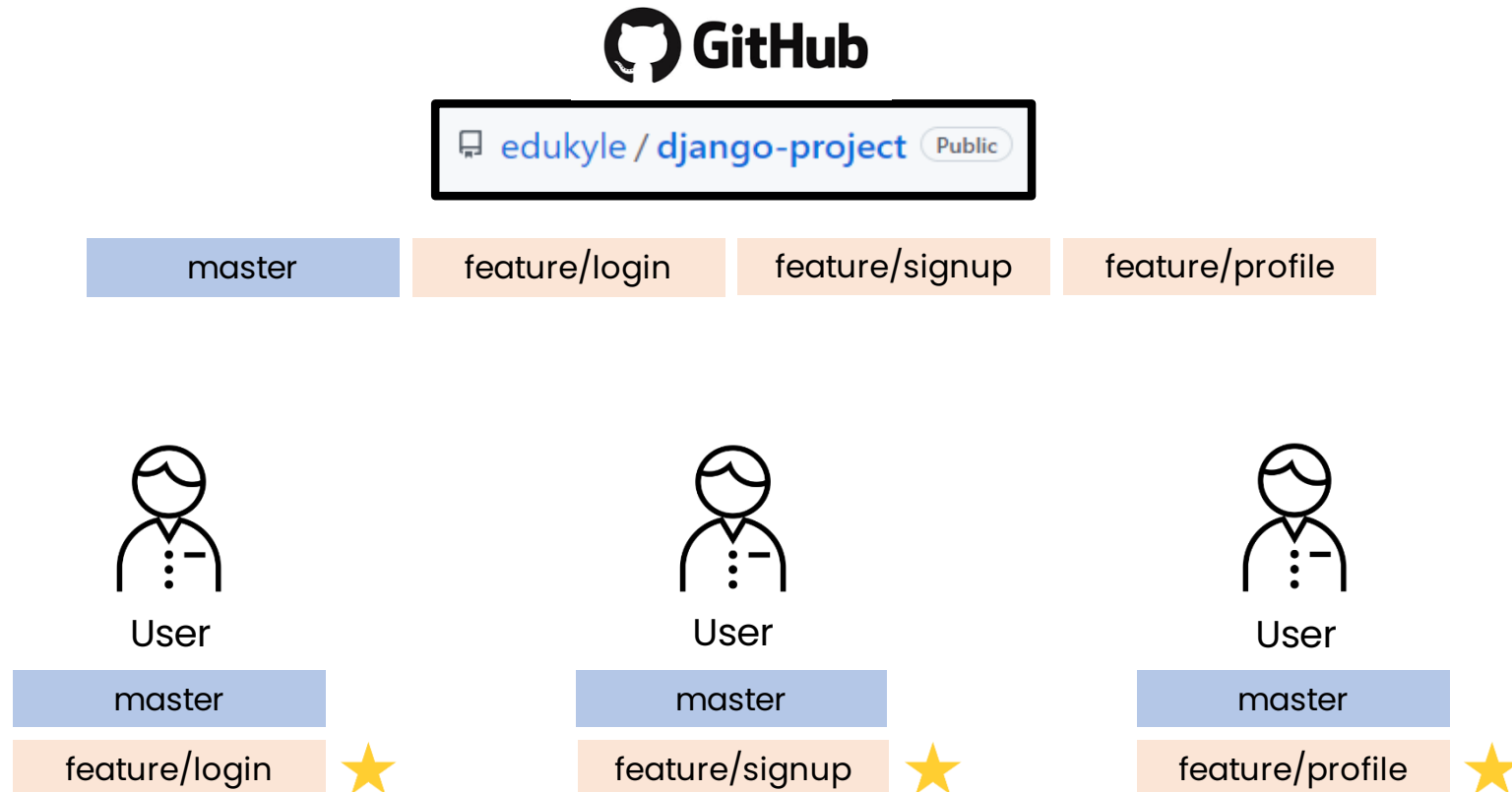
- 기능 구현이 완료되면, 원격 저장소에 해당 브랜치를 Push



Shared repository model

따라하기 (4/10)

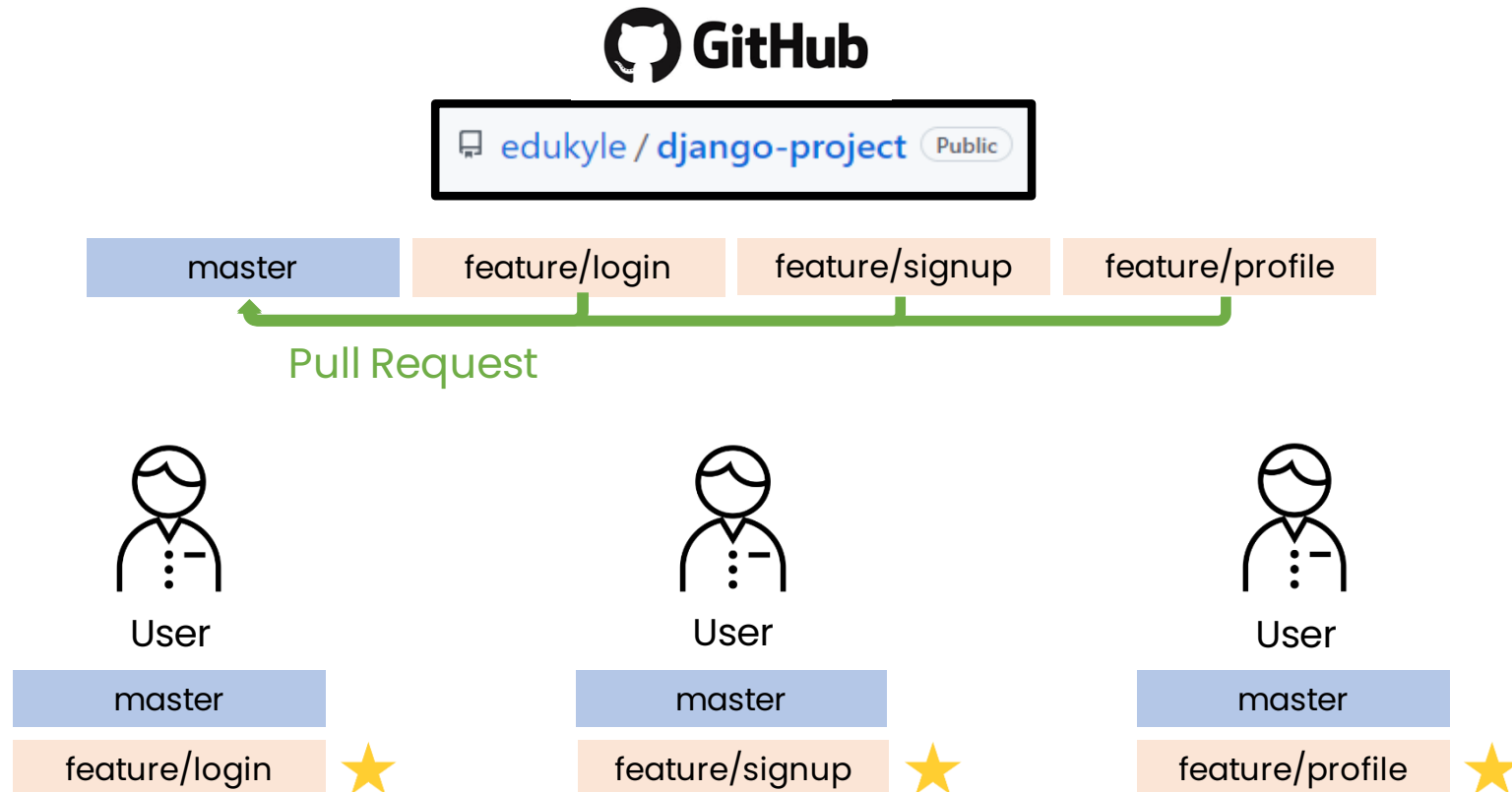
- 원격 저장소에 각 기능의 브랜치가 반영됨



Shared repository model

따라하기 (5/10)

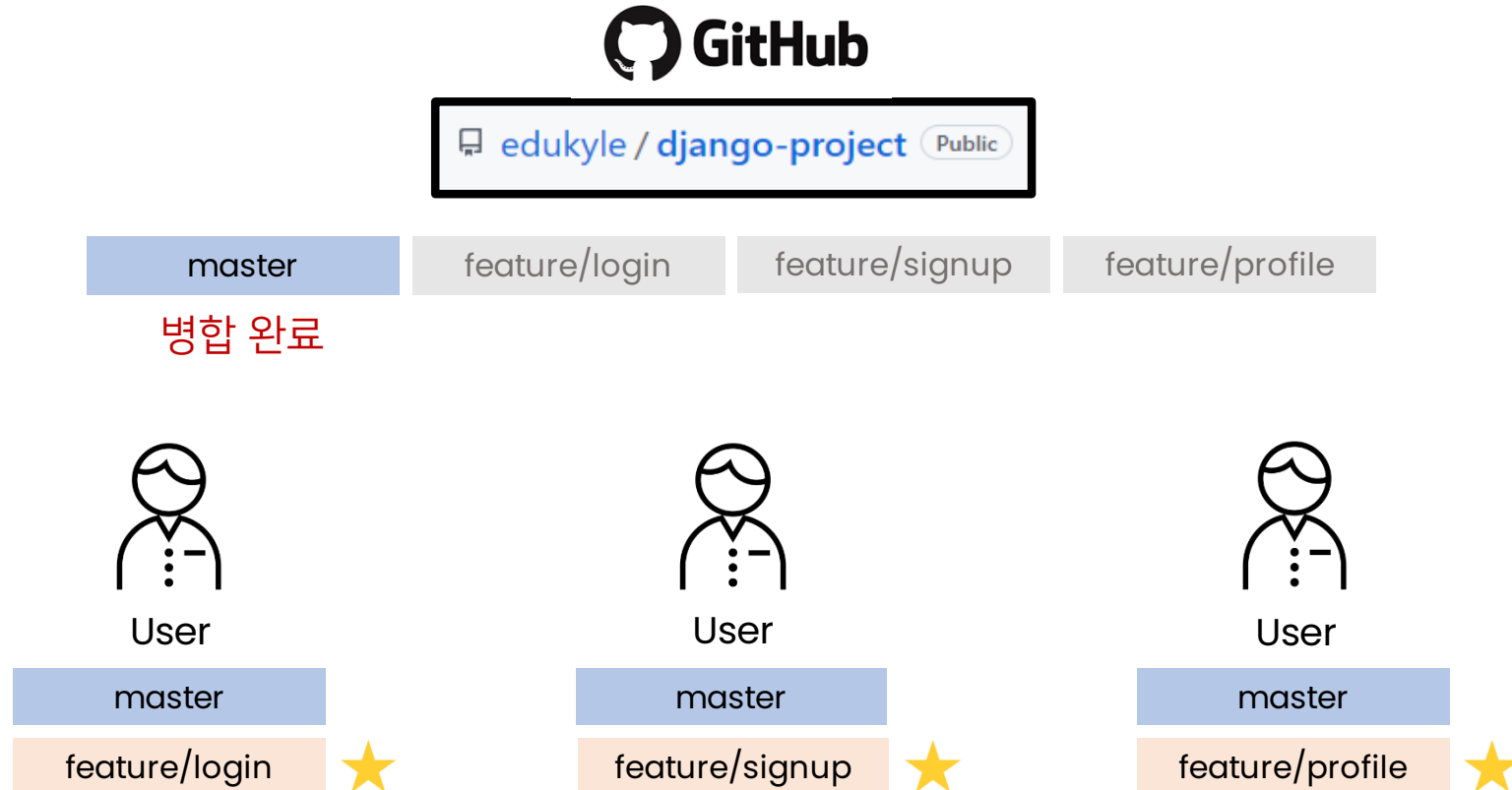
- Pull Request를 통해 브랜치를 master에 반영해달라는 요청을 보냄



Shared repository model

따라하기 (6/10)

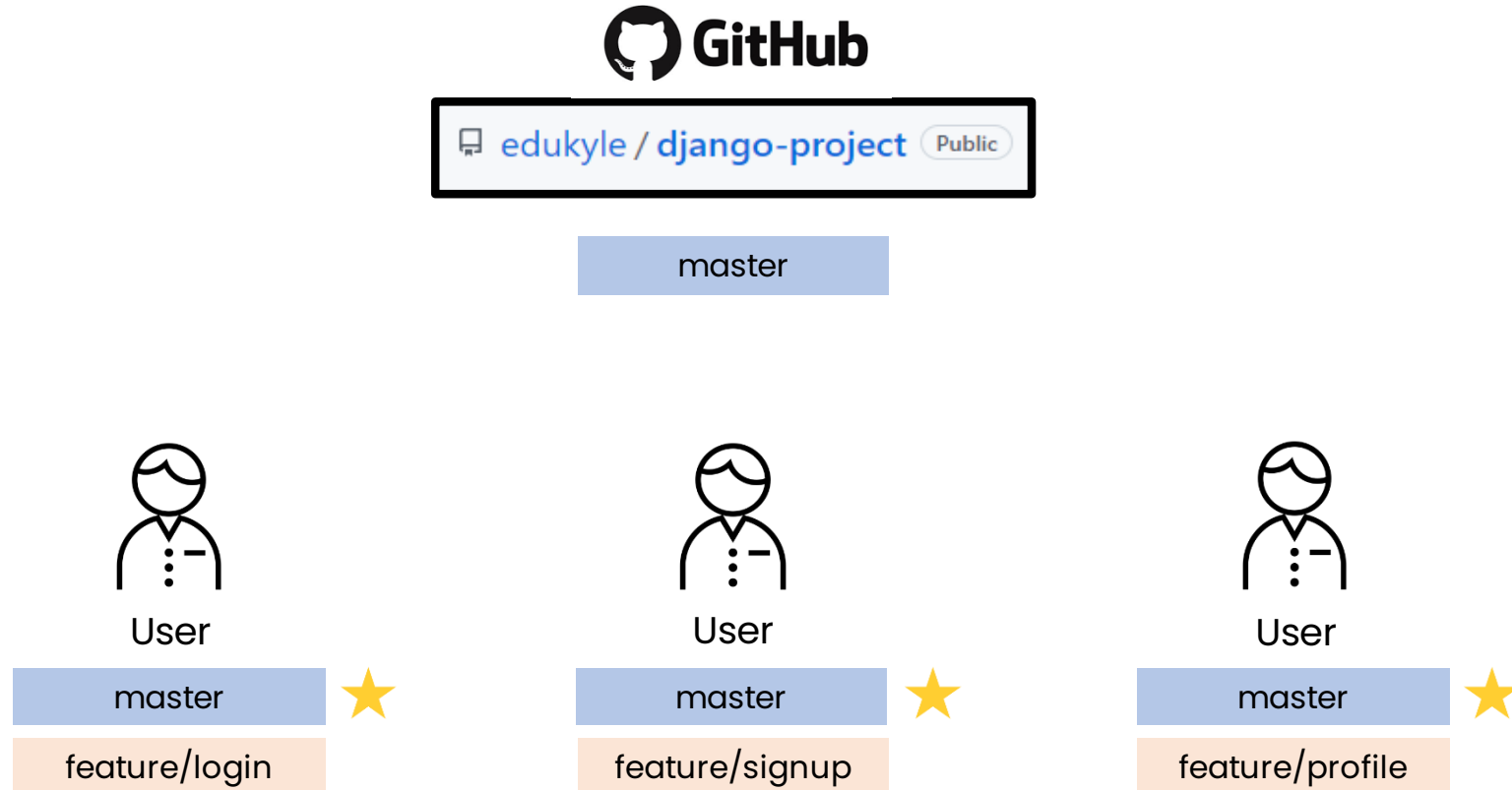
- 병합이 완료된 브랜치는 불필요하므로 원격 저장소에서 삭제



Shared repository model

따라하기 (7/10)

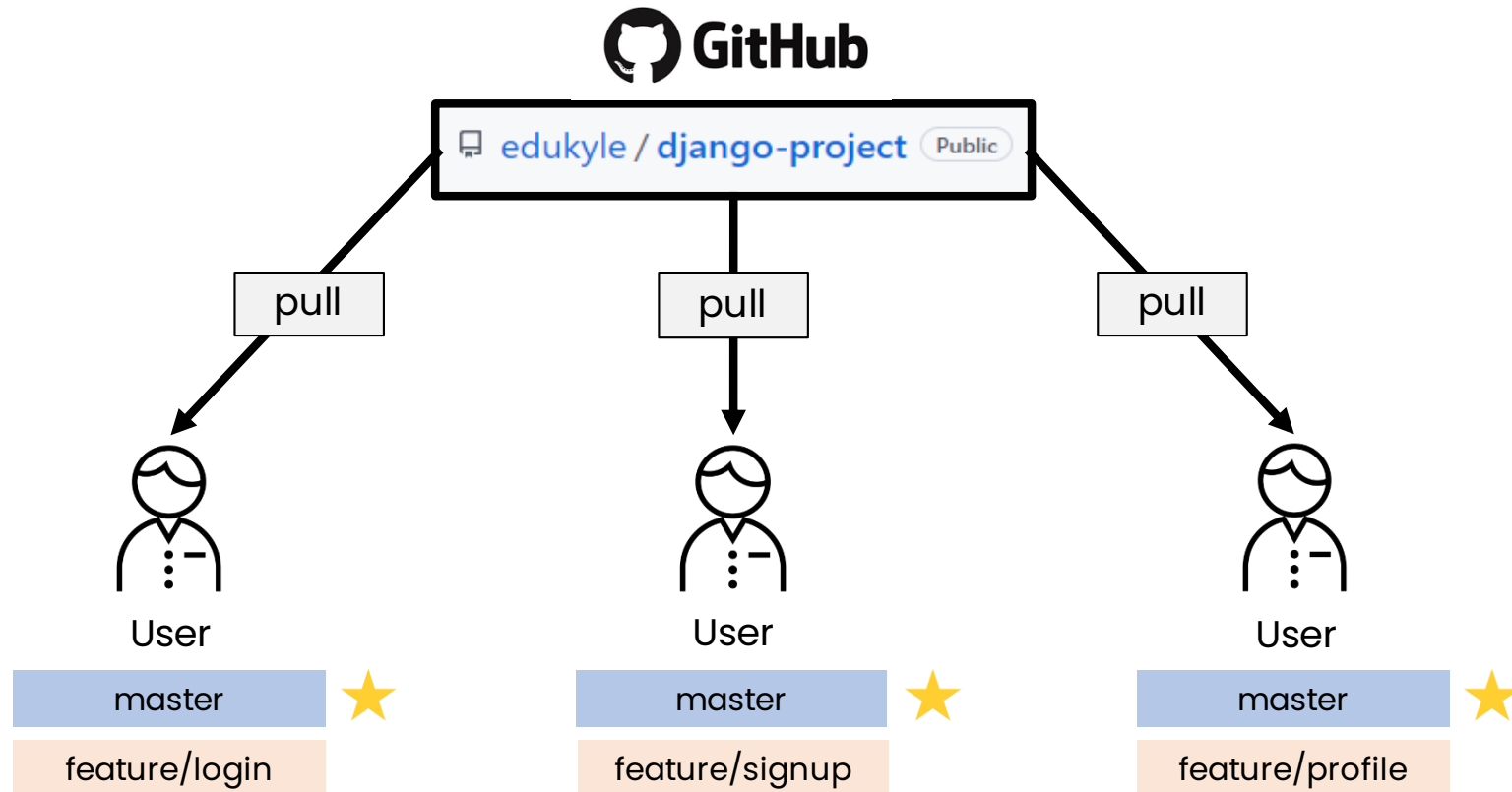
- 원격 저장소에서 병합이 완료되면, 사용자는 로컬에서 master 브랜치로 switch



Shared repository model

따라하기 (8/10)

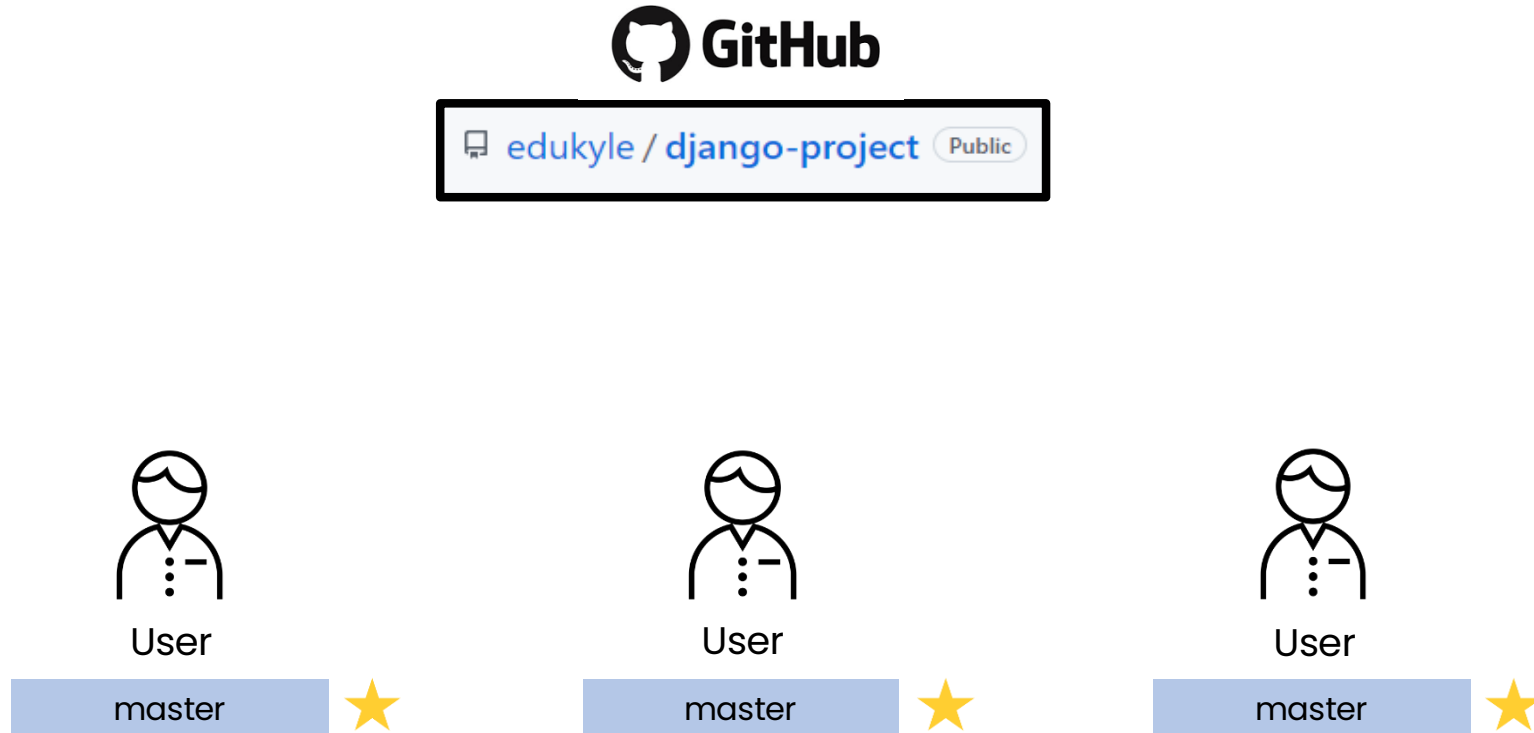
- 병합으로 인해 변경된 원격 저장소의 master 내용을 로컬에 Pull



Shared repository model

따라하기 (9/10)

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



Shared repository model

따라하기 (10/10)

- 새 기능 추가를 위해 새로운 브랜치를 생성하며 지금까지의 과정을 반복



edukyle / django-project Public



User

master

feature/delete



User

master

feature/logout



User

master

feature/update



Fork & Pull model

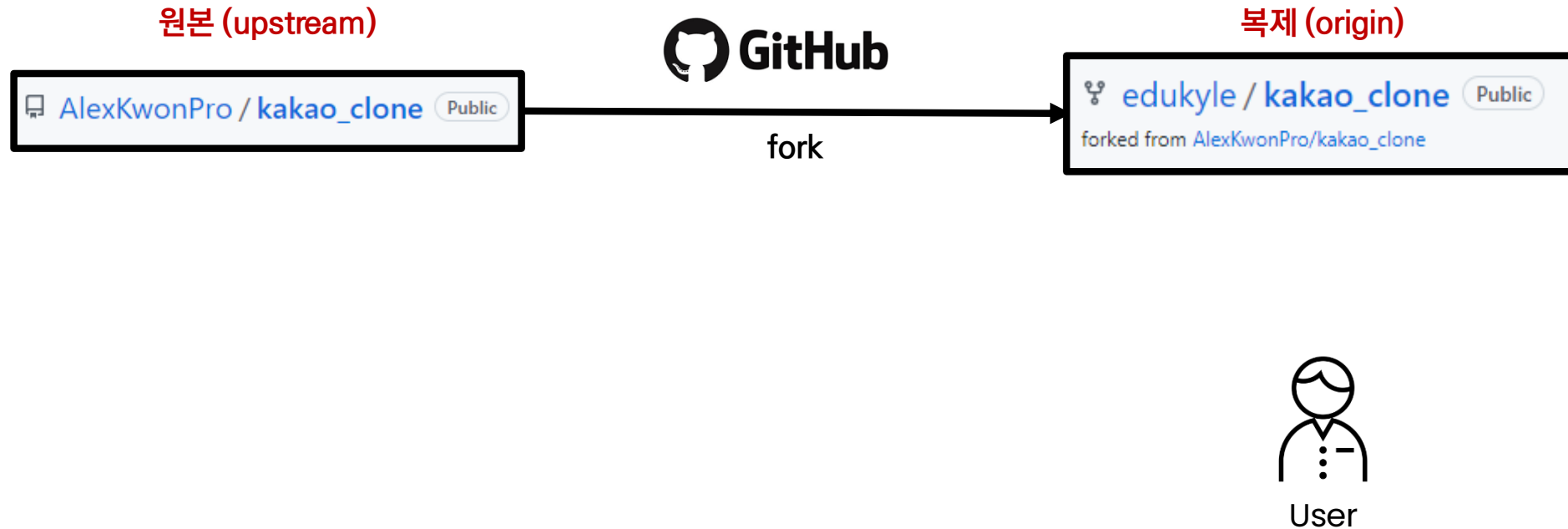
| 개요

- 오픈소스 프로젝트와 같이, 자신의 소유가 아닌 원격 저장소인 경우
- 원본 원격 저장소를 그대로 내 원격 저장소에 복제 (이러한 행위를 **Fork**라고 함)
- 기능 완성 후 복제한 내 원격 저장소에 Push
- 이후 **Pull Request**를 통해 원본 원격 저장소에 반영될 수 있도록 요청함

Fork & Pull model

따라하기 (1/12)

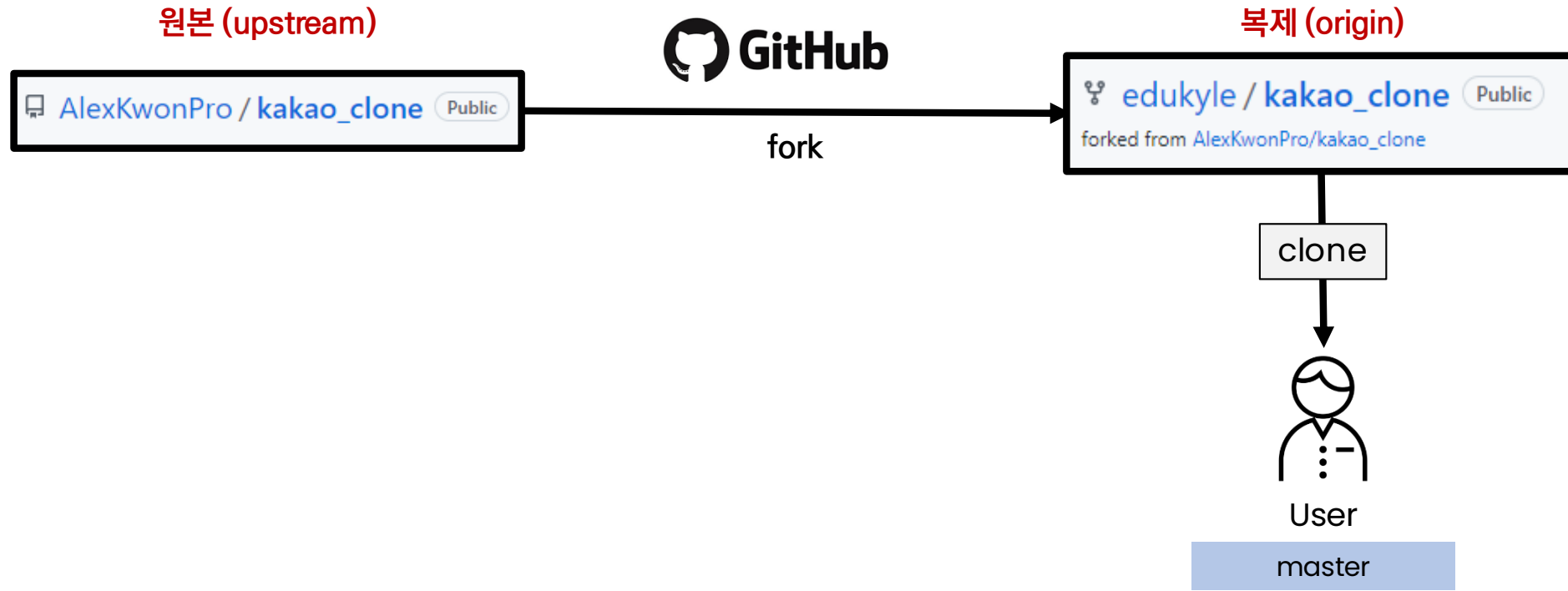
- 소유권이 없는 원격 저장소를 fork를 통해 내 원격 저장소로 복제



Fork & Pull model

따라하기 (2/12)

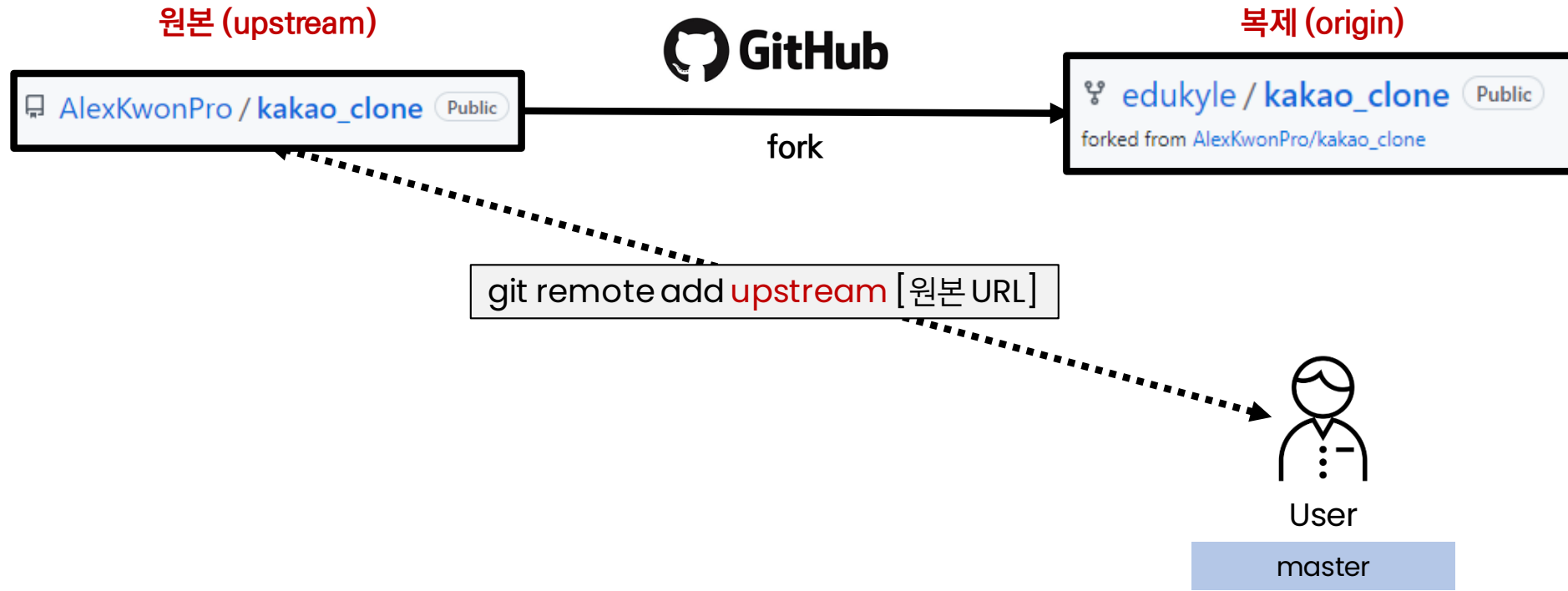
- fork 이후, 복제된 내 원격 저장소를 로컬 저장소에 clone



Fork & Pull model

따라하기 (3/12)

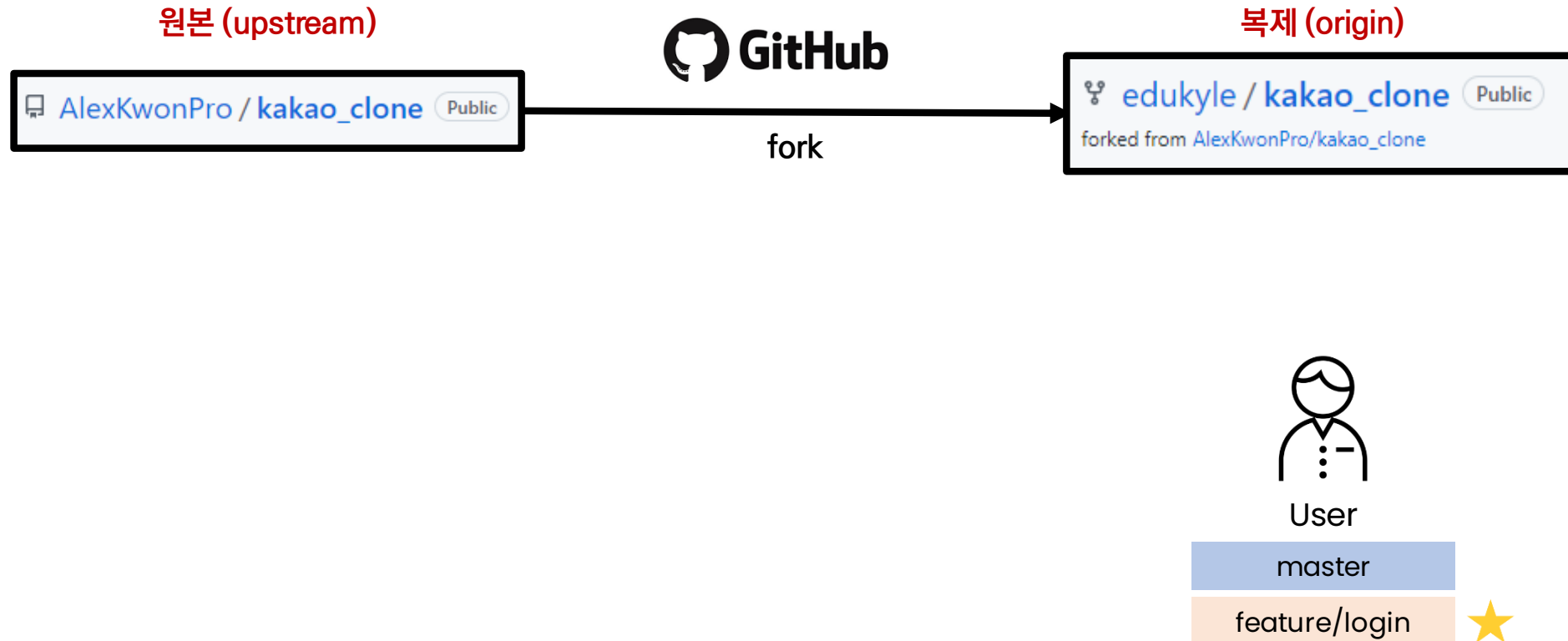
- 이후에 로컬 저장소와 원본 원격 저장소를 동기화 하기 위해 연결



Fork & Pull model

따라하기 (4/12)

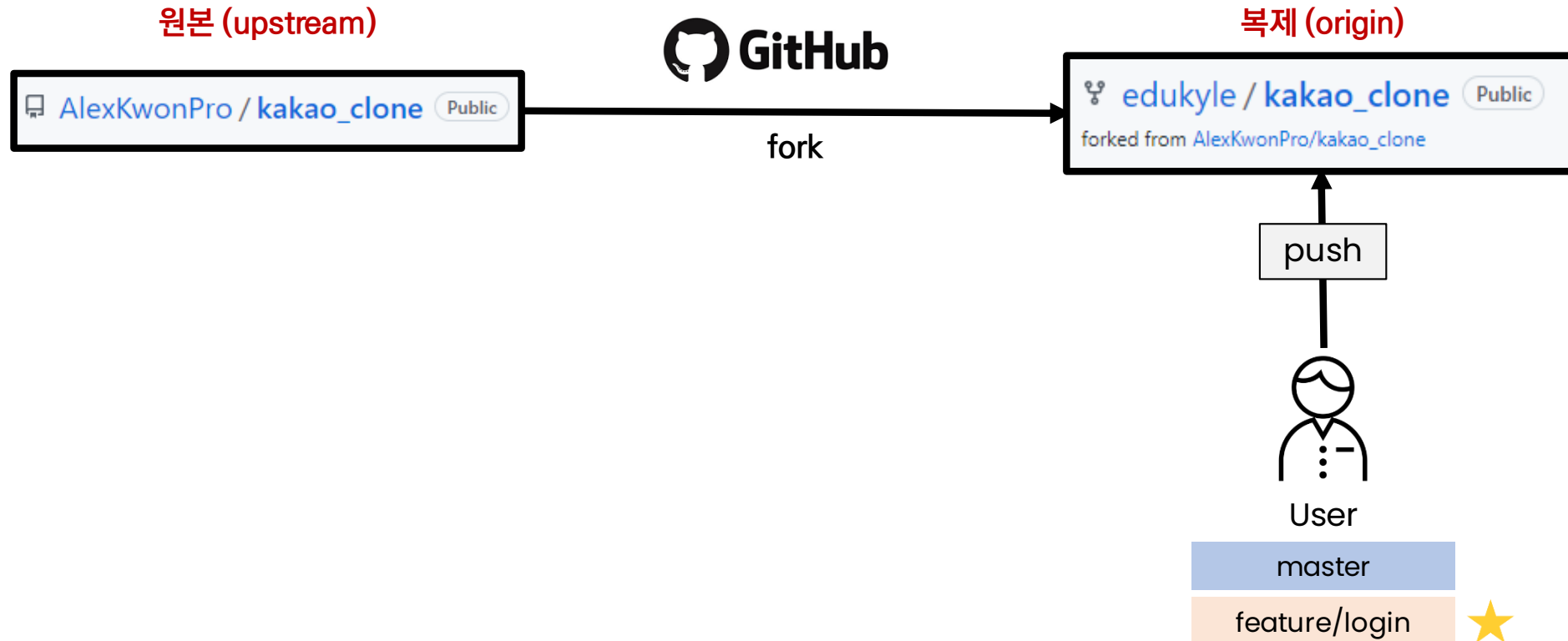
- 사용자는 자신이 작업할 기능에 대한 브랜치를 생성하고, 그 안에서 기능을 구현



Fork & Pull model

따라하기 (5/12)

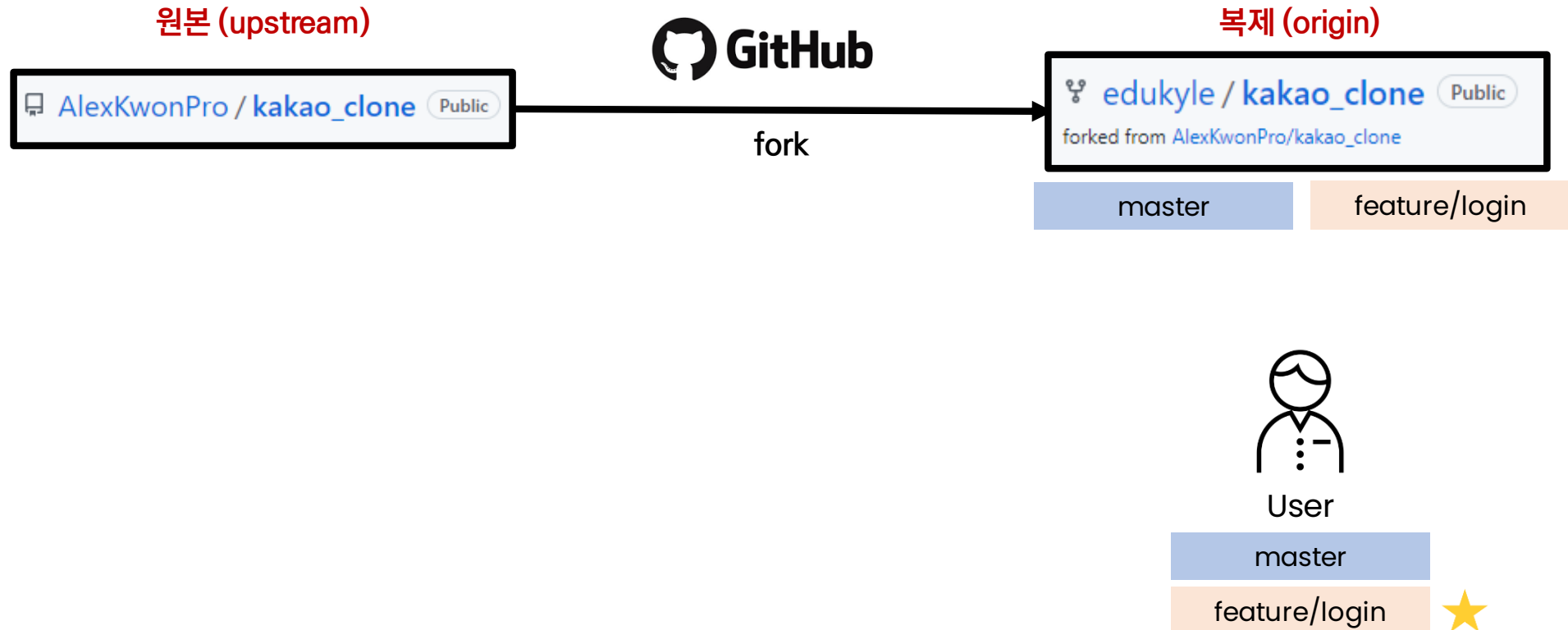
- 기능 구현이 완료되면, 복제 원격 저장소(origin)에 해당 브랜치를 Push



Fork & Pull model

따라하기 (6/12)

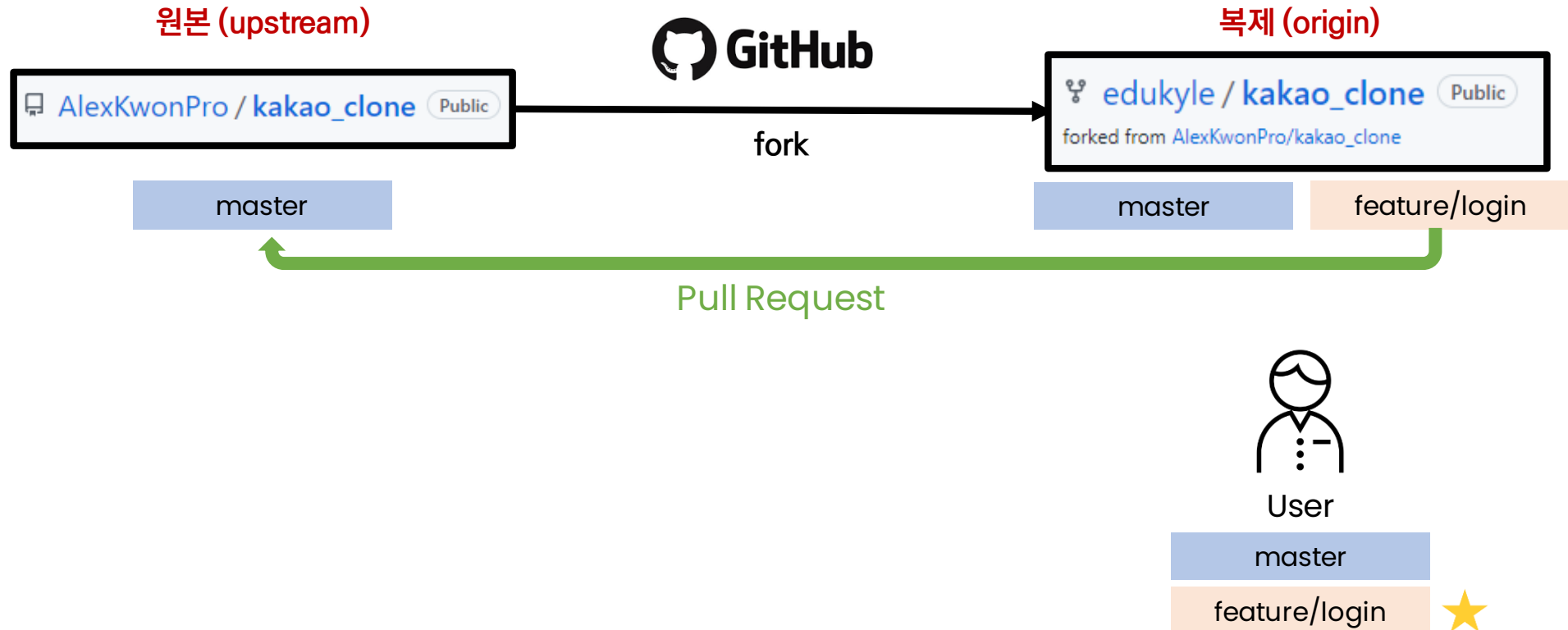
- 복제 원격 저장소(origin)에 브랜치가 반영됨



Fork & Pull model

따라하기 (7/12)

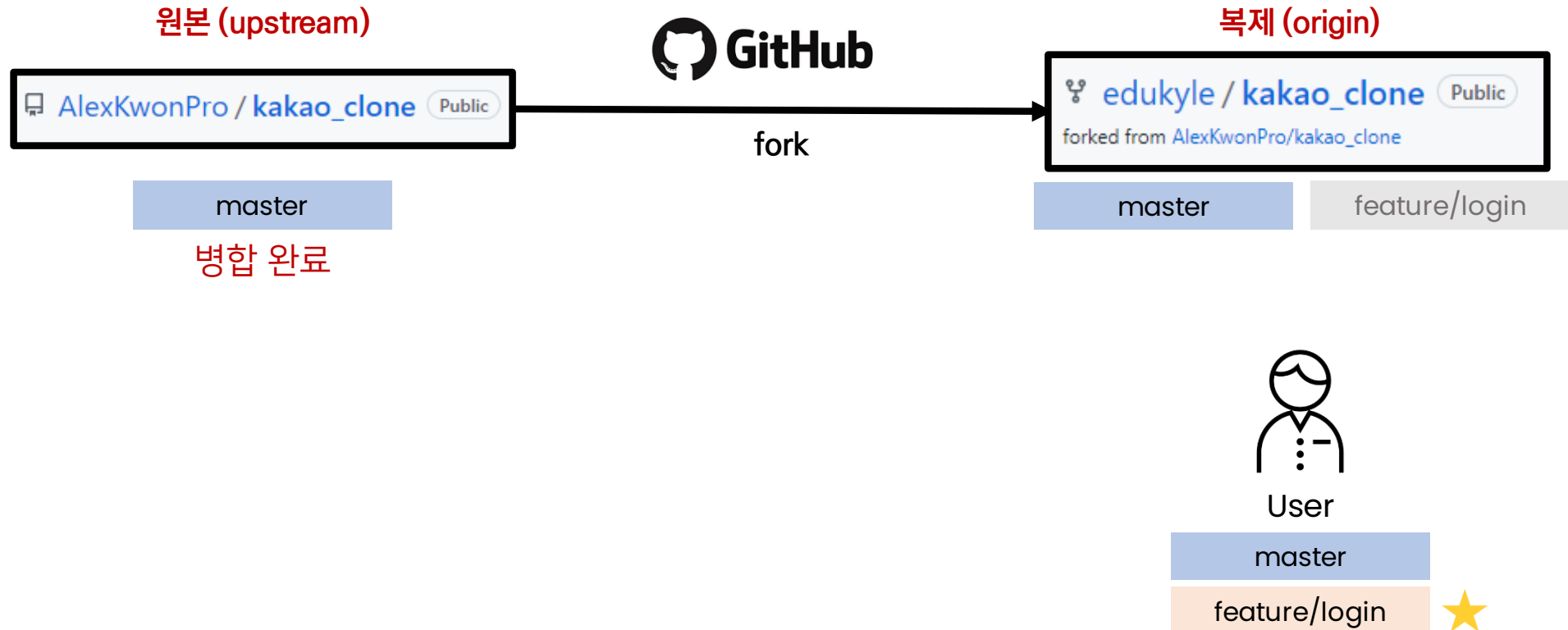
- Pull Request를 통해 origin의 브랜치를 upstream에 반영해달라는 요청을 보냄



Fork & Pull model

따라하기 (8/12)

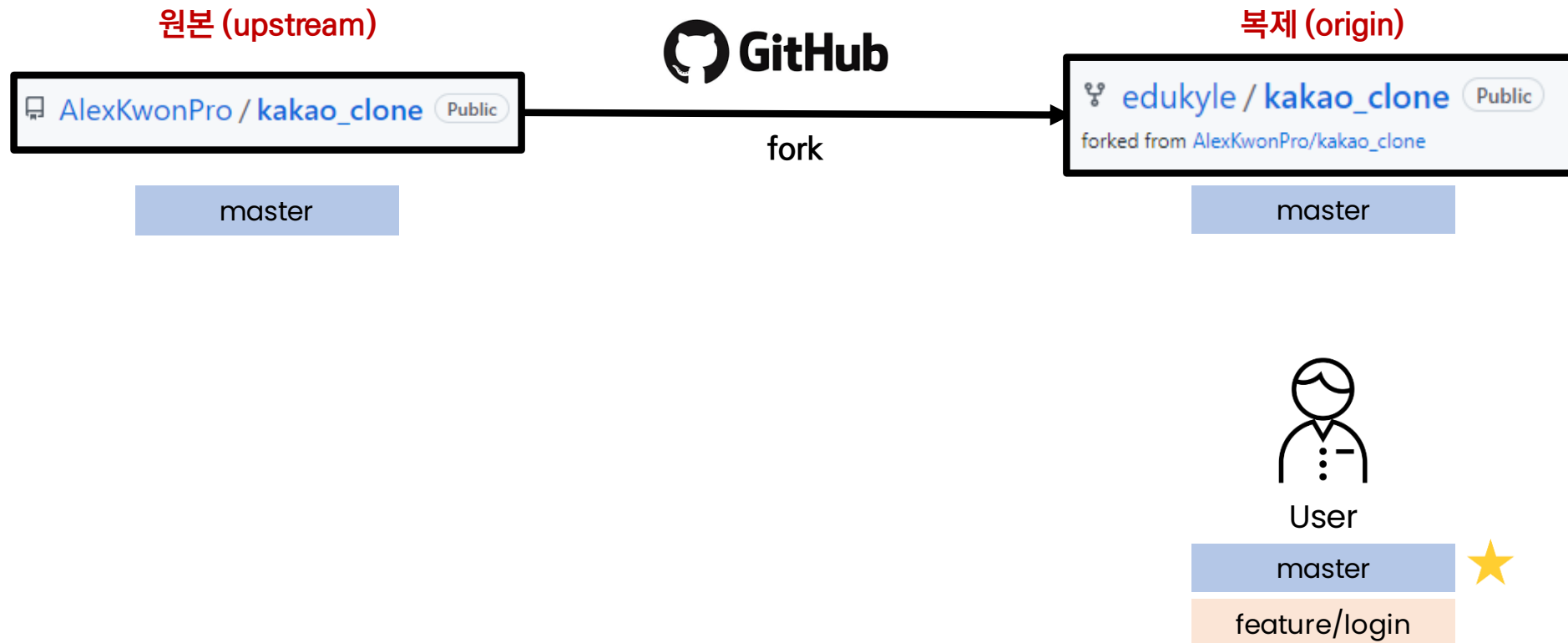
- upstream에 브랜치가 병합되면 origin의 브랜치는 삭제



Fork & Pull model

따라하기 (9/12)

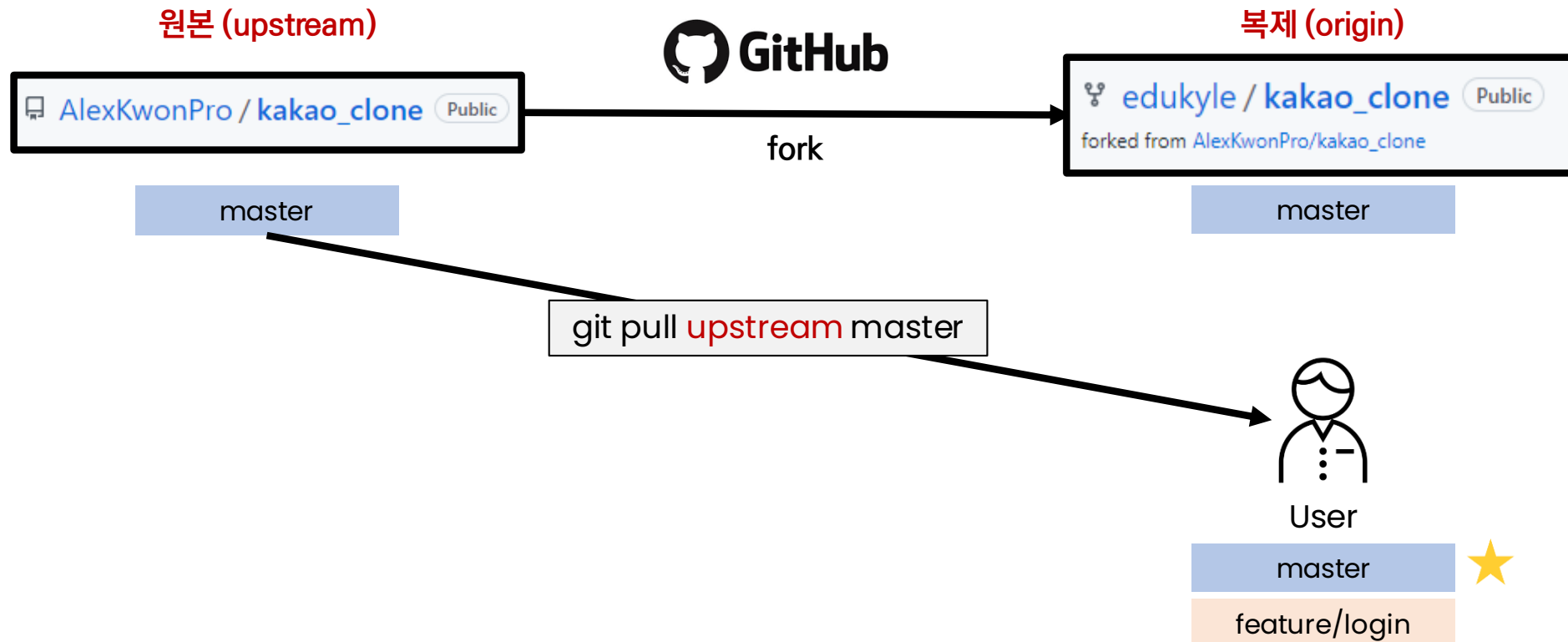
- 이후 사용자는 로컬에서 master 브랜치로 switch



Fork & Pull model

따라하기 (10/12)

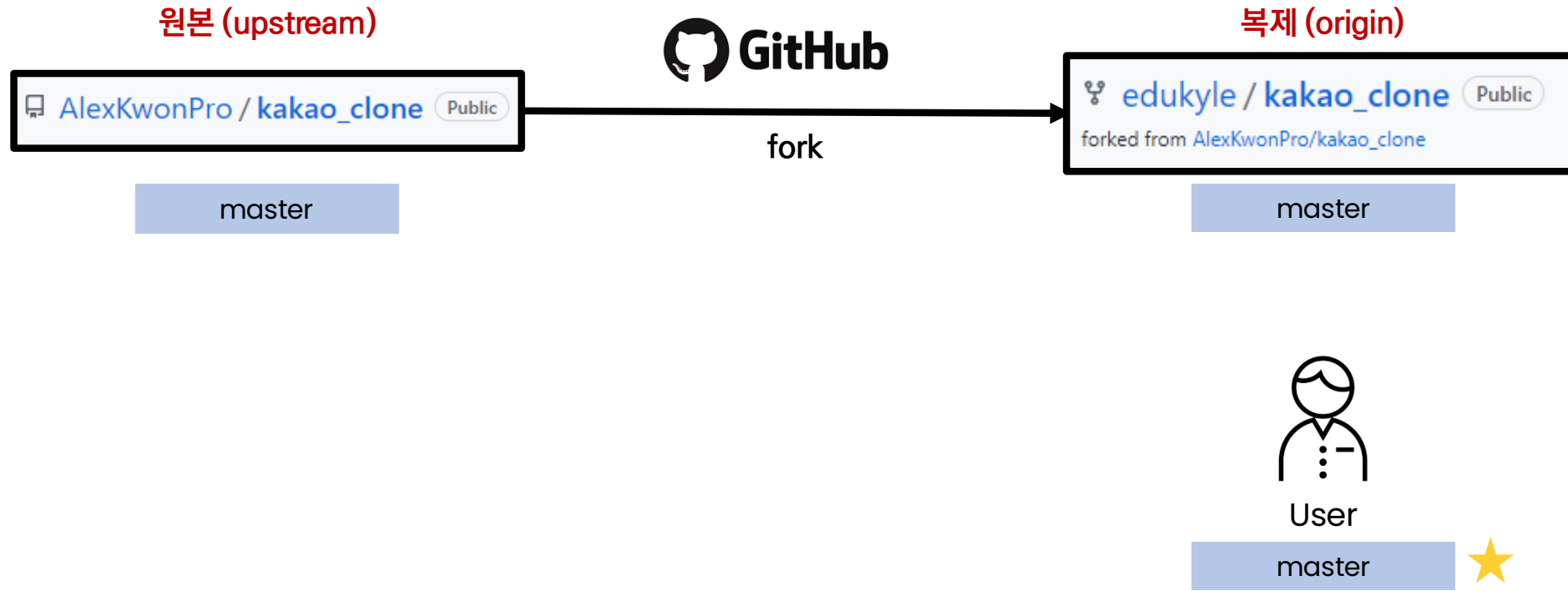
- 병합으로 인해 변경된 upstream의 master 내용을 로컬에 Pull



Fork & Pull model

따라하기 (11/12)

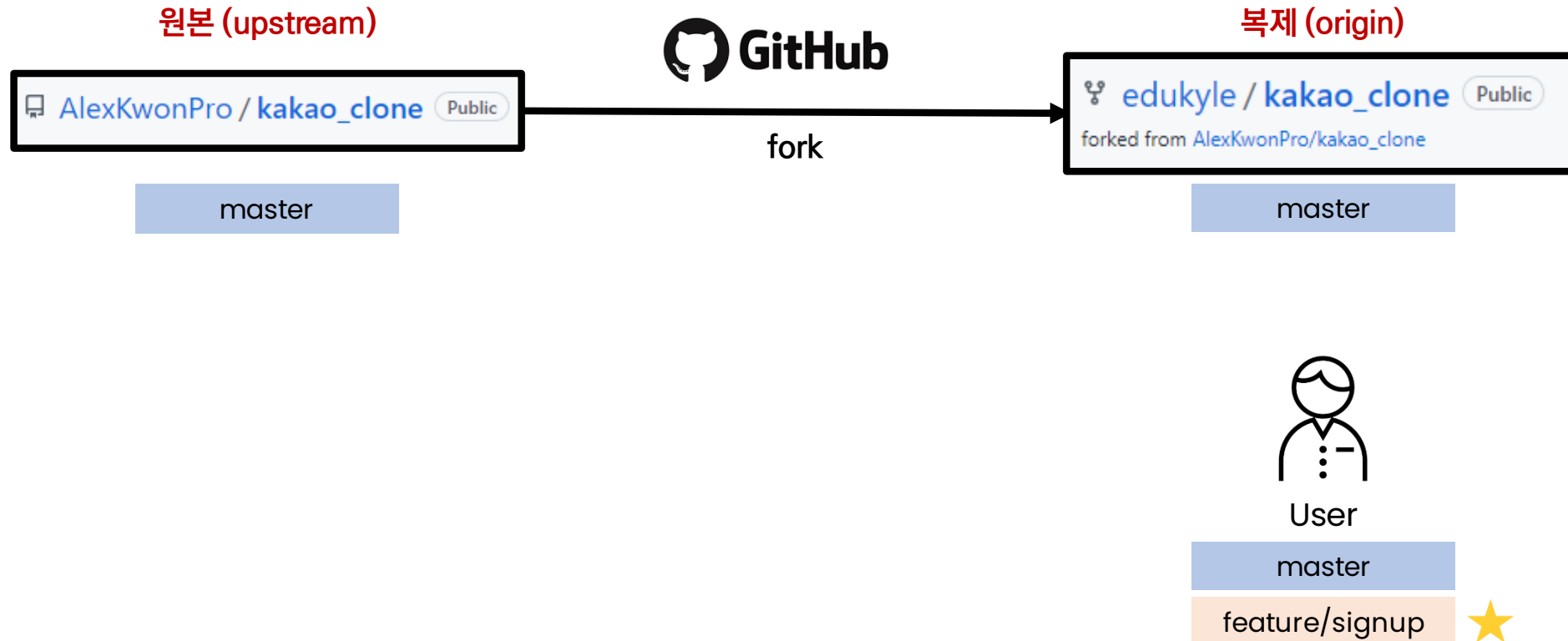
- upstream의 master 내용을 받았으므로, 기존 로컬 브랜치 삭제 (한 사이클 종료)



Fork & Pull model

따라하기 (12/12)

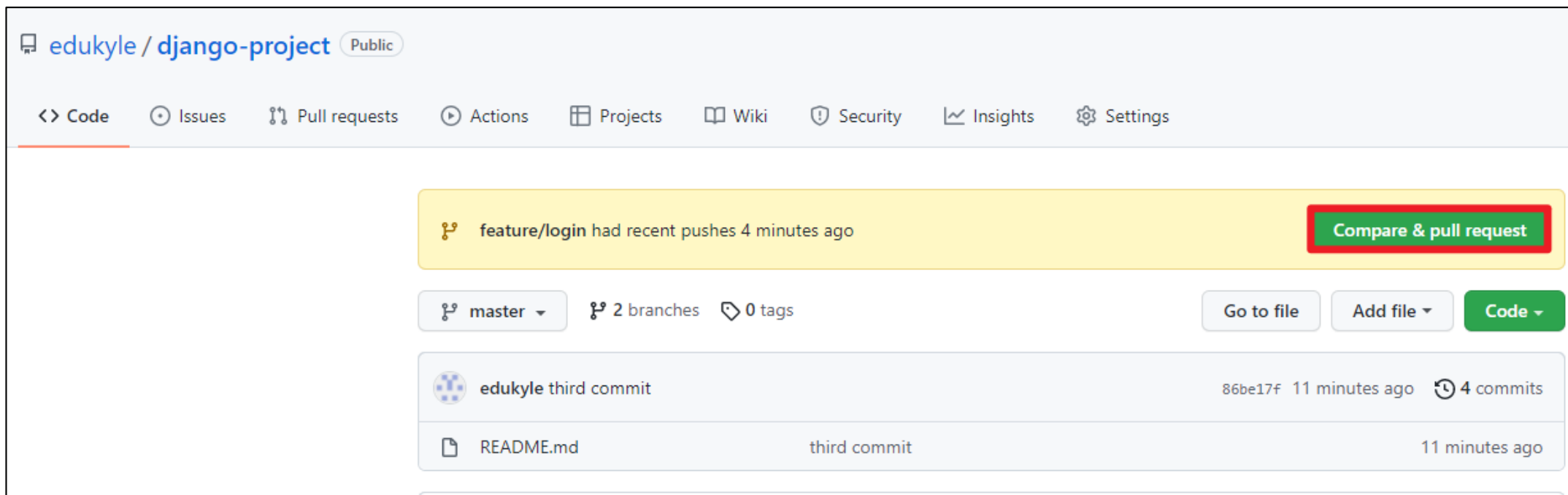
- 새로운 기능 추가를 위해 새로운 브랜치를 생성하며 위 과정을 반복



[참고] PR 자세히 알아보기

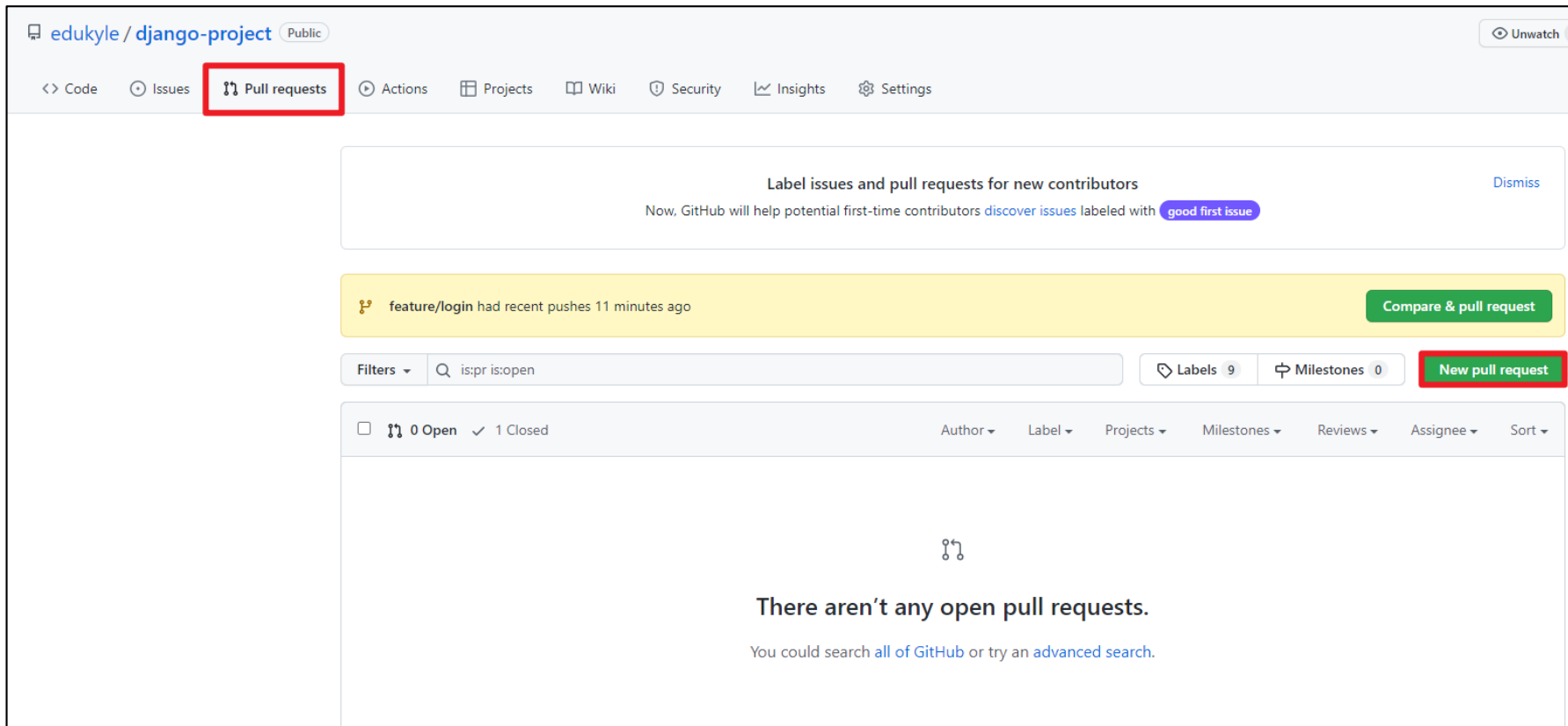
Github에서 Pull Request 보내기 (1/10)

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



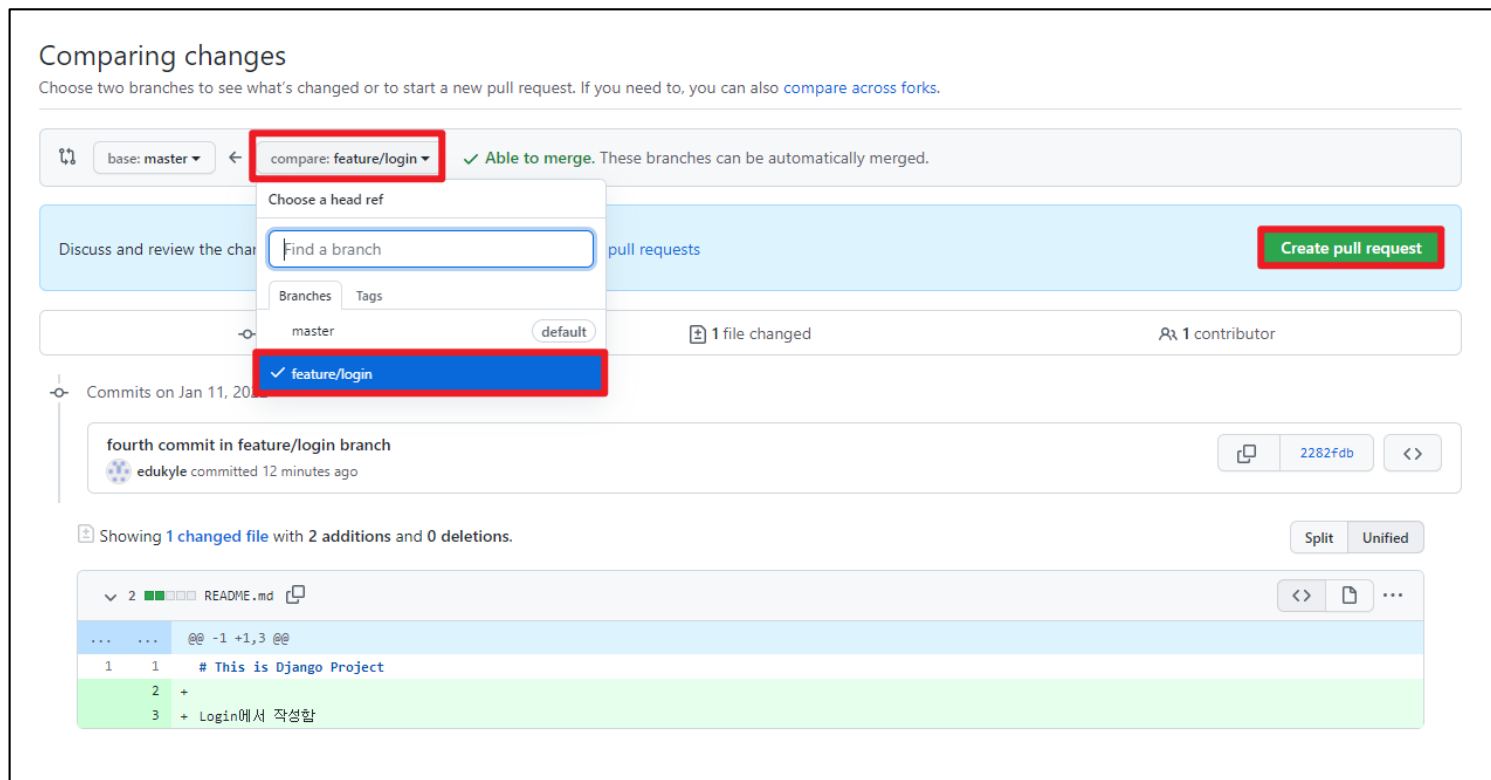
Github에서 Pull Request 보내기 (2/10)

- 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



Github에서 Pull Request 보내기 (4/10)

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

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: master ← compare: feature/login ✓ Able to merge. These branches can be automatically merged.

이곳에 Pull Request의 제목을 작성

Write Preview

이곳에 상세 내용을 작성

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

- Reviewers : 현재 PR에 대해 코드 리뷰를 진행해 줄 담당자
- Assignees : 현재 PR에 대한 작업을 맡고 있는 담당자

Reviewers
No reviews

Assignees
No one—assign yourself

Labels
None yet

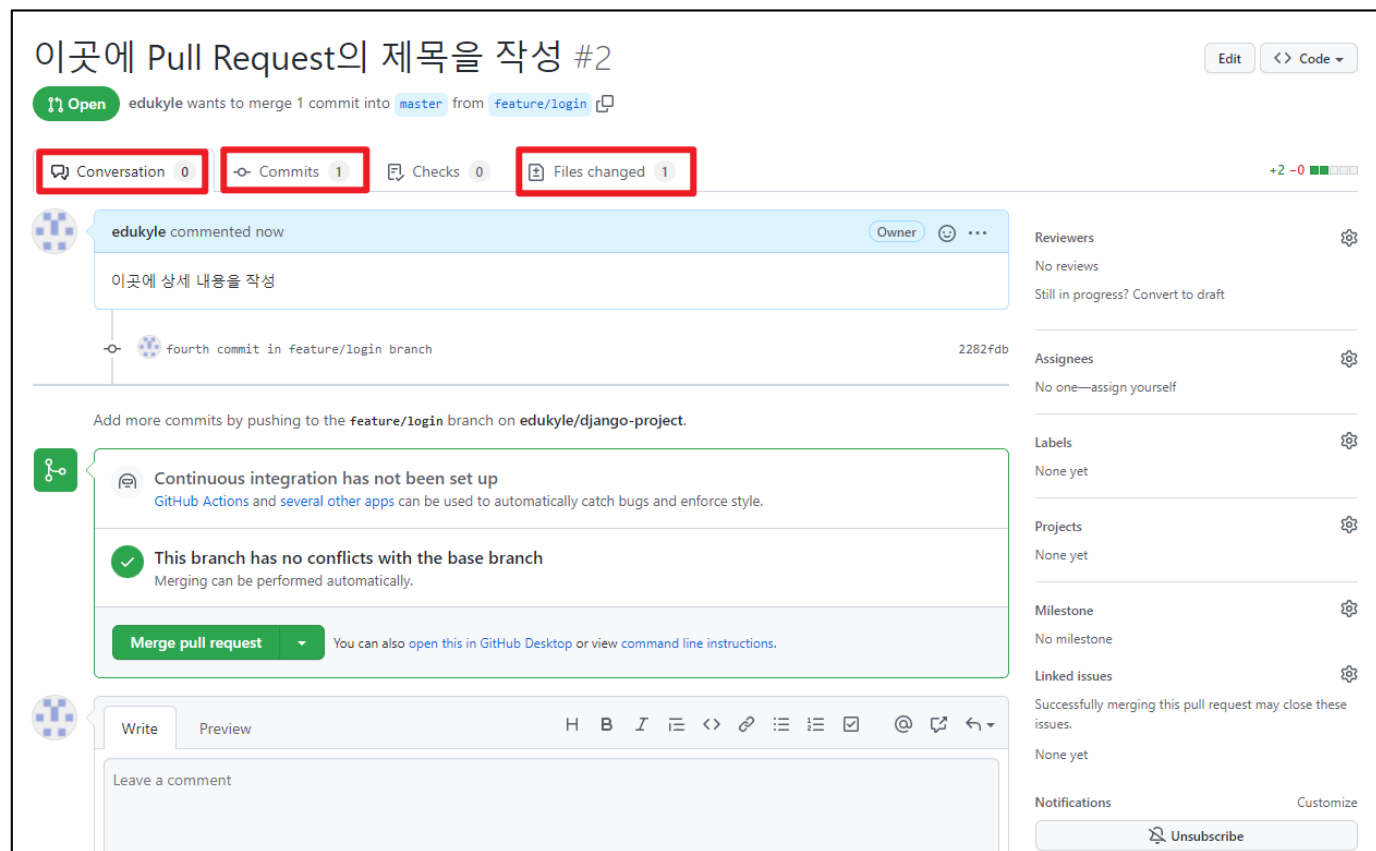
Projects
None yet

Milestone
No milestone

Linked issues
Use Closing keywords in the description to automatically close issues

Github에서 Pull Request 보내기 (5/10)

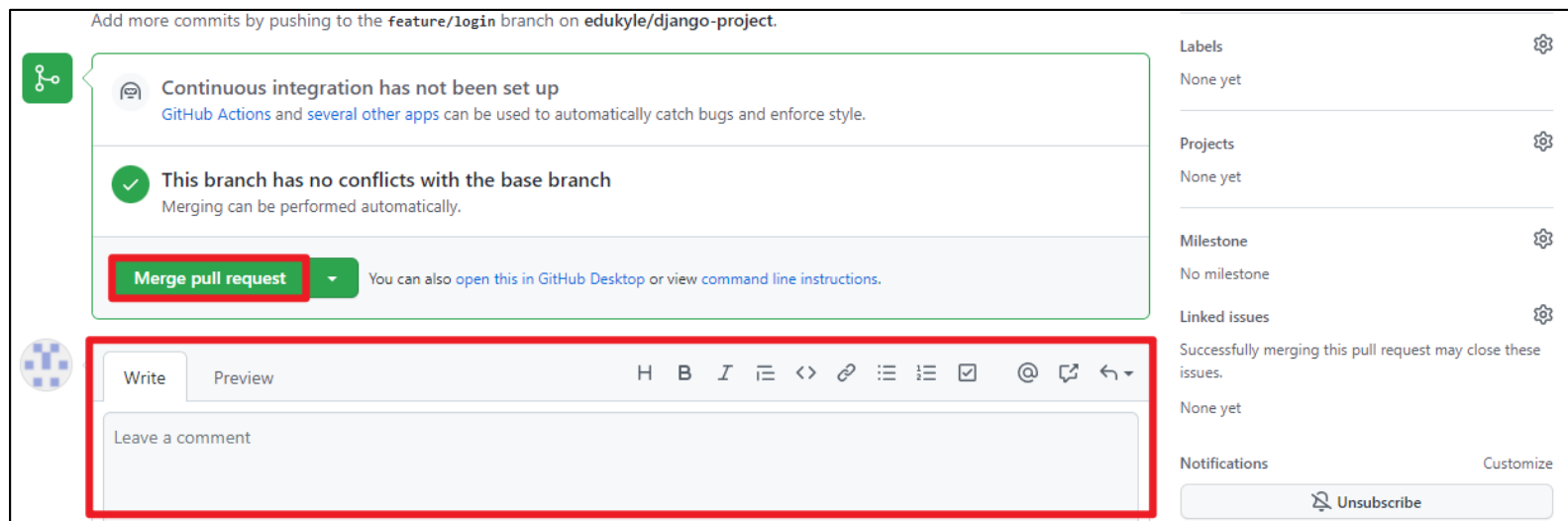
- PR이 생성되면 Conversation, Commits, Files changed 화면을 확인 가능



Github에서 Pull Request 보내기 (5/10)

1. Conversation

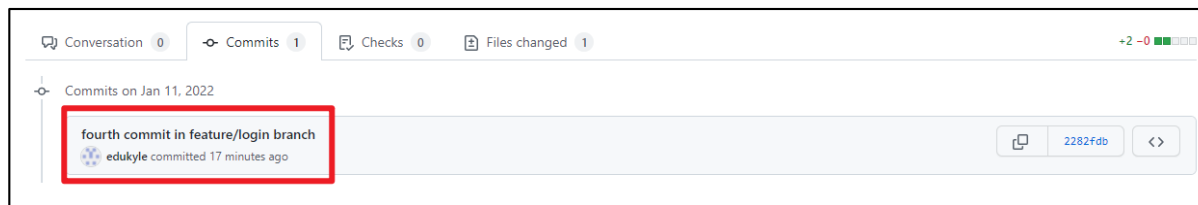
- 아래 Write 부분에서 별도로 comment를 작성할 수 있음
- Merge pull request 버튼을 누르면 병합 시작
- 충돌(conflict) 상황에서는 충돌을 해결하라고 나타남



Github에서 Pull Request 보내기 (5/10)

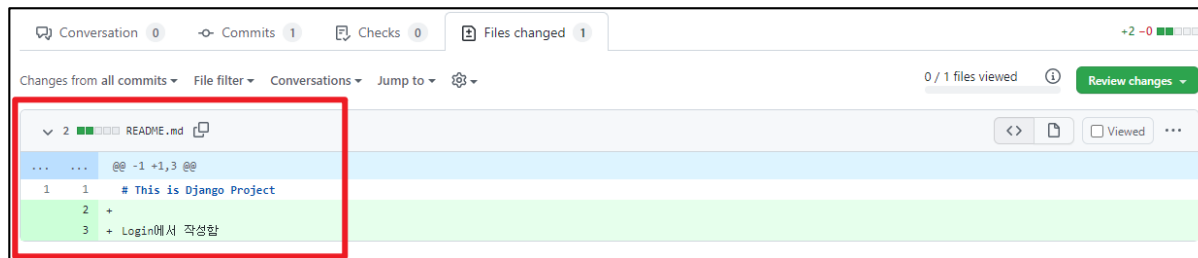
2. Commits

- PR을 통해 반영될 커밋들을 볼 수 있음



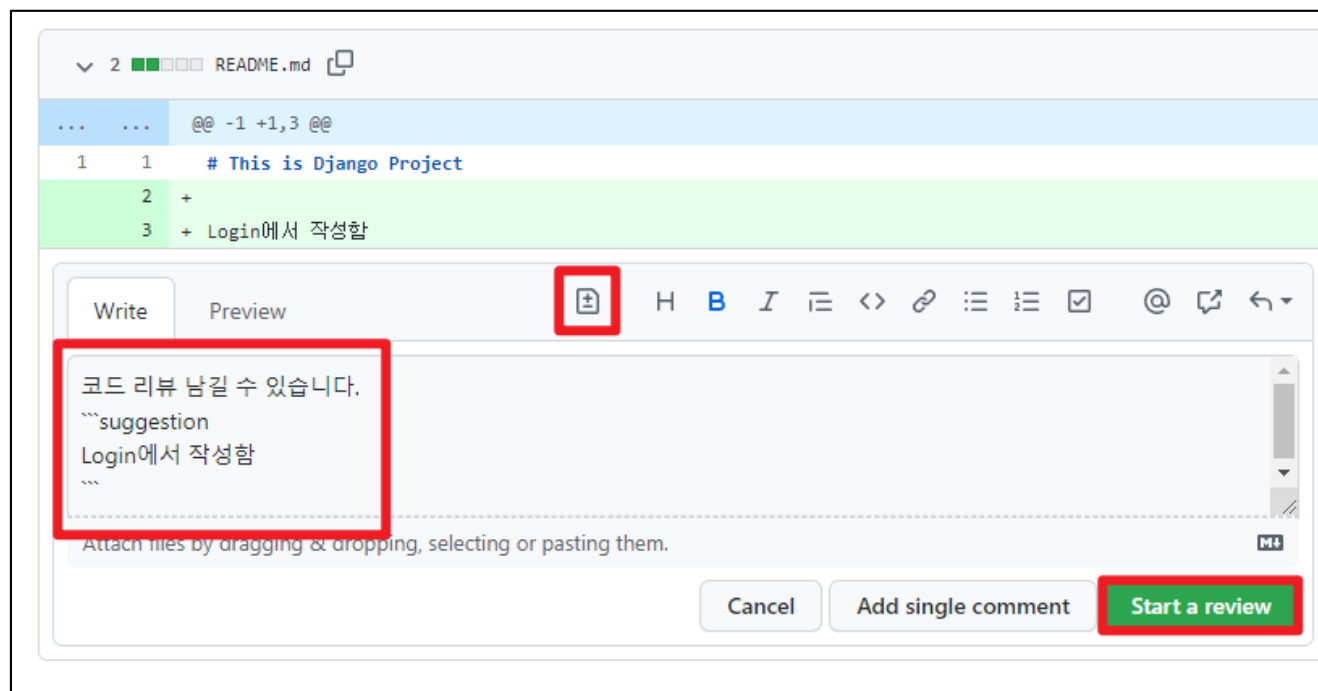
3. Files changed

- 파일의 변화 내역들을 볼 수 있음



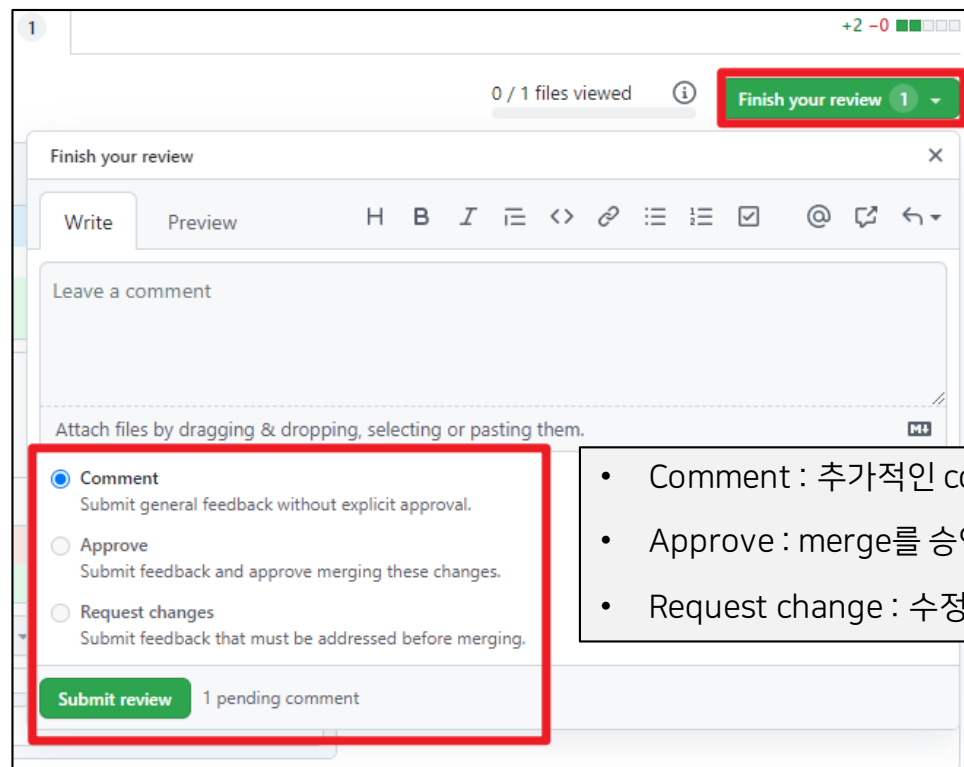
Github에서 Pull Request 보내기 (6/10)

- 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



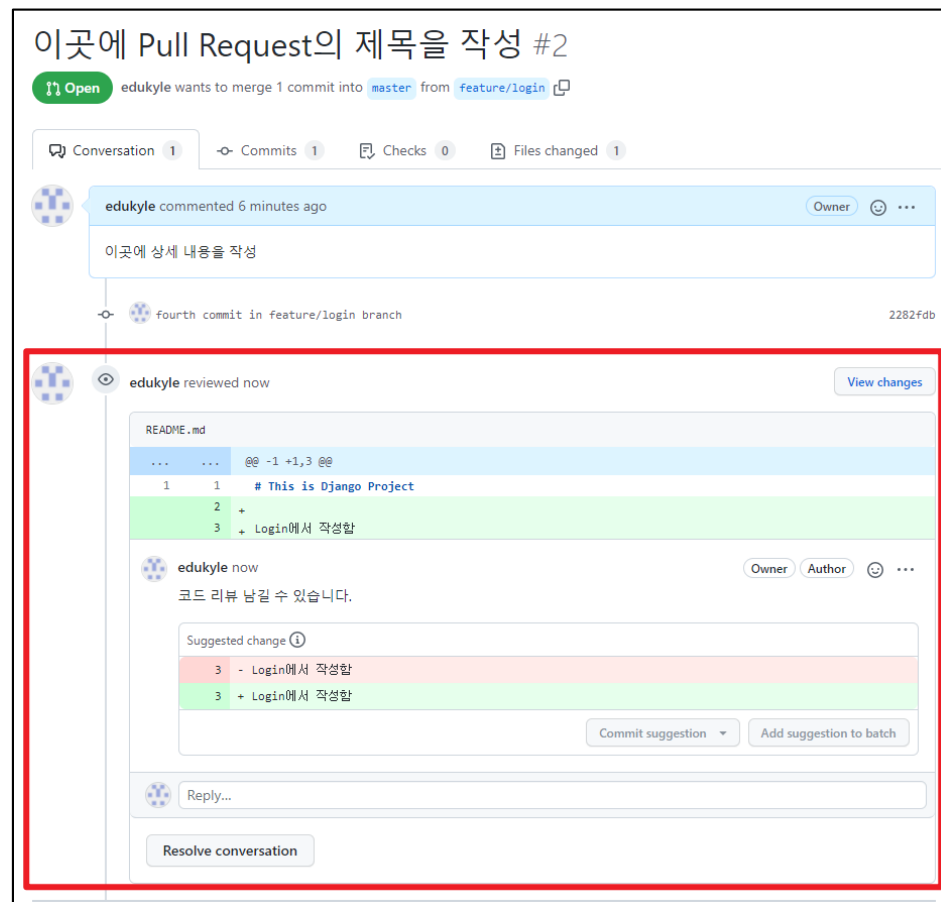
Github에서 Pull Request 보내기 (7/10)

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



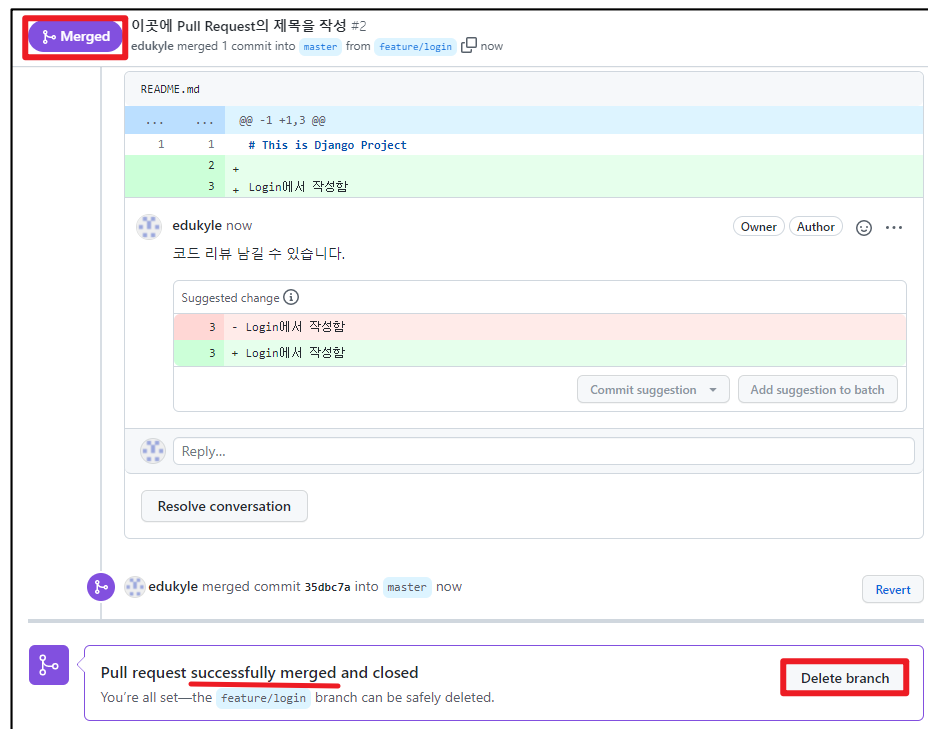
Github에서 Pull Request 보내기 (8/10)

- 다시 conversation으로 가보면 진행했던 리뷰가 나타난 것을 확인 가능



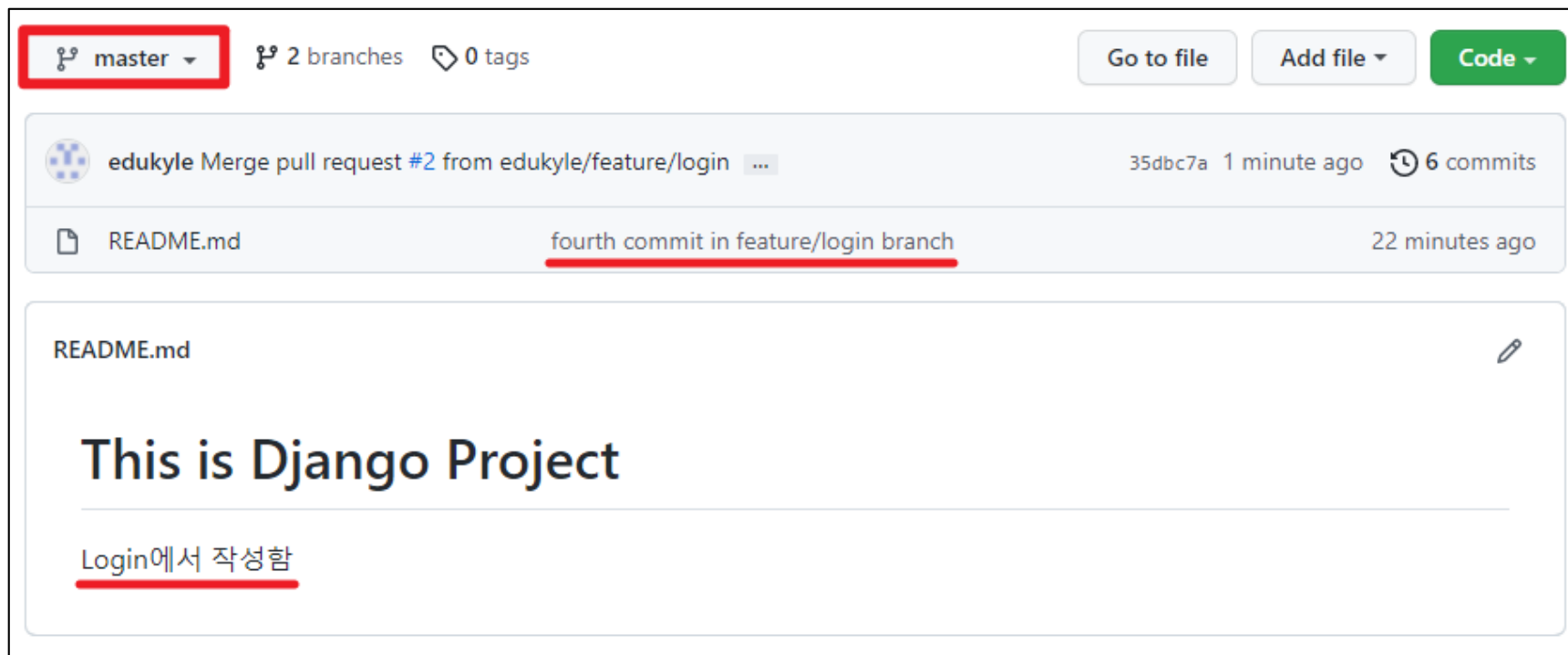
Github에서 Pull Request 보내기 (9/10)

- 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



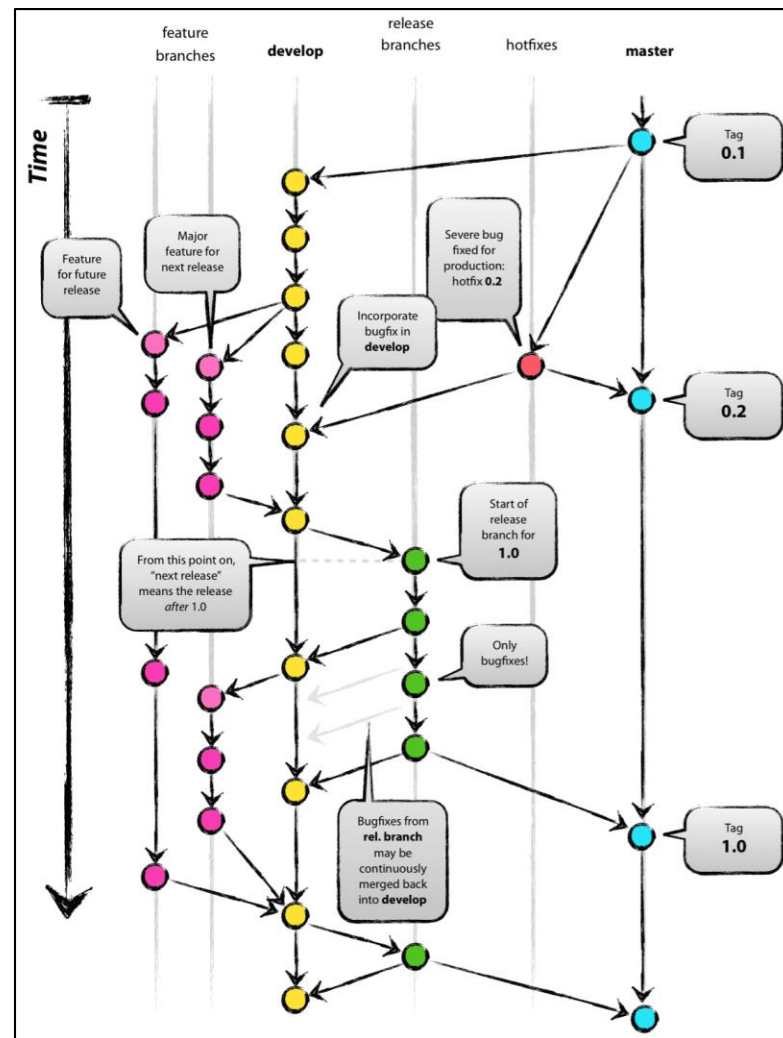
[참고] Git 브랜치 전략

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

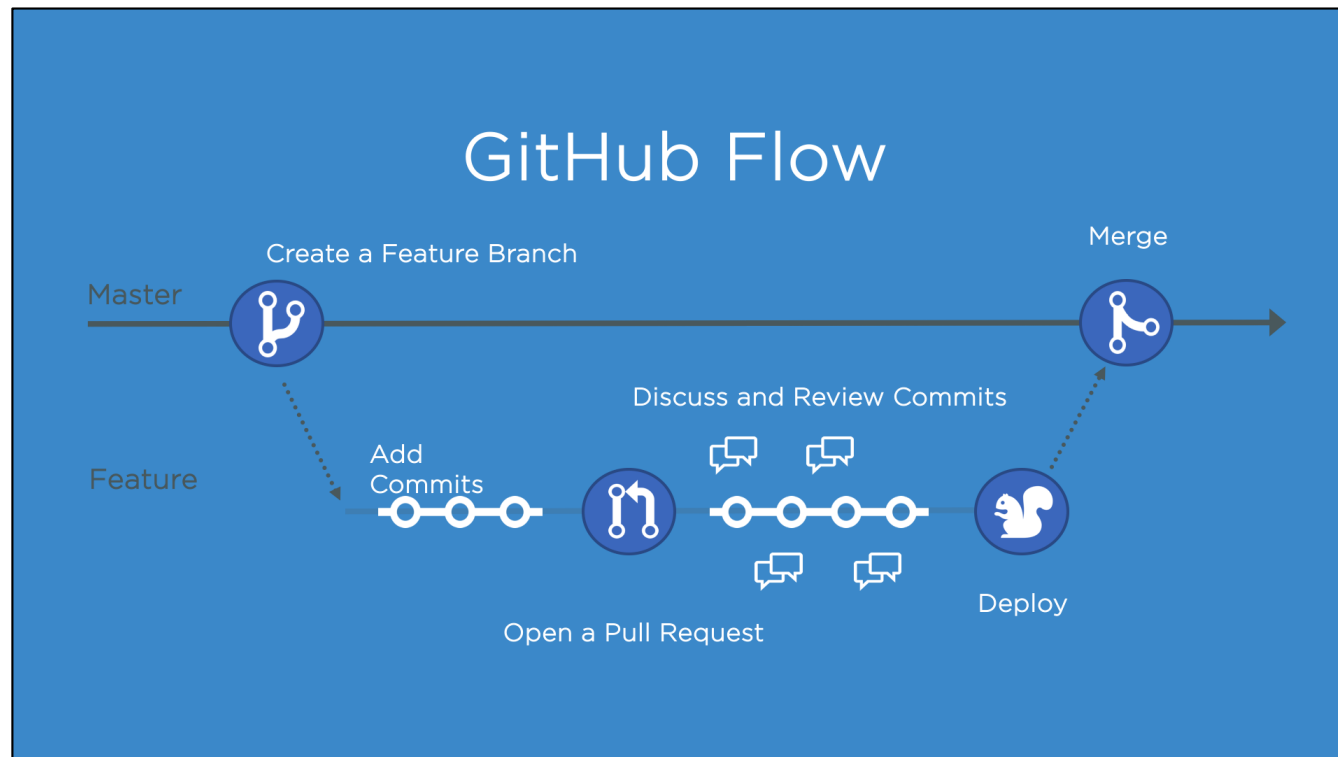
git-flow

- 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 수정하는 브랜치
- 대규모 프로젝트에 적합한 브랜치 전략



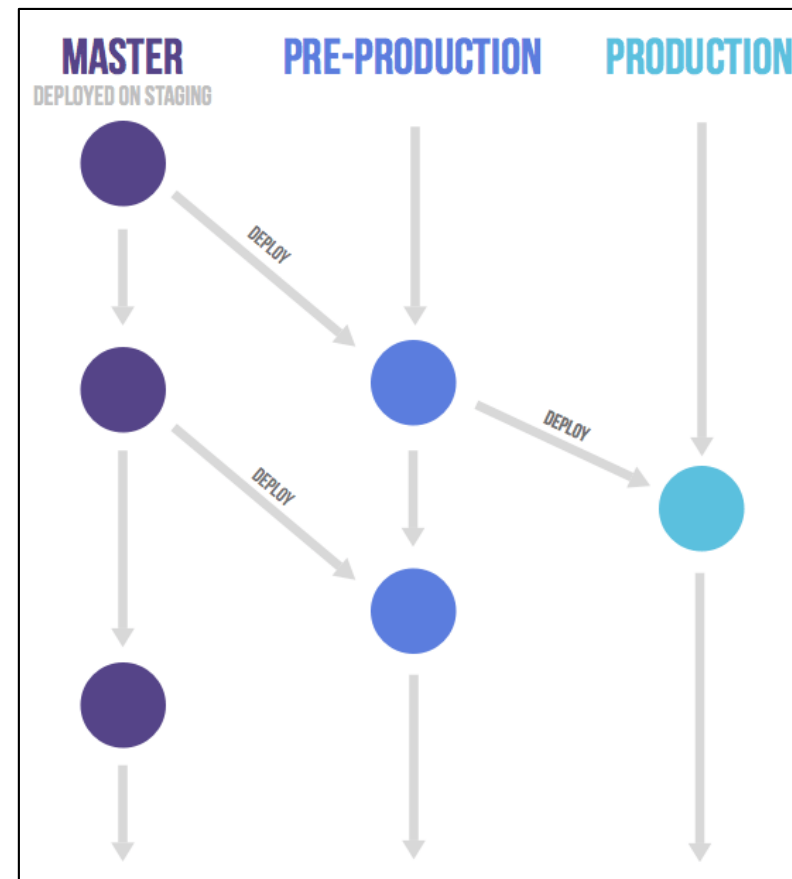
github-flow

- 복잡한 git-flow를 개선하여 github에서 사용하는 방식
- Pull Request 기능을 사용하도록 권장하며, 병합 후 배포가 자동화로 이루어짐



gitlab-flow

- github-flow의 배포 이슈를 보완하기 위해 gitlab에서 사용하는 방식
- master 브랜치와 production 브랜치 사이에 pre-production 브랜치를 두어 개발 내용을 바로 반영하지 않고, 배포 시기를 조절함



정리

- 결국 어떤 브랜치 전략을 사용할 것 인지는 팀에서 정하는 문제
- 소개된 git, github, gitlab 브랜치 전략이 아닌 우리 팀 고유의 브랜치 전략도 가능
- 브랜치를 자주 생성하는 것을 강력히 권장하며,
main(master) 브랜치 하나로만 작업하는 형태는 지양해야 함



다음
방송에서
만나요!

삼성 청년 SW 아카데미