

과제1

Git & Github

계정1 - https://github.com/SeongHunTed/Git_Project

계정2 - https://github.com/JAY-Winter/Git_Project

충실판교 글로벌미디어학부

김성훈 20181065

1. 과제 개요

Git과 Github를 이용하여 팀 프로젝트의 협업, 형상 관리에 대해 학습한다. 로컬 파일을 원격 저장소에서 관리하고 공유함으로써 발생하는 상황들에 대해 알맞은 git 명령어를 학습한다. 팀프로젝트라는 가상 프로젝트를 진행함으로써 생기는 문제상황을 해결한다.

a. Environments

<Software>

- a. Mac iOS
- b. Git version 2.32.0 (Apple Git-132)
- c. Terminal : iTerm2

b. account

프로젝트를 2명(Ted, Jay)라는 사람이 진행한다는 가정하에 진행했기에 2개의 Github계정을 사용했다.

1. Ted : https://github.com/SeongHunTed/Git_Project
2. Jay : https://github.com/JAY-Winter/Git_Project

2. 과제 수행

a. Synario

Team Leader : Ted

Team Member : Jay

Jay는 Ted와 함께 공동 프로젝트를 진행하게 되었다. Ted가 우선 원격저장소를 만들고 Jay는 이를 fork해간다.

Jay는 fork해온 저장소를 이용하여 Ted에게 기능 단위로 수행하여 Pull & Request를 보내고 Ted는 이를 검토하여 본인이 작업하고 있는 main 브랜치에 merge할지말지 판단하며 본인의 작업을 수행한다.

Introduction

Git & Github

Git과 Github : 형상 관리와 협업을 돋는 툴

<Git>

하나의 긴 글을 작성한다고 하자. 글을 작성하다가 보면 수정과 입력이 계속 일어날 것이다. 어느 순간 글이 맘에 들지 않아 이전의 내용으로 돌아가고 싶다면, 기억해내어 비슷하게 다시 작성해야 할 것이다. 특정 시점의 글 상태로 돌아갈 수 있다면 굉장히 편할 것이다.

Git은 이를 가능케 한다. 프로그래밍을 할 때 이 전의 내용으로 돌아가 확인하거나 수정해야 할 일이 있다. 변경사항이 있을 때마다 버전별로 파일을 저장한다면, 확인할 수 있겠지만, 효율적이지 못하다.

Git은 하나의 맥락 속에서 프로젝트의 버전을 관리할 수 있게 한다. 이를 가능케 하는 단위가 **commit**이다. commit을 이용하여 원하는 시점으로 돌아갈 수 있다.

<Github>

Github는 Git으로 버전 관리 중인 내 컴퓨터의 Git파일을 서버에 저장하는 플랫폼이다.

이는 오픈된 공간이며, 다른 사람들의 코드를 볼 수도 있고, 내 코드들도 열람할 수 있다.

그래서 우리는 Git과 Github를 이용하여 협업을 진행한다.

Introduction

Git & Github



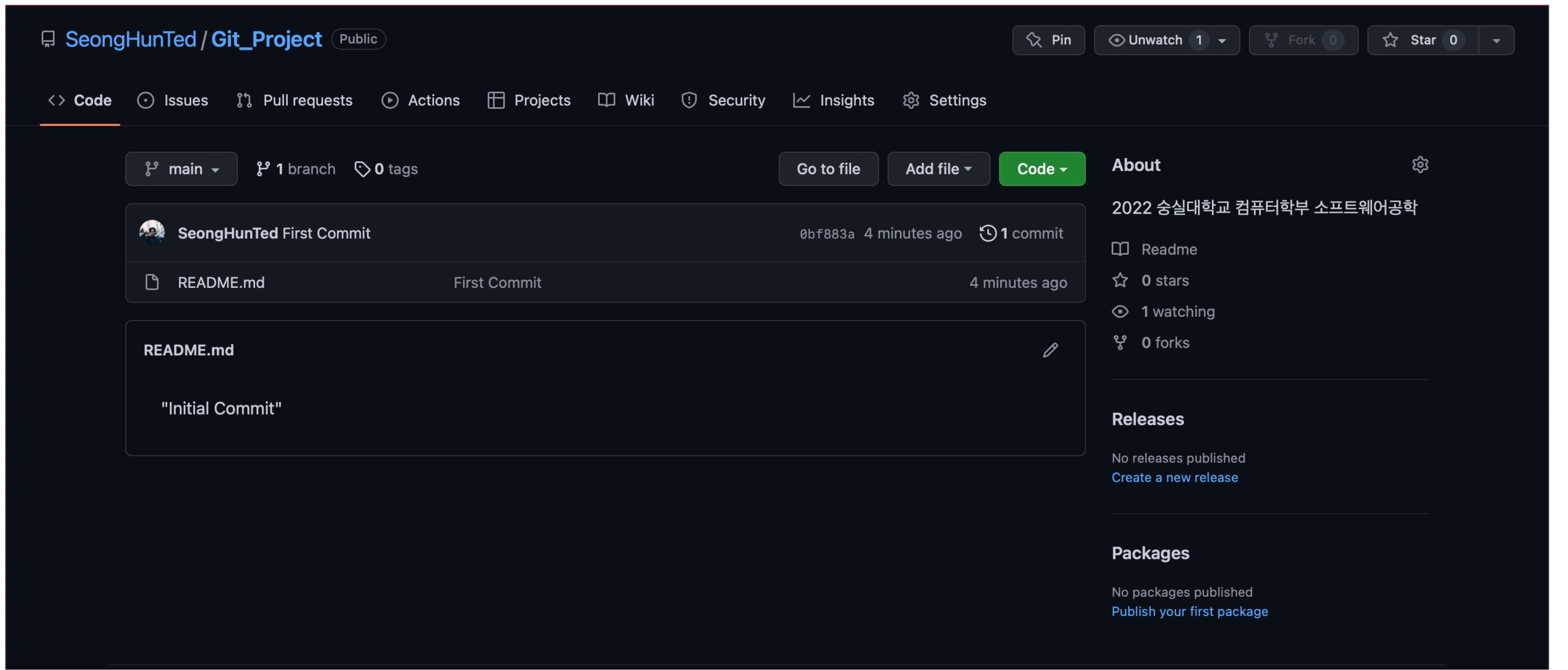
<들어가기에 앞서>

Git에서 버전의 단위는 commit이다. 현재 상태의 스냅샷이라고 이해하면 된다.

Working Directory에서 바로 Commit을 하지 않고 Staging Area를 거치는 이유는, 의도한대로 더 안전한 커밋을 위함이다.

이를 알고 시작하자.

시나리오1 - 원격저장소 만들기 : Ted



1. Ted는 프로젝트 협업과 형상 관리를 위해 Github에서 Git_Project라는 원격저장소 공간을 하나 만들었다.
- New Repository 클릭 후 이름 설정 후 나머지 옵션은 default로 한다.
2. 앞으로 이 공간에 Ted는 본인의 commit을 올릴 것이다.
3. Jay는 이 원격저장소를 fork한 후 프로젝트를 시작한다.

시나리오1 - 로컬에서 Git 시작하기 : Ted

```
> mkdir Ted  
> Ted  
> git init  
hint: Using 'master' as the name for the initial branch. This default br  
anch name  
hint: is subject to change. To configure the initial branch name to use  
in all  
hint: of your new repositories, which will suppress this warning, call:  
hint:  
hint:   git config --global init.defaultBranch <name>  
hint:  
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and  
hint: 'development'. The just-created branch can be renamed via this com  
mand:  
hint:  
hint:   git branch -m <name>  
Initialized empty Git repository in /Users/Hoon/Ted/.git/  
> vim README.md  
> git add README.md  
> git commit -m "First Commit"  
[master (root-commit) 0bf883a] First Commit  
 1 file changed, 1 insertion(+)  
  create mode 100644 README.md  
> git branch -M main  
> git remote add origin https://github.com/SeongHunTed/Git_Project.git
```

1. Ted는 Ted라는 폴더를 하나 만들었다. (터미널 명령어 : mkdir <디렉토리이름>)
2. README.md 라는 파일 하나를 만들고 이에 내용을 추가했다.
3. 로컬 git 저장소와 github 원격 저장소를 연결한다.

- **git init** : 현재 위치한 디렉토리에 .git이라는 폴더를 생성한다. 이 폴더를 통해 현 로컬 디렉토리를 관리하게 된다.
- **git add** : 작업한 내용, 변경 내용을 Staging Area에 올리는 과정이다. 이는 변경사항이 있을 때만 동작하며 파일은 Tracked 상태로 변경된다.
`git add .` : 변경된 모든 사항을 Staging Area에 올림
`git add <파일이름>` : 특정 파일의 수정된 사항 만을 Staging Area에 올림
- **git commit** : Staging Area에 올라온 파일에 대한 스냅샷을 기록하는 것이다. 파일의 모든 내용을 저장하는 것은 아니고, 변경된 수정 사항만을 저장한다. 이러한 방식 때문에 최초의 commit이 아니면 각 commit은 그 위의 부모 commit을 가르킨다.
`git commit -m` : 메세지를 인라인으로 첨부 할 때 쓰는 명령어. 보통 기능 단위로 커밋을 하며 ""을 이용하여 메세지를 첨부한다.
- **git branch** : 현재의 branch를 확인하는 명령어.
git에는 branch가 있다. 번역하면 가지인데, 이 branch에는 commit과 그 부모 commit들을 포함하는 작업내역들이 있다. branch라는 단어에서 알 수 있듯이 가지처럼 다양한 가지를 가질 수 있다.
쉽게 이해하자면 branch에는 사용자의 커밋들이 저장되었는 곳이라고 보면 된다.
`git branch <name>` : name이라는 브랜치를 현재 commit 상태에서 만듦.
`git branch -m` : branch의 이름을 변경할 때 쓴다. Default branch이름은 master인데, 인종차별 문제로 master대신 main을 쓰는 추세이다.
- **git remote** : 현재 git에 연결된 원격 저장소를 확인하는 명령어.
`git remote add <단축이름> <원격저장소 url>` : 해당 GitHub 원격저장소 공간과 local git 연결 추가
`git remote rename <대상> <수정할 이름>` : 단축이름을 변경
`git remote remove <대상>` : 대상 리모트를 연결 해제

위 상황 처럼 git init으로 시작하여 로컬 git과 github를 연결하는 경우 순서대로 따라야 정상적으로 로컬과 원격저장소가 연결된다.

시나리오1 - 로컬에서 Git 시작하기 : Ted

```
> git config --list
> git config --global user.email 4047ksh@naver.com
> git config --global user.name SeongHunTed
    git config —list 내역

credential.helper=osxkeychain
user.name=SeongHunTed
user.email=4047ksh@naver.com
push.default=current
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
core.ignorecase=true
core.precomposeunicode=true
remote.origin.url=https://github.com/SeongHunTed/Git_Project.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.main.remote=origin
branch.main.merge=refs/heads/main
(END)
```

- **git config** : git 기본 이메일, 이름 설정 명령어

협업을 하다보면 누가 코드를 쓴건지 확인해야할 때가 있다. git config로 user.mail & name을 등록 해놓으면 누가 코드를 쓴 것인지 확인이 가능하다.

git config —global user.name <name> : git 이름 설정
git config —global user.mail <mail> : git 이메일 설정
git config —list : git에 설정된 모든 내역을 출력해준다.

1. Ted는 git config 명령어를 통해서 mail 과 name을 등록했다.

- **git push** : 변경된 commit 내역을 원격저장소에 올리는 명령어

```
> git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 234 bytes | 234.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SeongHunTed/Git_Project.git
 * [new branch]      main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

변경내역을 commit하고 나서 반드시 원격저장소에 push 해주어야한다.
이렇게 해야 로컬 저장소와 원격 저장소의 상태가 동일해지며 다른 사람도 참조할 수 있기 때문이다.

git push -u <원격저장소이름> <branch이름> : 최초 한 번만 이 명령어를 쓰면 이 다음부터는 생략 가능하다.

git push -f <원격저장소이름> <branch이름> : 원격 저장소와 로컬 저장소의 상태가 달라 정상적 push가 되지 않는 경우 오류를 감안하고 push 하는 경우.
(본인은 초보자 일때 자주 이 명령어를 썼으나 손실이 일어날 수 있으므로 협업시에는 사용을 지양)

2. Ted는 commit을 원격 저장소로 push 했다.

시나리오1 - git tag : Ted

```
› git tag v1.0
```

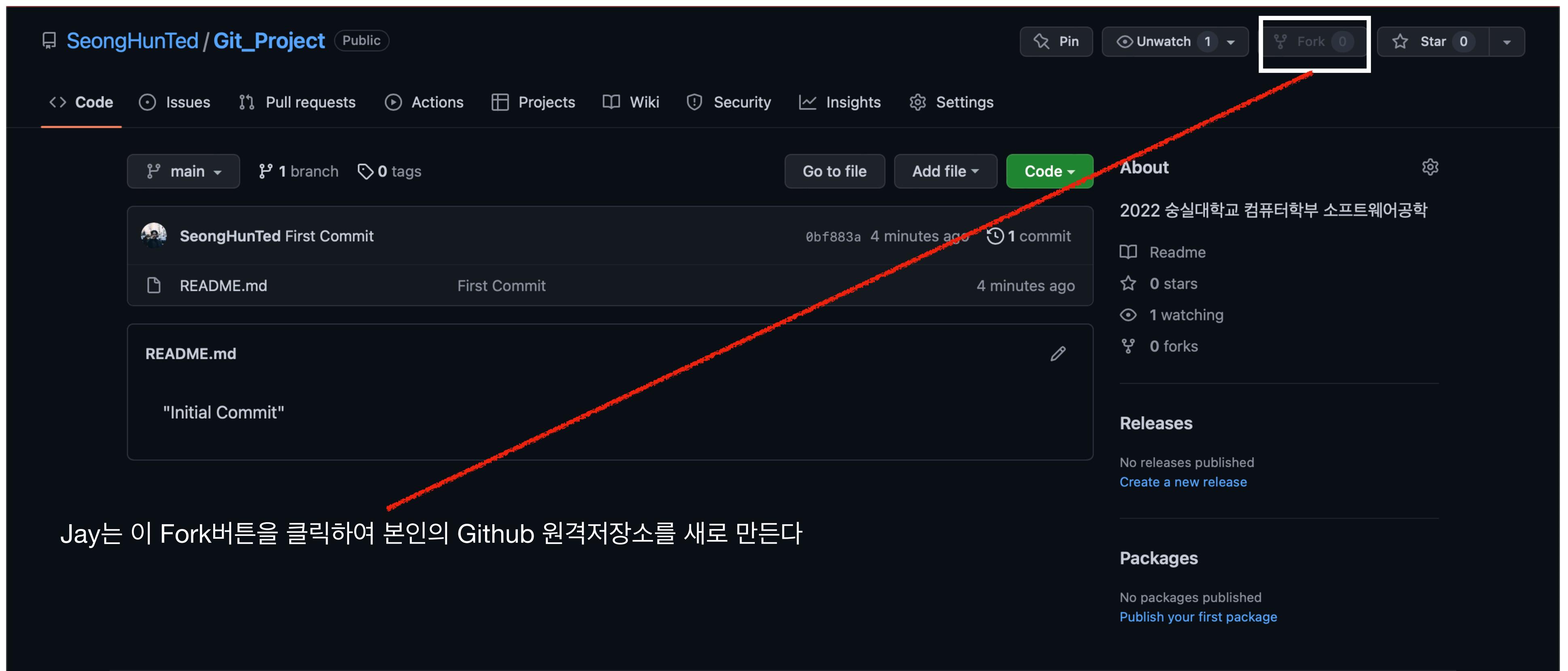
<Ted : git tag>

1. **git tag <버전 명>** : 프로젝트 릴리즈 할 때 사용하는 명령어.
버전 정보를 저장한다.

git tag : 이미 만들어진 태그가 있는지 확인

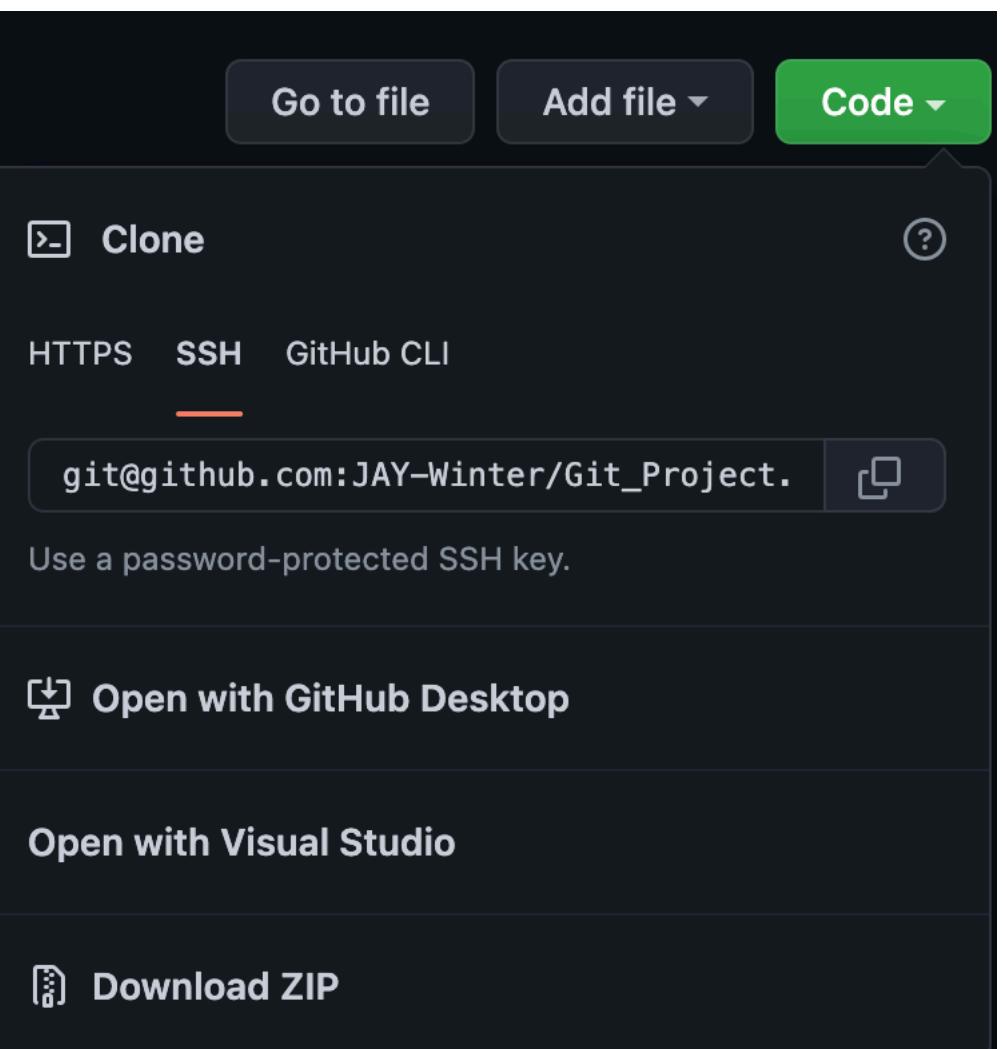
```
v1.0  
(END)
```

시나리오2 - Github에서 Git 시작하기 : Jay



1. Jay는 Ted의 원격저장소에서 Fork를 통해 본인의 Github에 새로운 원격 저장소를 만든다

시나리오2 - Github에서 Git 시작하기 : Jay



```
> git clone git@github.com:JAY-Winter/Git_Project.git
Cloning into 'Git_Project'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.

> ls
Git_Project
> cd Git_Project
> ls
README.md
> vi README.md
> mkdir src
> src
> vim src.md
> cd ..
> git add src
> git commit -m "Jay : 기능1 구현 완료"
[main 10b1540] Jay : 기능1 구현 완료
1 file changed, 4 insertions(+)
create mode 100644 src/src.md
> git add README.md
> git commit --amend
[main 18bfaac] Jay : 기능1 구현 완료
Date: Fri May 13 17:08:58 2022 +0900
2 files changed, 5 insertions(+), 1 deletion(-)
create mode 100644 src/src.md
> git push -u origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 475 bytes | 475.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:JAY-Winter/Git_Project.git
  0bf883a..18bfaac main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

1. Jay는 Github에서 Code버튼을 눌러 SSH의 clone할 수 있는 값을 가져온다.
2. git clone <가져온 값 혹은 url>
3. Ted가 서면으로 README.md 수정과 src.md를 만들고 기능1 구현을 요청했다.

시나리오2 - Github에서 Git 시작하기 : Jay

```
> git clone git@github.com:JAY-Winter/Git_Project.git
Cloning into 'Git_Project'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
> ls
Git_Project
> Git_Project
> ls
README.md
> vi README.md
> mkdir src
> src
> vim src.md
> cd ..
> git add src
> git commit -m "Jay : 기능 1 구현 완료"
[main 10b1540] Jay : 기능 1 구현 완료
 1 file changed, 4 insertions(+)
 create mode 100644 src/src.md
> git add README.md
> git commit --amend
[main 18bfaac] Jay : 기능 1 구현 완료
  Date: Fri May 13 17:08:58 2022 +0900
  2 files changed, 5 insertions(+), 1 deletion(-)
  create mode 100644 src/src.md
> git push -u origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (5/5), 475 bytes | 475.00 KiB/s, done.
Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:JAY-Winter/Git_Project.git
  0bf883a..18bfaac  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

- **git clone <url || ssh value>** : 원격 저장소에 존재하는 커밋 이력을 로컬 디렉토리로 복사해오는 명령어.

Ted가 git을 시작한 방법과 달리 Jay는 이미 원격저장소에 Ted의 저장소를 fork해왔기 때문에 로컬에 원격저장소를 clone으로 연결할 수 있다.

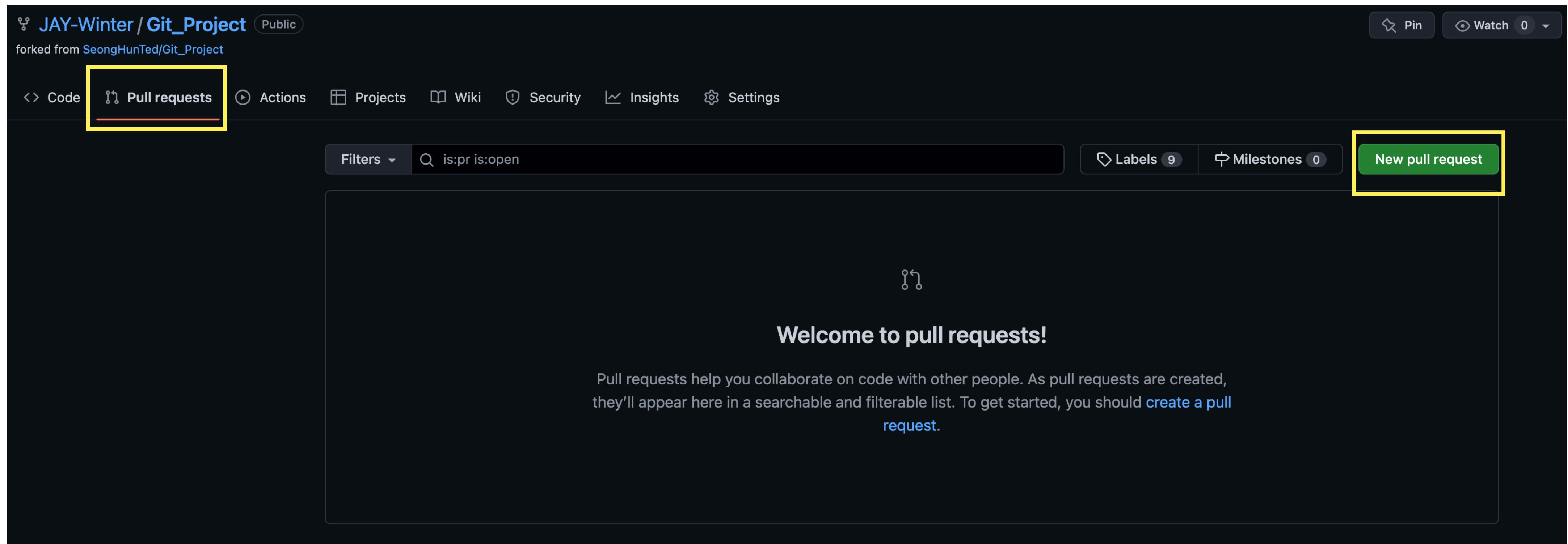
(보통은 git clone <url>을 사용하나, 본인은 한개의 컴퓨터에서 두개의 계정을 이용하기 위해 Jay의 계정은 ssh 키 값을 이용하여 프로젝트 관리를 하였다.)

- **git commit —amend** : 이미 commit을 완료했으나 빠트린 파일이 있을 때 commit 내역 정정

Jay는 실수로 README.md 파일 커밋을 안했다. 이럴 때 git add <file> 후 해당 명령어를 쓰면 커밋이력이 바뀜을 알 수 있다.

1. Jay는 요청사항대로 README.md 수정과 src.md 생성 후 기능 추가를 마쳤다.
2. commit을 끝내고 생각해보니 해당 커밋에 README.md 수정본을 추가하는 것을 잊었다.
3. commit —amend 명령어로 커밋을 수정하여 push를 완료했다.

시나리오2 - Pull & Request : Jay

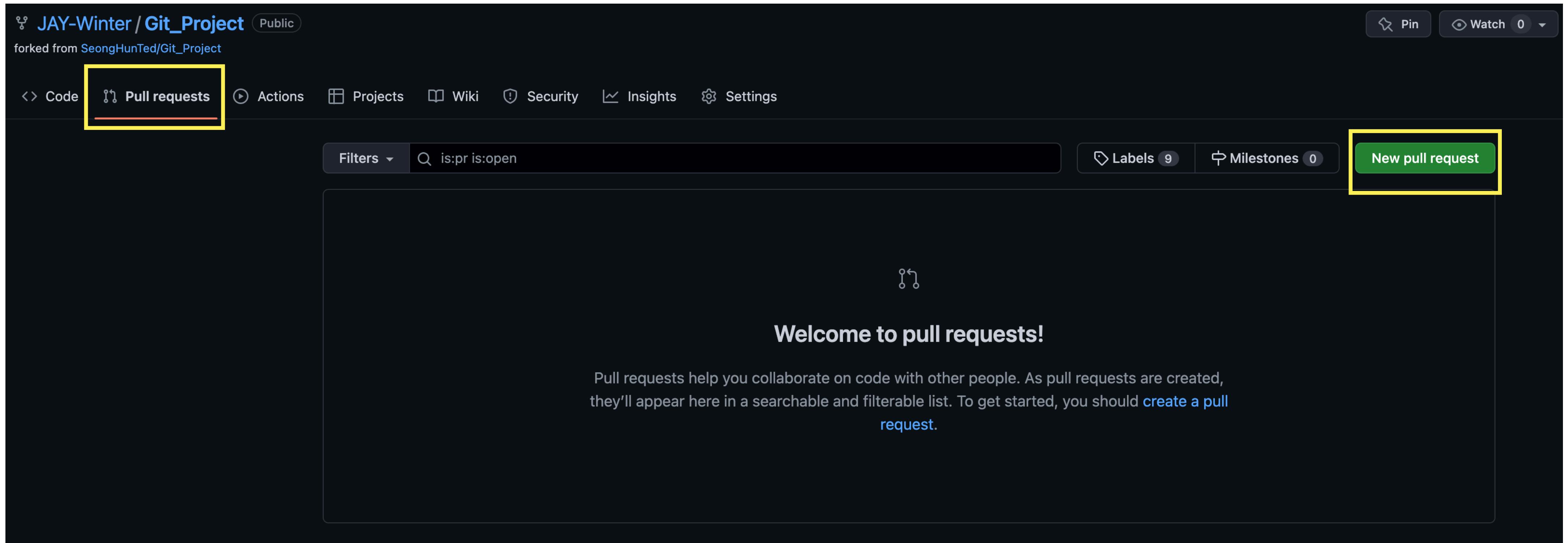


Pull & Request란?

전체 프로젝트 자체에 push 권한이 없는 개발자가 수정한 코드를 올렸으니 검토후 합쳐달라 요청하는 것.

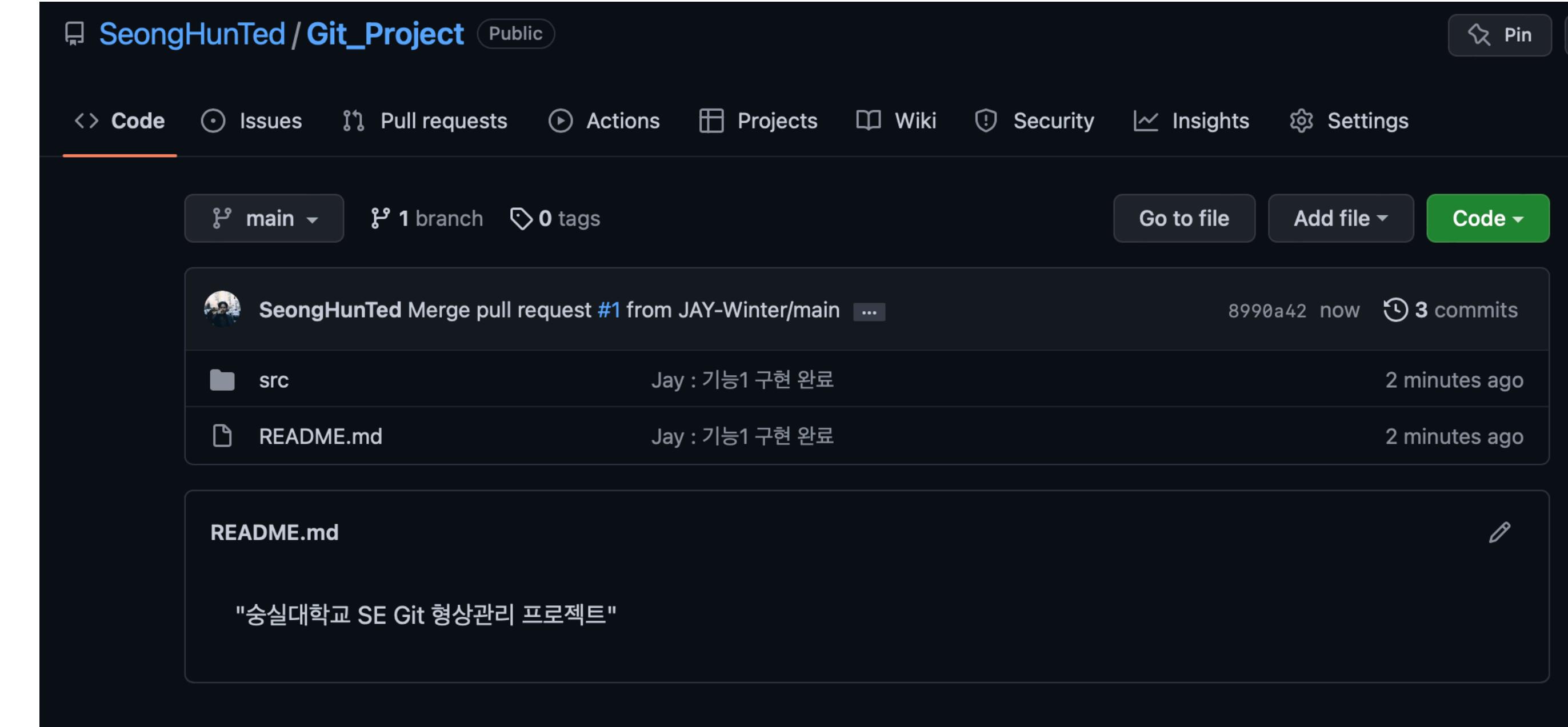
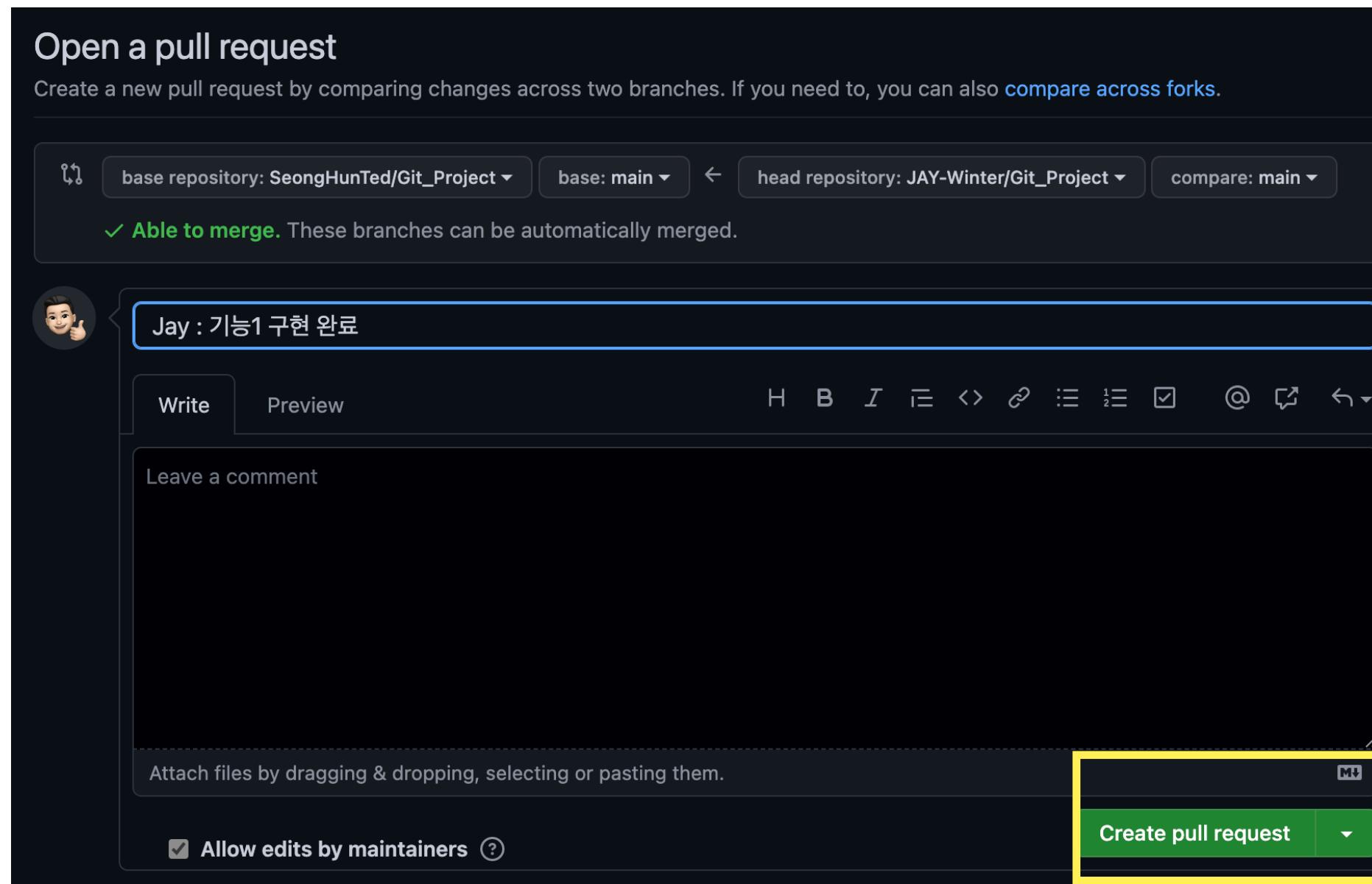
프로젝트 전체 관리자는 pull request를 확인 후 이상이 없으면 전체 코드에 merge하여 최신 상태를 유지한다.

시나리오2 - Pull & Request : Jay



1. Ted와 Jay는 협업 중이고 Ted의 main 브랜치를 중심으로 프로젝트의 기능을 추가한다.
2. Jay는 Jay의 원격저장소의 main 브랜치에서 작업을 관리한다.
3. Jay가 만든 기능을 Ted의 main 브랜치에 직접적으로 push 할 수 없다.
4. 따라서 Jay는 Ted의 브랜치에 Pull & Request를 통해 프로젝트에 기능을 추가한다.

시나리오2 - Pull & Request & Merge: Jay, Ted



<Jay : 기능 1 구현 완료 commit을 pull request>

Jay의 main branch를 Ted의 main branch로 Pull Request(이하 PR) 요청
(본래 프로젝트의 main 브랜치는 비워두고 위 상황에서는 Ted의 원격 저장소에
main, Ted, Jay branch를 각각 만들고 완성 단계에서 Ted와 Jay 브랜치를 main
브랜치에 합치는 것이 옳은 방법이다.)

<Ted : Jay의 PR을 accept한 Ted의 원격 저장소>

문제가 없다고 생각한 Ted는 기능 1이 추가된 커밋이 있는 Jay의 PR을 수락하였고
그 이후의 Ted의 원격 저장소

시나리오3 - pull : Ted

The screenshot shows a GitHub repository named 'SeongHunTed/Git_Project'. The 'Code' tab is selected. A merge pull request from 'JAY-Winter/main' to 'main' is visible, with 3 commits. The commit history includes changes to 'src' and 'README.md'. The 'README.md' file contains the text: "승실대학교 SE Git 형상관리 프로젝트". A large black arrow points from the GitHub interface to the terminal command on the right.

<Ted : Jay의 PR을 accept한 Ted의 원격 저장소>

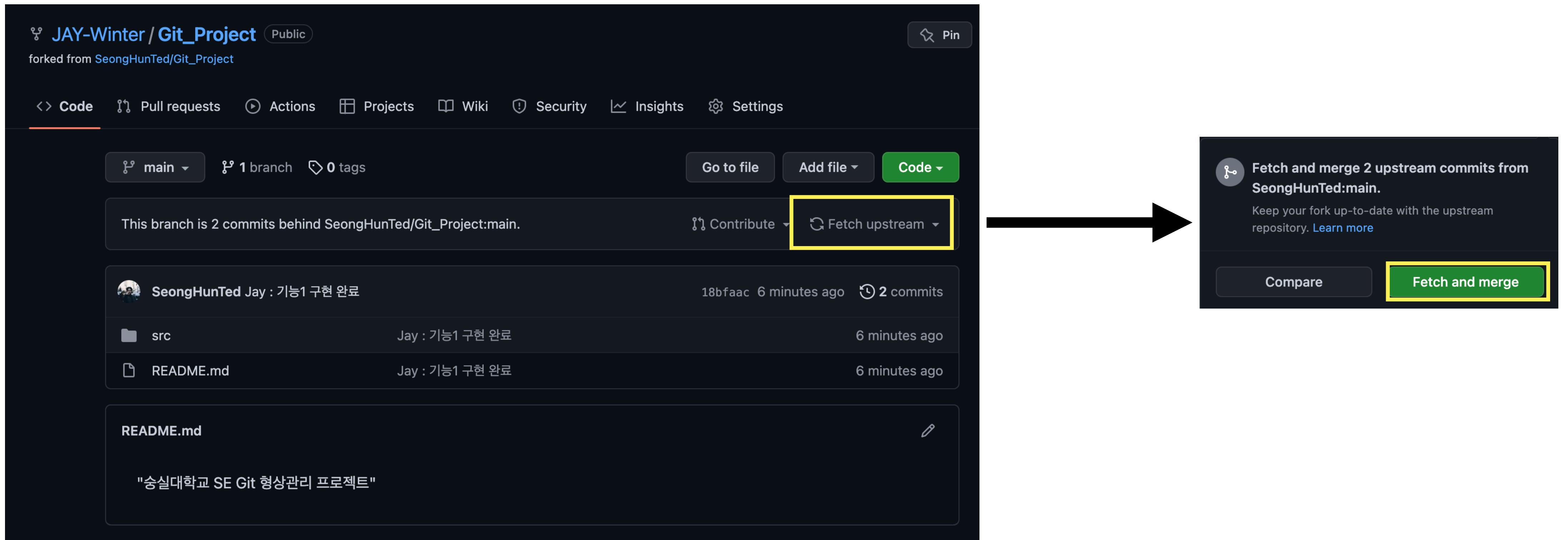
1. Ted의 원격 저장소는 최신화가 되었지만 Ted의 로컬 저장소는 아직 최신상태가 아니다.
2. 이때 Ted는 git pull 명령어를 이용하여 원격저장소와 로컬저장소를 동기화할 수 있다.
3. 동기화 이후 Ted는 기능 2를 추가하고 src.md 파일에 요청사항을 적어 push한다.

```
git pull origin main
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 6 (delta 0), reused 5 (delta 0), pack-reused 0
Unpacking objects: 100% (6/6), 1.08 KiB | 277.00 KiB/s, done.
From https://github.com/SeongHunTed/Git_Project
 * branch            main      -> FETCH_HEAD
   0bf883a..8990a42  main      -> origin/main
Updating 0bf883a..8990a42
Fast-forward
 README.md | 2 +-+
 src/src.md | 4 +++
 2 files changed, 5 insertions(+), 1 deletion(-)
 create mode 100644 src/src.md
> src
> vi src.md
> git add .
> git commit -m " Ted : 기능 2 구현 , 요청사항 "
[main c52c57f] Ted : 기능 2 구현 , 요청사항
 1 file changed, 5 insertions(+), 1 deletion(-)
> git push -u origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 439 bytes | 439.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SeongHunTed/Git_Project.git
 8990a42..c52c57f  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```

• git pull <원격저장소> <로컬브랜치>

: 원격저장소의 변경된 데이터를 로컬 저장소로 가져옴

시나리오3 - Fetch upstream & merge : Jay



<Ted : Ted의 원격 저장소와 Jay의 원격 저장소의 차이>

1. Ted가 요청 사항과 기능2를 구현한 커밋이력은 Ted의 원격저장소에만 존재한다.
2. Ted와 Jay의 원격저장소는 Fork로 인해 연결되어있다.
3. Jay는 Fetch upstream and merge를 통해 원격저장소의 변경사항을 동기화 한다.

시나리오3 - Jay

```
> git pull origin main
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), 1.05 KiB | 179.00 KiB/s, done.
From github.com:JAY-Winter/Git_Project
 * branch            main      -> FETCH_HEAD
   18bfaac..c52c57f  main      -> origin/main
Updating 18bfaac..c52c57f
Fast-forward
 src/src.md | 6 +++++-
 1 file changed, 5 insertions(+), 1 deletion(-)

> src
> vi src.md
> git add .
> cd ..
> git commit -m "Jay : 기능 3 구현"
[main a6fbf10] Jay : 기능 3 구현
 1 file changed, 4 insertions(+), 2 deletions(-)
> git push -u origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 406 bytes | 406.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:JAY-Winter/Git_Project.git
  c52c57f..a6fbf10  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```



<Jay : 원격 저장소를 fetch 후 git pull>

1. Jay의 원격 저장소만 최신화가 되었으므로 동일하게 git pull을 이용하여 로컬 저장소도 최신화한다.
2. 이후 요청사항에 있던 기능 3을 구현한다.
3. commit, push, PR, accept를 이전과 같이 동일하게 진행한다

시나리오4 - Rebase : Jay

```
> git pull origin main
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 5 (delta 0), reused 4 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), 1.01 KiB | 258.00 KiB/s, done.
From https://github.com/SeongHunTed/Git_Project
 * branch            main      -> FETCH_HEAD
   c52c57f..ad0c648  main      -> origin/main
Updating c52c57f..ad0c648
Fast-forward
 src/src.md | 6 +---+-
 1 file changed, 4 insertions(+), 2 deletions(-)
> vi src.md
> git add .
> git commit -m " Ted : 기능 4 구현 중, 요청 사항 "
[main 6162352] Ted : 기능 4 구현 중, 요청 사항
 1 file changed, 4 insertions(+), 1 deletion(-)
> git push -u origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 431 bytes | 431.00 KiB/s, done.
Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/SeongHunTed/Git_Project.git
  ad0c648..6162352  main -> main
Branch 'main' set up to track remote branch 'main' from 'origin'.
```



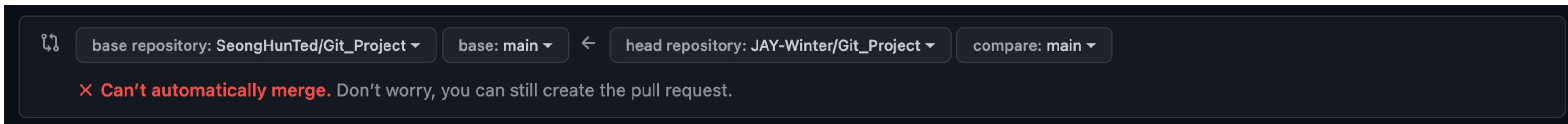
<Jay : 원격 저장소를 fetch 후 git pull>

1. Ted가 기능4를 구현 중이며 Jay에게 기능 5 구현을 요청했다.
2. Jay는 이전과 동일하게 fetch -> git pull로 최신화 및 동기화를 진행했다.
3. Jay가 기능5를 구현하여 push 후 PR을 보내려는데, 그 전에 Ted가 기능 4를 구현해서 push했다.



어떤 일이 발생할까???

시나리오4 - Rebase : Jay



<Jay : PR 진행중 Ted의 main 브랜치가 업데이트 되어버린 상황>

1. 자동 merge가 되지 않는다.
2. 수동으로 코드를 고쳐서 conflict 상황을 해결 가능하다.
혹은 Jay의 원격저장소를 수정하는 방법도 있다.

```
> git remote -v
origin  git@github.com:JAY-Winter/Git_Project.git (fetch)
origin  git@github.com:JAY-Winter/Git_Project.git (push)
> git remote add upstream https://github.com/SeongHunTed/Git_Project
> git remote -v
origin git@github.com:JAY-Winter/Git_Project.git (fetch)
origin git@github.com:JAY-Winter/Git_Project.git (push)
upstream      https://github.com/SeongHunTed/Git_Project (fetch)
upstream      https://github.com/SeongHunTed/Git Project (push)
> git fetch upstream main
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 4 (delta 1), reused 4 (delta 1), pack-reused 0
Unpacking objects: 100% (4/4), 364 bytes | 121.00 KiB/s, done.
From https://github.com/SeongHunTed/Git_Project
 * branch            main      -> FETCH_HEAD
 * [new branch]      main      -> upstream/main
> git rebase upstream/main
Auto-merging src/src.md
CONFLICT (content): Merge conflict in src/src.md
error: could not apply dbf0e85... Jay : 기능5 구현 완료
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase --abort".
Could not apply dbf0e85... Jay : 기능5 구현 완료
```



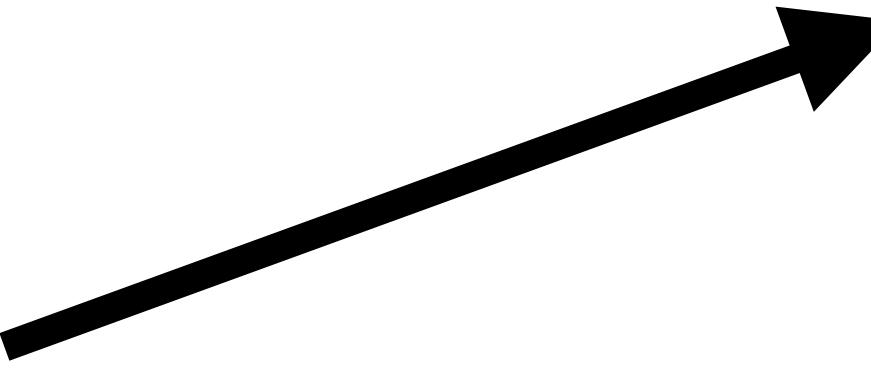
<Jay : Rebase로 Conflict 해결하기>

1. **remote -v** 로 현재 연결된 원격 저장소 확인 후 Ted의 원격저장소를 upstream이란 이름으로 추가한다.
2. **git fetch <원격저장소> <브랜치>** : 원격 저장소의 데이터를 해당 브랜치도 가져온다.
여기서 브랜치는 새로운 브랜치를 생성하여 그곳에 데이터를 가져오는데 Jay는 이제 origin/main, upstream/main 브랜치가 있다. (보통은 브랜치 이름을 같게 하지 않는다)
3. **git rebase <합쳐질 branch>** : 브랜치끼리의 작업을 접목하는 명령어.
커밋들을 모두 모아서 복사한뒤 다른 브랜치에 놓는다. 즉 Jay는 Ted가 만든 기능 4를 포함하는 커밋이 있는 브랜치에 본인이 구현한 것을 merge 한 것이다.

시나리오4 - Rebase : Jay

```
pick 0bf883a First Commit
pick 18bfaac Jay : 기능 1 구현 완료
pick c52c57f Ted : 기능 2 구현, 요청사항
pick a6fbf10 Jay : 기능 3 구현
pick 8102352 Ted : 기능 4 구현 중, 요청사항
edit dbf0e85 Jay : 기능 5 구현 완료

# Rebase dbf0e85 onto 4cef4f2 (6 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 c
#                               ase
#                               keep only this commit's message; -c is same as -C b
# ut
#                               ut
#                               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 --continu
# e')
# d, drop <commit> = remove commit
# l, label <label> = label current HEAD with a name
# t, reset <label> = reset HEAD to a label
# m, merge [-C <commit> | -c <commit>] <label> [# <oneline>]
# .
#     create a merge commit using the original merge commit's
# .
#     message (or the oneline, if no original merge commit was
# .
#     specified); use -c <commit> to reword the commit message
#
# These lines can be re-ordered; they are executed from top to bottom.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will be aborted.
#
~ ~ ~ ~
-- INSERT --
```



```
> git rebase upstream/main
Auto-merging src/src.md
CONFLICT (content): Merge conflict in src/src.md
error: could not apply dbf0e85... Jay : 기능 5 구현 완료
Resolve all conflicts manually, mark them as resolved with
"git add/rm <conflicted_files>", then run "git rebase --continue".
You can instead skip this commit: run "git rebase --skip".
To abort and get back to the state before "git rebase", run "git rebase
--abort".
Could not apply dbf0e85... Jay : 기능 5 구현 완료
> git add .
> git rebase --continue
[detached HEAD 9e64ec9] Jay : 기능 5 구현 완료
  2 files changed, 7 insertions(+), 1 deletion(-)
    create mode 100644 .DS_Store
Successfully rebased and updated refs/heads/main.
> git commit -m " Jay : 기능 5 구현 완료 , rebase "
[main a58e28a] Jay : 기능 5 구현 완료 , rebase
  1 file changed, 1 insertion(+), 7 deletions(-)
> git push origin main -f
```

<Jay : Rebase Continue>

1. 커밋이 달라 발생한 conflict 해결후 **git rebase --continue** 명령어로 rebase를 재진행한다.
 2. commit을 진행하고 push -f 명령어를 사용하였다.
-> Ted가 새로 구현한 기능 4로 인해 커밋이 바뀌었다
-> 그 상태 자체를 Jay's base로 바꾸었으므로 Jay의 원격 저장소와 커밋이 다른 문제가 발생한다.
-> 따라서 강제로 push 하여 커밋을 업데이트 해준다.
 3. 이후에 Jay, Ted는 모두 원격저장소와 로컬저장소를 모두 최신화 했다.
- main, Ted, Jay라는 브랜치가 있다고 가정했을 때,
main은 비어둔 상태로 Ted 와 Jay가 각각 다른 기능을 구현해서 push한다고 하자.
Ted가 먼저 완성해서 main 브랜치와 Ted 브랜치를 합쳤다고 하면, Jay는 이 합쳐진 main 브랜치에
동기화가 안되어있어 합칠 때 문제가 생기므로, 미리 최신화된 main 브랜치로 rebase를 하여
본인 브랜치를 합치는것이 맞다.

<Jay : upstream/main과 origin/main의 충돌>

1. 이제 base는 Ted의 원격저장소에 가져온 최신 상태가 base가 된다.
2. 하지만 그 상태와 Jay가 기능 5를 구현한 상태가 다르다
-> git rebase를 하면 위와같은 화면이 나오는데 변경된 commit 앞에 pick을 edit으로 바꾼다.

시나리오5 - Cherry-pick : Ted

<Ted : 기능4와 연동될 class 1, 2, 3생성>

```
> git branch test
> git checkout test
Switched to branch 'test'
> ls
src.md
> vi src.md
> git add .
> git commit -m " Ted : test branch class 1 "
[test bd7275a] Ted : test branch class 1
1 file changed, 3 insertions(+), 1 deletion(-)
> vi src.md
> git add .
> git commit -m " Ted : test branch class 1,2 "
[test c816eb4] Ted : test branch class 1,2
1 file changed, 2 insertions(+)
-> git add .
> git commit -m " Ted : test branch class 1,2,3 "
[test 1faa9bf] Ted : test branch class 1,2,3
1 file changed, 2 insertions(+)
```

1. Ted가 기능 4와 연동될 class1, class2, class3을 하나씩 구현하기 위해 branch test를 만들었다.
2. **git checkout <브랜치명>** : 현재 브랜치에서 <브랜치>로 이동할때 사용한다.
3. 1번 커밋 : class 1만 존재
2번 커밋 : class 1,2 존재
3번 커밋 : class 1,2,3 존재
4. 하지만 후에 Ted가 class3은 필요 없음을 깨달았다.



#test branch 생성하여 기능 6에서 사용할 class 만들어보는 중 .y

```
[기능 1]
구현완료

[기능 2]
구현완료

[기능 3]
구현완료

[기능 4]
구현완료

[기능 5]
구현완료

[class 1]
```

#test branch 생성하여 기능 6에서 사용할 class 만들어보는 중 .y

```
[기능 1]
구현완료

[기능 2]
구현완료

[기능 3]
구현완료

[기능 4]
구현완료

[기능 5]
구현완료

[class 1]
[class 2]
```

#test branch 생성하여 기능 6에서 사용할 class 만들어보는 중 .y

```
[기능 1]
구현완료

[기능 2]
구현완료

[기능 3]
구현완료

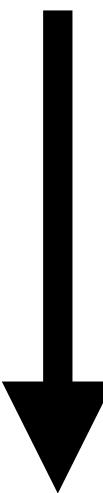
[기능 4]
구현완료

[기능 5]
구현완료

[class 1]
[class 2]
[class 3]
```

시나리오5 - Cherry-pick : Ted

```
> git log  
> git checkout main  
Switched to branch 'main'  
Your branch is up to date with 'origin/main'.  
> git log  
> git cherry-pick c816eb  
Auto-merging src/src.md  
CONFLICT (content): Merge conflict in src/src.md  
error: could not apply c816eb... Ted : test branch class 1,2  
hint: after resolving the conflicts, mark the corrected paths  
hint: with 'git add <paths>' or 'git rm <paths>'  
hint: and commit the result with 'git commit'
```



```
commit 1faa9bf717e3b0fe58406350bc8e46d407105ee5 (HEAD -> test)  
Author: SeongHunTed <4047ksh@naver.com>  
Date: Fri May 13 18:23:41 2022 +0900  
  
Ted : test branch class 1,2,3  
  
commit c816eb|c292555ffaba17442e89a3448c18627d7  
Author: SeongHunTed <4047ksh@naver.com>  
Date: Fri May 13 18:22:38 2022 +0900  
  
Ted : test branch class 1,2  
  
commit bd7275a1bda3134ab755a19a3ec4df4d9862b2b0  
Author: SeongHunTed <4047ksh@naver.com>  
Date: Fri May 13 18:21:35 2022 +0900  
  
Ted : test branch class 1
```

```
> vi src.md  
> git add .  
> git cherry-pick --continue  
[main 7917c1c] Ted : test branch class 1,2  
Date: Fri May 13 18:22:38 2022 +0900  
1 file changed, 4 insertions(+)  
> git push origin main  
> git branch -D test  
Deleted branch test (was 1faa9bf).  
> git branch
```

<Ted : 기능4와 연동될 class 1, 2, 3생성>

1. **git log** : 저장소의 커밋 히스토리를 시간순으로 보여준다. 즉 가장 커밋부터 위에서 아래로 나온다. 각 커밋은 SHA-1체크섬, 저자이름 저자 이메일, 날짜, 커밋 메세지를 보여준다.

git log -p : 각 커밋의 diff(다른점)을 보여준다.

2. 현재 test branch에서 체크섬 값(해쉬값) 앞 6자리를 가져온다.
-> 커밋은 SHA-1체크섬 앞 6자리로 구분할 수 있다.
즉 2번째 커밋은 c816eb로 구분된다.

3. **git cherry-pick <해당커밋>** : 현재 커밋 위치 아래에 커밋의 복사본은 만드는 명령어.

class1,2만 있는 커밋을 가져오면 되므로 해당 커밋 해쉬값을 가져와 main 브랜치 최신 커밋에 복사한다.

4. conflict 상황 또한 소스코드 수정을 통해 완료한다.
- conflict 상황은 이후 merge 설명에서 더 자세히 다루겠다.

5. **git cherry-pick --continue** : conflict 수정후 cherry-pick을 완료한다.

6. 이제 더 이상 test branch가 필요하지 않음으로 **git branch -D <삭제할 브랜치>** 를 이용해 삭제한다.

```
* main  
(END)
```

7. 확인해보면 main 브랜치만 존재한다.

시나리오5 - Cherry-pick : Ted

The screenshot shows a GitHub repository page for 'SeongHunTed/Git_Project'. The 'Code' tab is selected. At the top, it shows 'main' branch, 1 branch, 0 tags, and a green 'Code' button. Below this, a commit list is displayed:

Author	Commit Message	Time
SeongHunTed	Ted : test branch class 1,2	7917c1c 2 minutes ago
	Ted : test branch class 1,2	2 minutes ago
	Jay : 기능5 구현완료	40 minutes ago
	Jay : 기능1 구현 완료	1 hour ago

Below the commit list, there is a file viewer for 'README.md' containing the text: "승실대학교 SE Git 형상관리 프로젝트".

<Ted : class 1,2만 성공적으로 push 된 Ted의 원격저장소>

시나리오6 - stash: Jay

```
> vi src.md  
> cat src.md  
#기능 5-2 구현 중입니다. Jay  
  
[기능 1]  
구현 완료  
  
[기능 2]  
구현 완료  
  
[기능 3]  
구현 완료  
  
[기능 4]  
구현 완료  
  
[기능 5]  
구현 완료  
  
[class 1]  
  
[class 2]  
  
[기능 5-2]  
구현 중  
> git status  
On branch main  
Your branch is up to date with 'origin/main'.  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
modified: ./DS_Store  
modified: src.md  
  
no changes added to commit (use "git add" and/or "git commit -a")
```



```
> git stash  
Saved working directory and index state WIP on main: 7917c1c Ted : test  
branch class 1,2  
> git status  
On branch main  
Your branch is up to date with 'origin/main'.  
  
nothing to commit, working tree clean
```

git stash : 디렉토리에서 수정한 파일들만을 임시로 저장한다.
modified이면서, Tracked상태인 파일과, staging area에 있는 파일을 다른곳에 임시 저장한다.

이후 status를 보면 깨끗한 상태임을 알 수 있다.

```
> vi README.md  
> cd ..  
> vi README.md  
> git add .  
> git commit -m "Jay : Add Info on README"  
[main d2f08fd] Jay : Add Info on README  
1 file changed, 1 insertion(+)  
> git push origin main  
Enumerating objects: 5, done.  
Counting objects: 100% (5/5), done.  
Delta compression using up to 4 threads  
Compressing objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 441 bytes | 441.00 KiB/s, done.  
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
To github.com:JAY-Winter/Git_Project.git  
    7917c1c..d2f08fd main -> main
```

<Jay : 작업중 Ted의 갑작스러운 요청>

- 모든 동기화 이후 Jay가 5-2 기능을 구현중이다. Ted의 갑작스러운 요청으로 README.md 파일에 개발자 정보를 추가 해서 바로 push해달라는 요청이 왔다.
- 아직 기능을 구현 중이라 commit 할 수 없는 Jay는 stash를 사용한다.
- git status** : 현재 branch를 알려주며 파일의 상태를 알려준다. Tracked/ Untracked 여부 확인이 가능하다. Tracked -> 커밋에 넣어진 파일

📁 src	Ted : test branch class 1,2	37 minutes ago
📄 .DS_Store	Jay : 기능5 구현완료	1 hour ago
📄 README.md	Jay : Add Info on README	23 seconds ago

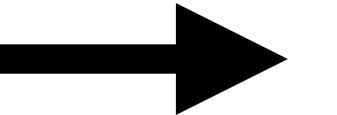
<Ted 요청사항 완료>

시나리오6 - stash: Jay

```
> git stash pop
On branch main
Your branch is up to date with 'origin/main'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   .DS_Store
    modified:   src/src.md

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (3c7587606318fd73d4f5dbe5233eddefc91d8d4a)
> git add .
> git commit -m "Jay : 기능 5-2 구현"
[main f97ba52] Jay : 기능 5-2 구현
 1 file changed, 4 insertions(+), 1 deletion(-)
> git push origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 398 bytes | 398.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:JAY-Winter/Git_Project.git
  d2f08fd..f97ba52  main -> main
```



	SeongHunTed Jay : 기능 5-2 구현	f97ba52 19 seconds ago	15 commits
	src	Jay : 기능 5-2 구현	19 seconds ago
	.DS_Store	Jay : 기능5 구현완료	1 hour ago
	README.md	Jay : Add Info on README	3 minutes ago

<Jay : 요청사항 완료 후 5-2 구현>

1. **git stash pop** : 임시저장한 stash를 꺼내온다
Jay는 임시 저장된 5-2 를 다시 꺼내 와서 작업을 완료 했다

git stash list : 저장된 stash 확인기능

시나리오7 - git reset –hard: Jay

<Jay : 기능 8 구현 실수>

```
> git add .
> git commit -m " Jay : 기능 8 구현 완료 "
[main 4bfkad9] Jay : 기능 8 구현 완료
 1 file changed, 1 insertion(+)
> cat src.md
#기능 5-2 구현 중입니다. Jay

[기능 1]
구현 완료

[기능 2]
구현 완료

[기능 3]
구현 완료

[기능 4]
구현 완료

[기능 5]
구현 완료

[class 1]

[class 2]

[기능 5-2]
구현 완료

# 기능 6은 스택 오버플로우에서 가져왔습니다.
# 같이 수정해야 합니다.
[기능 6]
a = a + 1

# Ted 구현부
[기능 7]

# Jay 구현부
[기능 8]
구현 완료 (실수 있음)
```

1. **git reset –hard <해당커밋>** : 브랜치의 예전의 커밋으로 이동하는 방식이다.
애초에 커밋하지 않은 것 처럼 예전 커밋으로 돌릴 수 있다.
2. Ted가 프로토타입을 만들어 원격 저장소에 올리고 Jay는 이를 동기화 했다.
3. Jay가 커밋, 푸쉬 이후 본인의 함수에 문제가 있음을 깨달았다.
-> 수정 후 push -> conflict 해결로 해도 되지만 git reset –hard를 사용하기로 했다.

```
commit 4bfkad9f2682c7b44831daa797b515749106100b
Author: SeongHunTed <4047ksh@naver.com>
Date:   Fri May 13 19:19:25 2022 +0900

  Jay : 기능 8 구현 완료

commit 1844f90e25c1cdc6ab687d946868236e1285b6d0
  D)
Author: SeongHunTed <4047ksh@naver.com>
Date:   Fri May 13 19:15:05 2022 +0900

  Ted : 마지막 요청사항

commit bdb7a3549dc61949915a555519ceecb1f64fedbb
Merge: 7917c1c f97ba52
Author: Ted <42074365+SeongHunTed@users.noreply
Date:   Fri May 13 19:12:16 2022 +0900

  Merge pull request #4 from JAY-Winter/main

  Add Info Readme, 기능 5-2 구현

commit f97ba520f061b64b4e22ebc9c42047d772bbff75
Author: SeongHunTed <4047ksh@naver.com>
Date:   Fri May 13 19:10:51 2022 +0900

  Jay : 기능 5-2 구현

commit d2f08fd5150d914dbd5f611ea76a7057ab9d7f61
Author: SeongHunTed <4047ksh@naver.com>
Date:   Fri May 13 19:08:23 2022 +0900

  Jay : Add Info on README

commit 7917c1cda116635b9f584ef906c5b398f7903354
Author: SeongHunTed <4047ksh@naver.com>
Date:   Fri May 13 18:22:38 2022 +0900

  Ted : test branch class 1,2

commit 2b84599cc81d6905cdhbcd55f58d2b20bba65581c
```

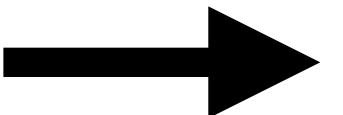


이곳으로 돌아와야한다.

시나리오7 - git reset --hard: Jay

```
> git log  
> git reset --hard 1844f9  
HEAD is now at 1844f90 Ted : 마지막 요청사항  
> vi src.md  
> git add .  
> git commit -m " Jay : 기능 8 구현 완료 _Perfect "  
[main 54d629b] Jay : 기능 8 구현 완료 _Perfect  
1 file changed, 1 insertion(+)
```

```
commit 54d629bdb3c457caafe77699add99e7d0ef5583b (HEAD -> main)  
Author: SeongHunTed <4047ksh@naver.com>  
Date:   Fri May 13 19:23:06 2022 +0900  
  
Jay : 기능 8 구현 완료 _Perfect  
  
commit 1844f90e25c1cdc6ab687d946868236e1285b6dd (origin/main, origin/HEAD)  
Author: SeongHunTed <4047ksh@naver.com>  
Date:   Fri May 13 19:15:05 2022 +0900  
  
Ted : 마지막 요청사항
```

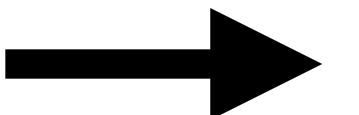


<Jay : 기능 8 구현 실수>

1. Jay는 git reset --hard로 수정된 파일의 내용과 커밋 내역을 모두 지웠다.
2. Jay는 구현한다가 기능 6을 위해선 기능 9가 필요함을 알고 추가도 해놓았다.
3. 기능 8를 다시 구현 한 후 push -f 진행했다. (원격 저장소 커밋과 충돌되기 때문에)
4. git log를 보면 이전에 실수한 내역은 사라지고 새로 구현한 함수의 커밋만을 확인 가능

시나리오7 - git revert: Ted

```
> git add .
> git commit -m " Ted : 기능 7 구현 완료_mistake "
[main 819754c] Ted : 기능 7 구현 완료_mistake
 1 file changed, 1 insertion(+), 1 deletion(-)
> git push origin main
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 4 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 399 bytes | 399.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/SeongHunTed/Git_Project.git
  af8d83a..819754c main -> main
> git revert HEAD
[main 200c3eb] Revert " Ted : 기능 7 구현 완료_mistake"
 1 file changed, 1 insertion(+), 1 deletion(-)
```



<Ted : 기능 7 구현 실수>

1. Ted는 Jay와 비슷하게 실수를 했다.
2. 하지만 Jay와 협업하는 브랜치를 사용하고 있으므로 커밋내역까지 삭제해버리는 git reset —hard를 쓰긴 위험하다.
3. **git revert HEAD** : 특정 커밋으로 돌아가 내용을 수정한다. 강제로 커밋을 삭제하는 git reset —hard와 달리 수정했음을 알리는 커밋이 하나 추가된다.

```
> git add .
> git commit -m " Ted : 기능 7 구현 완료_Perfect "
[main f2a02d0] Ted : 기능 7 구현 완료_Perfect
 1 file changed, 1 insertion(+), 1 deletion(-)
> git push origin main
```

시나리오8 - git merge : Ted

<직전 시나리오 정리>

1. Ted는 Jay에게 기능 8과 6을 구현할 것을 요청
2. 기능 6은 StackOverFlow에서 가져온 예시코드이므로 둘이 각자 구현 해보기로 결정
-> Jay는 구현하여 PR을 보내고 Ted는 이를 수락.
3. Ted는 기능 7과 6을 구현 완료
4. 프로젝트의 막바지라 Ted의 로컬 저장소에 Jay의 코드와 Ted의 코드를 merge한 후 push 할 예정

```
> git branch Jay
> git checkout Jay
Switched to branch 'Jay'
> git pull origin main
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 2), reused 4 (delta 2), pack-reused 0
Unpacking objects: 100% (5/5), 1.09 KiB | 160.00 KiB/s, done.
From https://github.com/SeongHunTed/Git_Project
 * branch      main      -> FETCH_HEAD
   fb1c5c1..c802102  main      -> origin/main
Updating fb1c5c1..c802102
Fast-forward
 src/src.md | 8 +++++---
 1 file changed, 5 insertions(+), 3 deletions(-)
> git checkout main
Switched to branch 'main'
Your branch is behind 'origin/main' by 2 commits, and can be fast-forwarded.
(use "git pull" to update your local branch)
> vi src.md
> git add .
> git commit -m " Ted : 기능 7 구현 및 6 수정 "
[main 12f24eb] Ted : 기능 7 구현 및 6 수정
 1 file changed, 1 insertion(+), 1 deletion(-)
```



<Jay의 코드 새로운 브랜치에 pull>

1. Ted는 Jay의 코드를 새로운 Jay 브랜치에 pull해왔다.
2. 본인의 main 브랜치와 Jay 브랜치를 merge할 것이다.
3. Ted도 기능 7과 6 수정한 것을 커밋 해놓은 상태

시나리오8 - git merge : Ted

```
> git merge Jay
Auto-merging src/src.md
CONFLICT (content): Merge conflict in src/src.md
Automatic merge failed; fix conflicts and then commit the result.

> git branch -D Jay
Deleted branch Jay (was c802102).

> git add .
> git commit -m "Ted : conflict 수정"
[main eb7e477] Ted : conflict 수정
> git push origin main
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 4 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (8/8), 767 bytes | 767.00 KiB/s, done.
Total 8 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/SeongHunTed/Git_Project.git
  c802102..eb7e477  main -> main
```

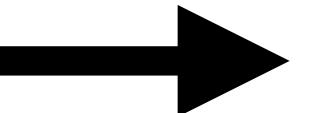
<Ted : 2명의 코드 merge>

1. **git merge <타겟 브랜치>** : 현재 브랜치에 다른 브랜치를 병합하는 명령어.
: 병합시 충돌에 주의해야함. 각자 다른 기능은 자동으로 merge가 되지만,
현재 기능 6은 같은 기능을 다른 두사람이 구현하였으므로 합치는 과정에서
어떤 변경사항을 합쳐야 하는지 모르기 때문에 충돌이 일어난다.

git merge —abort : 병합전 상태로 돌아가는 명령어.

충돌 해결하기

1. git merge —abort : 병합 전으로 돌아가 충돌을 피한다.
2. git reset —hard HEAD : 직전 커밋의 상태로 돌아간다.
3. 수동으로 merge 한다.



```
#기 능 6 구 현 중 입 니 다 . Jay
[기 능 1]
구 현 완 료

[기 능 2]
구 현 완 료

[기 능 3]
구 현 완 료

[기 능 4]
구 현 완 료

[기 능 5]
구 현 완 료

[class 1]

[class 2]

[기 능 5-2]
구 현 완 료

# 기 능 6은 스택 오 버플로 우 에서 가져 왔습니다 .
# 같 이 수 정 해 야 합 니 다 .
[기 능 6]
<<<<< HEAD
a = a + 3; b = b + 2; 기 능 5;
=====
a = a - 1; b = b + 2; 기 능 5;
>>>>> Jay

# Ted 구 현 부
[기 능 7]
구 현 완 료
# Jay 구 현 부
[기 능 8]

[기 능 9]
구 현 완 료
#기 능 6을 사용 하기 위 해 기 능 9를 추 가 했 습 니 다 .
"src.md" 42L, 556B
```

수동merge

자동merge

<Jay의 코드 새로운 브랜치에 pull>

1. Ted는 Jay의 코드를 새로운 Jay 브랜치에 pull해왔다.
2. 본인의 main 브랜치와 Jay 브랜치를 merge할 것이다.

시나리오8 - git merge : Ted

```
#기능 6 구현 중입니다. Jay
```

```
[기능 1]  
구현 완료
```

```
[기능 2]  
구현 완료
```

```
[기능 3]  
구현 완료
```

```
[기능 4]  
구현 완료
```

```
[기능 5]  
구현 완료
```

```
[class 1]
```

```
[class 2]
```

```
[기능 5-2]  
구현 완료
```

```
# 기능 6은 스택 오버플로우에서 가져왔습니다.  
# 같이 수정해야 합니다.
```

```
[기능 6]  
<<<<< HEAD  
a = a + 3; b = b + 2; 기능 5;  
=====  
a = a - 1; b = b + 2; 기능 5;  
>>>>> Jay
```

```
# Ted 구현부  
[기능 7]  
구현 완료  
# Jay 구현부  
[기능 8]
```

```
[기능 9]  
구현 완료  
#기능 6을 사용하기 위해 기능 9를 추가했습니다.  
"src.md" 42L, 556B
```

→ 수동merge

<수동으로 merge>

1. <<<<< HEAD >>>>> 까지가 충돌이 일어나는 부분이다.
2. <<<<< HEAD 부터 ===== 까지는 main 브랜치의 소스코드
===== 부터 >>>>> Jay까지는 Jay 브랜치의 소스코드이다.
3. git이 알려준 이 부분을 알맞은 코드로 수정해주면 된다.
4. 수정 후 commit하면 올바르게 merge가 완료된다.

→ 자동merge

Git 명령어 Index

Git 명령어	Index	사용여부
add	6	o
branch	6	o
checkout	21	o
cherry-pick	22	o
clone	11	o
commit —amend	11	o
commit -m	6	o
config	7	o
init	6	o
log	22	o
merge	30	o
pull	15	o
push	7	o
rebase	19	o
remote	6	o
reset —hard	26	o
revert	28	o
stash	24	o
status	24	o
tag	8	o