

Git 강의 #2

안드로이드 부트 캠프 4기

강사 : 모티즌 소프트 김갑석

Overview

1. 브랜치 (Branch)
2. 브랜치 전략 (Branch Strategy)
3. 원격 저장소 (Remote Repository)
3. GitHub Flow & Pull Request & 리뷰
5. 협업에 필요한 필수 Git 명령어
6. 프로그램 사용하기(sourceTree)
7. 팀별 실습

1. 브랜치(branch)

- 브랜치란?
- 브랜치 명령어
- 브랜치 병합(Merge)
- 충돌(Conflict) & 해결(Resolve)
- 개인 실습

1.1.브랜치란?

- 브랜치(branch) = ‘개발 작업 흐름의 복사본’
- 동시에 여러 개발을 독립적으로 수행 가능

[사용예]

- main: 운영 중인 안정적인 코드(배포용)
- feature/login: 로그인 기능 개발 브랜치
- feature/product : 상품 관련 목록, 상세 화면 개발 브랜치
- feature/order : 주문관련 기능 개발 브랜치
- hotfix/order-failure : 주문 실패 오류 대처 브랜치

#대규모 팀 프로젝트

- main ->feature/product/edit : feature/product의 하위 상세 브랜치

1.2.브랜치 명령어

1. 기본 명령어

- git branch : 현재 브랜치(*) 및 목록 확인
- git branch -a : 로컬 + 원격 브랜치 목록 확인
- git branch -r : 원격 브랜치 목록
- git branch <브랜치이름> : 브랜치 생성
- git branch -d 브랜치명 : 브랜치 삭제
- git branch -m 새이름 : 브랜치 이름 변경
- git branch -m 이전이름 새이름 : 특정 브랜치 이름 변경
- git branch -v : 각 브랜치의 마지막 커밋 메시지를 함께 출력
- git branch --no-merged : 현재 브랜치에 아직 병합되지 않은 브랜치 목록
- git branch --merged : 현재 브랜치에 병합된 브랜치 목록

2. 브랜치 전환 및 생성

- git switch 브랜치이름 : git checkout 보다 명확한 전환 명령어(git 2.23+ 버전부터 지원함.)
- git switch -c 새브랜치 : 브랜치 생성 + 전환(git branch -b 와 동일)

1.3.브랜치 병합(Merge)

1.병합(merge)의 개념

브랜치 작업이 끝났다면 → **main** 브랜치에 합치는 단계

- git merge 브랜치 : 현재 브랜치에 대상 브랜치의 변경 내용을 병합

2.병합 방식

- Fast-forward: 한쪽 브랜치만 변경 이력이 있는 경우, 병합 커밋 이력 X

```
Main :      A - - - B
                \
Feature :      C - - D
```

```
Main :      A - - - B - - - C - - - D
```

-> git merge --no-ff branchName -m comment : 강제로 병합 커밋 이력 남김

- 3-way merge: 양쪽 브랜치 모두 변경 이력이 있는 경우, 병합 커밋 이력 생성, 충돌 발생 가능

```
Main :      A - - - B - - - D
                \
Feature :      C - - E
```

```
Main :      A - - - B - - - D - - - E
                \  /
                  C
```

- git status : 현재 브랜치 상태 조회
- git log - - oneline - - graph : 커밋 히스토리를 시각적으로 표시해준다.

1.4.충돌(Conflict)

1.충돌(Conflict)의 개념

두 대상의 변경 내용을 git이 스스로 병합을 할 수 없을 때 발생하는 현상

- 같은 파일의 같은 줄을 서로 다르게 수정
- 한쪽은 수정, 다른쪽은 삭제
- 두 브랜치에서 동일한 파일을 생성

-> Git 이 자동으로 병합을 못하고 사용자에게 결정을 요구 하는 것입니다.

2.충돌 발생 시점

- Merge : 병합
- Rebase : 커밋 기반 적용
- Cherry-pick : 커밋 선택 적용
- Stash pop / apply: 임시 변경 복원
- Revert : 커밋 되돌리기
- Pull : 원격 저장소 변경사항 적용
- Switch : 브랜치 전환

1.5.충돌(Conflict) 유도

- main 브랜치에 파일 생성 - conflict.txt

```
git switch -c feature/conflict-branch  
git add . & git commit
```

브랜치 생성 및 전환
파일 수정 후 커밋

- 파일 수정 후 커밋(feature/conflict-branch 브랜치와 같은 줄을 편집)

```
git switch main(master)  
git add . & git commit  
git merge feature/conflict-branch
```

main 브랜치로 전환
파일 수정 후 커밋
main 브랜치로 병합

- 충돌 발생

```
Auto-merging conflict.txt  
CONFLICT (content) : Merge conflict conflict.txt  
Automatic merge failed; fix conflicts and then commit the result.
```

해당 파일에 충돌이 발생함

1.6.충돌(Conflict) 확인 및 해결(Resolve)

```
git status
```

충돌 확인

```
- - - result - - -  
On branch main  
Your branch is ahead of 'origin/main' by 1 commit.  
  (use "git push" to publish your local commits)  
You have unmerged paths.  
  (fix conflicts and run "git commit")  
  (use "git merge --abort" to abort the merge)  
Unmerged paths:  
  (use "git add <file>..." to mark resolution)  
      both modified:   conflict.txt  
no changes added to commit (use "git add" and/or "git commit -a")
```

conflict.txt 충돌난 파일을 열어서 충돌된 부분을 수정합니다.

```
<<<<<< HEAD  
내 브랜치 작성한 줄 master  
=====  
다른 브랜치 작성한 줄  
>>>>>> feature/conflict
```

충돌 해결

```
git add conflict.txt  
git commit -m 'resolve conflict in text'  
[main 8d5fad5] fix merge conflict
```

1.7.실습(브랜치 생성 & 충돌 & 해결)

1.브랜치 생성 :

- 브랜치 생성 및 전환 : `git switch -c 'branch name'`
- 브랜치 병합 : `git merge 'target branch name'`
- 스테이징 & 커밋 : `git add . && git commit -m 'commit comment'`
- 작업 공간 상태 : `git status`

2.충돌 상황 유도

- main branch 에 충돌 테스트 할 파일 (conflict.txt) 생성 간단한 내용작성 후 커밋
- feature/conflict 브랜치 생성 : `git switch -c feature/conflict`
- feature/conflict 브랜치의 conflict.txt 파일을 열어 해당문구 수정 혹은 새라인추가 후 커밋
- 다시 main branch로 돌아와서 conflict.txt 파일을 열어 해당문구 수정 혹은 새라인추가 후 커밋
- 머지 실행 : `git merge feature/conflict`

2.충돌 확인

- 충돌이 난 파일 확인 : `git status`

3.충돌 해결

- 충돌난 파일 열어서 내용 수정
- 커밋

2. 브랜치 전략(Branch Strategy)

- **GitHub Flow**
→ 단순(main 브랜치), PR 기반 협업, 빠른 배포, 소규모 팀, 스타트업
- **Git Flow**
→ 명확한 릴리즈 주기, 다양한 브랜치 분리, 대규모 프로젝트
- **Trunk-Based Development(TBD)**
→ 단일 브랜치 기반, 짧은 브랜치 주기, 실시간 배포 환경

2.1. GitHub Flow

Pull Request 기반으로 main 브랜치에 기능을 병합하며, 빠른 피드백과 배포를 지향하는 전략입니다.

- 브랜치 구조

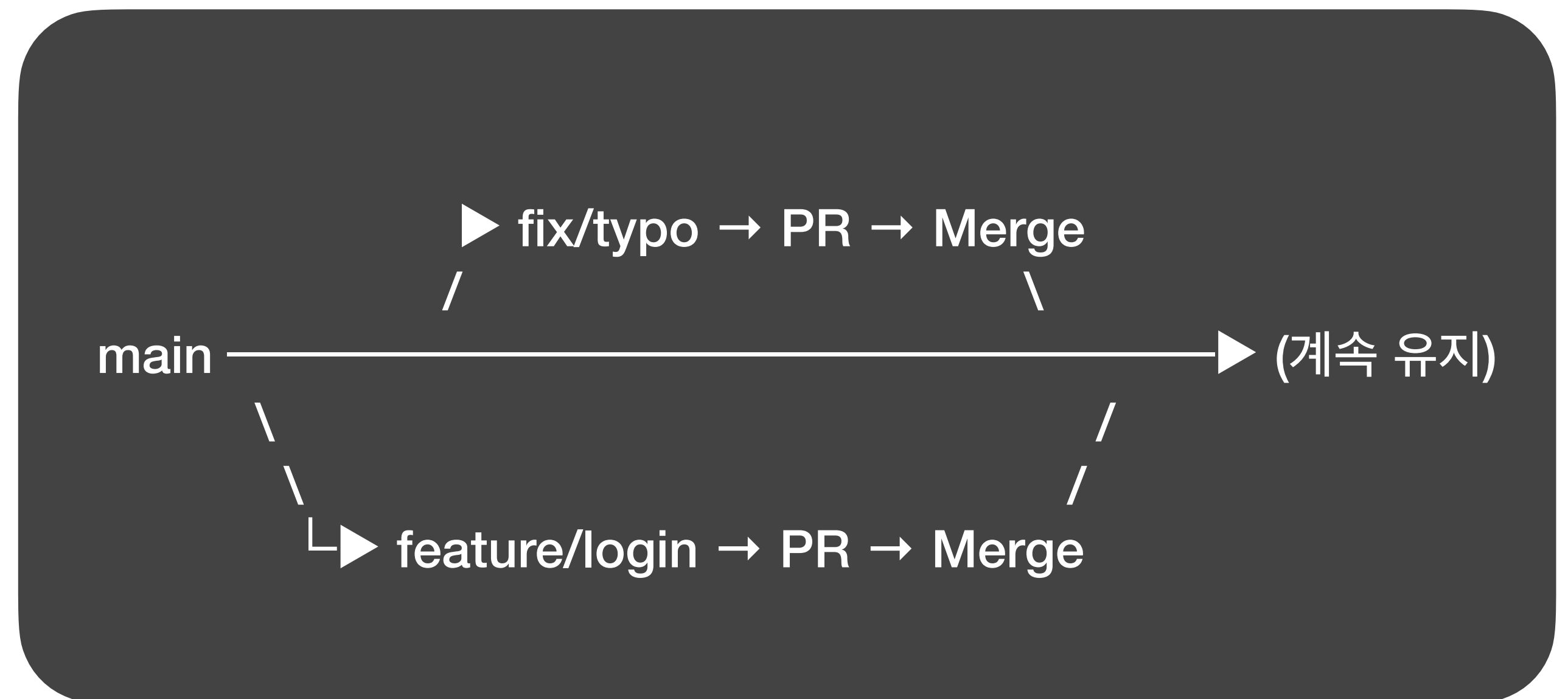
- main : 실제 운영 버전
- feature/* : 기능 개발 브랜치
- hotfix/* : 운영 중 긴급 수정

- 특징

- 단순하고 직관적인 전략
- main 브랜치 중심으로 개발
- 기능(이슈) 단위 브랜치 생성
- PR 강제로 소스 품질 관리
- CI/CD 자동화와 연동 적합
- 중규모 이하의 팀에 적합

- 주의

- PR이 필수 사항이나 리뷰 문화가 없으면 소스 품질 관리 어려움
- 머지시 전체에 영향 -> 팀내 관리



2.2. Git Flow

Git Flow는 릴리즈 주기가 긴 프로젝트나 엔터프라이즈 환경에서 자주 사용됩니다.

- 브랜치 구조

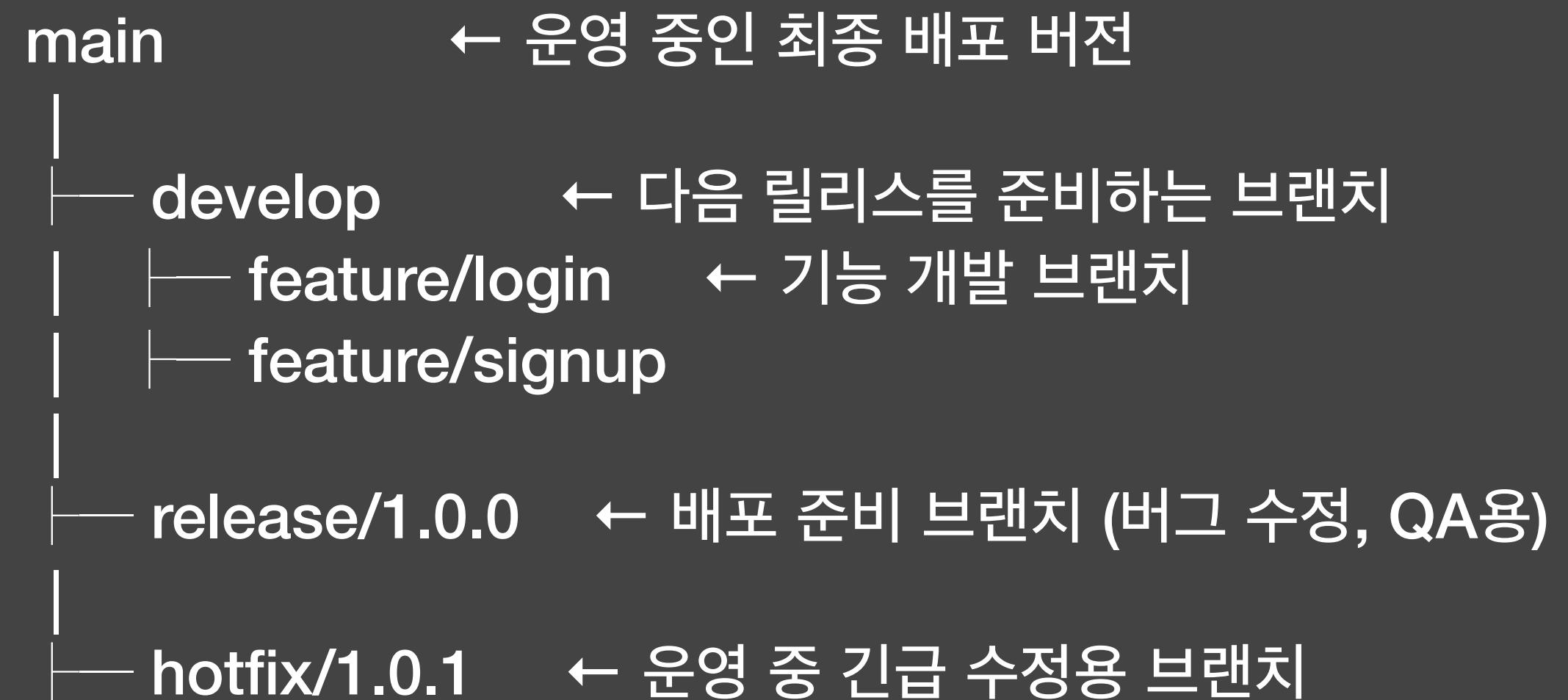
- main : 실제 운영 버전
- develop : 개발 통합 브랜치
- feature/* : 기능 개발 브랜치
- release/* : 배포전 QA & 테스트
- hotfix/* : 운영 중 긴급 수정

- 특징

- 배포/릴리즈 사이클 명확하게 관리
- 브랜치 역할 분리 -> 대규모 프로젝트에 적합
- 릴리즈 안정성 강화

- 단점

- 브랜치 수가 많고 관리 복잡
- PR 리뷰 사이클이 느릴 수 있음



2.3. Trunk-Based Development(TBD)

Trunk-Based Development는 main브랜치(또는 trunk)에 모든 커밋을 집중시키는 전략입니다.

- 브랜치 구조

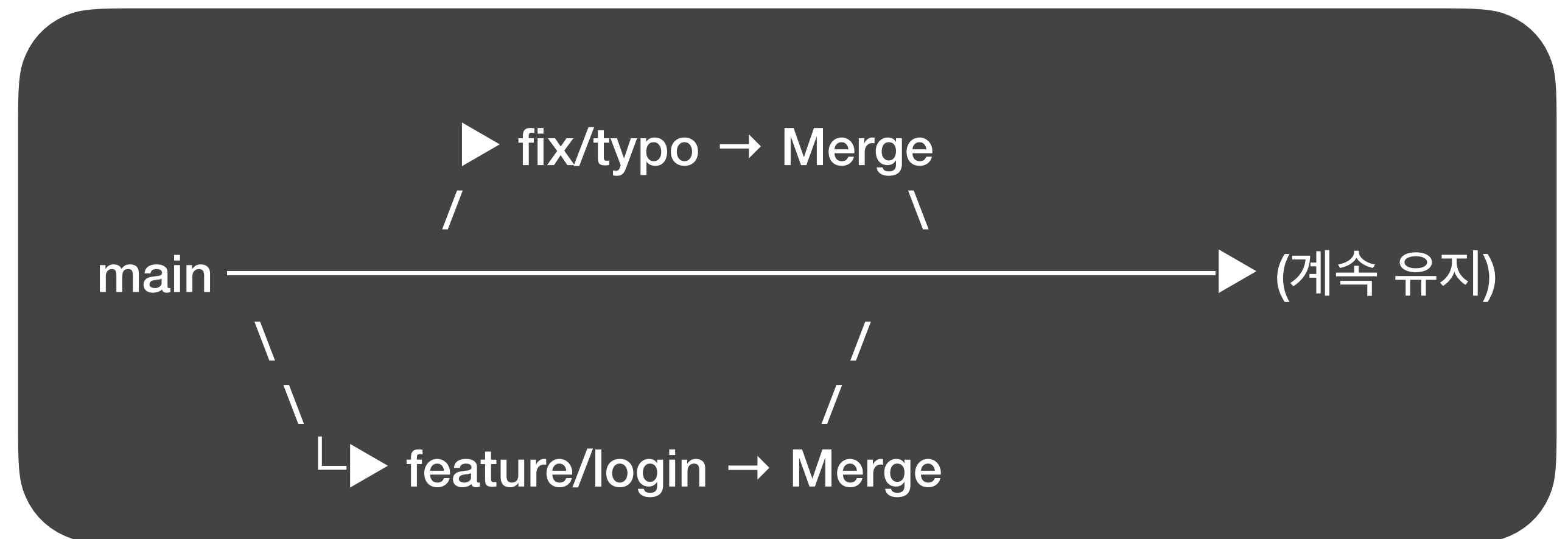
- main(trunk) : 실제 운영 버전
- feature/* : 기능 개발 브랜치(1일 이내)
- hotfix/* : 운영 중 긴급 수정

- 특징

- 단순하고 직관적인 전략
- main 브랜치 중심으로 개발
- 기능(이슈) 단위 브랜치 생성
- 빠른 통합으로 팀원 간의 소스 동기화가 잘됨
- CI/CD 자동화와 연동 필수
- 스타트업

- 주의

- PR이 선택 사항으로 코드 품질 관리 필수(자동 테스트, 코드 리뷰 필수)
- 머지시 전체에 영향 -> 팀내 관리



2.4. 비교 요약 및 선택 기준

전략	특징	적합팀/상황	주요 브랜치
Github Flow	단순하고 직관적 PR 기반 리뷰 수동(반자동) 배포	스타트업 소규모팀	main, feature/*
Git Flow	릴리즈 중심, 브랜치 역할 분리로 안정적 릴리 즈 가능	대규모팀 릴리즈 주기 명확한팀	main, develop, feature/*, release/*, hotfix/*
Trunk-Based	단일 브랜치 중심 짧은 브랜치 수명 지속 통합(CI)에 최적화	스타트업 DevOps 환경 짧은 배포주기	main 또는 trunk, short-lived branch

3.원격 저장소(Remote Repository)

- 원격 저장소란?
- GitHub 원격 저장소 생성
- 원격 저장소 명령어
- GitHub 원격 저장소 연동
- 실습 : 로컬 -> GitHub 푸시 & 팀원 클론

3.1.원격 저장소란?

- 인터넷 상에 존재하는 Git 저장소
- 협업을 위한 공동 저장소 역할을 수행
- 대표적인 플랫폼 : GitHub, GitLab, Bitbucket 등

[로컬. 저장소] <— Push / Pull —> [원격 저장소 (GitHub)]

 A의 노트북

 B의 노트북

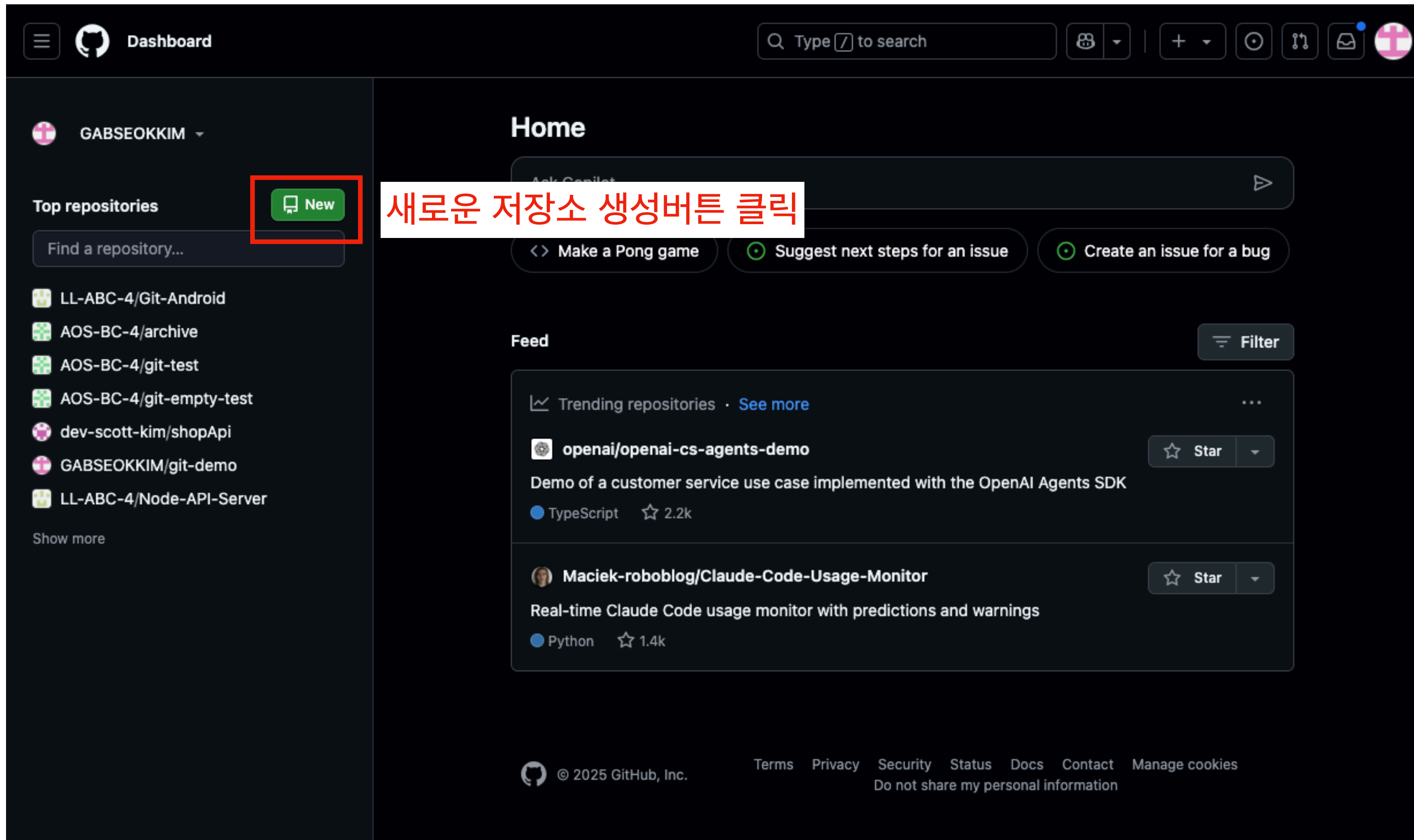
 GitHub 서버

 GitHub 서버

3.2.GitHub 원격 저장소 생성

1. GitHub에 로그인 -> <https://github.com>
2. 새로운 저장소 생성 -> [New] 클릭
3. 저장소 이름 지정
4. 공개 여부 선택
 1. Public : 누구나 볼수 있음, 하지만 소스 변경은 안됨(초대된 사람만 변경 가능)
 2. Private : 본인과 초대한 사람만 접근 가능
5. Initialize 선택X
6. [Create repository] 클릭

3.2.GitHub 원격 저장소 생성



3.2.GitHub 원격 저장소 생성

New repository

Q Type 17 to search

+

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Required fields are marked with an asterisk (*).

Owner *

GABSEOKKIM

/

Repository name *

저장소 이름 설정

Great repository names are short and memorable. Need inspiration? How about [studious-octo-meme](#) ?

Description (optional)

☒

Public

Anyone on the internet can see this repository. You choose who can commit.

☐

Private

You choose who can see and commit to this repository.

저장소 공개 여부 설정

Initialize this repository with:

☐ Add a README file

This is where you can write a long description for your project. [Learn more about READMEs.](#)

저장소 초기화 설정

Add .gitignore

.gitignore template: None

Choose which files not to track from a list of templates. [Learn more about ignoring files.](#)

Choose a license

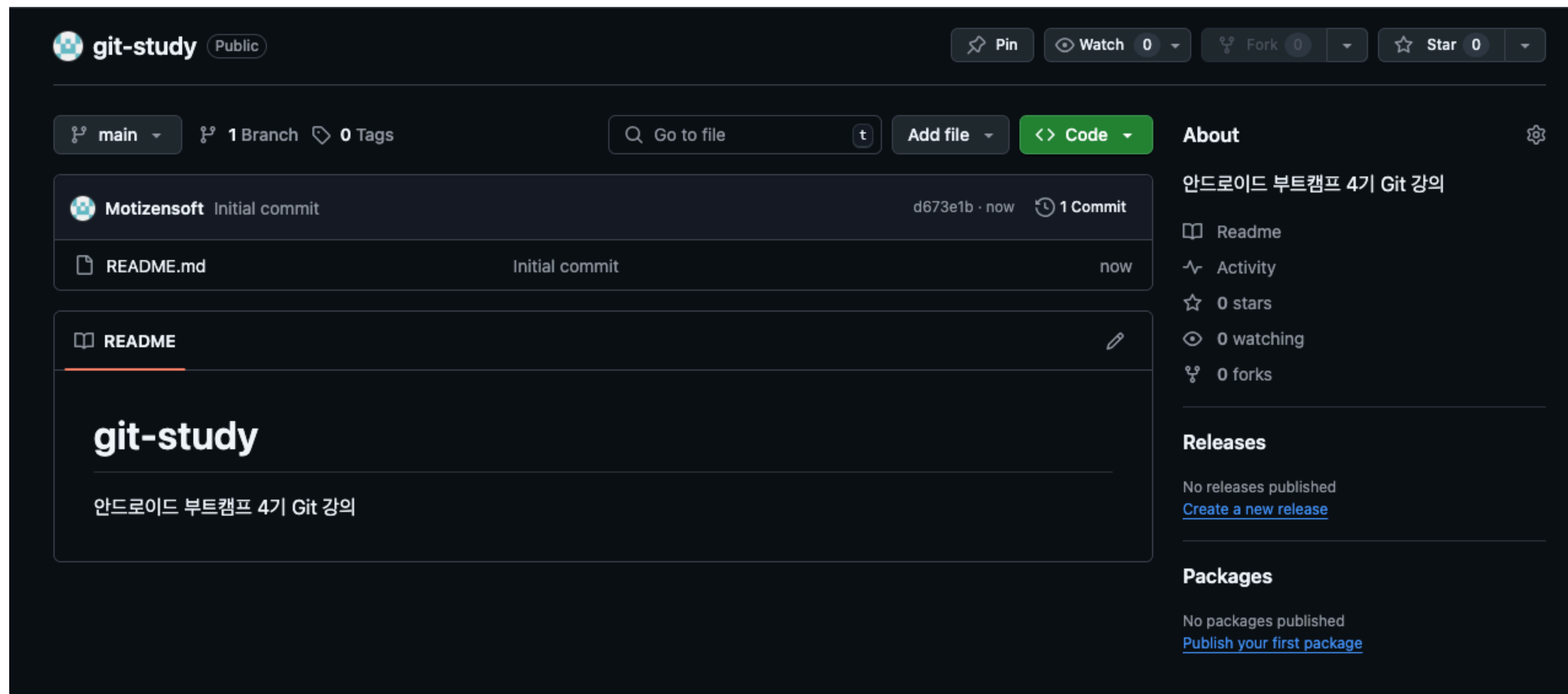
License: None

A license tells others what they can and can't do with your code. [Learn more about licenses.](#)

You are creating a public repository in your personal account.

Create repository

3.2.GitHub 원격 저장소 생성



3.3.원격 저장소 명령어

1.기본 명령어(로컬)

- git remote add origin <url> : 원격 저장소 등록
- git remote remove origin : 원격 저장소 연결 제거
- git remote : 등록된 원격 저장소 목록 보기
- git remote -v : 등록된 원격 저장소 목록(URL) 보기
- git remote set-url origin : 원격 저장소 주소 갱신
- git remote rename <기존이름> <새이름> : 원격 저장소 이름 변경

2.저장소간 연동 명령어(로컬 <-> 원격)

- git push : 로컬 변경 사항 -> 원격 저장소로 전송
- git pull : 원격 저장소 변경 사항 -> 로컬 저장소로 가져오고 병합 (fetch + merge)
- git fetch : 원격 저장소 변경 사항 -> 로컬 저장소로 가져오기만함(병합X)

3.복제 명령어 (로컬 <- 원격)

- git clone <url> : 원격 저장소를 로컬에 복제

3.4.원격 저장소 연동

- GitHub 원격 저장소 연결 : 원격에 브랜치가 있는 경우

```
git remote add origin https://github.com/AOS-BC-4/git-test.git
git fetch origin

...
From https://github.com/AOS-BC-4/git-test
* [new branch]    main    -> origin/main
git pull origin main
```

```
# 강사 테스트 GitHub Repository
# 원격 저장소의 최신 커밋과 브랜치 정보를 가져옴

# 결과 -> origin에 main 브랜치를 받을수 있음.

# 원격 저장소의 데이터 로컬에 병합
```

- GitHub 원격 저장소 연결 : 원격에 브랜치가 없는 경우

```
git remote add origin https://github.com/AOS-BC-4/git-empty-test.git

# 로컬 브랜치 생성 후
git push origin main

...
To https://github.com/AOS-BC-4/git-empty-test.git
* [new branch]    main    -> main
```

```
# 강사 테스트 GitHub Repository

# 원격 저장소 로컬 저장소의 최신 커밋과 브랜치 정보를 전송

# 결과 -> 원격지에 main 브랜치를 전송함
```

- GitHub 원격 저장소 복제 : 원격에 브랜치가 있는 경우

```
git clone https://github.com/AOS-BC-4/git-clone.git
```

```
# 강사 테스트 GitHub Repository
```

3.5.원격 저장소 데이터 연동

- Git PUSH : 로컬 저장소의 변동 사항을 원격 저장소로 업로드

```
git switch -c feature/push-test
echo "first push" > push.txt
git add . && git commit -m "first push test"
git push origin feature/push-test
```

...

To <https://github.com/AOS-BC-4/git-test.git>

* [new branch] feature/push-test -> feature/push-test

푸시 테스트용 기능 브랜치 생성

파일 생성

커밋

원격 저장소의 feature/push-test 브랜치에 업로드

원격지에 저장소 생성됨

- Git PULL : 원격 저장소 변동 사항을 로컬 저장소로 병합

```
git switch main
git pull origin main
```

메인 브랜치로 이동

원격지의 변동 사항을 내려받아 로컬에 병합함

- Git FETCH : 원격 저장소의 변동 사항만 가져옴(병합안함)

```
git fetch origin
remote: Enumerating objects: 1, done.
From https://github.com/AOS-BC-4/git-test
06e6395..5845b98 main -> origin/main
git merge origin/main
```

원격지 브랜치의 변동 사항만 가져옴

원격 저장소에 변경이 생겼으니 업데이트 가능

로컬 메인 브랜치로 이동 원격 변동 사항을 로컬에 적용

3.6.실습(로컬 <-> 원격 저장소 연동)

1.GitHub에 원격 저장소(Repository)를 생성, main 브랜치 생성

2.로컬 저장소에 연결 : remote add or clone

- git remote add origin remoteRepositoryUrl
- git clone remoteRepositoryUrl

3.개인 기능 브랜치 생성(예 feature/order)

- git switch -c feature/order

4.기능 브랜치에 변경 사항 적용 : 파일 추가/수정, 커밋, 푸시

- git add filename, git commit -m comment, git push remote origin

5.병합

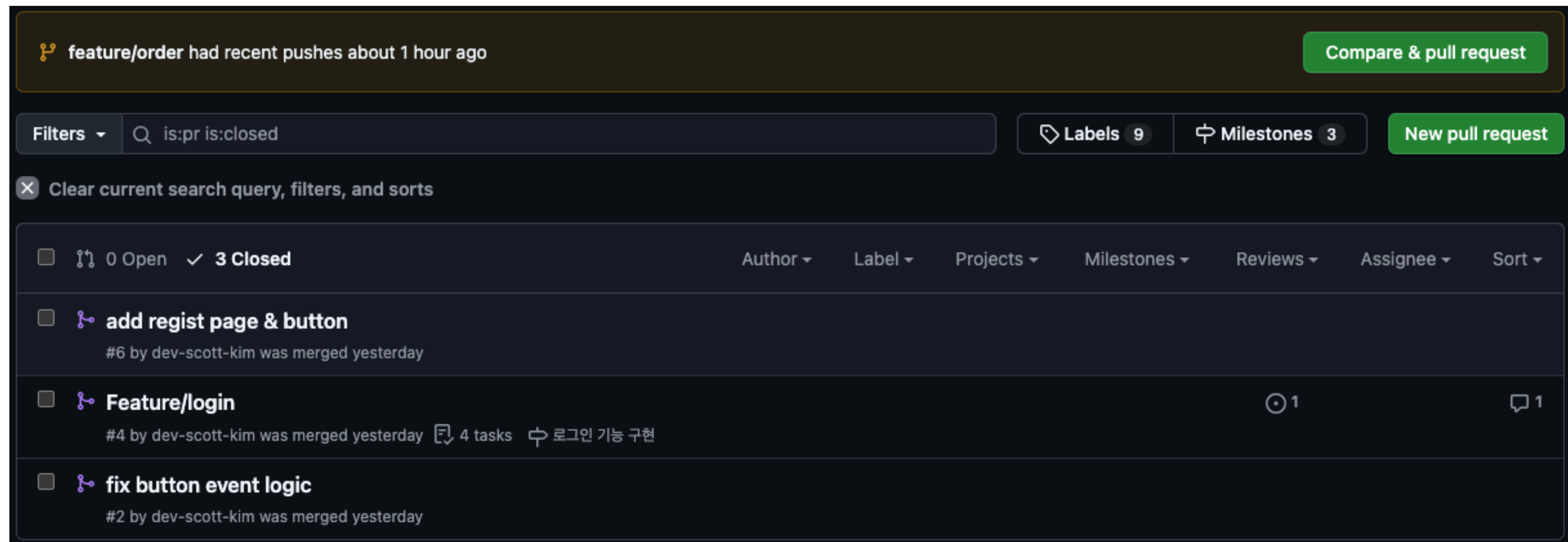
- [option#1]GitHub에서 PR 생성 및 병합
- [option#2]main 에 병합(merge) 후 푸시 : git merge feature/order

4. GitHub Flow & Pull Request & 리뷰

- Pull Request란?
- PR 생성 및 병합 흐름 설명
- PR 템플릿
- 충돌 상황 및 리뷰 문화 실습

4.1. Pull Request란?

- 작업한 내용을 다른 브랜치(보통 main)에 병합해달라고 요청하는 절차
- GitHub 협업의 핵심 도구
- 직접 병합하지 않고, 리뷰 과정을 통해 코드 품질을 높임



The screenshot displays the GitHub Pull Request interface. At the top, a dark blue banner indicates that the `feature/order` branch has recent pushes about 1 hour ago, with a green button labeled "Compare & pull request". Below this, a search bar shows filters for `is:pr is:closed`. To the right of the search bar are buttons for "Labels 9" and "Milestones 3", and a green button for "New pull request". A link to "Clear current search query, filters, and sorts" is also present. The main section shows a list of pull requests with columns for checkboxes, branch names, authors, labels, projects, milestones, reviews, assignees, and sort options. Three pull requests are listed: "add regist page & button" (#6 by dev-scott-kim, merged yesterday), "Feature/login" (#4 by dev-scott-kim, merged yesterday, with 4 tasks and a login feature implementation), and "fix button event logic" (#2 by dev-scott-kim, merged yesterday).

Checkbox	Branch Name	Author	Label	Projects	Milestones	Reviews	Assignee	Sort
<input type="checkbox"/>	<code>add regist page & button</code>	#6 by dev-scott-kim						
<input type="checkbox"/>	<code>Feature/login</code>	#4 by dev-scott-kim				1		
<input type="checkbox"/>	<code>fix button event logic</code>	#2 by dev-scott-kim						

4.2. PR 생성 및 병합 흐름 설명

1. 새로운 기능을 위해 feature 브랜치 생성
2. 기능 개발 완료 후 원격 저장소에 push
3. GitHub 웹에서 Pull Request 생성
4. 팀원 코드 리뷰
5. 리뷰 후 승인되면 병합 (merge)
6. 브랜치 삭제

4.2.1. PR 생성 및 병합 흐름 설명

1. 새로운 기능을 위해 feature 브랜치 생성 후 기능 개발 완료 후 원격 저장소에 push

```
git switch -c feature/login  
echo "add login function completed" > login.txt  
git add . && git commit -m "completed login"  
git push origin feature/login
```

기능 브랜치 생성
기능 개발
개발 완료 후 커밋
원격 저장소에 푸시

-> 현재 브랜치를 처음 원격 저장소에 푸시하려고 할때 아직 연결이 되지 않은 경우 오류

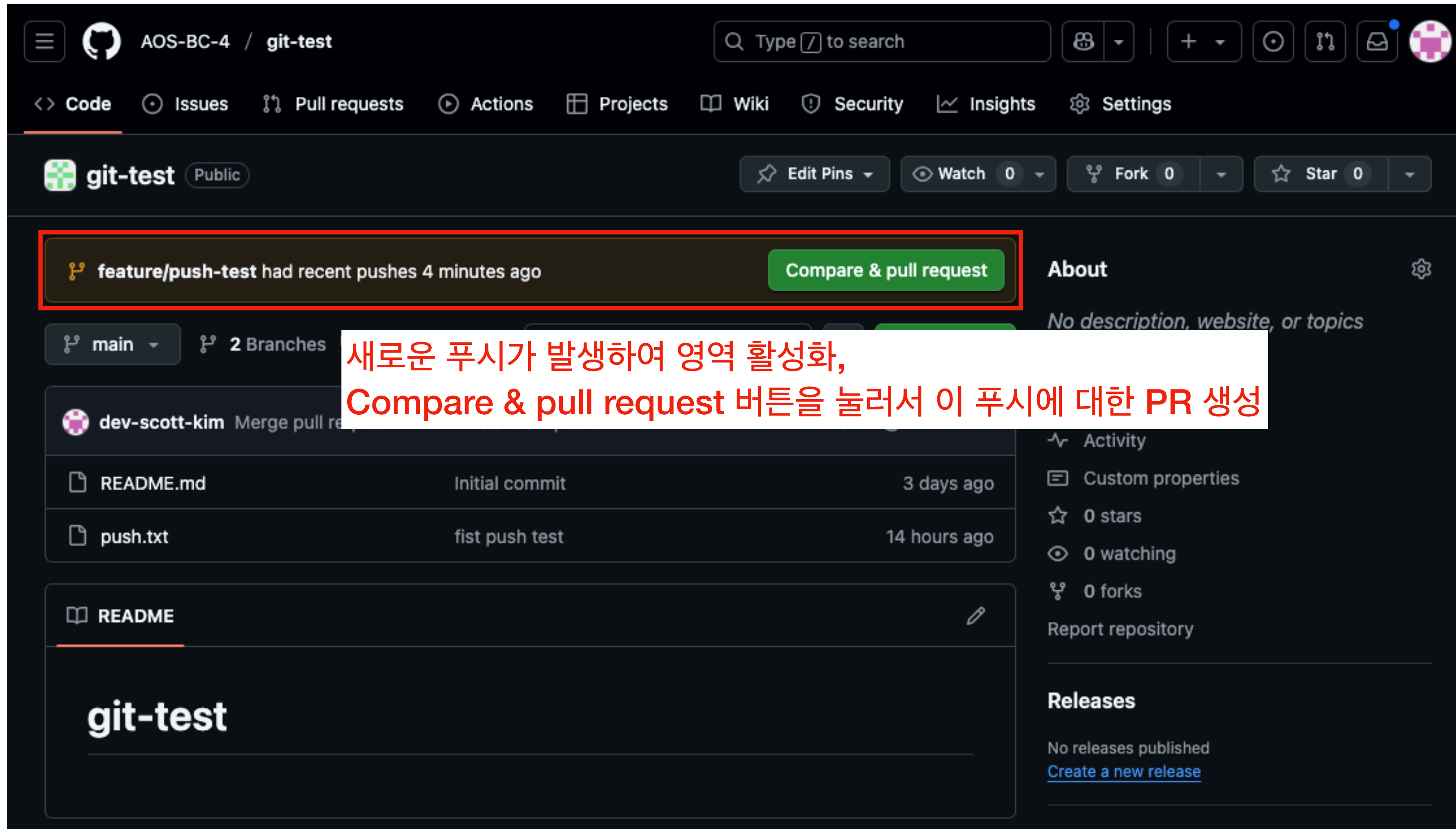
```
- - - - result message - - - - -  
fatal: The current branch feature/push-test has no upstream branch.  
To push the current branch and set the remote as upstream, use
```

```
git push --set-upstream origin feature/login
```

실행

4.2.2 PR 생성 및 병합 흐름 설명

2. GitHub 웹에서 상태 조회



The screenshot shows the GitHub interface for a repository named 'git-test' under the user 'AOS-BC-4'. The repository is public. A notification bar at the top indicates that the 'feature/push-test' branch had recent pushes 4 minutes ago, with a green 'Compare & pull request' button next to it. Below this, the repository's main branch is 'main', and there are 2 branches. A table lists recent commits: 'Initial commit' (3 days ago) and 'fist push test' (14 hours ago). The 'README' file is visible, and the repository name 'git-test' is displayed at the bottom. On the right side, the 'About' section shows no description, website, or topics, and the 'Releases' section shows no releases published with a link to 'Create a new release'.

새로운 푸시가 발생하여 영역 활성화,
Compare & pull request 버튼을 눌러서 이 푸시에 대한 PR 생성

4.2.3 PR 생성 및 병합 흐름 설명

3. GitHub 웹에서 Pull Request 생성

Open a pull request
Create a new pull request by comparing changes

base: main ← compare: feature/push-test

Add a title
second push commit

Add a description
Write Preview H B I ≡ < > @ ↩ ↪ ⌨
Add your description here...
리뷰를 요청할 내용을 작성

Markdown is supported Paste, drop, or click to add files

Reviewers
Suggestions
GABSEOKKIM Request

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Development

Create pull request

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

Helpful resources
[GitHub Community Guidelines](#)

1 commit 1 file changed 1 contributor

Compare & pull request 버튼을 눌러서 PR 생성시에는 자동 세팅됨

리뷰를 요청할 대상 설정

PR 최종 승인 책임자

리뷰를 설정이 완료되면 생성

4.2.4 PR 생성 및 병합 흐름 설명

4. GitHub 웹에서 Pull Request 상세 페이지

second push commit #2

dev-scott-kim wants to merge 1 commit into main from feature/push-test

Conversation 0Commits 1Checks 0Files changed 1

dev-scott-kim commented now

PR 요청 상세 내용

No conflicts with base branch

PR 병합

Add a comment

리뷰 메시지 작성영역

Reviewers

Suggestions

GABSEOKKIM

Request

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

None yet

Notifications

Customize

Unsubscribe

You're receiving notifications because you authored

conflict second line #3

dev-scott-kim wants to merge 1 commit into main from feature/push-test

Conversation 0Commits 1Checks 0Files changed 1

dev-scott-kim commented 1 minute ago

dddd

This branch has conflicts that must be resolved

충돌 발생 [Resolve conflicts]

눌러서 충돌 해결 화면으로 이동

Add a comment

Reviewers

Suggestions

GABSEOKKIM

Request

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

None yet

Notifications

Customize

Unsubscribe

You're receiving notifications because you authored the thread.

4.2.5 PR 생성 및 병합 흐름 설명

. GitHub 웹에서 Pull Request 충돌 해결

conflict second line #3

Resolving conflicts between feature/push-test and main and committing changes → feature/push-test

1 conflicting file

push.txt

1 conflict Prev Next

```
1 pr conflict
2 <<<<<< feature/push-test
3 modify second line
4 =====
5 conflict second line
6 >>>>>> main
7
```

충돌난 소스 확인 및 수정

conflict second line #3

Resolving conflicts between feature/push-test and main and committing changes → feature/push-test

1 conflicting file

push.txt

1 conflict Prev Next

Mark as resolved

```
1 pr conflict
2 modify second line
3 conflict second line
```

충돌 소스 수정 하면 버튼 활성화후 클릭

conflict second line #3

Resolving conflicts between feature/push-test and main and committing changes → feature/push-test

Commit merge

1 conflicting file

push.txt

✓ Resolved

```
1 pr conflict
2 modify second line
3 conflict second line
```

해결되면 [Commit merge] 클릭

4.2.6 PR 생성 및 병합 흐름 설명

. GitHub 웹에서 Pull Request 머지 완료

The screenshot shows a GitHub Pull Request (PR) titled "conflict second line #3". The PR is in a "Merged" state, with a message indicating that "dev-scott-kim merged 2 commits into main from feature/push-test now". The PR details show a commit history with a merge of "main" into "feature/push-test" and a subsequent merge of "feature/push-test" into "main". A red box highlights a notification message: "Pull request successfully merged and closed. You're all set — the feature/push-test branch can be safely deleted." Next to this message is a "Delete branch" button. Below the notification is a comment section with a "Write" tab and a "Preview" tab. The "Write" tab is active, showing a text area for adding a comment. The "Preview" tab shows the rendered comment. The "Comment" button is at the bottom right of the comment section. The right sidebar contains various settings and information, including "Reviewers", "Suggestions", "Assignees", "Labels", "Projects", "Milestone", "Notifications", and "Participants".

conflict second line #3

Merged dev-scott-kim merged 2 commits into main from feature/push-test now

Conversation 0 Commits 2 Checks 0 Files changed 1 +2 -1

dev-scott-kim commented 20 minutes ago

dddd

GABSEOKKIM and others added 2 commits 22 minutes ago

conflict second line

Merge branch 'main' into feature/push-test

dev-scott-kim merged commit c6721a6 into main now

Pull request successfully merged and closed

You're all set — the feature/push-test branch can be safely deleted.

Delete branch

Add a comment

Write Preview

Add your comment here...

Markdown is supported Paste, drop, or click to add files

Comment

mer지가 완료되면 PR은 Close 상태가 되고, [Delete branch] 활성화됨

4.3. PR Template

- PR을 생성할 때 자동으로 채워지는 기본 양식
- 팀원 간 코드 리뷰 기준을 통일하고, 커뮤니케이션을 체계화
- .github/pull_request_template.md 파일에 작성

<템플릿 예제>

```
## 변경 내용 요약  
- 어떤 기능을 추가/수정했는지
```

```
## 테스트 방법  
- 어떻게 테스트했는지
```

```
## 체크리스트  
- [ ] 기능 정상 동작 확인  
- [ ] 사이드 이펙트 확인
```

```
...
```

4.4.실습(GitHub Flow & Pull Request & 리뷰)

1. 새로운 기능을 위해 feature 브랜치 생성
2. 기능 개발 완료 후 원격 저장소에 push
3. GitHub 웹에서 Pull Request 생성
4. 코드 리뷰
5. 리뷰 후 승인되면 병합 (merge)
6. 브랜치 삭제

5. 협업에 필요한 필수 Git 명령어

- git stash : 현재 작업을 임시 저장
- git commit —amend : 커밋 편집
- git revert : 커밋을 되돌리기
- git rebase : 커밋 재배포치
- git reset : 커밋 초기화
- git restore : 변경 사항 취소
- git cherry-pick : 특정 커밋 선택 적용

#Git Stash - 임시 저장

- 작업 중인 변경사항을 임시로 저장
- 작업 브랜치 초기화
- `git stash` : 변경 사항을 임시로 저장하고 워킹 트렉 토리를 원래대로 되돌림
- `git stash push -m "임시 저장명"`
- `git stash list` : stash 목록
- `git stash pop` : 가장 최근 stash를 적용하고 stash 에서 제거
- `git stash apply` : 가장 최근 stash를 적용하고 stash 에는 남김
- `git stash drop` : 특정 stash 항목 삭제
- `git stash clear` : 모든 stash 항목 삭제

1. 임시 저장

```
git stash push -m "기능 개발중"
```

2. 임시 저장 목록 확인

```
git stash list
```

- - - result - -

```
stash@{0}: 0n feature/login : 구현중
```

```
stash@{1}: 0n feature/login : 버튼 생성중
```

3. 특정 임시 저장 데이터 적용

```
Git stash apply stash@{1}
```

4. 가장 최근 내용 적용

```
git stash pop #스테이시 삭제됨
```

#Git Commit —amend

- 최근 커밋 수정

1.사용 시점

- 커밋 메세지 수정
- 커밋시 파일 누락

2. 주의 사항

- 이미 push 한 커밋이면 조심
- main에 병합한 커밋이면 위험
- 혼자 작업중인 브랜치에서 사용 권장

1. 누락된 파일만 추가 할때

```
git add xxx.txt  
git commit -amend
```

2. 누락된 파일을 추가 및 메세지도 함께 수정 할때

```
git commit -amend -m '수정된 메세지'
```

3. 수정된 커밋 확인

```
git log - - oneline
```

#Git Revert - 커밋 되돌리기

- 커밋 되돌리기

1. 해당 커밋의 변경 사항만 취소, 새 커밋 생성
2. 기존 커밋 히스토리 그대로 유지

main : A - - - B - - - C - - - D <- 기존

revert C 실행

reverted : A - - - B - - - C - - - D - - - D' - - - C' <- 되돌리는 커밋 D',C' 생성

#main : 소스 상태는 최종 B 상태임.

#Git Rebase - 커밋 재배치

- 커밋 재배치

1. 브랜치의 베이스를 재비치하여 깔끔한 커밋 히스토리 유지
2. 작업 브랜치를 최신 상태로 유지

```
Main :      A - - - B - - - D - - - E
           |
Feature :    C
```

```
Main :      A - - - B - - - D - - - E
           |
Feature (rebased): C'
```

<주의 사항>

- 공유(main,develop) 브랜치에서 사용 금지 : 충돌 및 히스토리 꼬임.
- 충돌 발생시 해결하기 어려움 : 충돌 발생시 각 커밋마다 모두 수동으로 해결을 해야함.
- 커밋 해시가 바뀌므로 기존 커밋 참조가 무효화 될수 있음. -> 강제 푸시 해야함(git push --force)

#Git Rebase : 시나리오

기능 브랜치에서 개발중 혹은 배포전, main 이 변경된 경우 최신 소스를 반영 하여 배포 해야할 상황

1. 메인 브랜치로 이동 A,B,C 커밋

```
git switch main
```

```
git add A.txt && git commit -m 'commit A'
```

```
git add B.txt && git commit -m 'commit B'
```

2. 새 브랜치 생성

```
git switch -c feature/rebase
```

```
git add C.txt && git commit -m 'commit C'
```

3. Main 에 E 커밋 추가

```
git switch -c main
```

```
git add D.txt && git commit -m 'commit D'
```

```
git add E.txt && git commit -m 'commit E'
```

1. 메인 브랜치로 이동 후 최신화

```
git switch feature/rebase
```

```
git pull origin feature/rebase
```

2. rebase 실행

```
git rebase main
```

3. 충돌 발생 시

→ 충돌 해결

→ git add .

→ git rebase --continue

4. 원격에 푸시

```
git push origin feature/rebase
```

7. 메인에 머지

```
git switch main
```

```
git merge feature/rebase
```

#Git Reset - 커밋 초기화

- 커밋 히스토리 와 작업 디렉토리를 이전 상태로 되돌림
 - `git reset - - soft <Head>` : 커밋 이동, staged 상태
 - `git reset - - mixed <Head>` : 커밋 이동, staged -> unstated
 - `git reset - - hard <Head>` : 커밋 이동, staged, 작업 디렉토리를 모두 이전 상태로
 - `git reset - - mixed` : staged -> unstated
 - `git reset - - hard` : 작업 디렉토리를 모두 이전 상태로

<주의 사항>

- hard 옵션은 변경사항을 완전히 삭제하므로 복구 불가
- 협업 중인 브랜치의 커밋을 reset 하면 히스토리 충돌 발생

#Git Reset - 커밋 초기화 : 시나리오

1. 현재 브랜치 상태 확인

`git status`

2. 커밋 로그를 확인

`git log - - oneline`

D 커밋 해시: def5678

3. - - soft 리셋 후 상태 확인

`git reset - - soft def5678`

`git status`

4. 잘못 커밋한 내용을 수정하여 재 커밋

`git add . && commit -m '커밋 수정'`

1. - - mixed 리셋 후 상태 확인

`git reset - - mixed def5678`

`git status`

2. 잘못 커밋한 내용을 수정하여 재 커밋

`git add . && commit -m '커밋 수정'`

1. - - hard 리셋 후 상태 확인

`git reset - - hard def5678`

`git status`

커밋된 파일들 모두 삭제됨.

#Git restore <filename>

- 현재 작업 디렉토리의 변경 사항을 복원
 - **restore <file>** : 작업중인 파일의 변경 사항 폐기
 - **restore - - staged** : 실수로 add 한 파일을 취소
 - **restore - - source <Hash> <file>** : 특정 커밋 시점으로 파일을 돌리고 싶을때

1. 특정 파일 복원

git restore <file>

2. 여러 파일 복원

git restore <file1> <file2>

3. 디렉토리 전체 복원

git restore <directory> <- 디렉토리명

4. 워킹 디렉토리 복원

git restore .

5. 워킹 디렉토리 스테이지에서 unstage(add 되돌리기)

git restore - - staged .

6. 특정 커밋 시점으로 파일을 돌리고 싶을때

git restore - - source <hash> <file>

#Git cherry-pick

- 특정 커밋만 골라 현재 브랜치에 적용
- 브랜치간 커밋 이력을 합치지 않고 부분 기능만 가져오기

```
main :      A - - - B - - - C - - - E      <- 기존
              \
feature :      D - - - F      <- cherry-pick D만 newfeature로

              cherry-pick D 실행

newFeature :  A - - - B - - - C - - - E - - - D'  <- 새로운 커밋으로 복사됨
```

#Git cherry-pick : 시나리오

기능 브랜치에서 개발중, main 에서 기능 브랜치의 일부 커밋만 포함하여 배포 해야할때

1. 메인 브랜치로 이동 A,B,C 커밋

```
git switch main
```

```
git add A.txt && git commit -m 'commit A'
```

```
git add B.txt && git commit -m 'commit B'
```

```
git add C.txt && git commit -m 'commit C'
```

2. 새 브랜치 생성

```
git switch -c feature
```

```
git add D.txt && git commit -m 'commit D'
```

3. Main 에 E 커밋 추가

```
git switch -c main
```

```
git add E.txt && git commit -m 'commit E'
```

4. 기능 브랜치 생성

```
git switch -c feature
```

```
git add F.txt && git commit -m 'commit F'
```

1. 메인 브랜치로 이동 후 최신화

```
git switch main
```

```
git pull origin main
```

2. 새 브랜치 생성

```
git switch -c feature-d
```

3. feature 브랜치에서 커밋 해시 확인

```
git log feature --oneline
```

D 커밋 해시: def5678

4. cherry-pick 실행

```
git cherry-pick def5678
```

5. 충돌 발생 시

→ 충돌 해결

→ git add .

→ git cherry-pick --continue

6. 원격에 푸시

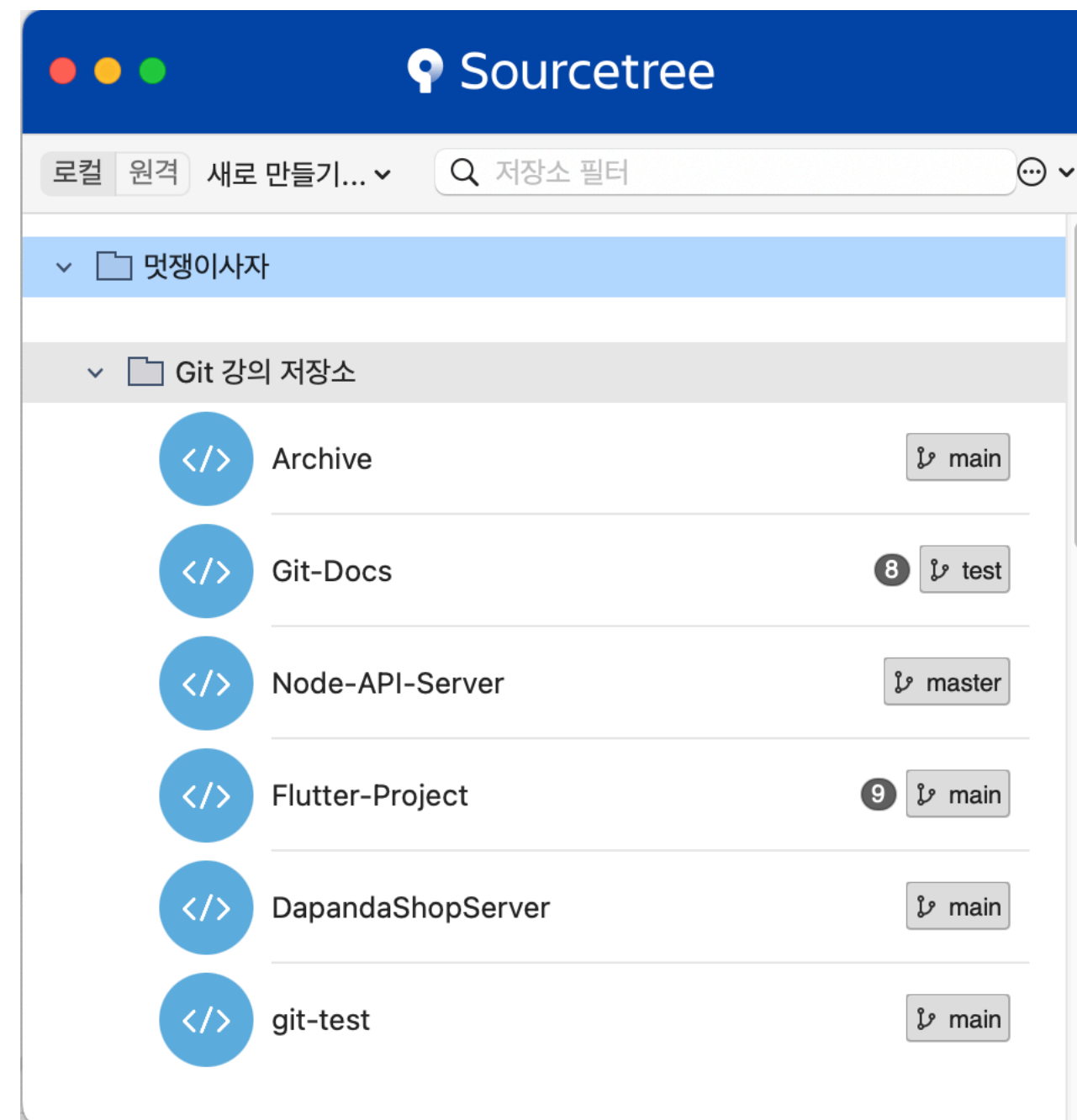
```
git push origin feature-d
```

7. 메인에 머지

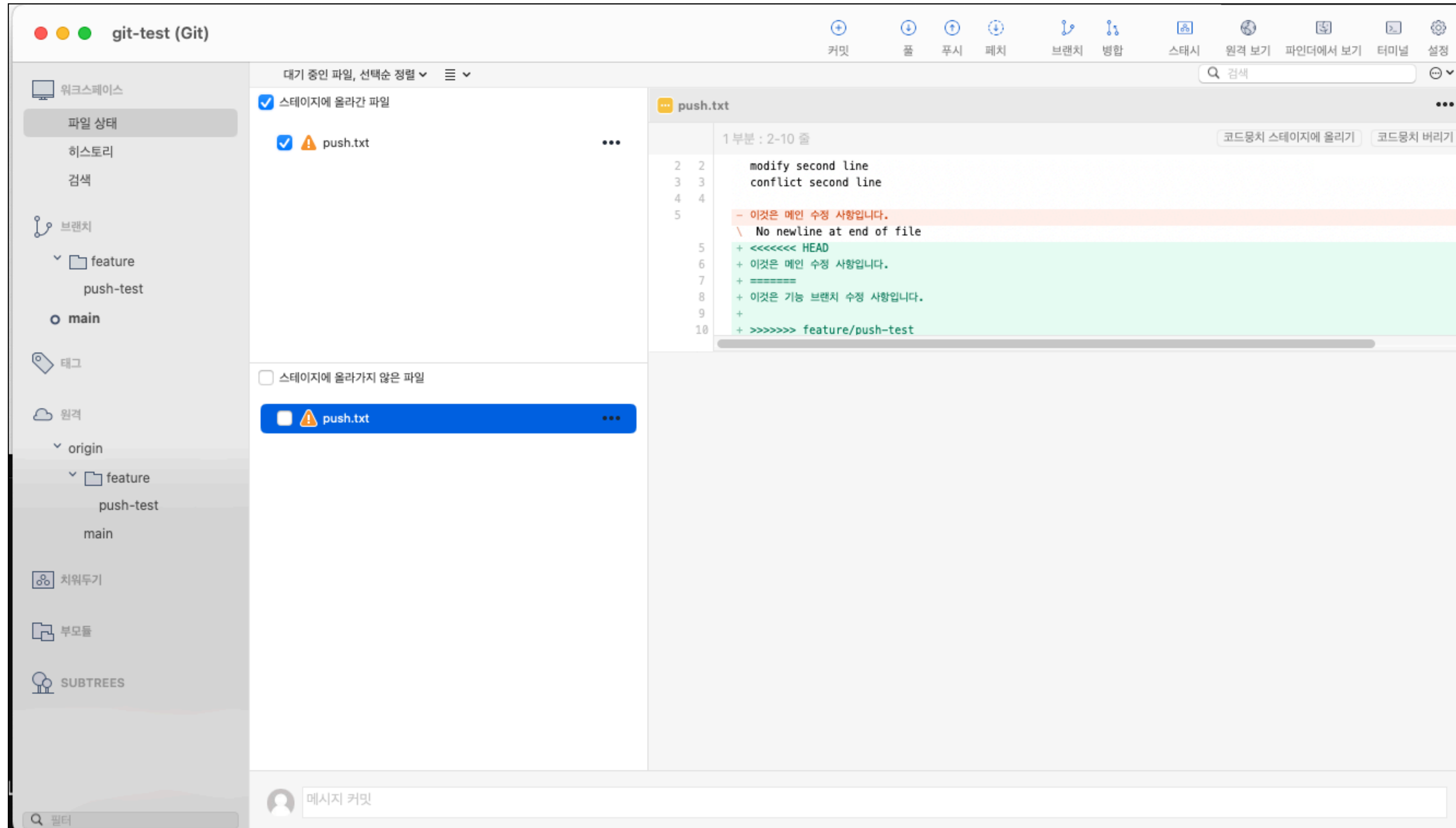
```
git switch main
```

```
git merge feature-d
```

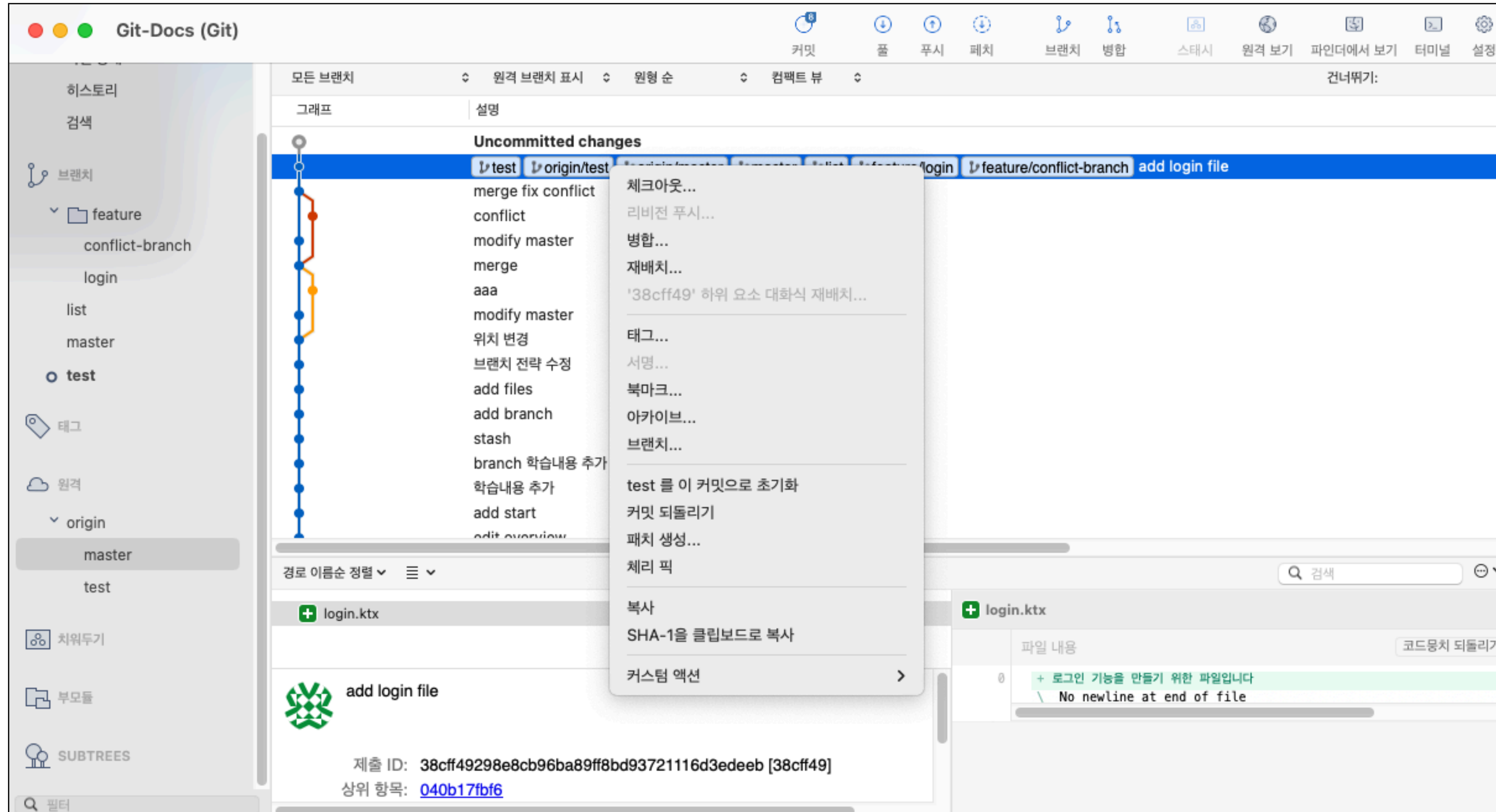
6. Git 활용툴 리뷰 - SourceTree



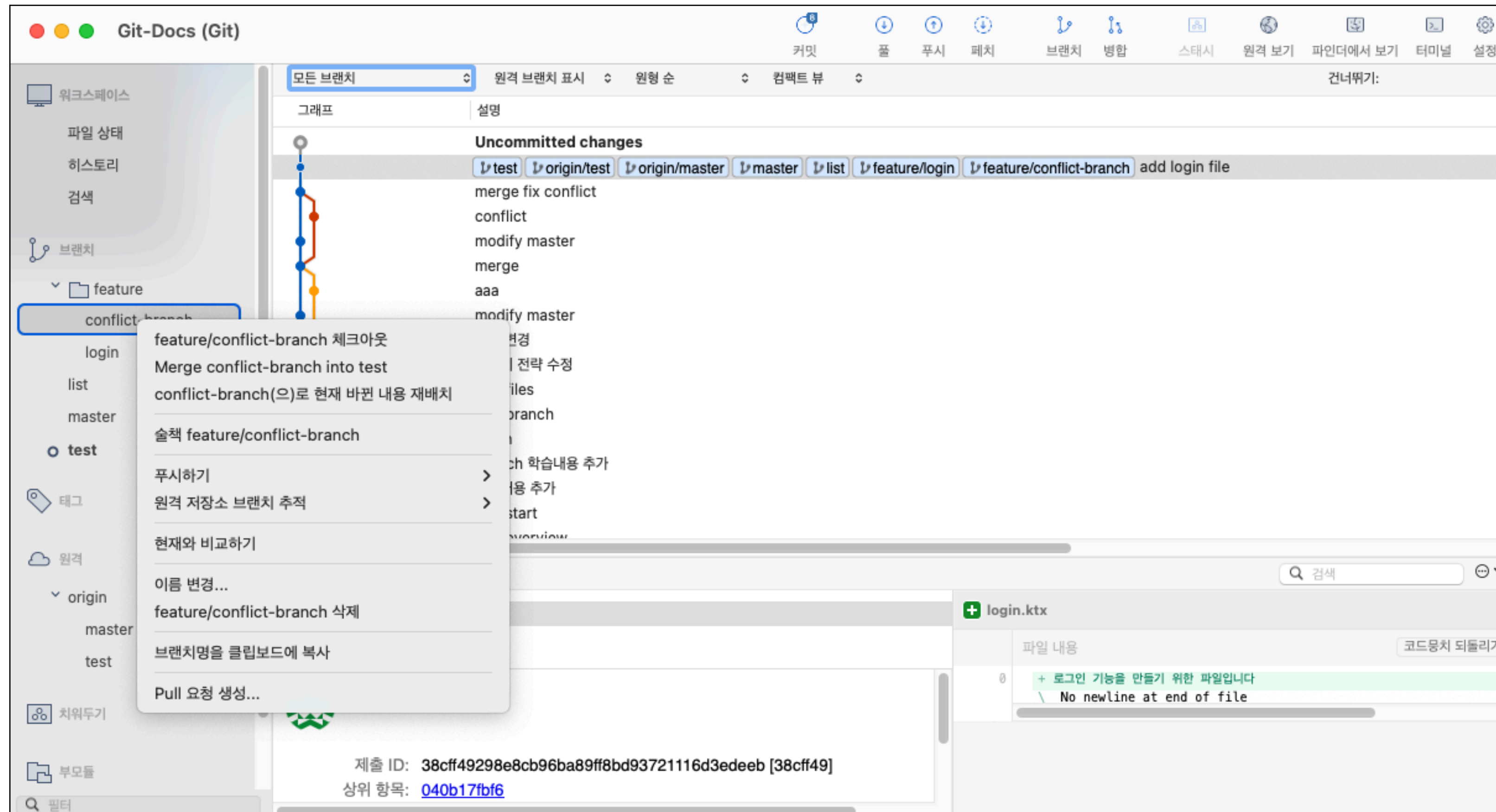
sourceTree 리뷰 - 충돌 파일 확인



sourceTree 리뷰 - 기능들



sourceTree 리뷰 - 기능들



7. 팀별 실습

이름	역할	주요 업무
A팀장	저장소 초기 설정, 승인된 PR에 대한 병합,충돌 유도 소스, 리뷰 참여	.github/pr템플릿 작성, 최종 병합
B 개발	기능 1 개발, PR 생성, 리뷰 참여	feature/*
C 개발	기능 2 개발, PR 생성, 리뷰 참여	feature/*
D 개발	기능 3 개발, PR 생성, 리뷰 참여	feature/*

수고하셨습니다