

Lecture 2.

GitHub 개인 설정

김영빈 교수

01 강의 개요

학습 목표

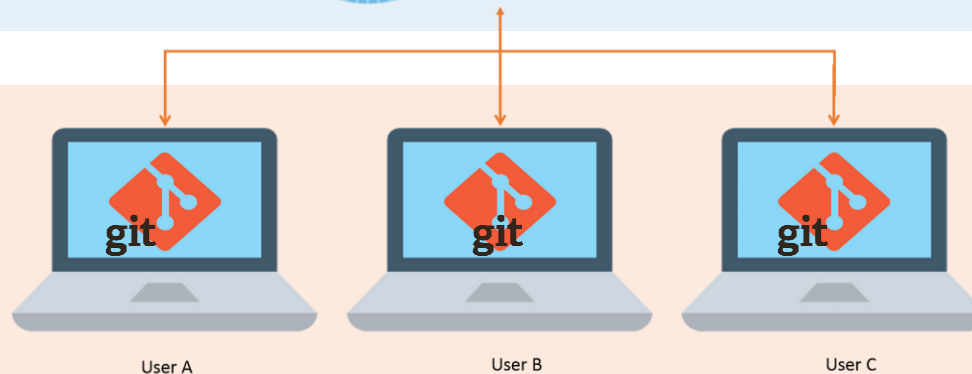
- Git/GitHub의 개념과 차이를 명확하게 이해
- 버전 관리 시스템(VCS)의 필요성 파악
- Git의 내부 동작 원리(Snapshot 방식, SHA-1 해시 기반 구조) 이해
- GitHub에서 협업하는 방법(Branch, Pull Request, Issues 등)
- 실무에서 GitHub가 어떻게 사용되는지 사례를 통해 학습

Cloud :



원격 저장소, 팀 협업 지원
ex) git push, git pull, PR

Local :



Local 컴퓨터에서 작업, 버전 관리
ex) git add, git commit

버전 관리 시스템(VCS)

버전 관리가 왜 필요한가?

- 실수로 기존 코드 덮어쓰기 → 복구 불가
- 여러 명이 동시에 수정 → 충돌 발생
- 비효율적인 수동 백업
(project_v1, project_v2_final, project_v3_final_final 같은 방식으로 파일을 관리하면 혼란 발생)



해결책 : 버전 관리 시스템 (VCS, Version Control System)

- 모든 변경 사항 자동 저장 → 필요할 때 과거 버전으로 쉽게 되돌리기 가능
- 동시 작업 및 충돌 관리 → 여러 명이 협업해도 효율적으로 코드 관리 가능
- 변경 이력 추적 → 누가, 언제, 무엇을 수정했는지 기록하여 프로젝트 관리 용이

VCS의 유형

중앙 집중형 VCS	분산형 VCS
<ul style="list-style-type: none">• 중앙 서버에 모든 버전이 저장됨• 인터넷이 없으면 작업 불가능• 서버 장애 시, 모든 데이터 손실 가능• CVS(Concurrent Version System), SVN(Subversion)	<ul style="list-style-type: none">• 각 사용자가 Local에 모든 기록을 저장 → 인터넷이 없어도 작업 가능• 중앙 서버가 손상돼도 각 사용자 PC에 모든 기록이 남아 있음• 빠르고 유연한 Branch 관리 가능• Git

Git과 GitHub의 차이



Local 컴퓨터에서 버전 관리를 담당하는 소프트웨어로, 파일 변경 사항을 추적하고 관리

```
$ git init
Initialized empty Git repository in /tmp/tmp.IMBYSY7R8Y/.git/
$ cat > README << 'EOF'
> Git is a distributed revision control system.
> EOF
$ git add README
$ git commit
[master (root-commit) e4dcc69] You can edit locally and push
to any remote.
 1 file changed, 1 insertion(+)
 create mode 100644 README
$ git remote add origin git@github.com:cdown/thats.git
$ git push -u origin master
```



GitHub

Git을 기반으로 한 웹 호스팅 서비스로, 코드 저장소와 다양한 협업 기능을 제공



OpenAI

Verified

92.5k followers

<https://openai.com/>

[Overview](#) [Repositories 190](#) [Projects](#) [Packages](#) [People 19](#)

Pinned

[openai-cookbook](#) Public

Examples and guides for using the OpenAI API

MDX 62.1k 10k

[whisper](#) Public

Robust Speech Recognition via Large-Scale Weak Supervision

Python 77.7k 9.3k



git



GitHub

	git	GitHub
기능	버전 관리 시스템(VCS)	클라우드 기반 협업 플랫폼
설치 위치	Local 컴퓨터 (개발자의 PC)	웹 서버 (github.com)
주요 역할	파일 변경 기록 저장, Branch 관리	코드 저장소 제공, 협업 기능 (PR, Issue)
네트워크 필요 여부	Local에서도 사용 가능	인터넷 필요
사용 방식	git commit, git merge	git push, git pull, PR

GitHub의 활용 사례

코드 저장소 외, GitHub를 활용하는 방법

- 개발자 협업의 중심
- 자동화 (CI/CD) : GitHub Actions로 자동 배포 및 테스트
- 포트폴리오 활용



기업의 오픈소스 프로젝트

Google

TensorFlow, Flutter

Facebook

React, PyTorch

Microsoft

VS Code, TypeScript

OpenAI

Whisper, GPT 모델 공개



GitHub를 사용하는 기업 사례

Netflix

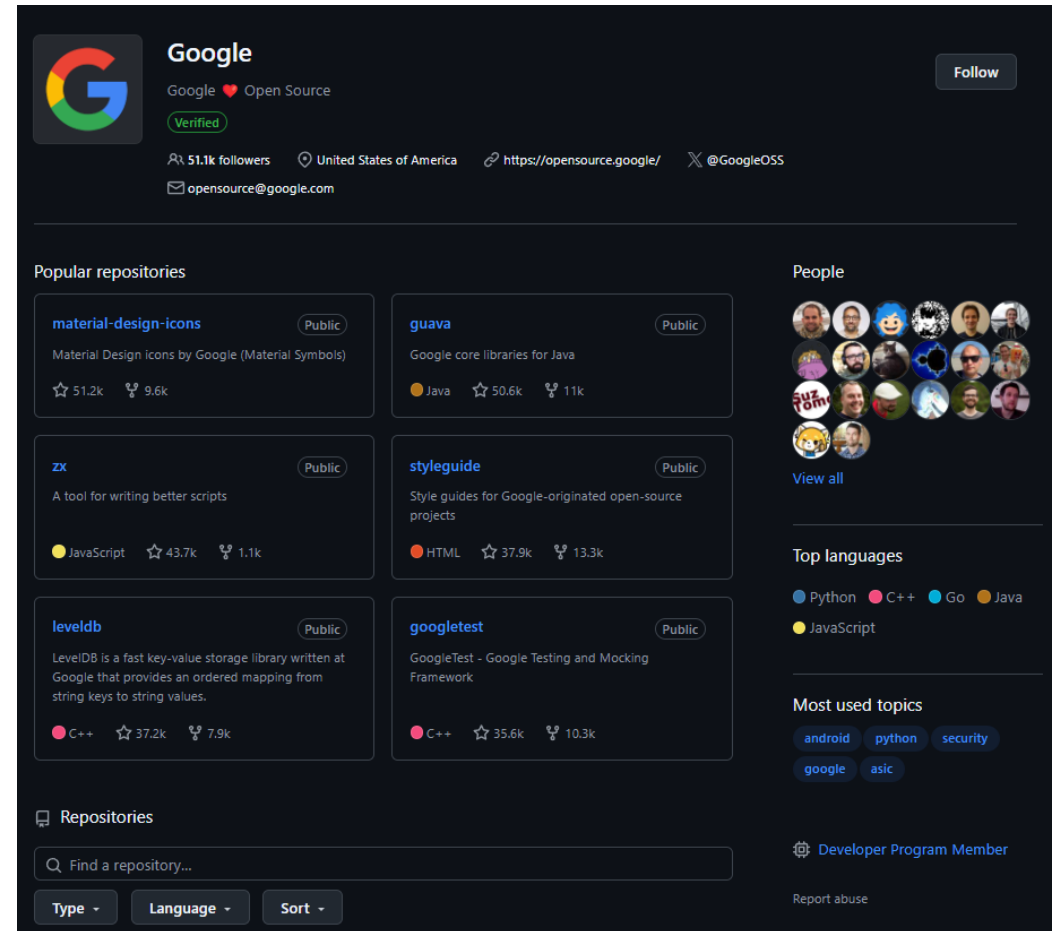
GitHub을 활용한 마이크로서비스 관리

NASA

GitHub을 이용해 우주 탐사 데이터 공유

Tesla

GitHub을 통해 자율주행 관련 코드 오픈소스화



▲ Google의 GitHub 사용 예시

02 Git의 내부 구조와 동작 원리

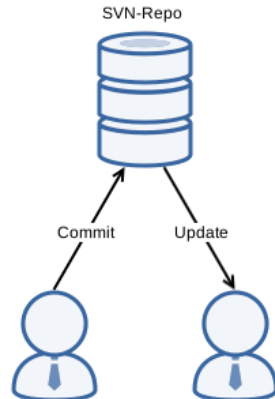
Git의 구조

SVN vs Git 저장방식 비교



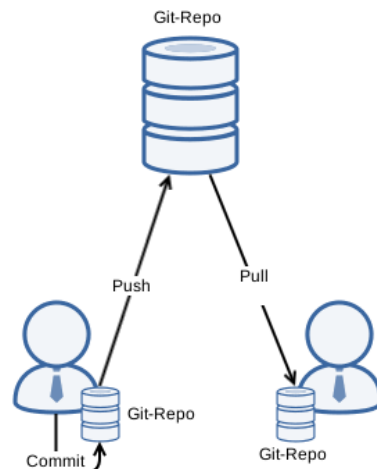
: 델타(Delta) 방식

- 중앙 집중 관리식
- 파일의 변경 사항만 저장하는 "Diff 기반 저장"
- 과거 버전으로 돌아가기 위해 **많은 연산 필요**
 - 초기 버전에서 변경 사항을 순차적으로 적용하여 현재 버전 계산
 - 오래된 버전으로 돌아가려면 변경 사항을 역순으로 적용해야 함



: 스냅샷(Snapshot) 방식

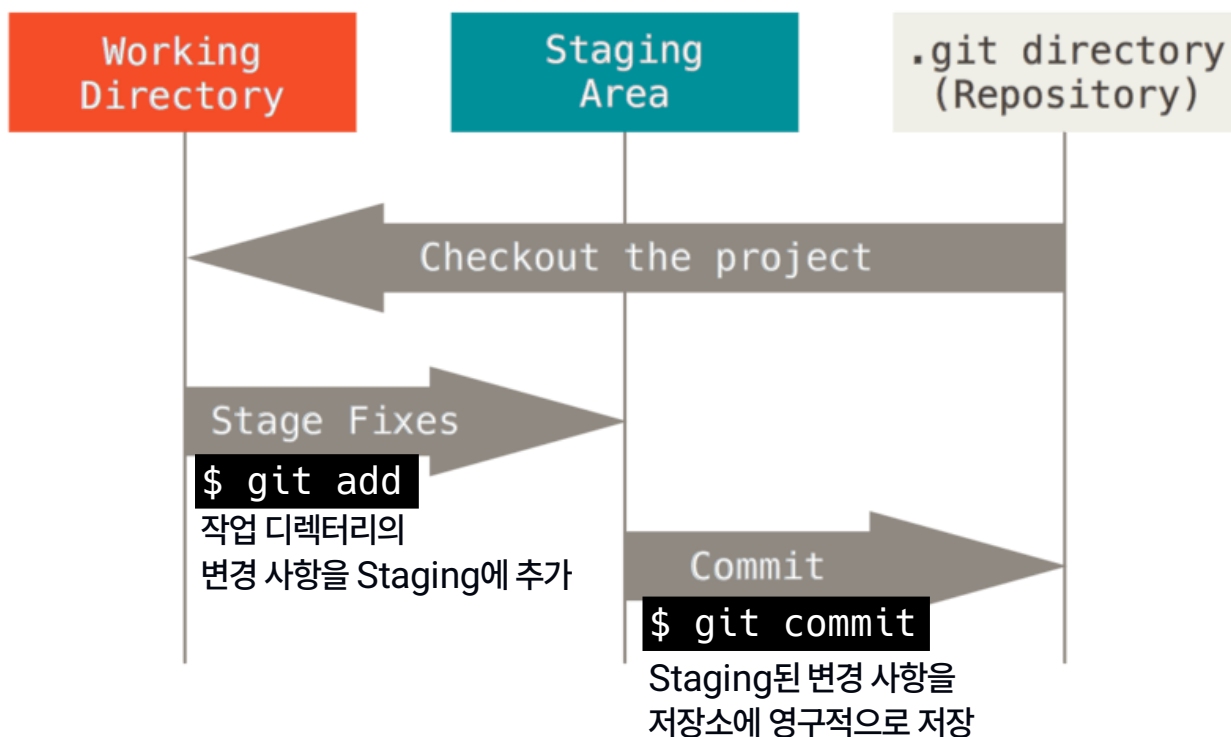
- 분산 관리식
- 변경 시점마다 파일 전체를 **Snapshot**으로 저장
- 변경이 없는 파일은 이전 버전 참조
- 어떤 시점이든 빠르게 이동 가능



Git의 3가지 주요 영역

- Working Directory / Staging Area / Repository

▼ Git의 3가지 주요 영역



Working Directory

- 현재 작업 중인 파일이 위치하는 공간
- Git이 관리하지 않는 파일도 포함

Staging Area

- `$ git add` 명령어로 변경 사항을 임시 저장
- Commit할 파일을 선택하는 단계

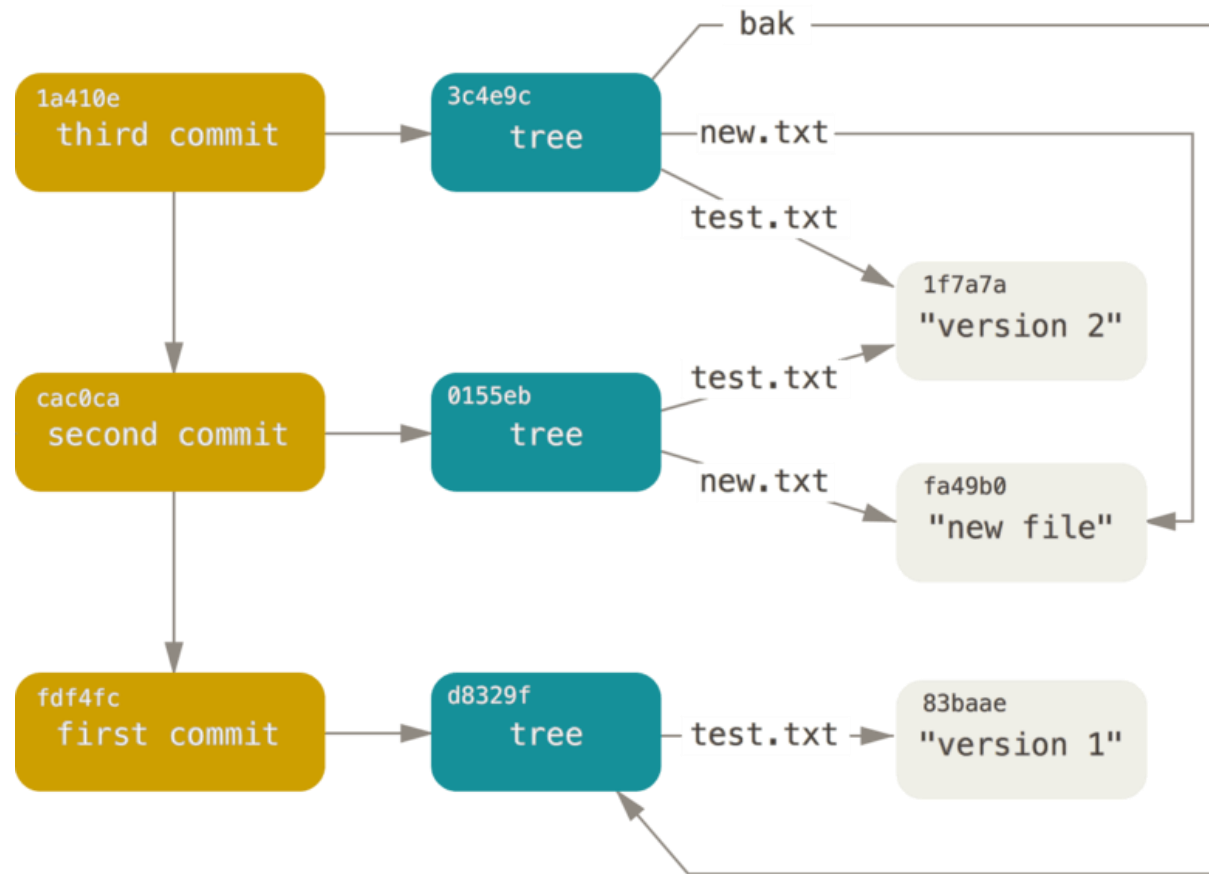
Repository

- `$ git commit` 로 변경 사항을 확정 저장
- SHA-1 해시 값과 함께 저장소에 영구 저장

Git의 Object 모델

- Blob / Tree / Commit

▼ Git 저장소 내의 모든 개체



Blob

- 파일의 내용 자체를 저장하는 객체
- 파일이 변경될 때마다 새로운 Blob 객체가 생성됨
- SHA-1 해시 값으로 파일을 관리

Tree

- 디렉터리와 파일 구조를 관리하는 객체
- 여러 개의 Blob 또는 다른 Tree 객체를 포함 가능
- 각 파일의 이름과 Blob 객체를 연결하는 역할

Commit

- 특정 시점의 프로젝트 상태를 저장하는 객체
- Tree 객체 + 이전 Commit 객체(부모) + Commit 메시지 + 작성자 정보 포함
- Tree와 Blob을 참조하는 해시 값 포함
- Branch를 이동할 때 기준이 되는 객체

Git - Commit

`$ git commit` : Staging Area에 있는 **변경 사항**을 **Repository**에 영구적으로 **기록**하는 작업 수행

1. Commit할 파일을 Staging Area에서 가져오기

- `$ git add` 로 추가된 파일을 대상으로만 작업
(Staging Area에 있는 파일들만 Commit 가능)

```
$ git status # 현재 Staging Area에 있는 파일 확인
$ git ls-files --stage # Staging Area의 Blob 객체 확인
```

2. Blob과 Tree 생성

- 변경된 파일의 내용을 SHA-1 해시로 변환하여 Blob 객체 생성
- 파일이 변경되지 않았으면 기존 해시 값 유지 (중복 데이터 저장 방지)

```
$ git hash-object -w test.txt # test.txt 파일을 Blob 객체로 저장
$ git write-tree # 현재 Staging Area의 트리 객체 생성
$ git cat-file -p $(git write-tree) # 생성된 트리 객체 확인
```

3. 새로운 Commit 객체 생성

- Commit 객체에 포함되는 정보:
 - 부모 Commit(이전 Commit 과의 연결), Tree 객체 (파일 및 디렉토리 구조), 작성자 정보 (이름, 이메일), Commit 메시지

```
$ git commit -m "커밋 메시지" # 새로운 Commit 객체 생성
$ git log --oneline # 커밋 로그 확인
```

4. HEAD 업데이트

- 현재 Branch가 가리키는 최신 Commit을 새로운 Commit으로 업데이트 (HEAD가 최신 Commit을 가리키도록)

```
$ git rev-parse HEAD # 현재 HEAD가 가리키는 커밋 확인
$ git log --oneline # 최신 커밋 로그 확인
```

Git - 파일 변경 사항 추적

Git은 파일의 변경 사항을 자동으로 감지하고, 이를 3가지 상태 중 하나로 분류한다.

상태	의미	예시
Untracked	Git이 관리하지 않는 파일	새로 추가한 파일 (ex: new_file.txt)
Modified	기존 파일이 수정됨	기존 코드 변경 (ex: index.html 수정)
Staged	Commit할 준비 완료	<code>git add</code> 로 Staging Area에 추가

◆ 파일이 추가되었을 때

1. `new_file.txt` 를 새로 생성하면 → **Untracked** 상태
2. `git add new_file.txt` 실행 → **Staged** 상태로 이동
3. `git commit -m "Add new file"` 실행 → Committed 상태

◆ 파일을 수정했을 때

1. 기존 파일을 수정하면 → **Modified** 상태
2. `git add` 실행 시 **Staged** 상태로 이동
3. `git commit` 실행 시 Committed 상태로 저장

```
# 1. 새로운 파일 생성 (Untracked 상태)
$echo "Hello, Git!" > new_file.txt
$git status
# 실행 결과:
# On branch main
# Untracked files:
#   (use "git add <file>..." to include in what will be committed)
#       new_file.txt

# 2. Staging Area에 추가 (Staged 상태)
$git add new_file.txt
$git status
# 실행 결과:
# On branch main
# Changes to be committed:
#   (use "git restore --staged <file>..." to unstage)
#       new file:   new_file.txt

# 3. 커밋 수행 (Committed 상태)
$git commit -m "Add new file"
$git status
# 실행 결과:
# On branch main
# nothing to commit, working tree clean
```

▲ 파일 추가 예시

Git - HEAD

HEAD

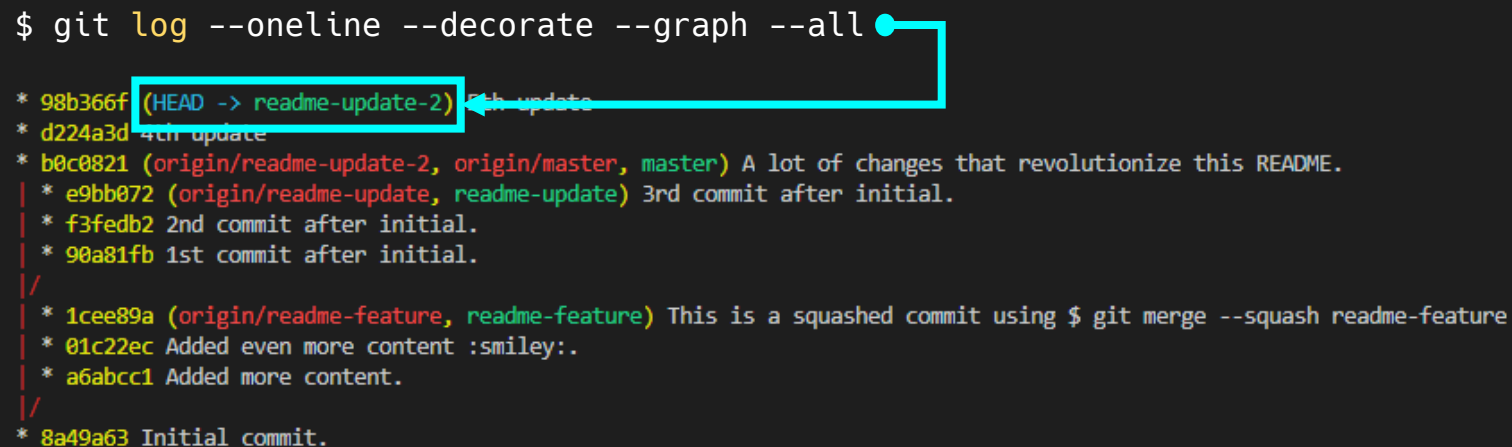
- 현재 Checkout된 Commit 또는 Branch를 가리키는 Git의 포인터
- `git checkout <branch>` 실행 시, HEAD가 해당 Branch를 가리키게 됨

HEAD가 이동하는 방식

- `git commit` 실행 시, HEAD가 새로운 Commit을 가리킴
- `git checkout <branch>` 실행 시, HEAD가 해당 Branch로 변경됨

▼ HEAD를 포함한 모든 Branch의 Commit 이력 확인

```
$ git log --oneline --decorate --graph --all
```



```
* 98b366f (HEAD -> readme-update-2) 5th update
* d224a3d 4th update
* b0c0821 (origin/readme-update-2, origin/master, master) A lot of changes that revolutionize this README.
* e9bb072 (origin/readme-update, readme-update) 3rd commit after initial.
* f3fedb2 2nd commit after initial.
* 90a81fb 1st commit after initial.
/
* 1cee89a (origin/readme-feature, readme-feature) This is a squashed commit using $ git merge --squash readme-feature
* 01c22ec Added even more content :smiley:.
* a6abcc1 Added more content.
/
* 8a49a63 Initial commit.
```

사진출처: <https://prsantos.com/posts/2021-03-07-git-log-graph>

Detached HEAD

: HEAD가 특정 Branch가 아닌,
특정 Commit을 직접 참조하고 있는 상태

`git checkout <commit ID>`

상태에서 새로운 commit을 만들면, 해당 commit은 branch와 연결되지 않아
고립됨.

이후 branch를 변경하면, 새 commit을 다시 찾기 어렵거나 잃어버릴 가능성이
높음.

1. Attached HEAD state (정상적인 상태)

HEAD -> branch -> commit

2. Detached HEAD state (특정 commit을 직접 가리킴)

HEAD -> commit

usually, you check out a branch:
`$ git checkout master`



...and not a specific commit:
`$ git checkout a05ef02`

해결 방법

1. 기존 Branch(Main)로 이동

```
$ git checkout main # 기존 브랜치(main)로 이동
```

2. 새로운 Branch를 생성하여 Commit 연결 유지

```
$ git checkout -b new-branch # 새로운 브랜치 생성 후 HEAD를 해당 브랜치로 이동
$ git commit -m "Save detached state"
$ git checkout main # 다른 브랜치로 이동해도 커밋이 유지됨
```

Git - Log

`$ git log` : 현재 Branch의 Commit History를 역순으로 출력하는 Git 명령어

```
commit 110f334a69a9fcb5320cb92e045a988aaa633f92 (HEAD -> master)
Author: Ankit Mahajan <ankitmahajan852@gmail.com>
Date:   Sun Dec 26 16:48:59 2021 +0530

    Second Commit

commit a38502e3b656b84f47b58eb1ada5490703ab269b (origin/master)
Author: Ankit Mahajan <ankitmahajan852@gmail.com>
Date:   Sat Dec 25 20:26:20 2021 +0530

    Initial Commit
```

▲ 실행 결과 예시

명령어	기능
<code>git log --oneline</code>	한 줄 요약 형태로 Commit History 표시
<code>git log --graph</code>	Branch와 Merge 관계를 그래픽으로 표현
<code>git log --decorate</code>	HEAD, Branch 이름을 표시
<code>git log --author="Kim"</code>	특정 작성자의 Commit만 검색

▲ git log 옵션 정리

`$ git log --oneline --graph --all` : 모든 Branch의 Commit을 한 줄 요약 및 그래프 형식으로 출력하는 Git 명령어

```
* 98b366f (HEAD -> readme-update-2) 5th update
* d224a3d 4th update
* b0c0821 (origin/readme-update-2, origin/master, master) A lot of changes that revolutionize this README.
| * e9bb072 (origin/readme-update, readme-update) 3rd commit after initial.
| * f3fedb2 2nd commit after initial.
| * 90a81fb 1st commit after initial.
|/
| * 1cee89a (origin/readme-feature, readme-feature) This is a squashed commit using $ git merge --squash readme-feature
| * 01c22ec Added even more content :smiley:.
| * a6abcc1 Added more content.
|/
* 8a49a63 Initial commit.
```

▲ 실행 결과 예시

Git - Refs

◆ Git의 주요 참조 개념

- HEAD : 현재 Checkout된 Branch 또는 Commit
- Branch : 특정 개발 흐름을 가리키는 포인터
- Tag : 특정 시점(릴리즈 버전 등)을 고정하는 참조

참조	역할	특징
HEAD	현재 Commit을 가리키는 포인터	이동 가능 (git checkout)
Branch	여러 Commit을 관리하는 참조	git branch로 생성 및 삭제 가능
Tag	특정 Commit을 고정하는 참조	git tag -a v1.0 -m "Version 1.0"

◆ git show로 참조 정보 확인하기

```
$ git show HEAD # HEAD가 가리키는 커밋 확인
$ git show main # main 브랜치의 최신 커밋 확인
$ git show v1.0 # 태그 v1.0이 가리키는 커밋 확인
```

Git - Merge 방식 비교

사진출처: <https://kotlinworld.com/277>

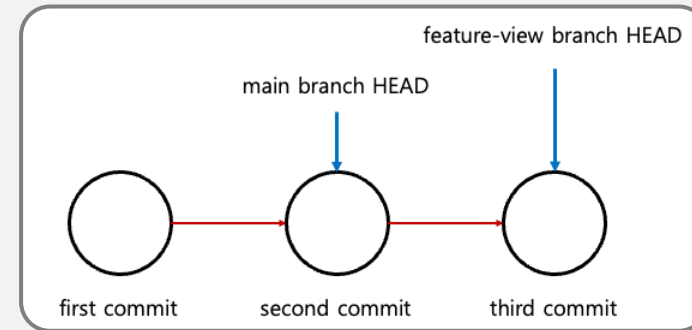
Fast-forward Merge

: Branch가 분기되지 않고,
한쪽 Branch가 다른 Branch의 최신 Commit을 그대로 포함하는 직선적 흐름
(단순한 병합에 사용)

- History가 깨끗하게 유지됨
- 새로운 병합 Commit이 생성되지 않음

한계

- 중간에 다른 Commit이 있을 경우 동작하지 않음
- 동일한 파일을 수정한 Commit이 여러 Branch에 존재하면 충돌발생



- HEAD 포인터를 앞으로 이동
- HEAD 포인터를 앞으로 이동하여 병합 (새로운 Commit 생성 없음)

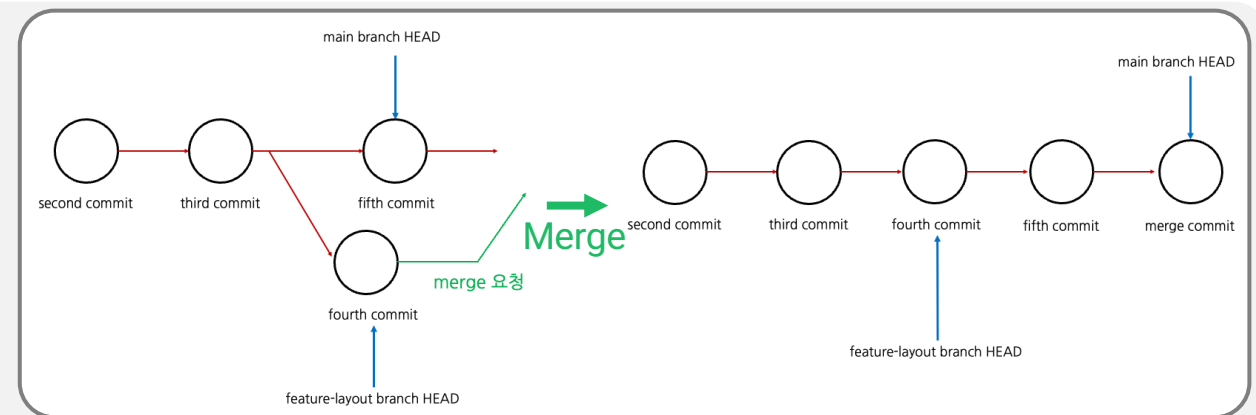
```
$ git checkout -b feature # feature 브랜치 생성 및 이동
$ git commit -m "Add feature" # 새로운 커밋 생성 (feature 브랜치에서)
$ git checkout main # main 브랜치로 이동
$ git merge feature # Fast-forward Merge 수행
```

Three-way Merge

: 병합하려는 두 Branch가 서로 다른 Commit을 가지고 있어 공통 조상이 존재
(분기된 Branch 병합에 사용)

- Branch가 독립적으로 변경된 후 병합될 때 사용됨
- 새로운 병합 Commit이 생성되어 History에 병합 흔적이 남음

```
$ git checkout main # main 브랜치로 이동
$ git merge feature # 병합 커밋 생성
```



- 3개의 Commit 비교로 새 병합 Commit 생성
- 공통 조상과 두 Branch의 최신 Commit을 비교하여 새로운 Merge Commit 생성

Git - Merge Conflict

충돌 발생 원인

같은 파일의 같은 위치를 여러 사람이 수정했을 때 충돌 발생

예시) 각 Branch에서 README.md의 동일한 부분을 다르게 수정

```
Main branch >>
Welcome to my project!

Feature branch >>
Welcome to our awesome project!
```



```
$ git merge feature
CONFLICT (content): Merge conflict in README.md
```

→ Main Branch와 Feature Branch가 같은 줄을 다르게 수정했기 때문에 **충돌발생**

충돌 해결 방법

충돌이 발생하면 Git은 해당 파일 내에서 충돌 부분을 자동으로 표시

예시) 충돌 발생 후 README.md 파일 내용

```
<<<<<< HEAD
Welcome to my project!
=====
Welcome to our awesome project!
>>>>>> feature
```

현재(HEAD, 즉 main branch)의 내용

변경된 내용을 구분하는 기준선

feature branch의 내용



해결 단계)

1. 충돌 파일을 편집하여 최종 내용으로 수정
2. 충돌 마커 제거 (<<<, ==, >>>)
3. 수정된 파일을 Git에 추가 → **git add README.md**
4. 충돌 해결 완료 후 Commit → **git commit**

Git - 변경 사항을 되돌리는 3가지 방법

명령어	역할	특징
git reset	Commit History 수정	Local History 변경 (주의)
git revert	새 되돌리기 Commit 생성	History를 보존하며 변경 취소
git checkout	Branch/Commit 이동	작업 디렉토리 내용 변경

Reset

```
$ git reset --hard HEAD~1
```

- 특정 Commit으로 이동해 변경 사항을 되돌림

--soft : Commit만 되돌림
--mixed : Commit + Staging 되돌림
--hard : ⚠ 모든 변경 삭제

Revert

```
$ git revert HEAD
```

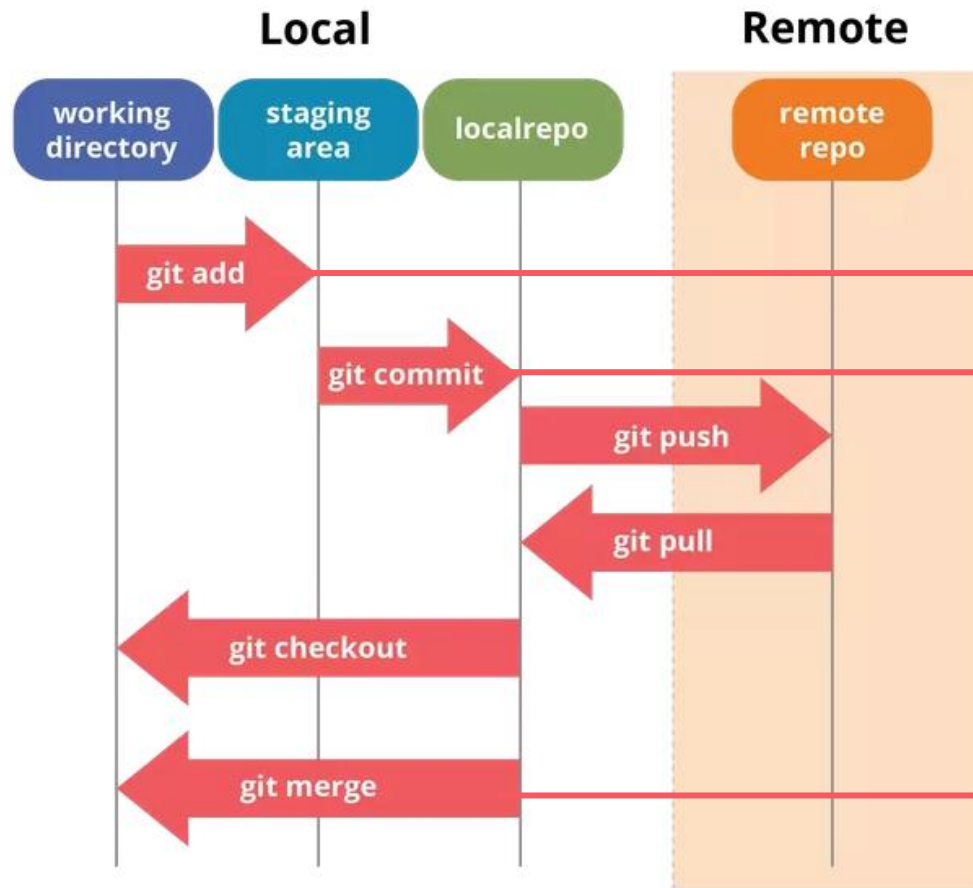
- History를 유지하면서 변경 취소
- 원격 저장소 Commit 취소에 안전
- 협업 시 권장됨

Checkout

```
$ git checkout feature  
$ git checkout a1b2c3d
```

- Branch 간 이동
- 특정 Commit으로 이동 가능
- Commit 이동 시 Detached HEAD

Git - 내부 동작 요약



◆ Git의 전체 동작 과정

1. 파일 추가 → **Untracked** 상태
2. `$ git add` → **Staged** 상태
3. `$ git commit` → Repository에 저장
4. Branch 작업 → `$ git merge` 로 병합
5. 필요시 되돌리기 → Reset / Revert

◆ Git 용어 및 핵심 개념

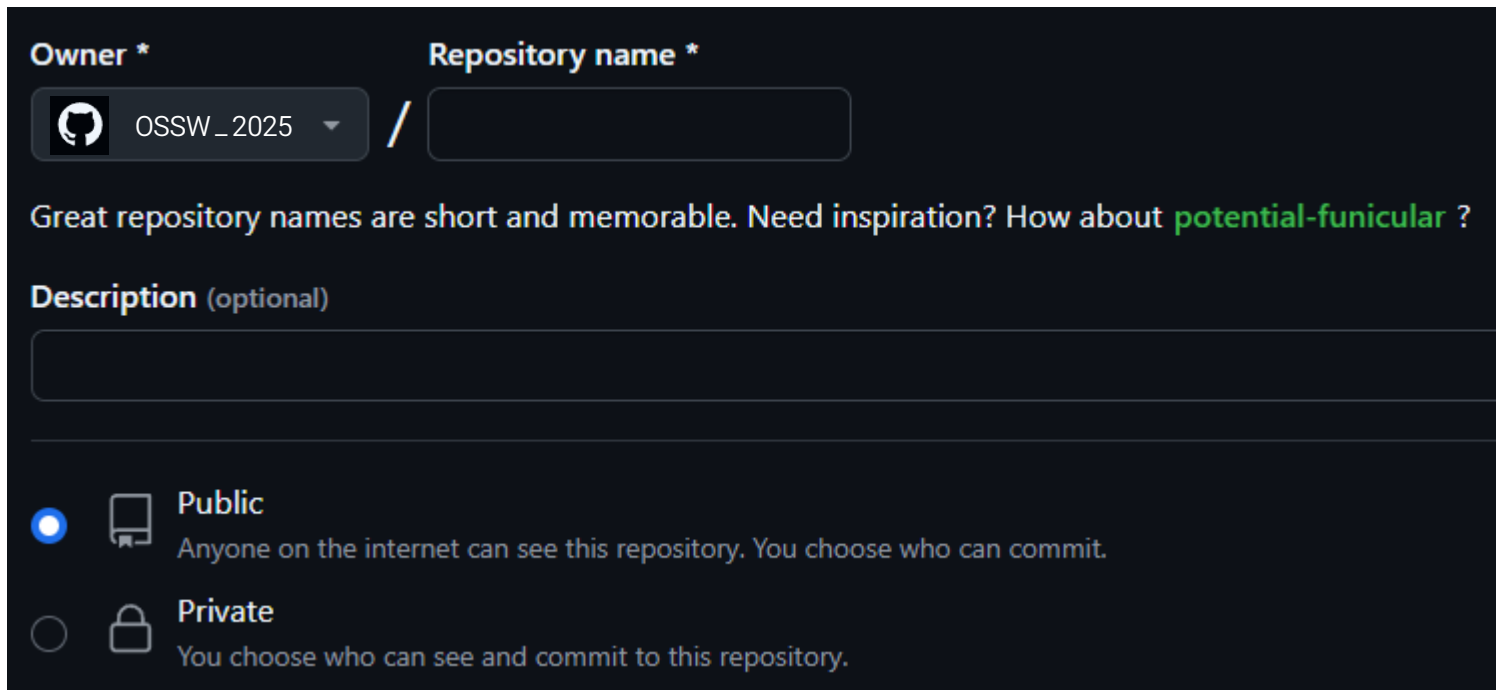
- Snapshot 기반 : 파일 전체를 Snapshot으로 저장
- SHA-1 해시 : 모든 객체는 해시로 식별
- 3가지 영역 : Working, Staging, Repository
- 3가지 객체 : Blob, Tree, Commit
- Branch : 독립적인 개발 흐름
- Merge : Branch 통합 방법
- 원격 저장소 : GitHub 기반 협업
- Pull Request : 코드 리뷰와 통합

03 GitHub에서 협업하는 방법

GitHub - Repository

Repository : 프로젝트 파일을 관리하고 협업할 수 있는 **공간**

- 프로젝트의 코드, 파일, 문서, 변경 이력(Commit 기록)이 저장됨
- GitHub에서 협업의 기본 단위



The screenshot shows the GitHub repository creation interface. At the top, there are two input fields: 'Owner *' with a dropdown menu showing 'OSSW_2025' and a GitHub icon, and 'Repository name *' with an empty text box. Below these fields is a hint: 'Great repository names are short and memorable. Need inspiration? How about potential-funicular?'. Underneath is a 'Description (optional)' text area. At the bottom, there are two radio button options: 'Public' (selected) with a lock icon and the text 'Anyone on the internet can see this repository. You choose who can commit.', and 'Private' with an unlocked lock icon and the text 'You choose who can see and commit to this repository.'

Public 저장소

- 누구나 볼 수 있고, Fork하여 기여 가능

Private 저장소

- 소유자와 권한이 부여된 사용자만 접근 가능

▲ GitHub에서 Repository를 만드는 화면

GitHub - Branch

Branch : 독립적인 개발 흐름을 유지하고 병합을 통해 통합할 수 있는 기능

Local Branch

- 개발자가 자신의 컴퓨터에서 직접 관리하는 Branch

```
$ git branch
* main
feature-branch
```

◀ Local Branch 사용

원격 Branch

- GitHub에 업로드 된 Branch

```
$ git branch -r
origin/main
origin/feature-branch
```

◀ 원격 Branch 사용

```
$ git checkout -b feature # 새 브랜치 생성 및 이동
$ git push -u origin feature # 브랜치 원격 저장소에 업로드
$ git checkout main # 기존 브랜치로 돌아가기
```

◀ Branch 관리 명령어

Pull Request (PR)

Pull Request(PR) : 개발자가 코드 변경 사항을 병합하기 전에 검토 받을 수 있도록 하는 GitHub의 협업 기능

- 새로운 기능을 개발한 후, **Main Branch에 Merge하기 전 코드 리뷰를 요청하는 과정**

새로운 기능 개발

- 작업 별 Branch를 생성하여 독립적으로 개발 진행

PR 생성

- 개발이 완료되면 기존 Main Branch로 병합을 요청하는 PR을 생성

코드 검토 및 피드백

- 팀원들이 코드 변경 사항을 검토하고 피드백을 제공
- 코드 스타일, 버그, 최적화 가능성 등을 논의

수정 및 반영

- 리뷰 내용을 반영하여 코드 수정 후 다시 푸시

Merge 및 Branch 정리

- 코드 리뷰가 완료되면 PR을 승인한 후 Main Branch로 병합
- 불필요한 Feature Branch는 삭제하여 관리

※ PR 흐름도

1. Fork 생성

→

2. 변경 작업

→

3. PR 생성

→

4. 코드 리뷰

→

5. Merge

GitHub - Pull Request (PR)

❖ Branch를 Push하고 PR을 생성하는 과정

1 Local에서 branch에서 작업 후 push

```
$ git checkout -b feature (Branch 이름)
# 파일 수정 작업... (변경사항)
$ git commit -m "Add feature"
$ git push origin feature
```

2 GitHub에서 PR 생성

1 Repository 이동

2 Pull requests

3 새로운 PR 생성

4 Branch 비교

5 PR message 작성

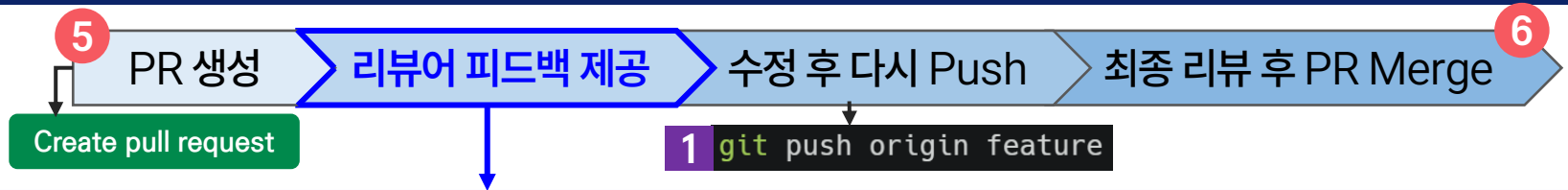
6 최종 PR 생성

Compare & pull request

GitHub - Pull Request (PR)

❖ PR 과정에서의 코드 리뷰

: 팀원 간 코드 품질을 높이고, 버그를 방지하는 과정



fix pycache folder being added instead of encoding info files #152

1 PR 페이지에서 Files changed 탭 클릭

2 변경된 코드에서 특정 줄을 선택하여 코멘트 작성

Files changed 1

0 / 1 files viewed

tiktoken/registry.py

27 # - it's a separate top-level package because namespace subpackages of non-namespace packages don't quite do what you want with editable installs

28 plugin_mods = pkgutil.iter_modules(tiktoken_ext.__path__, tiktoken_ext.__name__ + ".")

29 filtered_plugin_mods = [(loader, mod_name, ispkg) for loader, mod_name, ispkg in plugin_mods if not mod_name.endswith('__pycache__')]

30

Write Preview

Leave a comment

Cancel Add single comment Start a review

Ask Copilot Review in codespace Review changes

Finish your review

Write Preview

Leave a comment

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

3 리뷰 유형 선택

4 Submit review

Comment Submit general feedback without explicit approval.

Approve Submit feedback approving these changes.

Request changes Submit feedback suggesting changes.

Comment

- 특정 코드 라인에 대한 피드백을 남길 때 사용
- 의견만 제시, 승인/거부 없음

Approve

- 코드가 정상적이고 변경 사항이 필요 없을 때 승인
- PR 병합을 위한 신호

Request Changes

- 코드에 문제가 있거나 개선이 필요할 때 변경 요청
- PR 작성자의 수정 필요

GitHub - Pull Request (PR)

❖ PR 협업에서 발생하는 문제와 해결 방법

PR 협업에서 흔히 발생하는 문제

1. 코드 스타일 충돌 (Code Style Conflicts)

- 각 개발자가 코드 스타일 규칙을 다르게 적용할 경우, 팀 내 코드 일관성이 떨어짐
- Prettier, ESLint 같은 자동 Formatter가 없다면 스타일 혼선을 초래

* Prettier : 코드 구현과는 관련 없이, 일관된 텍스트 작성을 도와주는 도구

* ESLint : 코드의 품질을 보장하도록 도와주는 도구

2. 병합 순서 문제 (Merge Order Issues)

- 여러 개의 PR이 동시에 열린 경우, 어떤 PR을 먼저 Merge할지 결정해야 함
- 특정 PR이 먼저 병합되면 이후 PR에서 충돌이 발생할 가능성이 높음

3. PR이 너무 커서 리뷰가 어려운 경우

- 한 번에 너무 많은 변경 사항이 포함된 PR은 리뷰어가 검토하기 어려움
- 코드 리뷰 속도가 느려지고, 피드백을 반영하는 시간이 길어짐

해결 방법

1. 코드 스타일 통일하기

- 자동 코드 Formatter 적용 → `npx prettier --write "**/*.js"`

2. 병합 순서 문제 해결

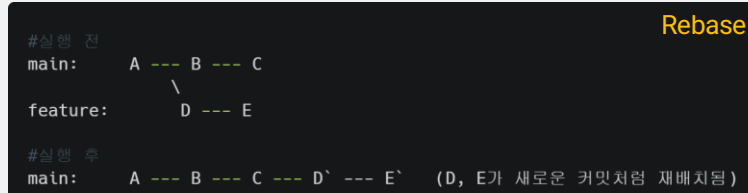
- Branch 병합 전략 선택: Rebase* vs Merge Commit*
- PR을 순차적으로 병합하거나, 중요한 PR을 우선 처리

3. 작은 단위로 PR 생성 (Atomic PR)

- 기능별로 작은 PR을 제출하여 리뷰 부담 줄이기
- 100줄 이하 변경 사항이 이상적

* Merge Commit : 두 Branch를 하나로 합치는 병합 Commit

* Rebase : 다른 Branch의 최신 Commit에 내 Branch의 Commit을 다시 적용



GitHub - Issues

GitHub Issues : 프로젝트에서 발생한 문제나 새로운 기능 요청을 체계적으로 관리하는 기능

Assignee 지정

Assignees 특정 팀원에게 이슈를 할당하여 책임감을 부여

 OSSW_2025

Labels

bug **enhancement** **question**

Label 목록

Type

No type

Projects

No projects

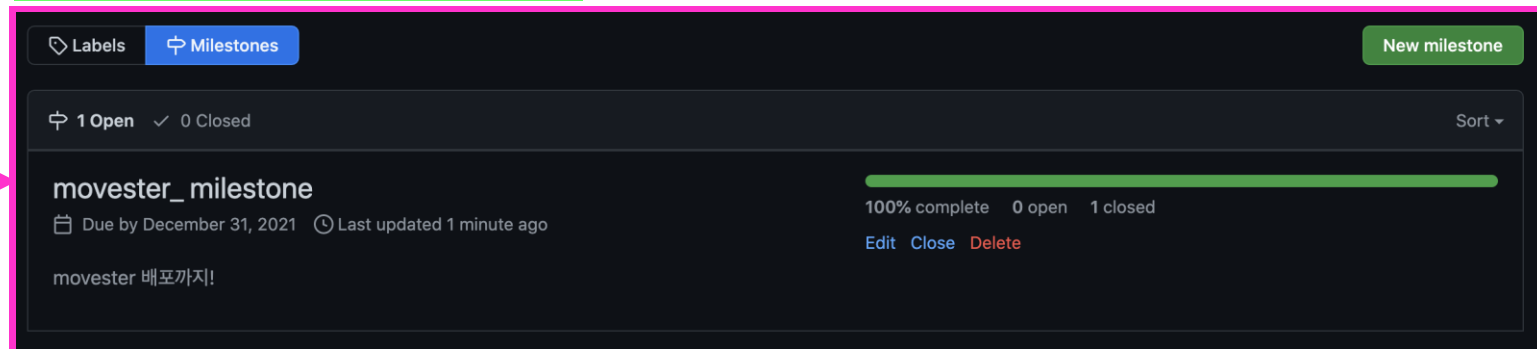
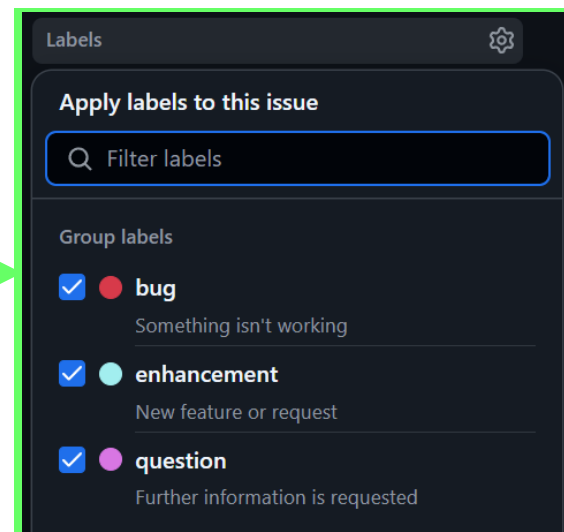
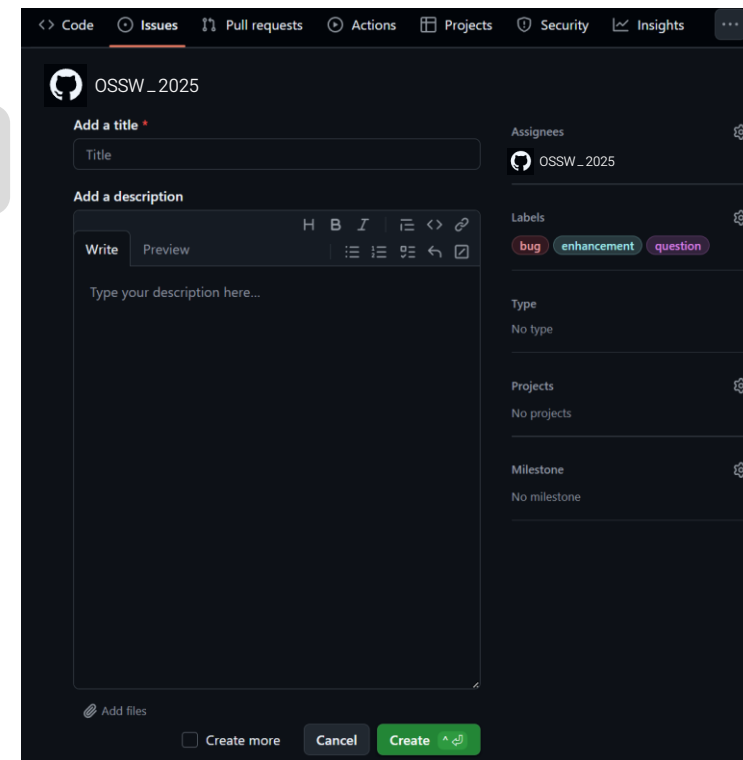
Milestone 설정

특정 기간 내 완료해야 할
작업 그룹 설정

Milestone

No milestone

GitHub Issues
생성 예제 ▶



GitHub - Discussions

GitHub Discussions

- 코드 리뷰와 무관한 일반적인 질문, 제안, 공지를 올릴 수 있는 공간
- 프로젝트 참여자 간 토론 활성화 가능
- Issues보다 자유로운 논의 공간 제공

활성화 경로

- 1) Repository Settings → Features → Discussions 활성화
- 2) Start a new discussion 클릭하여 새로운 주제 등록

Features

☐ Discussions

Discussions is the space for your community to have conversations, ask questions and post answers without opening issues.

Get started with Discussions

Engage your community by having discussions right in your repository, where your community already lives

Set up discussions

▲ Discussions 활성화

Discussions 활용 사례

- FAQ 작성
반복되는 질문을 정리하여 팀원 간 공유
- 기능 제안 공간
새로운 아이디어에 대한 피드백을 주고받음
- 커뮤니티 공지 게시판
프로젝트 업데이트 및 방향 공유

오픈소스 프로젝트 PR 사례

❖ 글로벌 기업의 오픈소스 프로젝트 PR 사례



Google의 PR 사례 : TensorFlow 프로젝트

- PR을 통한 모델 개선 및 버그 수정
- 내부 팀원의 검토
- 자동 테스트 실행
- 승인 후 병합



Facebook의 PR 사례 : React 프로젝트

- PR 제출자가 관련 Issues를 연결
- 자동 CI 테스트 통과 후 병합 가능
- 라이선스 확인 절차 필수



OpenAI의 PR 사례 : Whisper 프로젝트

- 기여자가 PR을 제출하면 OpenAI 연구팀이 리뷰
- 일정 수준의 코드 품질을 요구
- 문서 개선 PR도 적극 수용

GitHub - Insights

Insights

: Repository의 활동 통계, 기여 내역, 코드 빈도 등을 시각적으로 분석하는 기능



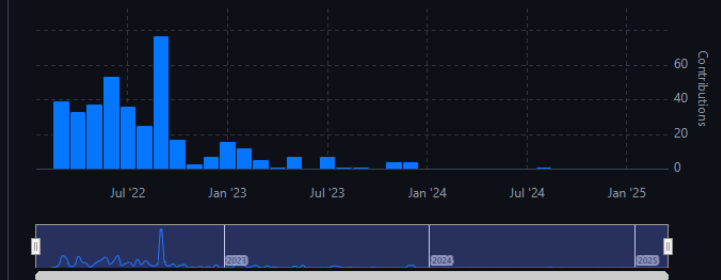
기여자 분석

Contributors

Contributions per week to main, excluding merge commits

Commits over time

Weekly from 2022년 2월 27일 to 2025년 3월 2일



Code frequency over the history of salesforce/LAVIS

Code frequency

Additions and deletions per week



코드 변경량

출처: <https://github.com/salesforce/LAVIS>

04 GitHub의 고급 기능

GitHub Actions 개요 (CI/CD)

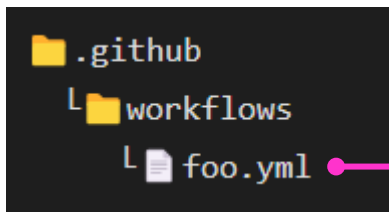
GitHub Actions : CI/CD를 구현할 수 있도록 하는 자동화 도구

◆ GitHub Actions의 역할

- 코드가 변경될 때마다 자동으로 빌드 및 테스트 실행
- 특정 이벤트(Push, PR, Merge 등)가 발생할 때 자동화된 작업 수행
- Docker 컨테이너 및 클라우드 배포 가능

◆ GitHub Actions를 활용한 CI/CD 자동화: (Node.js 프로젝트 예제)

1. GitHub Actions 워크플로우 파일 생성



- `.github/workflows/ci.yml` 폴더에 워크플로우 파일(yml 형식) 파일 추가
- Push 또는 Pull Request 발생 시 CI 실행

```
name: Node.js CI

on: [push, pull_request]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: 코드 체크아웃
        uses: actions/checkout@v2

      - name: Node.js 설정
        uses: actions/setup-node@v2
        with:
          node-version: 16

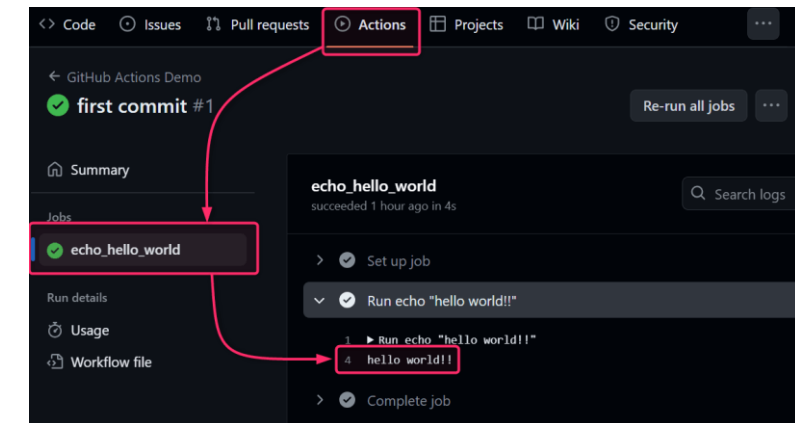
      - name: 의존성 설치 및 테스트 실행
        run: |
          npm install
          npm test
```

CI / CD

- Continuous Integration (CI)
: 개발자가 코드를 변경할 때마다 **자동으로 테스트를 실행**하여 코드 품질을 유지
- Continuous Deployment (CD)
: 코드가 검증된 후 **자동으로 배포**

2. GitHub에서 Actions 실행

- Actions 탭 클릭 → CI/CD 실행 상태 확인



▶ GitHub에서의 Actions Log

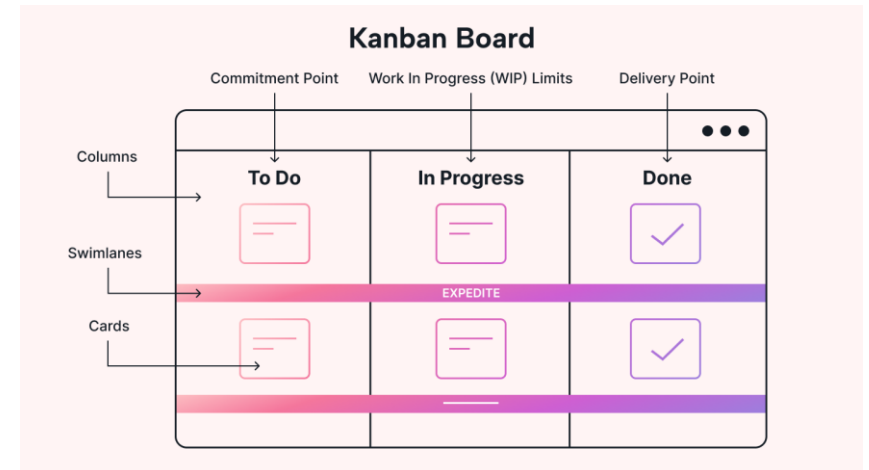
GitHub Projects 관리 - Kanban Board

Kanban 보드 : 소프트웨어 개발 및 작업 흐름 관리를 위해 카드와 컬럼을 사용하여 작업 상태를 시각적으로 표현하는 프로젝트 관리 도구

- "To Do → In Progress → Done" 방식으로 진행 관리

◆ GitHub Projects의 주요 기능

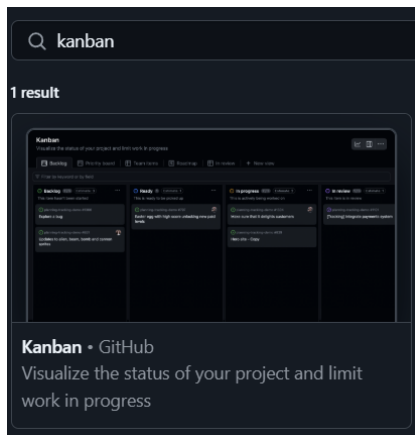
- Issue, Pull Request를 보드에 연결하여 진행 상황을 시각적으로 표시
- 자동화 규칙을 설정하여 이슈가 해결되면 자동 이동



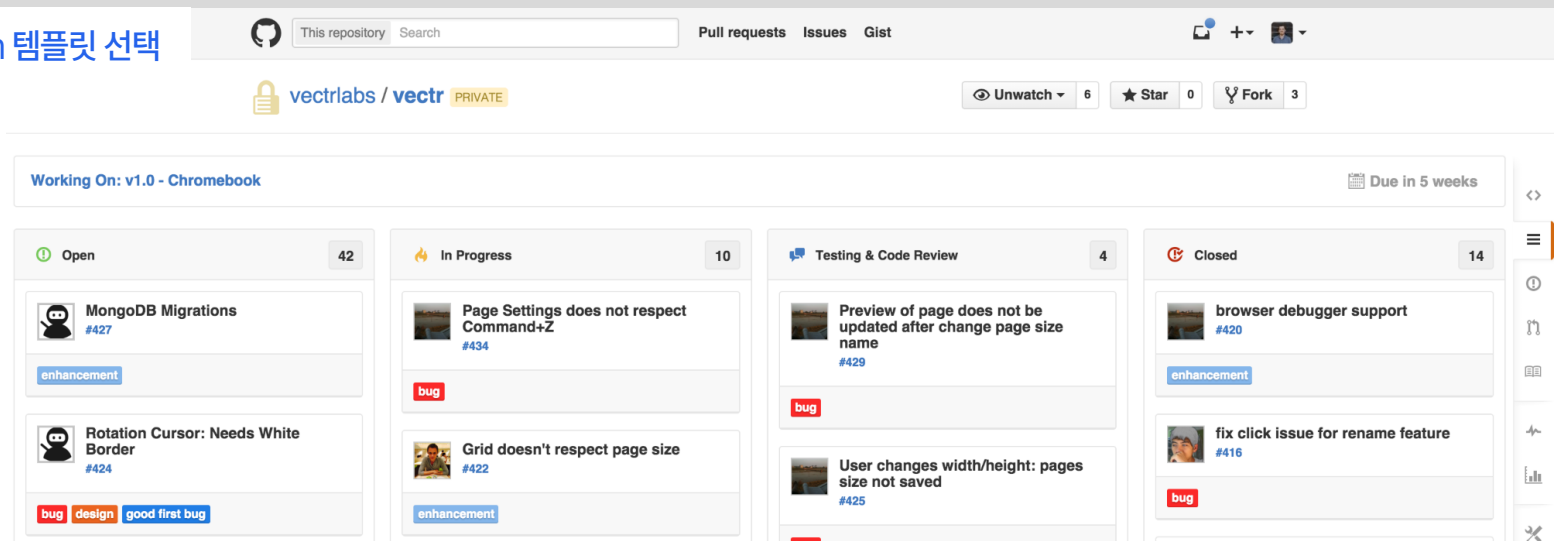
▲ Kanban Board 구조

◆ GitHub Projects 설정 방법

- Repository → Projects → New Project → [Kanban](#) 템플릿 선택



▲ GitHub에서의 Kanban Board



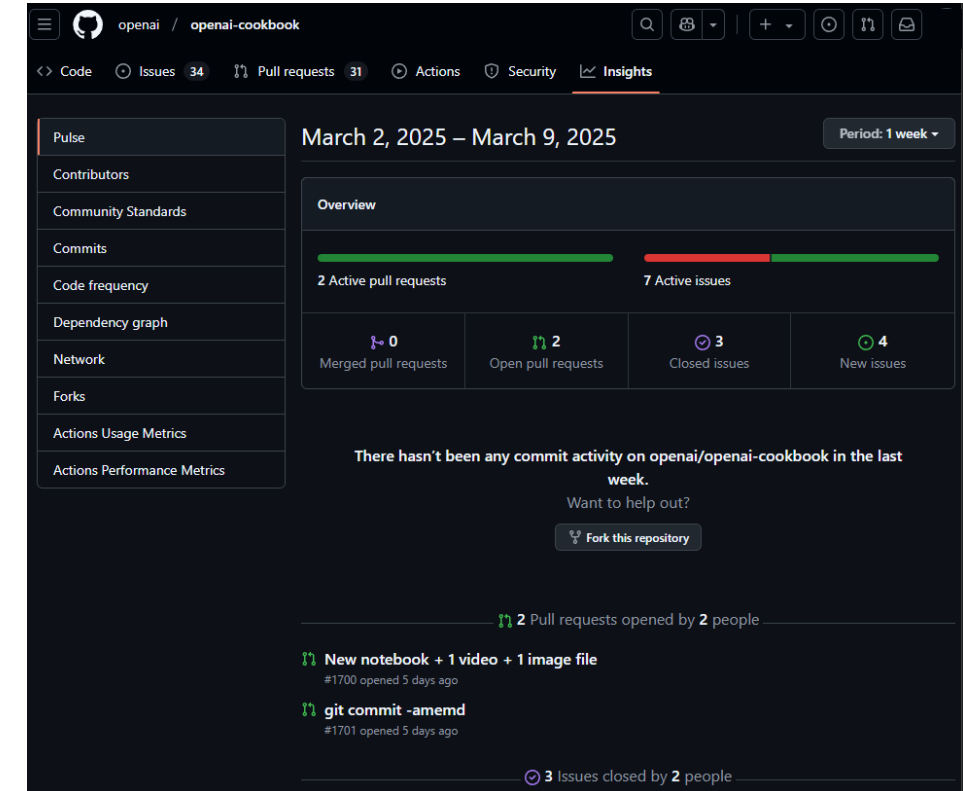
▲ GitHub에서의 Kanban Board 사용 예제

GitHub Projects 관리 - GitHub Insights

GitHub Insights : GitHub에서 프로젝트 활동, 코드 기여, 이슈 및 Pull Request 데이터를 분석하여 시각적으로 제공하는 기능



▲ GitHub Insights (p30 참고)



▲ GitHub Insights 예시

GitHub에서 제공하는 협업 설정

CODEOWNERS

: 특정 파일/디렉터리에 대해 자동으로 리뷰어를 지정하는 기능

활용 예시

- 문서 파일 변경 시 : @documentation-team 자동 리뷰 지정
- 백엔드 코드 변경 시 : @backend-devs 자동 리뷰 지정

```
# 특정 파일 담당자 지정
*.md @documentation-team
src/api/* @backend-devs
```

◀ 자동 리뷰어 지정

Issue 템플릿

: Issue 작성 시 미리 정의된 형식 제공

```
---
name: 버그 리포트
about: 프로젝트에서 발견된 버그를 보고해주세요
---

## 버그 설명
(버그가 발생한 상황을 자세히 설명해주세요.)

## 재현 방법
1. 특정 페이지로 이동
2. 버튼 클릭
3. 오류 발생

## 기대하는 동작
(정상적으로 동작해야 하는 방식 설명)
```

▲ .github/ISSUE_TEMPLATE/bug_report.md 예시

PR 템플릿

: PR 작성 시 일관된 형식을 유지하여 코드 리뷰를 효율화

```
## 변경 사항
- 기능 A 추가
- 버그 B 수정

## 변경 이유
(변경이 필요한 이유를 설명)

## 테스트 방법
- [ ] 로컬에서 테스트 완료
- [ ] CI 통과 확인
```

▲ .github/ISSUE_TEMPLATE/bug_report.md 예시

GitHub에서 Fork와 Clone의 차이

• Fork와 Clone 비교

개념	Fork	Clone
목적	오픈소스 프로젝트를 내 계정으로 복사하여 기여	Local에서 코드 작업 후, 원격 저장소와 연결
위치	GitHub 내에서 복사됨	개발자의 Local 환경에서 복사
연결 관계	원본 저장소와 연결되지만, 직접 변경은 불가능	원격 저장소와 직접 연결됨
사용 예시	외부 오픈소스 프로젝트에 기여	팀 프로젝트 참여, 개인 작업

• Fork 사용의 흐름

(오픈소스 프로젝트) → 내 GitHub 계정으로 Fork

→ Local로 Clone → 코드 수정 → PR 제출

• Fork를 이용한 오픈소스 기여 방법

1. 오픈소스 프로젝트 Fork하기



2. Fork한 저장소 Clone하기

```
$ git clone https://github.com/my-username/forked-repo.git
```

3. 변경 사항 반영 후 원본 저장소에 Pull Request(PR)

```
$ git checkout -b feature-branch
$ git commit -m "Add new feature"
$ git push origin feature-branch
```

GitHub의 최신 기능 (2024 기준)

❖ AI 코드 자동 완성

GitHub Copilot

- AI 기반 코드 자동 완성 기능 (OpenAI GPT 모델 기반)
- 주석을 입력하면 관련 코드 자동 생성
- VS Code, JetBrains, Neovim 등 다양한 에디터 지원



Copilot Chat (2024 업데이트)

- 코드 리뷰 중 AI ChatBot을 통해 Refactoring 추천, 오류 디버깅 지원
- 자연어로 질문하면 코드 수정 방법 제안



❖ AI 기반 코드 리뷰 기능

GitHub PR에서 자동 코드 리뷰

- GitHub PR에서 자동으로 코드 리뷰를 수행하는 AI 기능
- 보안 취약점 및 코드 스타일 오류를 검출
- 리뷰어가 직접 코드 분석하는 시간을 줄여줌

GitHub Codespaces 업데이트

- 클라우드 기반 개발 환경 (VS Code 기반)
- Local 환경 설정 없이 즉시 개발 가능
- 최신 Docker 컨테이너 지원



기업에서의 GitHub 활용 사례

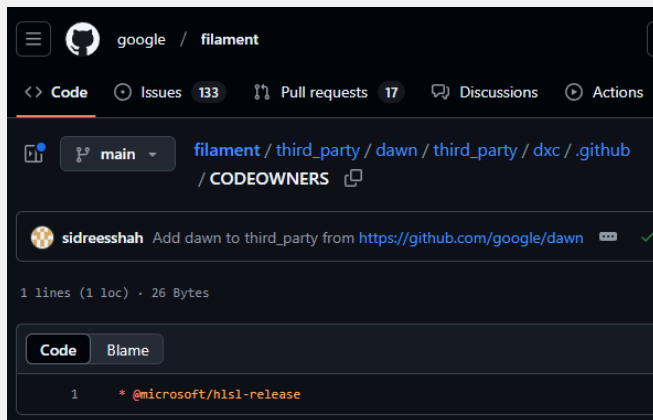
Google의 GitHub 활용 사례

TensorFlow 오픈소스 관리

- Pull Request 기반 코드 리뷰
- Google Bot을 이용한 자동 CI/CD 테스트

Google Open Source Program (OSPO)

- GitHub를 통해 사내 오픈소스 정책을 체계적으로 운영
- 프로젝트별 CODEOWNERS 지정



▲ Google이 project 내에서 Codeowners를 지정한 화면

Microsoft의 GitHub 활용 사례

VS Code, TypeScript 프로젝트 운영

- 모든 코드가 GitHub에서 공개적으로 관리
- 자동화된 GitHub Actions으로 CI/CD 실행

GitHub Copilot 개발

- GitHub과의 긴밀한 협력을 통해 AI 기반 코드 자동 완성 제공

```
name: CI/CD Pipeline
on: push
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - run: npm install
      - run: npm test
```

▲ Microsoft의 CI/CD 예제

Netflix의 GitHub 활용 사례

마이크로서비스 아키텍처 관리

- 여러 개의 독립적인 Repository에서 각각의 서비스 유지보수
- Netflix의 주요 프로젝트(Spinnaker, Chaos Monkey 등) 오픈소스 운영

GitHub Actions 기반 자동화

- 매일 수백 개의 배포 작업을 GitHub Actions로 자동화
- 테스트 완료 후, CD 파이프라인을 통해 자동 배포

```
name: Deploy to AWS
on: push
jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: AWS Login
        uses: aws-actions/configure-aws-credentials@v1
      - run: ./deploy.sh
```

▲ Netflix의 GitHub Actions 자동화 예제

05 마무리 및 Q&A

요약

◆ Git 핵심 개념 정리

1. 버전 관리 시스템 (VCS)의 필요성

- 코드 변경 사항을 기록하고, 과거 버전으로 되돌릴 수 있음
- 팀 프로젝트에서 코드 충돌을 방지하는 역할

2. Git의 내부 동작 원리

- Git은 Snapshot 방식으로 데이터를 저장 (SVN과 차이점)
- 주요 개념: **Working Directory, Staging Area, Repository**
- git commit 시, SHA-1 해시 기반으로 데이터를 관리

3. Branch 및 Merge

- Branch를 사용하면 독립적으로 기능 개발 가능
- **Fast-forward Merge vs Three-way Merge** 차이 이해

4. GitHub에서 협업하는 방식

- **Pull Request(PR)**를 활용한 코드 리뷰 & 협업
- **Issues, Projects, Discussions** 기능을 사용하여 프로젝트 관리

◆ GitHub 핵심 개념 정리

1. GitHub의 역할

- GitHub은 단순한 코드 저장소가 아니라, **협업과 자동화를 지원하는 플랫폼**
- **CI/CD (GitHub Actions), PR 리뷰, 프로젝트 관리** 기능 제공

2. GitHub의 최신 기능

- **GitHub Copilot & Copilot Chat**: AI 기반 코드 자동 완성
- **AI 코드 리뷰**: 보안 취약점 및 코드 개선 자동 추천

◆ GitHub 협업의 핵심 요소

1. Pull Request(PR) & 코드 리뷰

- 코드 품질을 유지하고, 팀원 간 의견 교환 가능
- 코드 리뷰 과정에서 버그 예방 & 성능 최적화

2. Issue & Project 기능을 통한 업무 관리

- GitHub Issues로 작업 항목을 명확하게 정의
- Kanban 보드를 사용하여 진행 상황을 한눈에 파악

3. GitHub Actions를 활용한 CI/CD 자동화

- 코드 변경 시 자동으로 빌드 & 테스트 실행
- 배포 프로세스를 자동화하여 실수 방지

◆ GitHub 협업을 중요한 이유

1. 버전 관리를 통한 안정적인 협업

- 코드 변경 사항을 체계적으로 관리하여 충돌 방지
- Branch를 활용한 독립적인 개발 가능 (feature, hotfix 등)
- 특정 시점으로 롤백(Rollback) 가능하여 코드 안정성 확보

2. 코드의 일관성 유지

- Pull Request(PR) & 코드 리뷰를 통해 코드 품질 보장
- 팀원 간 코딩 스타일 가이드라인 준수
- 기능 추가 및 수정 과정에서 중복 작업 방지

3. 원활한 의사소통 및 업무 분배

- Issue & Project 보드를 활용하여 역할 분배 및 일정 관리
- 실시간 협업을 위한 코드 리뷰, 코멘트 기능 제공
- 작업 진행 상황을 공유하여 누가 무엇을 하는지 명확하게 파악 가능

4. 자동화를 통한 개발 효율성 향상

- GitHub Actions를 활용한 CI/CD 파이프라인 구축
- 코드 변경 시 자동 테스트 및 배포 진행
- 반복적인 수동 작업을 최소화하여 개발 속도 향상

실무에서 GitHub를 효과적으로 활용하는 방법

◆ 실무에서 GitHub를 효과적으로 활용하는 방법

1. Branch 전략을 도입하라

- Git Flow, GitHub Flow 같은 branch 전략 활용
- 기능 Branch → PR → 코드 리뷰 → Merge 흐름 유지

2. PR & 코드 리뷰 문화 정착

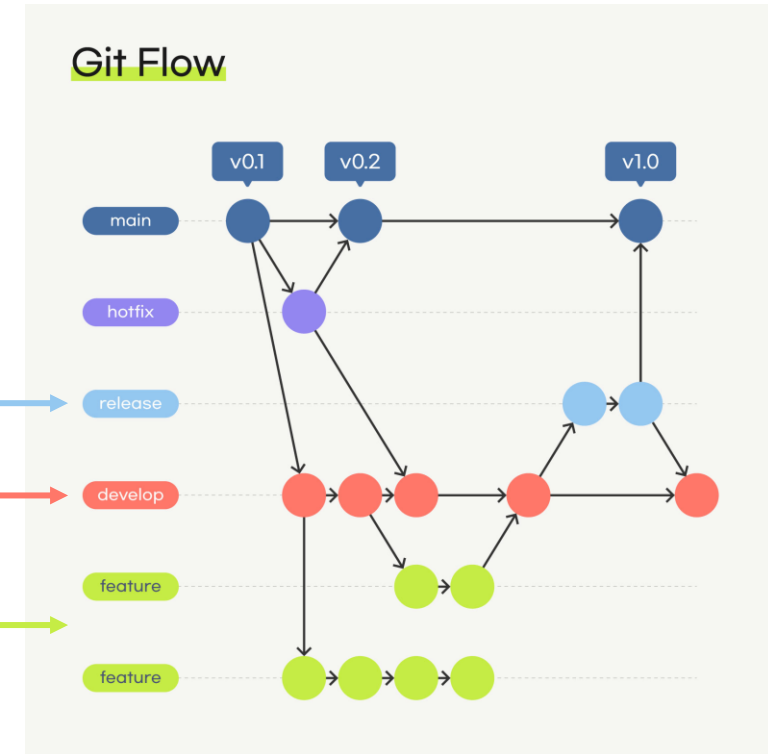
효과적인 PR 작성법:

- 명확한 제목과 설명 작성
- 작은 단위로 PR 생성 (리뷰 용이)
- git commit --signoff 활용

3. GitHub Actions 활용

Actions 활용 사례:

- 코드 품질 검사 자동화 (Lint)
- 테스트 자동 실행 (Unit/E2E)
- 배포 자동화 (Vercel, AWS 등)



▲ 실무 협업 워크플로우 예시

- Issue 생성: 작업할 내용 정의
- Branch 생성: 기능 개발을 위한 Branch
- 코드 작성 & Commit: 각 기능별로 코드 작성 및 Commit
- PR 생성: PR을 생성하여 Develop Branch로 Merge
- 코드 리뷰 & 토론: 코드 리뷰 후 수정 및 추가 Commit
- 자동 CI/CD: CI/CD 자동화 후 최종 테스트 및 배포 진행

GitHub 관련 자주 묻는 질문 (FAQ)

❖ Git을 사용할 때 흔히 겪는 문제와 해결책

❌ 'Fatal: Not a git repository' 오류

원인) 현재 디렉터리가 Git 저장소가 아님

```
fatal: not a git repository (or any of the parent directories): .git
```



해결방법

```
# 새 저장소 초기화
$ git init

# 또는 기존 저장소 복제
$ git clone [URL]
```

❌ 'Merge Conflict' 오류

원인) 같은 파일의 동일한 위치가 서로 다르게 변경됨

```
CONFLICT (content): Merge conflict in index.js
Automatic merge failed; fix conflicts and commit the result.
```



해결방법

```
# 충돌 파일 확인
$ git status

# 자동 병합 도구 실행 (옵션)
$ git mergetool

# 충돌 해결 후 커밋
$ git add .
$ git commit -m "Merge conflict resolved"
```

❌ 'git push rejected' 오류

원인) 원격 저장소의 변경 사항이 Local과 다름

```
! [rejected] main -> main (fetch first)
error: failed to push some refs to 'origin'
```



해결방법

```
# 방법 1: 변경사항 가져온 후 병합
$ git pull origin main

# 방법 2: 로컬 커밋을 원격 위에 재배치
$ git pull --rebase origin main

# 이후 다시 푸시
$ git push origin main
```

❖ 자주 발생하는 Git/GitHub 문제

• 이전 Commit 메시지 수정하기

```
# 가장 최근 커밋 메시지 수정
$ git commit --amend -m "새로운 커밋 메시지"

# 이미 푸시한 경우 강제 푸시 필요 (주의!)
$ git push -f origin branch-name
```

• Branch 삭제 후 복구하기

```
# 최근 삭제된 브랜치의 마지막 커밋 찾기
$ git reflog

# 해당 커밋에서 브랜치 새로 생성
$ git checkout -b recovered-branch 커밋해시
```

다음 실습 안내 : Git, GitHub 기초 실습 예정

