겨울방학 스터디
2회차

College of Art & Technology
Chung-Ang University

Complex Intelligent Systems Laboratory

https://cislab.cau.ac.kr

협업 규칙이 없다면?

😱 의미 없는 커밋 로그

`"fix", "update", "수정", "asdf"`

→ 버그 원인 추적 불가, git blame 무용지물

🔥 코드 병합 충돌 지옥

브랜치가 몇 주~몇 달씩 방치

→ Merge Hell, 충돌 해결에 반나절 소요

📝 형식적 코드 리뷰

1000줄 이상 거대 PR 등장

→ 리뷰어 피로 → "LGTM 👍" 버그 통과

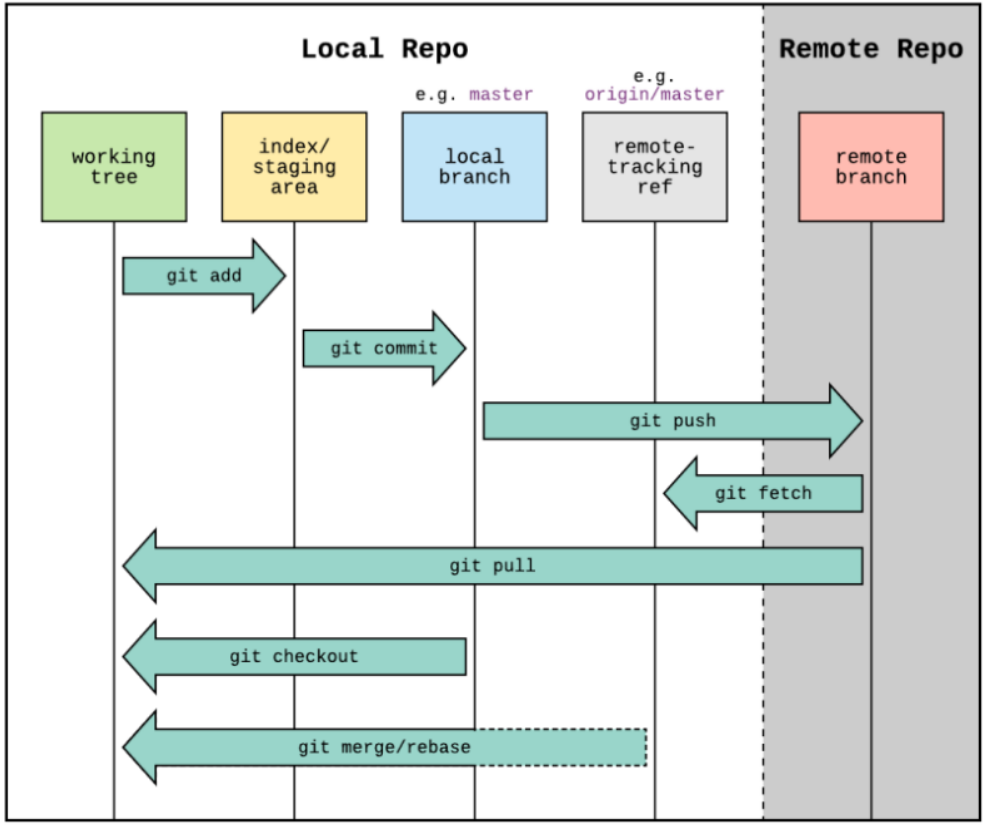
🚨 장애 대응 지연

문제 발생 시점 특정 불가

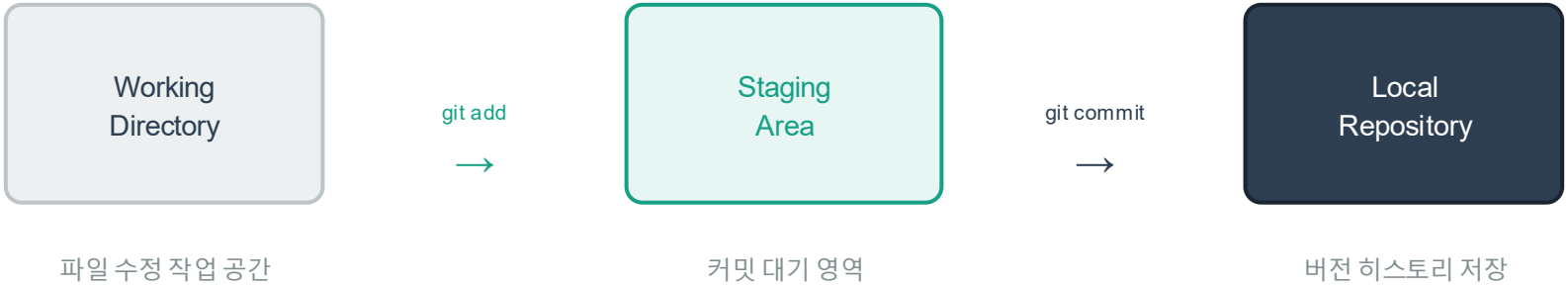
→ 롤백 범위 파악 어려움, 복구 지연

🕳️ 팀 협업 단절

코드 맥락 공유 안 됨 → 신규 멤버 온보딩 어려움, 같은 실수 반복


Local Repo
Remote Repo
e.g. master
e.g. origin/master
working tree
index/ staging area
local branch
remote-tracking ref
remote branch
git add
git commit
git push
git fetch
git pull
git checkout
git merge/rebase

Git의 3가지 영역



핵심: Git은 변경사항을 단계별로 관리하여 원하는 파일만 선택적으로 커밋 가능

Git의 3가지 영역

코드 충돌 최소화

여러 개발자가 동시에 작업할 때 충돌 관리

배포 안정성

프로덕션 코드와 개발 코드 분리

작업 추적

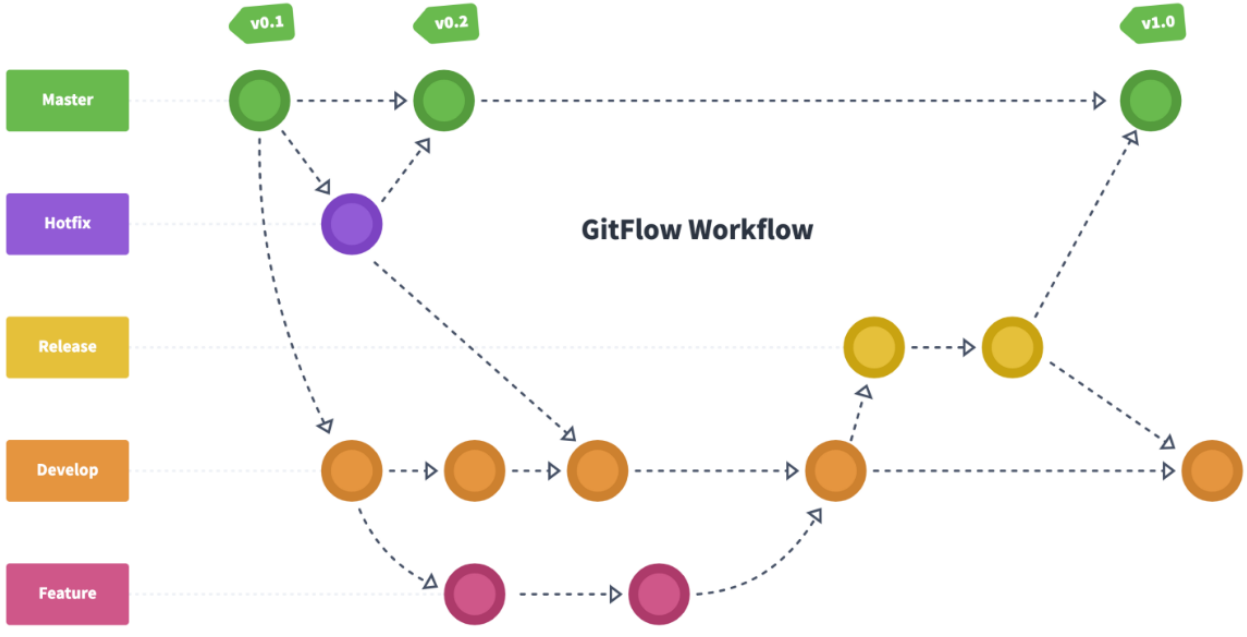
기능별, 이슈별 작업 히스토리 관리

팀의 규모, 배포 주기, 개발 문화에 따라 적합한 전략이 다름

Git Flow

5개 브랜치 구조

- **main** — 프로덕션 배포 코드
- **develop** — 다음 릴리즈 개발
- **feature/*** — 새 기능 개발
- **release/*** — 릴리즈 준비
- **hotfix/*** — 긴급 버그 수정



Git Flow의 문제점

1. 복잡한 브랜치 관리 규약

- release, hotfix 브랜치는 main과 develop에 동시 머지 필요
- feature vs hotfix 구분이 애매한 경우 발생

2. 브랜치가 오래 유지됨

- 큰 컨플릭트 발생 가능성 증가
- 코드 리뷰가 어려워짐 (변경사항이 너무 많음)
- 배포 주기가 길어짐

적합한 경우: 명시적 버전 릴리즈가 필요한 패키지/라이브러리, 다중 버전 유지보수

GitHub Flow

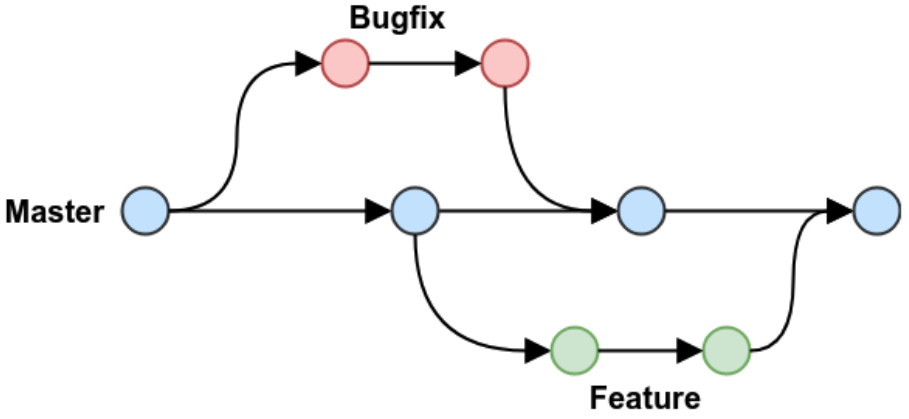
단순한 2단계 구조

main 항상 배포 가능한 상태 유지

feature 기능 개발 후 main에 머지

워크플로우

- main에서 브랜치 생성
- 작업 후 커밋
- Pull Request 생성
- 코드 리뷰
- main에 머지 → 배포



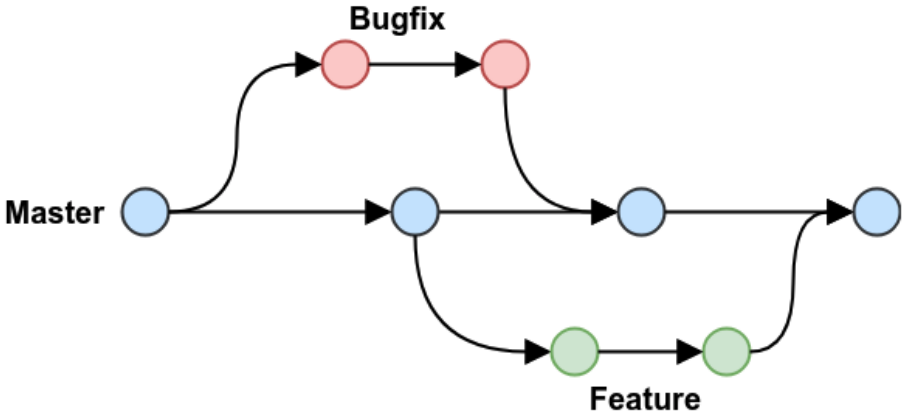
GitHub Flow

✅ **장점**

- 이해하기 쉬운 단순한 구조
- CI/CD 파이프라인과 연계 용이
- 빠른 배포 사이클

⚠️ **한계**

- feature 브랜치가 오래 유지될 수 있음
- 배포와 릴리즈 분리 어려움



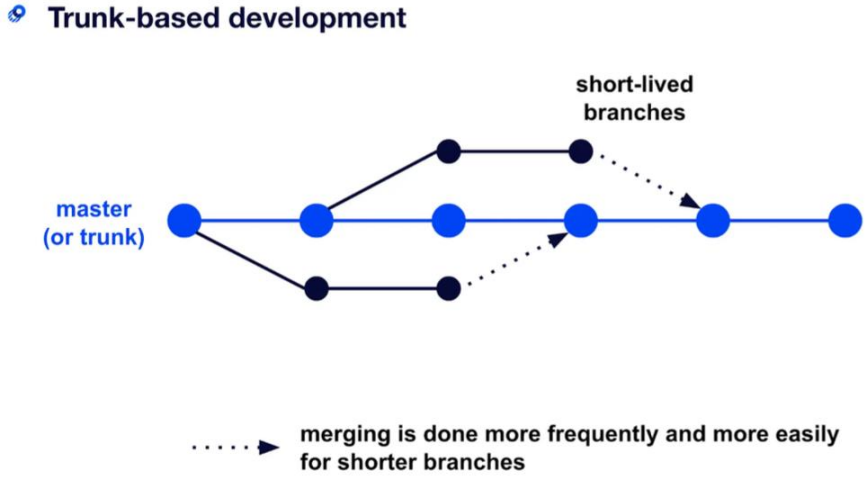
Trunk-Based Development (TBD)

핵심 개념

단일 브랜치(trunk/main)에 모든 개발자가 직접 커밋하거나,
아주 짧은 수명의 브랜치만 사용

주요 특징

- 브랜치 수명: **최대 2-3일**
- 작은 단위의 빈번한 통합
- main 머지 = 즉시 배포
- CI/CD 필수



TBD의 장점

✅ 브랜치 관리 부담 감소

복잡한 브랜치 규약 불필요, 자신의 피쳐 브랜치만 관리

✅ 빠른 배포

main 머지만으로 배포 완료, 하루 수십 회 배포 가능

✅ 작은 컨플릭트

며칠 단위 머지로 충돌이 작고 해결이 쉬움

✅ 빠른 피드백

실제 환경에서 즉시 검증, 문제 발견 시 빠른 롤백

✅ 쉬운 코드 리뷰

작은 변경사항 = 빠르고 정확한 리뷰 가능

✅ 환경 일관성

스테이지/운영 동일한 코드, 디버깅 용이

TBD의 어려움과 해결책

❌ 큰 기능 문제

몇 주~몇 달 걸리는 기능은 며칠 내 머지 어려움

✅ 해결: 작은 단위 배포

큰 작업을 여러 개의 작은 배포로 분할

❌ 의존적 기능 문제

기능 A가 기능 B에 의존 시 배포 순서 문제

✅ 해결: 피쳐 플래그 토글

코드는 배포하되, 토글로 기능 활성화 제어

❌ 불안정 기능 문제

검증 없이 바로 운영 배포는 위험

✅ 해결: 테스트 자동화

CI에서 자동 테스트로 품질 보장

TBD 핵심: 작은 배포 + 피쳐 플래그 토글 + 테스트 자동화

GitHub Flow vs TBD

GitHub Flow
기능(Feature) 중심

브랜치 수명
며칠 ~ 몇 주

통합 빈도
기능 완성 시 (낮음)

병합 방식
Merge (커밋 남김)

기능 격리
브랜치로 격리

코드 리뷰
완성 후 리뷰

Trunk-Based Development
통합(Integration) 중심

브랜치 수명
수 시간 ~ 1일

통합 빈도
매일 1회 이상 (높음)

병합 방식
Rebase (선형 히스토리)

기능 격리
피처 플래그로 격리

코드 리뷰
조금씩 자주 리뷰

💡 **결정적 차이:** GitHub Flow는 완성 후 리뷰 (수정 비용 큼) / TBD는 조금씩 자주 통합 (설계 오류 조기 발견)

3가지 전략 비교

	Git Flow	GitHub Flow	TBD
복잡도	높음	중간	낮음
배포 주기	주~월 단위	일~주 단위	시간~일 단위
브랜치 수명	주~월	일~주	시간~일
적합 환경	패키지, 다중버전	중소규모 웹앱	CI/CD 기반 웹앱
필수 조건	규칙 숙지	코드 리뷰	CI/CD, 테스트

TBD를 선택한 이유

1. 작은 팀 규모

복잡한 브랜치 관리보다 빠른 통합이 효율적

2. 웹 애플리케이션

배포 비용이 낮고 롤백이 용이한 환경

3. CI/CD 파이프라인 구축

6회차에서 배울 자동화된 테스트와 배포

4. 학습과 성장

현업에서 많이 쓰이는 실무 방식 체험

Git 브랜치 전략 이해

1. 작업 시작

```
git checkout main
git pull origin main
```

2. 브랜치 생성

```
git checkout -b feature/login
```

⭐ 3. 작업 및 커밋

```
git add src/login.js
git commit -m "feat: 로그인 API 연동"
```

4. 메인 변화 반영 (충돌 발생 가능 시점!)

```
git fetch origin
git rebase origin/main
```

5. 커밋 정리 (선택)

```
git rebase -i HEAD~3
```

6. 푸시

```
git push -f origin feature/login
```

7. 머지 (GitHub)

```
Rebase and merge ✓
```

🔁 **3~4단계 반복:** 작업 → 커밋 → rebase → 작업 → 커밋... (충돌은 4단계에서 발생!)

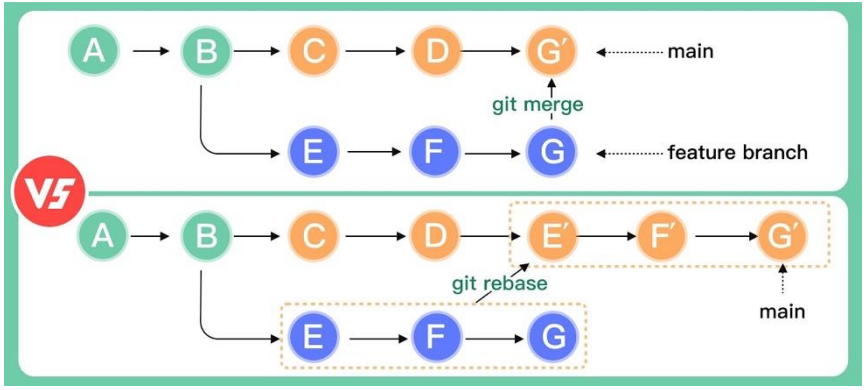
Merge vs Rebase

Merge

- 두 브랜치의 히스토리를 병합
- 머지 커밋 생성
- 히스토리가 복잡해질 수 있음

Rebase

- 커밋을 새로운 베이스로 재정렬
- 선형적인 히스토리 유지
- 깔끔한 커밋 로그



💡 TBD에서는 주로 Rebase를 활용하여 깔끔한 히스토리 유지

TBD에서 Rebase를 선호하는 이유

"자주, 짧게, 깨끗하게"

TBD는 브랜치가 오래 살아있으면 안됨

모든 변경은 최대한 빨리 Trunk에 합쳐져야 함

GitHub에서 머지 시:

`Rebase and merge ✓`

📊 히스토리 가독성

main 브랜치가 꼬임 없이 한 줄 유지

→ 기능 변화 추적이 쉬움

⚡ CI/CD 최적화

자동화된 배포 시스템이 문제 지점을

즉각 파악 가능

🛡️ 충돌 사전 방지

자주 리베이스 → 메인 변화를 조금씩 반영

→ Merge Hell 방지

TBD에서의 충돌 해결 방법

🚨 충돌 발생 시점: git rebase 실행 시

```
$ git rebase origin/main
CONFLICT (content): Merge conflict in src/login.js
error: could not apply abc1234...
```

📄 충돌 마커 예시 (src/login.js)

```
<<<<<<< HEAD
const API = "/api/v2/login";
=======
const API = "/api/v1/login";
>>>>>>> feat: 로그인 추가
```

HEAD = main의 코드 / 아래 = 내 코드

✅ 충돌 해결 4단계

Step 1. 충돌 파일 확인

```
git status
```

"both modified" 파일이 충돌 파일

Step 2. 파일 수정 (IDE 활용)

충돌 마커 제거 + 올바른 코드 선택/병합

Step 3. 스테이징

```
git add src/login.js
```

Step 4. 리베이스 계속

```
git rebase --continue
```

🔁 **반복:** 커밋 N개면 최대 N번

🚫 **포기:** git rebase --abort

TBD 심화

🚩 피쳐 플래그 (Feature Flag)

코드 배포와 기능 릴리즈를 분리

```
if (featureFlags.newCheckout) {
  return <NewCheckout />
}
```

📦 Micro PR

PR 크기를 최소화하여 리뷰 부담 감소

- 300줄 이하 권장
- 하나의 PR = 하나의 목적

가령, `git flow` 에서는 아래와 같이 처리하면 되었던 단순한 문구 변경조차도

AS-IS	TO-BE
`return <div>내 손 안의 백화점</div>;`	`return <div>오늘의 특가를 만나보세요!</div>;`

TBD에서는 이렇게 처리해야 합니다.

```
const isFlag = process.env.SOME_FLAG; // 환경변수일 수도 있고, 다른 방식일 수도 있습니다.

return <div>{isFlag ? '오늘의 특가를 만나보세요!' : '내 손 안의 백화점'}</div>;
```

피처 플래그

✅ 사용해야 하는 상황

기능이 너무 클 때
며칠 작업인데 중간 결과를 main에 합쳐야 할 때

A/B 테스트
사용자 그룹별 다른 UI 제공

리스크 관리
버그 발생 시 해당 기능만 즉시 OFF

⏸️ 안 써도 되는 상황

MVP/초기 개발 단계
실 사용자가 적어 즉각 수정 가능

소규모 팀
소통이 빨라 작업 영역 조율 가능

관리 오버헤드
플래그는 제거해야 할 기술 부채

현재 KMAP은 MVP 단계 + 소규모 팀이므로 피처 플래그 없이 진행

왜 커밋 메시지가 중요한가?

📋 변경 이력 추적

언제, 왜 코드가 변경되었는지 빠르게 파악

🤝 협업 효율화

팀원이 내 작업 의도를 묻지 않아도 이해 가능

🐛 버그 추적

git blame, git log로 문제 원인 빠르게 파악

📝 자동화 연계

CHANGELOG 자동 생성, 시맨틱 버저닝 연동

🔍 코드 리뷰 품질

커밋 메시지가 리뷰어에게 이정표 역할

커밋 메시지 = 코드의 문서화

나쁜 커밋 vs 좋은 커밋

❌ 나쁜 커밋 메시지

```
fix
```

```
Update
```

```
수정함
```

```
찐찐최종
```

```
asdf
```

✅ 좋은 커밋 메시지

```
feat: 로그인 폼 유효성 검사 추가
```

```
fix: 결제 버튼 더블클릭 방지
```

```
refactor: 사용자 인증 로직 분리
```

```
docs: API 문서 예제 코드 추가
```

```
test: 회원가입 단위 테스트 작성
```

Conventional Commits 구조

```
<type>(<scope>): <description>

[body]

[footer]
```

type (필수)
커밋의 종류를 나타냄
feat, fix, docs, style...

scope (선택)
영향 범위를 나타냄
auth, api, ui...

description (필수)
변경 내용 요약
50자 이내, 명령형

예시:

`feat(auth): 소셜 로그인 기능 추가`
`fix(api): 사용자 조회 null 처리 수정`

Type 종류

feat	새로운 기능 추가
refactor	코드 리팩토링
fix	버그 수정
test	테스트 코드 추가/수정
docs	문서 수정 (README 등)
chore	빌드, 패키지 매니저 설정
style	코드 포맷팅, 세미콜론 등
ci	CI/CD 설정 변경

💡 팀마다 사용하는 타입을 정의하여 일관성 유지

커밋 메시지 작성 규칙

제목은 50자 이내
한글 기준 약 25자 정도로 간결하게

제목은 명령형으로
"추가했음" X → "추가" O

What & Why 중심
How보다 무엇을, 왜 변경했는지

제목 끝에 마침표 X
제목은 문장이 아닌 제목

좋은 커밋 메시지 예시:

```
feat(auth): 카카오 소셜 로그인 추가

사용자 편의성 향상을 위해 카카오 OAuth2.0 연동
기존 이메일 로그인과 병행 사용 가능

Resolves: #123
```

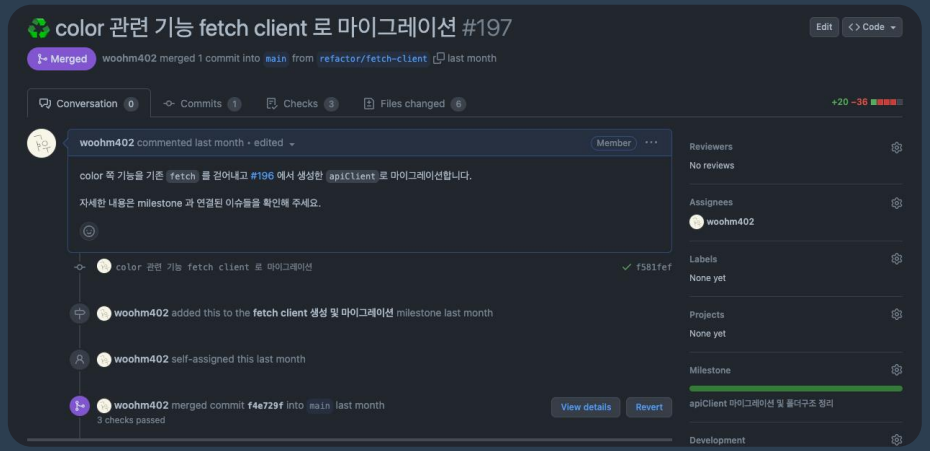
Pull Request란?

정의

내 브랜치의 변경사항을 다른 브랜치(주로 main)에 병합해달라는 요청

역할

- 코드 리뷰의 단위
- 변경 내용 문서화
- CI/CD 트리거
- 팀 내 커뮤니케이션 도구



좋은 PR의 4요소

1. Background

이 PR에 어떤 작업이 포함되었는지 요약

관련 문서, 티켓, 디자인 링크 포함

2. Changes

주요 수정 사항 기재

핵심 커밋 Hash와 함께 설명

3. Test Guide

테스트 방법 안내

UI 변경 시 전/후 스크린샷 첨부

4. Important

리뷰어에게 전달하거나 논의할 내용

특별히 봐줬으면 하는 부분 명시

💡 핵심만 정리 - 하나부터 열까지 설명하면 리뷰어가 피로감을 느낌

dhhyuk commented yesterday • edited by monibu1548

What is this PR? 🔍

- **프로덕트 스펙:** 내역 상세 리뉴얼 Product Spec
- **테크 스펙:** 내역상세 금액 에러표시 Tech Spec
- **Figma:** Figma
- **Jira Ticket:** BUDGET-405
- 기타 관련 문서:

Changes 📝

내역 상세 화면에서 금액 입력 범위 및 Validate 로직을 수정합니다.

- 내역 금액 초과에러를 InputField의 Error로 변경한다. (빨간 Underline)
- 금액 입력 범위를 -999,999,999,999 ~ 999,999,999,999원으로 변경한다.

Screenshot 📷

기능	스크린샷
GIF	

Test Checklist ☑️

- [x] 값이 999,999,999,999원을 초과하지 않는지 확인
- [x] 계산식 결과가 999,999,999,999원을 초과하지 않는지 확인

가독성 리뷰

banksaladTests/Assets Tab/List/InvestmentSectionModelTests.swift Outdated

```
+    }
+
+    func test_투자수익금_계산() {
+        let result1 = model.changedValue(valuation: 12000, principal: 20400)
```

e-sung on 30 Jun

p4: `result1` 이란 변수명이 눈에 잘 안 들어오는 것 같습니다.
각 시나리오별로 테스트 메소드를 별도로 만들고 메소드 이름을 더 상세하게 하면 어떨까요?

```
func test_손실일_경우_투자수익금_계산 {
  // given
  let valuation = 12000
  let principal = 20400

  // when
  let result = model.changedValue(valuation: valuation, principal: principal)

  // then
  expect(result).to ( equal("8,400원(-41.18%)"), description: "손실일 경우 마이너스 수익률을 표기")
}
```

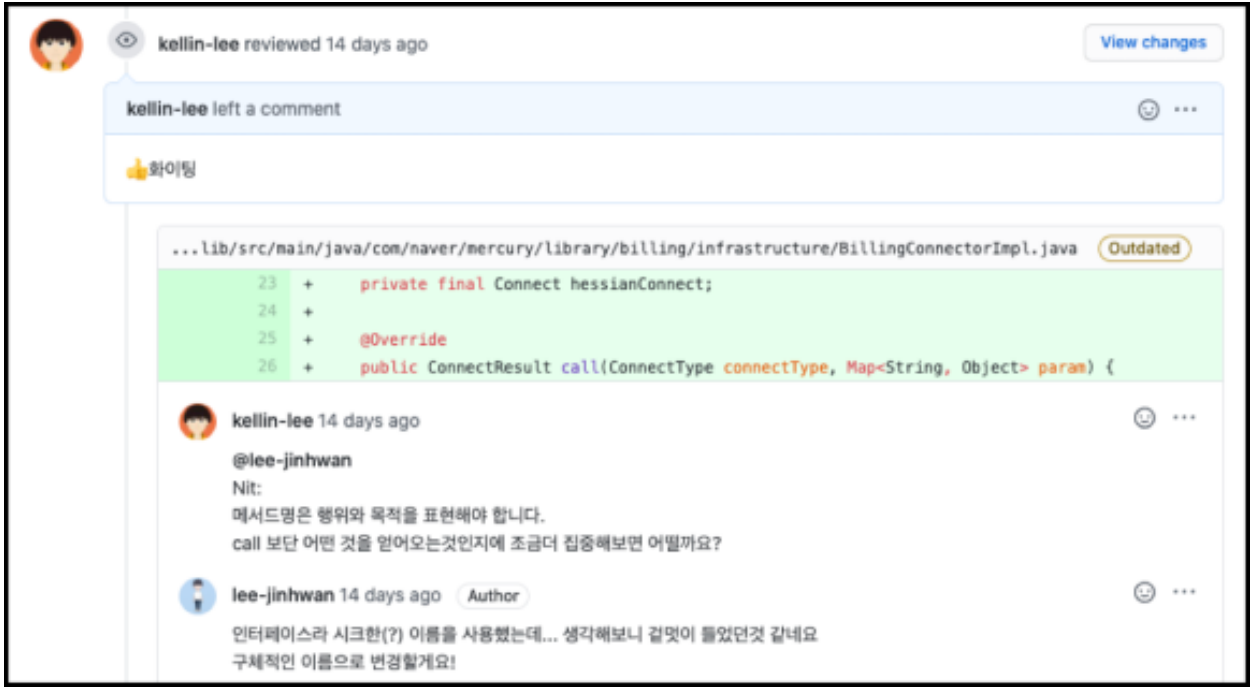
monibu1548 on 30 Jun Author

네, 좋습니다~! 분리해볼게요

Reply...

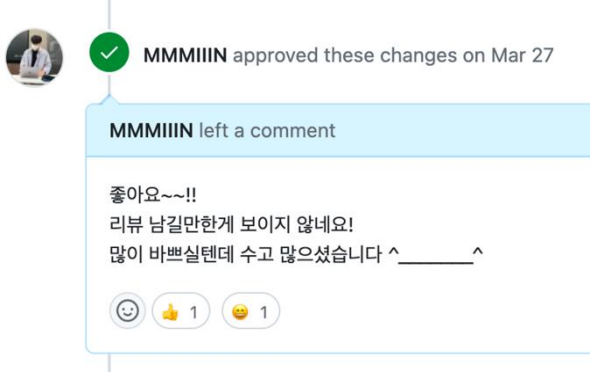
Resolve conversation

가독성 리뷰


kellin-lee reviewed 14 days ago
View changes
kellin-lee left a comment
화이팅
...lib/src/main/java/com/naver/mercury/library/billing/infrastructure/BillingConnectorImpl.java
Outdated
23 + private final Connect hessianConnect;
24 +
25 + @Override
26 + public ConnectResult call(ConnectType connectType, Map<String, Object> param) {
kellin-lee 14 days ago
@lee-jinhwan
Nit:
메서드명은 행위와 목적을 표현해야 합니다.
call 보단 어떤 것을 얻어오는것인지에 조금더 집중해보면 어떨까요?
lee-jinhwan 14 days ago
Author
인터페이스라 시크한(?) 이름을 사용했는데... 생각해보니 겁먹이 들었던것 같네요
구체적인 이름으로 변경할게요!

기술부채를 줄이기 위한 리뷰

LGTM(Looks Good To Me) 리뷰



LGTM(Looks Good To Me) 리뷰

코드 리뷰의 목적

🔍 품질 향상

버그 사전 발견, 코드 품질 개선

📚 지식 공유

팀원 간 기술/도메인 지식 전파

🎯 일관성 유지

코딩 스타일과 아키텍처 통일

좋은 코드 리뷰 문화

- 사람이 아닌 코드에 집중
- 비난 대신 건설적 제안
- 칭찬할 점도 언급하기
- 완벽보다 지속적 개선

AI 시대의 코드 리뷰

AI가 대신할 수 있는 것

- 문법/스타일 검사
- 일반적인 버그 패턴 탐지
- 엣지 케이스와 예외 처리

사람이 집중해야 할 것

- 비즈니스 로직의 정확성
- 아키텍처와 설계 결정
- 팀 컨텍스트 기반 판단

src/components/layout/useObserveSize.ts

```
22  +       observer.disconnect();
23  +     };
24  +   }, [observer, options, target]);
25  + }
```

sanchez-moon 2 weeks ago — Author

코드 리뷰를 진행하겠습니다.

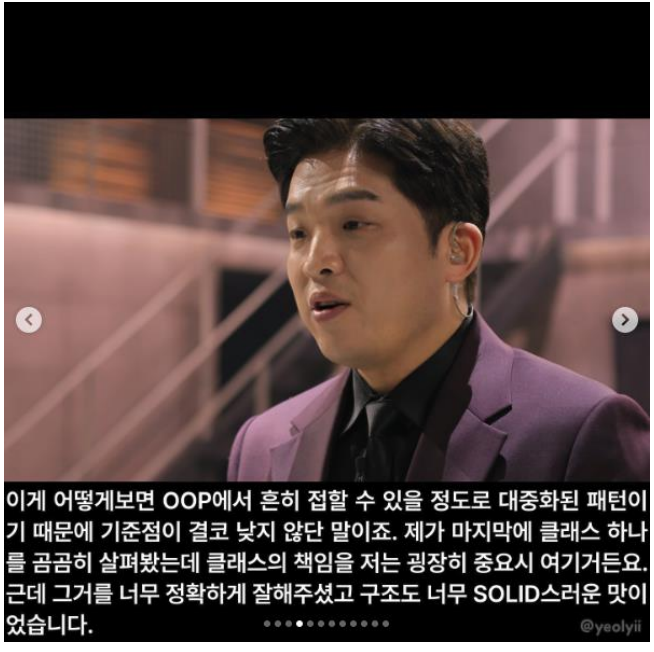
1. **메모리 누수 위험**: `useMemo`를 사용하여 `observer`를 생성하고 있지만, 컴포넌트가 언마운트될 때 `observer.disconnect()`를 호출하여 리소스를 해제하고 있습니다. 하지만, `useMemo`는 컴포넌트 업데이트 시 재사용될 수 있으므로, `observer`가 더 이상 필요하지 않을 때 안전하게 정리될 수 있도록 조건을 명확히 해야 합니다.

2. **ResizeObserver의 대표적인 부작용**: `ResizeObserver`는 비동기적으로 크기 변화를 감지하므로, 콜백 함수가 불필요하게 여러 번 호출될 수 있습니다. 불필요한 렌더링을 방지하기 위해, 콜백 함수를 debounce하거나 throttle하는 방법을 고려해볼 수 있습니다.

AI 시대의 코드 리뷰

AI 시대의 코드 리뷰


책임이 even하게 나눠지지 않았어요. 제 생각에는 클래스를 좀 더 쪼개셨어도 됐어요. 레거시가 가지고 있는 특유의 그 코드의 객체지 향이 너무 좋은데, 리팩터링 과정에서 제 생각에는 본인이 알고 있는 지식은 조금 모자란 거 같아요.
@yeolyii


이게 어떻게보면 OOP에서 흔히 접할 수 있을 정도로 대중화된 패턴이기 때문에 기준점이 결코 낮지 않단 말이죠. 제가 마지막에 클래스 하나를 곰곰히 살펴봤는데 클래스의 책임을 저는 굉장히 중요시 여기거든요. 근데 그거를 너무 정확하게 잘해주셨고 구조도 너무 SOLID스러운 맛이었습니다.
@yeolyii

Boris Cherny
@bcherny

I'm Boris and I created Claude Code. Lots of people have asked how I use Claude Code, so I wanted to show off my setup a bit.

My setup might be surprisingly vanilla! Claude Code works great out of the box, so I personally don't customize it much. There is no one correct way to use Claude Code: we intentionally build it in a way that you can use it, customize it, and hack it however you like. Each person on the Claude Code team uses it *very* differently.

sinsun_dorage
> ClaudeCode 1주

Threads

회사 내 1인 풀스택 개발자로 일하면서 클로드코드를 사용한지 7개월정도 된 것 같아요.
몰려오는 업무 지옥속에서 자동화를 위해 셀스크립트로 워크플로우들도 만들어보았지만(여전히 유용) 가장 좋은 방법은 스펙주도개발 방법론인것 같아요.

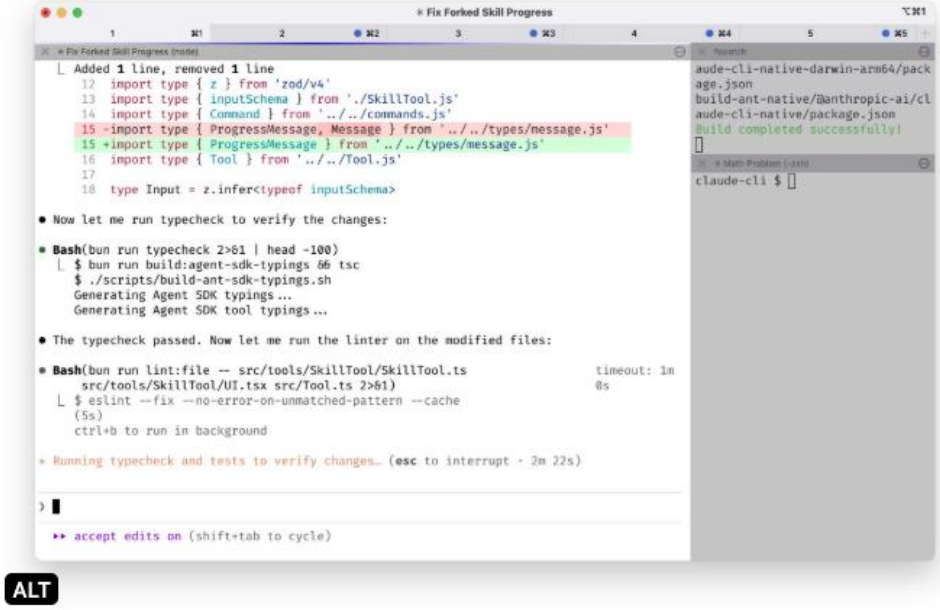
처음 들었을 때는 멘탈모델 수준의 방법론이여서 이해하기 어려웠어요.
하지만 클로드와 씨름 하다보니 자연스럽게 프롬프트가 아닌 문서를 작성하기 시작했어요.

제가 하는 스펙주도개발 방법론은 이어서 작성할게요. 1/4

60 4 11 31

멀티 세션 사용

1) 터미널에서 Claude 5개를 동시에 돌리기



ALT

nerd_makr 25. 11. 16. · 작성자

CLAUDE.md에 '작업 후에 커밋하라는 지침' 추가

- plan 수립 및 문서 작성 -> 구현 계획을 바탕으로 task(todo or progress)관련 문서 작성
- task 단위로 작업 수행 지시(작업이 끝날 때마다 커밋 지시, 커밋 해시를 문서에 기록해두기)

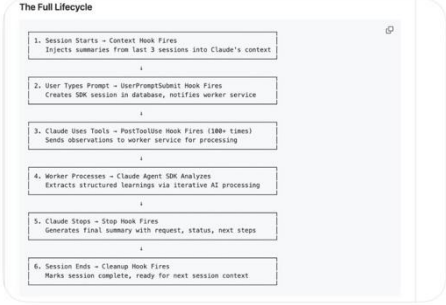
*커밋을 작업마다 해두면 롤백하기가 좋음

*실수로 llm이 코드를 엉망진창으로 만들었다면 롤백하고 다시 시작하면 됨

nerd_makr 25. 11. 16. · 작성자

- 컨텍스트 관리
- 기존에는 project_structure.md 를 만들어두고 CLAUDE.md 에 참조하라는 식으로 적어서 처리하고 있었는데 정확히 어떤 차이가 있을지는 테스트해봐야 함

> **ai.corder** 25. 10. 24.
>
> Claude code 컨텍스트 관리 시스템을 소개합니다. (추천)
>
> Claude Code를 사용하다 보면 익숙한 문제가 생깁니다... 1/3



1

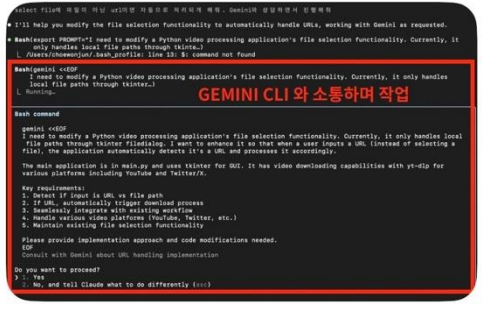
nerd_makr 25. 11. 16. · 작성자

많은 분들이 다음과 같은 방식을 많이 사용하심

1) Claude Code를 메인 코더로 사용

2) 플래닝, 의사결정 검토, 상의용으로 ChatGPT나 Codex CLI, Gemini CLI를 사용하는 경우가 많음.

> **choi.openai** 25. 6. 27.
>
> Claude Code와 Gemini CLI를 함께 쓰는 조합, 정말 좋습니다.
>
> ① Gemini CLI를 설치해서 바로 쓸 수 있게 세팅
>
> ② CLAUDE .md 파일에 아래 댓글 내용 작성...



♡ 2 💬 1

Q&A

다음 회차 안내

📅 1/22 (목) – 블로그 글 제출 및 PR 피드백

📝 과제: PR, 코드 리뷰 각자 1개 이상 수행
(다음주 목요일까지)