복잡함을 단순하게 바꾸는
프론트엔드 개발자

윤정연

010-2290-6150
yoouyeon.dev@gmail.com

About

팀의 일하는 방식을 더 효율적으로 만드는 일을 좋아합니다.
아이디어를 구체화해 문제를 해결하고,
함께 결과를 만들어가는 과정에서 즐거움을 느낍니다.
원리를 깊이 이해할 수 있는 개발자가 되기 위해 꾸준히 공부하고 있습니다.

Key Point

- TypeScript · React · Next.js 기반의 웹 서비스 개발 프로젝트 경험
- 기획 · 디자인 등 타 직군과 협업하여 서비스를 개발한 경험
- 문제를 기술적으로 해결하고 비효율적인 과정을 개선한 경험
- GitHub Actions를 활용한 반복 작업 자동화 경험
- 기술 블로그와 개인 위키 운영을 통해 지속적으로 학습하고 지식을 공유

윤정연

Contact yoouyeon.dev@gmail.com
010-2290-6150

Github https://github.com/yoouyeon

Blog https://blog.yoouyeon.dev

Skils JavaScript, TypeScript
React, Next.js

Educations

42 Seoul 2021 - 2023

C/C++ 기반 과제를 수행하며 CS 핵심 개념을 깊이 학습했습니다. 이 과정에서 자기주도적으로 학습하고 협력하며 성장하는 방법을 배웠습니다.

상명대학교 2017 - 2022

컴퓨터과학과 졸업

5 Projects

MODDO 2025

TypeScript, React
모임에서 발생한 지출을 분할 정산하는 서비스입니다. 프론트엔드 개발로 참여했습니다.

Githru 2024

TypeScript, Jest, Git
Git 데이터를 시각적으로 분석하는 VSCode Extension 오픈소스 프로젝트입니다.

Reviewer 2024

TypeScript, Next.js
광고 회사 어드민 페이지 개발 외주 프로젝트입니다. 프론트엔드와 PM으로 참여했습니다.

peer 2023 - 2024

TypeScript, Next.js
42 Seoul 학생들을 위한 프로젝트 팀빌딩 프로젝트입니다. 프론트엔드 개발로 참여했습니다.

42gg 2023

TypeScript, Next.js
42 Seoul 학생들의 탁구대 예약과 랭킹 서비스입니다. 프론트엔드 개발로 참여했습니다.

5 Experiences

IT 연합동아리 DND 2025

프론트엔드, 백엔드 개발자와 디자이너 총 6명으로 구성된 팀으로 8주간 서비스 기획부터 개발, 배포 후 피드백까지의 과정을 경험했습니다. 활동 결과 최우수상을 받았습니다.

2024 오픈소스 컨트리뷰션 아카데미 2024

Git 데이터를 시각적으로 분석하는 VSCode Extension 오픈소스 프로젝트 Githru에 참여해서 데이터 처리 속도 개선과 테스트 유지보수에 기여했습니다. 활동 결과 장려상을 받았습니다.

Next.js 한글 문서 번역 2024

Next.js 공식 문서의 한글 번역 페이지를 만드는 오픈소스 프로젝트에 기여했습니다.

스마일게이트 AI 해커톤 2024

프론트엔드, 백엔드 개발자와 기획자, 프롬프트 엔지니어로 구성된 팀으로 1박 2일간 생성형 AI를 이용해 편지를 작성하고 공유할 수 있는 서비스를 기획하고 개발했습니다.

유니톤 - IT 커뮤니티 연합 해커톤 2023

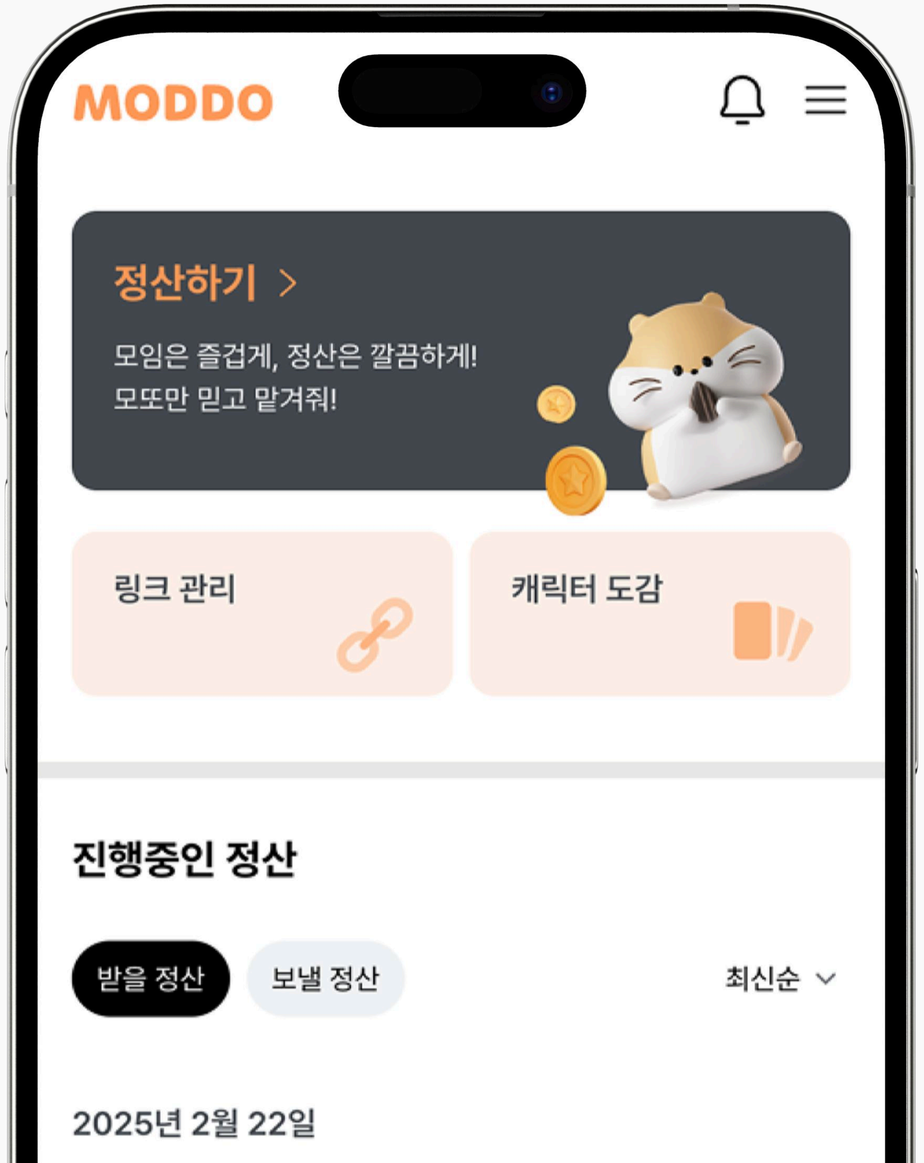
프론트엔드, 백엔드 개발자와 기획자, 디자이너로 구성된 팀으로 2박 3일간 K-POP 팬들이 좋아하는 연예인과 관련된 장소를 기록하고 공유할 수 있는 서비스를 기획하고 개발했습니다. 활동 결과 우수상을 받았습니다.

YOON JEONGYEON

MODDO (모또)

모임 정산 서비스

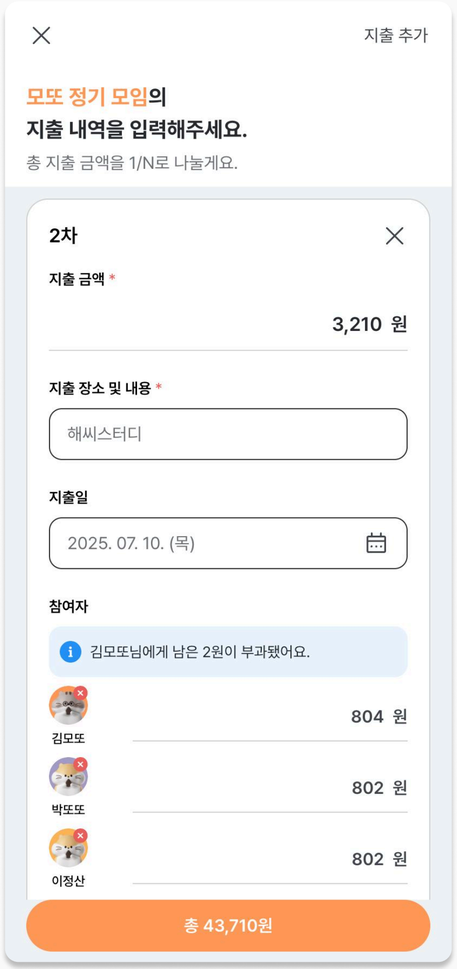
모또는 모임 후 정산 과정에서 총무의 부담을 줄이기 위한 프로젝트입니다.
모임에서 발생한 지출을 간편하게 분할 정산하는 기능을 제공하고
캐릭터 보상 시스템으로 정산 독촉의 심리적 부담을 낮춰
자연스러운 정산 참여를 유도합니다.

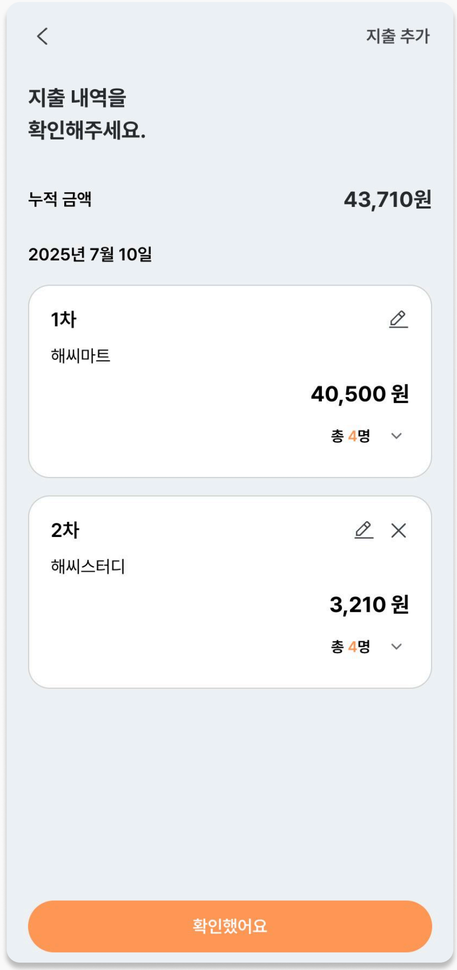


배포 사이트	https://moddo.kr/
깃허브	https://github.com/moddo-kr/moddo-frontend
팀 구성	프론트엔드 2명 / 백엔드 2명 / 디자이너 2명
개발 기간	2025.01 ~ present
기술 스택	TypeScript, React, Vite, TanStack Query, styled-components
역할	· 프론트엔드 · 모임 지출 추가와 공유 기능 개발, 디자인 시스템 구축

지출 기록

정산할 지출을 추가하는 단계입니다. 지출 장소와 날짜를 함께 입력할 수 있고, 지출 금액을 입력하면 참여자들에게 자동으로 1/N 분할합니다.



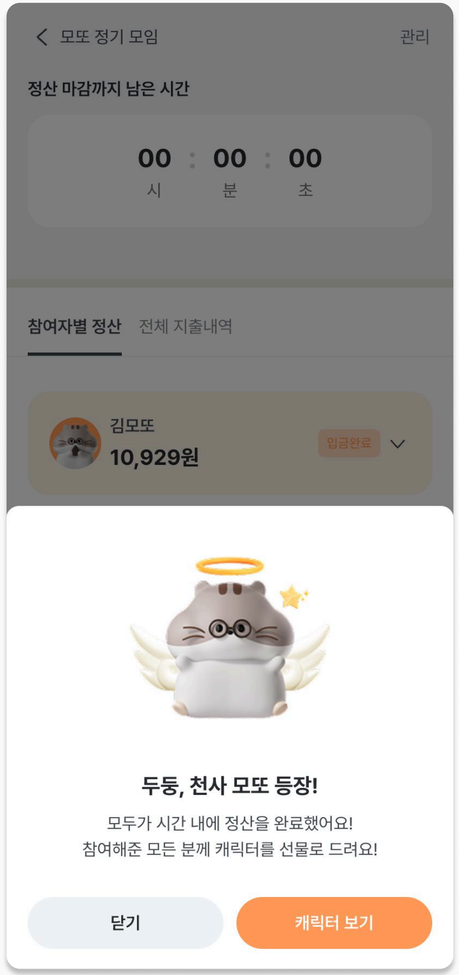


지출 내역 확인

지금까지 추가한 지출들을 확인할 수 있습니다. 추가한 지출을 수정하거나 삭제할 수 있고, 새로운 지출을 추가할 수 있습니다.

정산 공유

지출 내역을 참여자들에게 링크로 공유할 수 있습니다. 바로 접속할 수 있는 QR 코드를 만들 수 있고, 카카오톡과 슬랙으로 공유할 수 있는 기능을 제공합니다.



정산 성공

참여자들이 모두 제한된 시간 안에 정산을 완료하면 햄스터 캐릭터 보상을 얻을 수 있습니다.

Error Boundary를 이용해 에러 처리 로직을 선언적으로 분리하고 코드 가독성과 유지보수성을 높였습니다.

Problem

❶ 에러 처리 로직의 분산과 일관성 부족

각 컴포넌트에 에러 처리 로직이 흩어져 있어 동일한 코드를 여러 곳에서 반복적으로 수정해야 했습니다. 또한 화면마다 에러 처리 방식이 달라 비슷한 오류 상황에서도 결과가 달랐고, 새로운 예외가 생기면 처리 과정이 누락되기도 해 서비스 신뢰도와 사용자 경험이 떨어질 위험이 있었습니다.

❷ 렌더링 로직과의 결합

비즈니스 · 렌더링 로직 안에 예외 처리가 섞여 있어 코드의 핵심 로직을 파악하기 어렵고, 에러 화면을 재사용하거나 처리 패턴을 통일하기 어려웠습니다.

Solution

서비스에서 발생하는 에러를 **Router 외부 · Router 내부 · loader 계층**으로 나누고, 각 계층별로 **Error Boundary**를 적용해 렌더링 로직과 에러 처리를 분리해 **가독성과 유지보수성을 높였습니다.**

또한 **커스텀 useQuery/useMutation 훅**을 설계해 API 응답의 **상태 코드별 에러를 일관된 방식으로 처리**하도록 리팩토링했으며, 그 결과 유사한 오류 상황에서 일관된 에러 화면을 제공할 수 있게 되어 **사용자 경험이 개선**되는 효과도 있었습니다.

▸ 관련해서 작성했던 블로그 글

https://blog.yoouyeon.dev/error-boundary-error-handling

▼ 비즈니스 로직과 섞이고 일관성이 없었던 에러 처리 코드

```
const {
  data: headerData,
  isLoading,
  isError,
} = useGetGroupHeader(groupToken!);

// ...생략

if (isError) {
  return <div>error...</div>;
}
```

▼ 커스텀 useQuery를 이용해 일관된 방식으로 에러와 처리 방법을 정의하는 코드

```
// 커스텀 useQuery 사용 예시
const { data, isLoading, isError } = useGetGroupBasicInfo(
  groupToken,
  {
    403: () => {
      throw new BoundaryError({
        title: '접근 권한이 없어요',
        description: '모임의 총무만 참여자를 추가할 수 있어요.',
      })
    },
  },
  [403]
)
```

MSW를 이용해 프론트엔드 개발 기간을 단축시켰습니다.

Problem

프론트엔드와 백엔드를 **병렬로 개발**하는 일정이었고, 백엔드 API는 **개발 일정 후반부에 배포**될 예정이었습니다.

개발할 화면 대부분이 **API 응답에 의존**하고 있었기 때문에 실제 API를 받은 뒤에 연동을 시작하면 프론트엔드 구현과 QA를 마무리할 시간이 거의 없는 상황이었습니다.

이대로 진행하면 QA가 충분히 이뤄지지 않아 프로젝트 완성도가 떨어지고, 발표 시점에 **안정적인 데모를 보여주기 어려울 것**이라는 우려가 있었습니다.

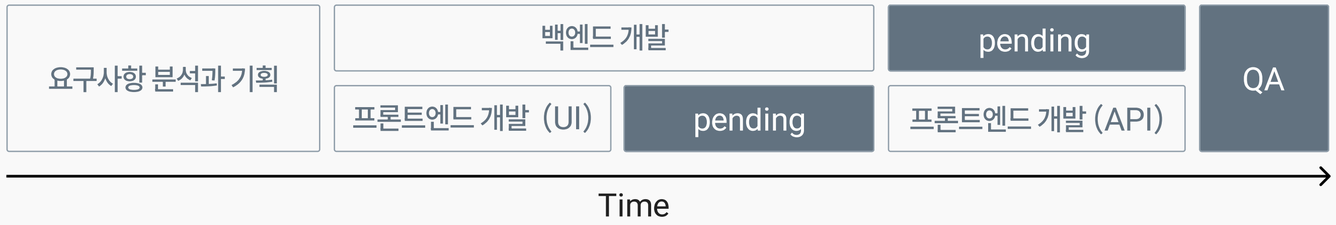
Solution

MSW를 활용해 API를 모킹해서 프론트엔드 개발을 **백엔드 일정과 무관하게 진행**할 수 있도록 했습니다.

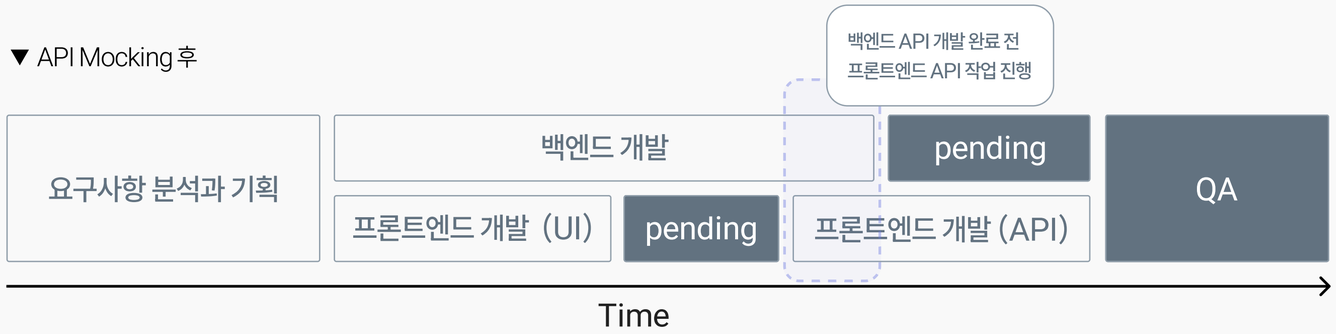
모킹 과정에서는 **백엔드 개발자와 함께 API 스펙을 사전에 조율**해서 **실제 API 변경에 따른 리스크를 줄였습니다.**

이 결과 **실제 API 배포 이전에 대부분의 기능을 구현**하고, 발표 전까지 **충분한 QA 시간을 확보**할 수 있었습니다.

▼ API Mocking 전



▼ API Mocking 후



▼ Axios 요청 인터셉터에 useMock 옵션을 추가해, MSW 모킹 API로 전환할 수 있게 했습니다.

```
if (newConfig.useMock) {
  newConfig.baseURL = 'http://localhost:3000/api/v1';
  newConfig.headers = AxiosHeaders.from({
    ...newConfig.headers,
    'X-Mock-Request': 'true',
  });
}
```

URL 기반 퍼널 구조를 적용해
멀티스텝 폼의 단계 전환을 일관적으로 제어할 수 있도록 개선했습니다.

Problem

❶ 프로세스마다 단계 관리 방식이 달라 일관성이 부족함

모임 생성 프로세스에서는 URL 기반으로 단계를 관리하는 반면
지출 생성 프로세스에서는 컴포넌트 상태 기반으로 단계를 관리해서
방식 간 일관성이 부족했습니다.

❷ 일부 단계에서 브라우저 네비게이션이 동작하지 않음

컴포넌트 상태 기반으로 단계를 관리하던 일부 프로세스에서는
뒤로가기 시 진행 중이던 단계가 초기화되는 문제가 있었습니다.
이 문제는 일부 프로세스에만 존재해서 일관된 사용자 경험을 제공하지
못하는 문제가 있었습니다.

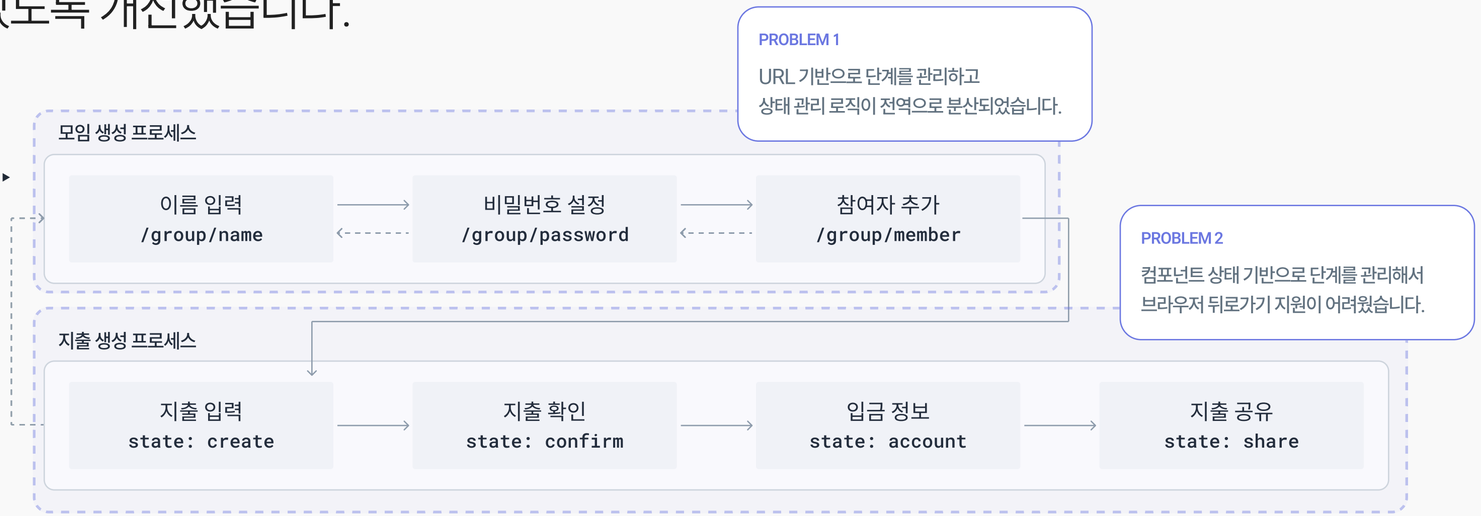
Solution

모임 생성과 지출 생성 프로세스의 단계 관리 방식을
URL 기반 퍼널 구조로 통일했습니다.

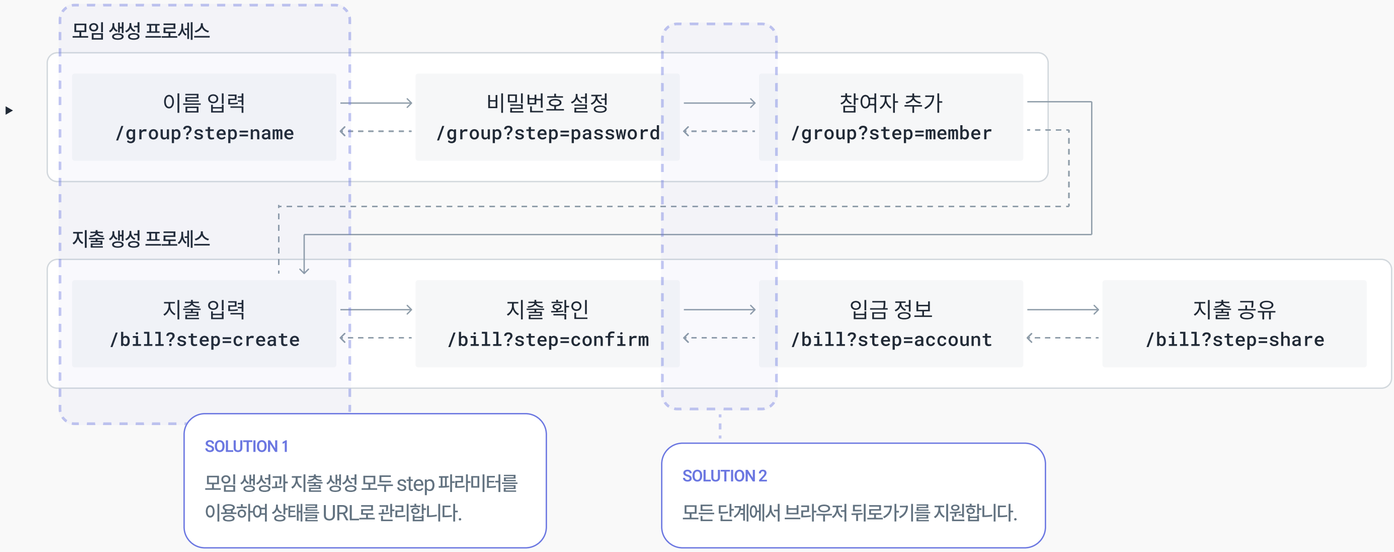
각 단계의 상태와 전환 로직을 **루트 페이지 한 곳에서 관리**
하도록 재설계해서 **코드 응집도**를 높였습니다.

URL을 기반으로 단계 흐름을 제어해서 뒤로가기 등
브라우저 네비게이션이 자연스럽게 동작하도록 개선했습니다.

개선 전 다이어그램 ▸



개선 후 다이어그램 ▸



GitHub Actions를 이용해 반복 작업을 자동화했습니다.

Problem

❶ PR 생성 시 반복되는 수동 작업

라벨 지정, 리뷰어 할당, Zenhub 태스크 상태 변경을
PR을 생성할 때 마다 수동으로 처리해야 하는 불편함이 있었습니다.

❷ 작업 누락과 처리 방식 불일치

수동 작업 과정에서 일부 단계가 누락되거나, 개발자마다 처리 방식이 달라
일관성이 떨어졌습니다.

Solution

GitHub Actions 워크플로를 구축해 PR 생성과 병합 과정의
반복 작업을 자동화했습니다.

브랜치명을 기반으로 관련 이슈의 **라벨을 자동 부여**하고,
Zenhub 태스크 상태를 자동으로 업데이트하도록 구성했습니다.

▸ 관련해서 작성했던 블로그 글

https://blog.yoouyeon.dev/github-action-zenhub-code-review

▼ action 실행 결과


```yaml
name: Automate PR Reviews

on:
  pull_request:
    types: [opened, ready_for_review]

jobs:
  assign:
    steps:
      - uses: hkusu/review-assign-action@v1
        with:
          assignees: ${{ github.actor }}
          reviewers: ${{ vars.REVIEWERS }} # 리뷰할 수 있는 사람의 목록
  automate-pr-review:
    steps:
      - uses: ./.github/actions/sync-labels
      - uses: ./.github/actions/zenhub-move-issue
```

▲ PR 생성 시 Assignee 지정·라벨 할당·Zenhub 이동 단계를 자동화한 워크플로 (수도 코드)

디자인 시스템의 토큰을 정의하고 타입 안정성을 개선했습니다.

디자인 시스템의 **컬러 토큰을 테마 객체로 구조화**하고,
이를 기반으로 **타입을 생성하는 유틸리티 타입**을 정의했습니다.

이를 통해 IDE 자동완성 기능을 통해 **사용할 수 있는 토큰을 쉽게 찾을 수 있게 되었고**, **컴파일 타임에 잘못 사용된 토큰을 감지**할 수 있었습니다.

또한 **타입이 테마 객체를 기준으로 자동 생성되도록 구성**하여 컬러 토큰이 변경되더라도 **타입 정의를 별도로 수정할 필요가 없어 유지보수성이 향상**되었습니다.

▸ 관련해서 작성했던 블로그 글

https://blog.yoouyeon.dev/convert-object-keys-to-type

```ts
type ColorTokenType =
    | 'primary.subtle'
    | 'primary.default'
    | 'primary.strong'
    | 'primary.heavy'
    | 'state.danger'
    | 'state.warning'
    | 'state.info'
    | 'state.success';
```

▲ 생성된 컬러 토큰 타입

```ts
/**
 * @description
 * ColorTokenPath의 재귀 깊이를 줄이기 위한 유틸리티 타입
 */
type DepthDecrement = [never, 0, 1, 2, 3, 4, 5];

/**
 * @description
 * ColorType의 키의 경로를 타입으로 변환하는 유틸리티 타입
 * @template T - 디자인 토큰 객체 타입
 * @template P - 현재까지의 경로 접두사
 * @template D - 최대 재귀 깊이 (기본값: 5)
 */
type ColorTokenPath<
  T,
  P extends string = '',
  D extends number = 5,
> = D extends 0
  ? never // 무한 루프를 방지하기 위해 재귀 깊이 제한에 도달하면 종료
  : T extends Record<string | number, unknown>
    ? {
        [K in keyof T]: K extends string | number
          ? T[K] extends Record<string | number, unknown> // 객체일 때 재귀 호출
            ? ColorTokenPath<T[K], `${P}${K}.`, DepthDecrement[D]>
            : `${P}${K}` // 객체가 아닌 경우(leaf)에는 현재 키를 포함한 경로 반환
          : never; // K가 string | number가 아닌 경우는 무시
      }[keyof T]
    : never; // T가 객체가 아닌 경우는 무시
```

▲ 컬러 토큰 생성 유틸리티 타입

YOON JEONGYEON

Githru (깃쓰루)

Git의 시각적 분석을 위한 VSCode Extention

Githru는 Git 로그를 시각화하여
개발 히스토리를 더 쉽게 이해할 수 있도록 돕는
오픈소스 VSCode 확장 프로그램입니다.

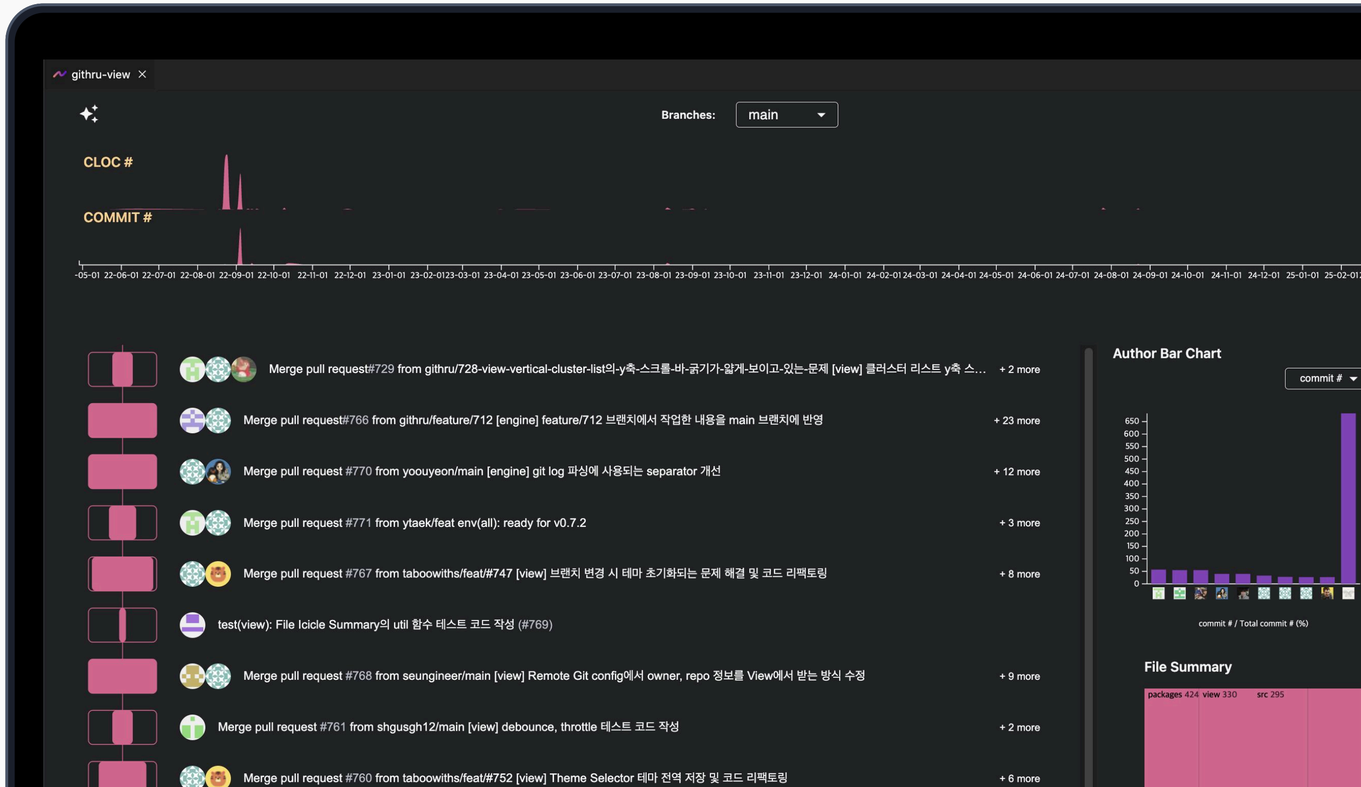
깃허브 https://github.com/githru/githru-vscode-ext

개발 기간 2024.07 ~ 2024.11

기술 스택 TypeScript, jest, Git

역할
- 대규모 git 로그 처리 성능 최적화
- 테스트 안정성 개선

기여한 PR https://github.com/githru/githru-vscode-ext/pulls?q=is%3Apr+assignee%3Ayoouyeon+



Git 로그의 파싱 로직을 최적화했습니다.

Problem

기존 Git 로그 파싱 로직은 fuller 포맷으로 로그를 가져온 뒤,
필요한 필드를 추출하기 위해 **총 3번의 순회**를 수행했습니다.

이 과정에는 괄호나 공백 제거 같은 **불필요한 문자열 처리**가 많았고,
데이터를 **분리했다가 다시 조합하는 비효율적인 부분**도
존재했습니다.

```
commit fbb514f6136ec4f 491c8469533e5d (HEAD -> main, origin/main, origin/HEAD)
Author:     yoouyeon <jyeon.yoon59@gmail.com>
AuthorDate: Fri Aug 30 12:58:10 2024 +0900
Commit:     yoouyeon <jyeon.yoon59@gmail.com>
CommitDate: Fri Aug 30 12:58:10 2024 +0900

    fix(engine): lint error

1       1       packages/analysis-engine/src/parser.spec.ts
1       1       packages/analysis-engine/src/parser.ts
```

▲ fuller 포맷으로 출력한 Git 로그 예시

```
// 각 필드를 따로따로 배열에 저장
ids.push(...);
parentsMatrix.push(...);
branchesMatrix.push(...);
tagsMatrix.push(...);
// ...

// 마지막에 다시 합침
for (let i = 0; i < ids.length; i++) {
  commitRaws.push({
    id: ids[i],
    parents: parentsMatrix[i],
    // ...
  });
}
```

▲ 여러 배열로 관리 후, 다시 합쳐야 하는 복잡한 순회 구조

Solution

fuller 포맷 대신 **커스텀 포맷**을 설계해
필요한 필드를 **한 번에 파싱**할 수 있도록 개선했습니다.

각 필드를 특정 구분자로 구분하고, String.split() 을 활용해
단 두번의 연산만으로 파싱하도록 변경했습니다.

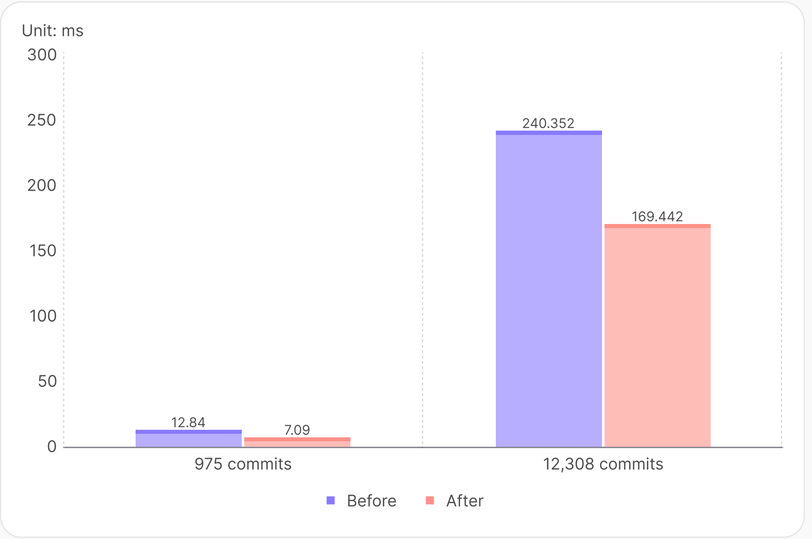
이를 통해 **3회 순회하던 파싱 로직을 단일 작업으로 최적화**했고,
1,000개 커밋에서 44.78%, **10,000개 커밋에서 26.44%**의
처리 시간을 단축했습니다.

▸ **관련해서 작성했던 블로그 글**

https://blog.yoouyeon.dev/ossca-week-7-record

```
<commit hash>
<parent hashes>
<refs(branches, tags>
<author name>
<author email>
<author date>
<committer name>
<committer email>
<committer date>
<commit message subject>
<commit message body>
<diffstat>
```

▲ 수정한 Git 로그 포맷



▲ Git Log 파싱 성능 비교 (As-Is vs To-Be)

```
// split 2회로 곧바로 파싱
const commits = log.split(COMMIT_SEPARATOR);
const [hash, parents, ref, ...rest] =
  commit.split(GIT_LOG_SEPARATOR);

commitRaws.push({ hash, parents, ref, ...rest });
```

▲ split 2회로 파싱 후 바로 JSON 구성

Git 로그 파싱 함수의 테스트 커버리지를 향상했습니다.

Git 로그 파싱 로직을 개선하는 과정에서,
기존 테스트가 **단순한 사례만 다루고 있다는 한계**를 발견했습니다.

이를 보완하기 위해 실제 커밋 로그 데이터를 수집해 분석하고,
git log 매뉴얼을 검토해 아래와 같은 **엣지 케이스**를 확인했습니다.

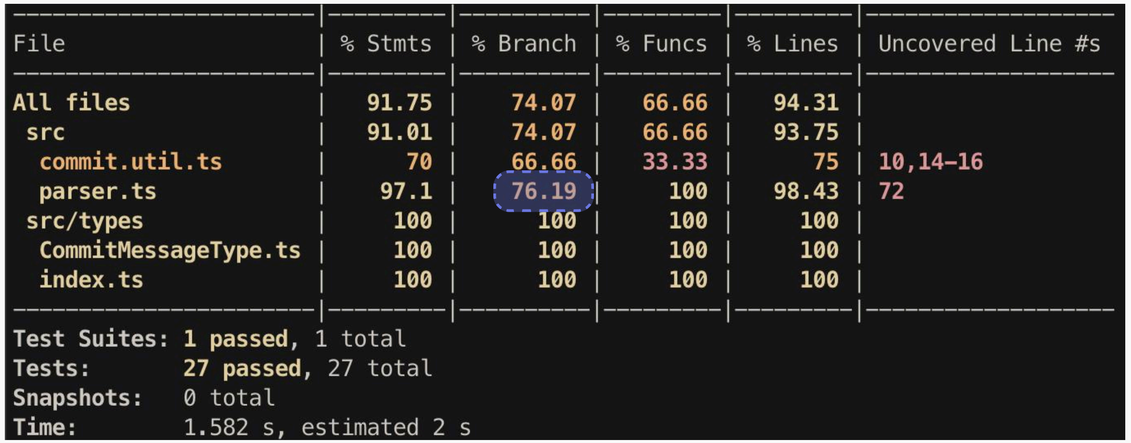
- **첫 번째 커밋**: parent 해시가 없음
- **Merge 커밋**: parent 해시가 2개 존재
- **빈 커밋**: 커밋 메시지 body가 없음

이 외에도 실제 사용 맥락을 고려해서

- **body가 포함된 커밋**
- **여러 커밋이 포함된 로그**

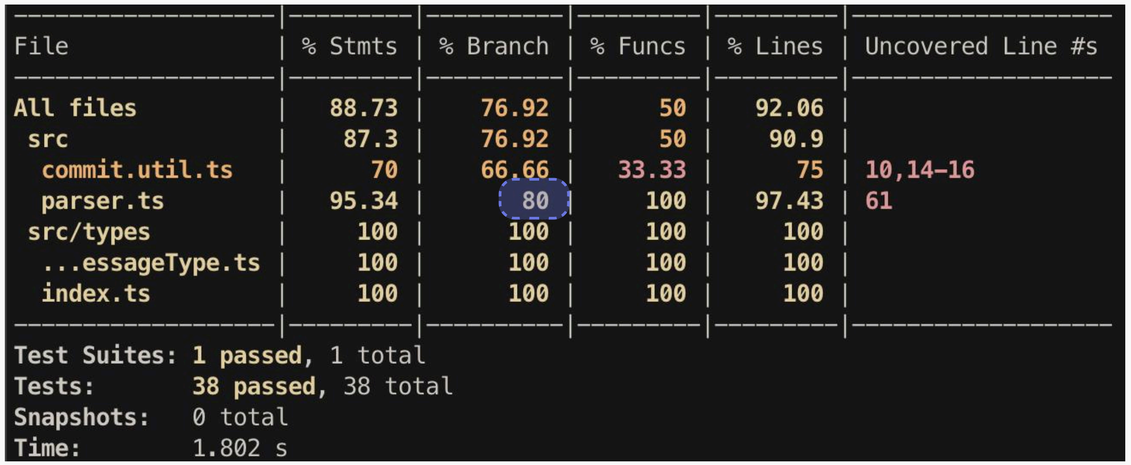
에 대한 테스트 시나리오를 추가했습니다.
그 결과 **Branch coverage를 76.1% → 80%로 향상**시켰으며
핵심 로직의 **분기 처리 안정성**을 높일 수 있었습니다.

▸ **개발 작업 Pull Request**
https://github.com/githru/githru-vscode-ext/pull/735



```
----------------------|---------|----------|---------|---------|-------------------
File                  | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
----------------------|---------|----------|---------|---------|-------------------
All files             |   91.75 |    74.07 |   66.66 |   94.31 |
 src                  |   91.01 |    74.07 |   66.66 |   93.75 |
  commit.util.ts      |      70 |    66.66 |   33.33 |      75 | 10,14-16
  parser.ts           |    97.1 |    76.19 |     100 |   98.43 | 72
 src/types            |     100 |      100 |     100 |     100 |
  CommitMessageType.ts|     100 |      100 |     100 |     100 |
  index.ts            |     100 |      100 |     100 |     100 |
----------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       27 passed, 27 total
Snapshots:   0 total
Time:        1.582 s, estimated 2 s
```

▲ 테스트 케이스 추가 전. Branch coverage 76.19%



```
--------------------|---------|----------|---------|---------|-------------------
File                | % Stmts | % Branch | % Funcs | % Lines | Uncovered Line #s
--------------------|---------|----------|---------|---------|-------------------
All files           |   88.73 |    76.92 |      50 |   92.06 |
 src                |    87.3 |    76.92 |      50 |    90.9 |
  commit.util.ts    |      70 |    66.66 |   33.33 |      75 | 10,14-16
  parser.ts         |   95.34 |       80 |     100 |   97.43 | 61
 src/types          |     100 |      100 |     100 |     100 |
  ...essageType.ts  |     100 |      100 |     100 |     100 |
  index.ts          |     100 |      100 |     100 |     100 |
--------------------|---------|----------|---------|---------|-------------------
Test Suites: 1 passed, 1 total
Tests:       38 passed, 38 total
Snapshots:   0 total
Time:        1.802 s
```

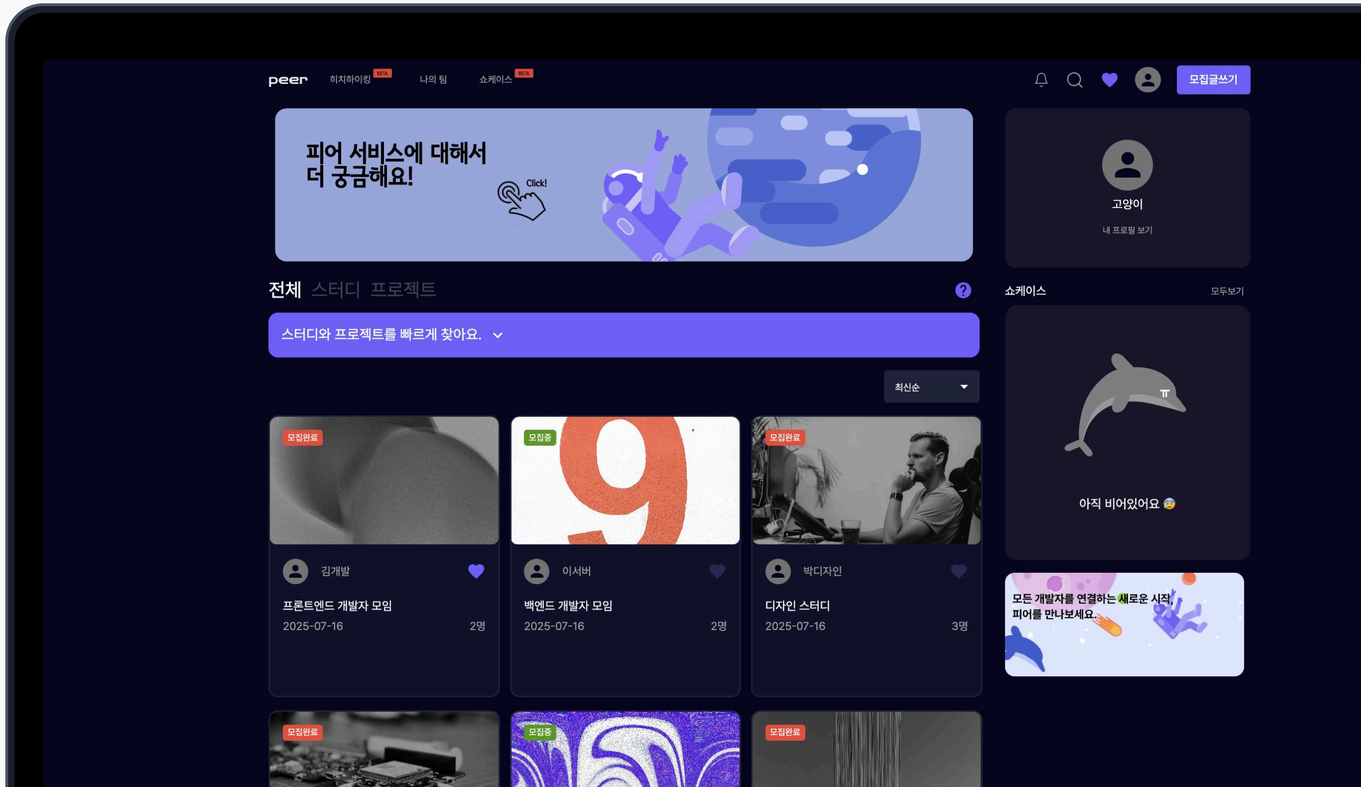
▲ 테스트 케이스 추가 후. Branch coverage 80%

YOON JEONGYEON

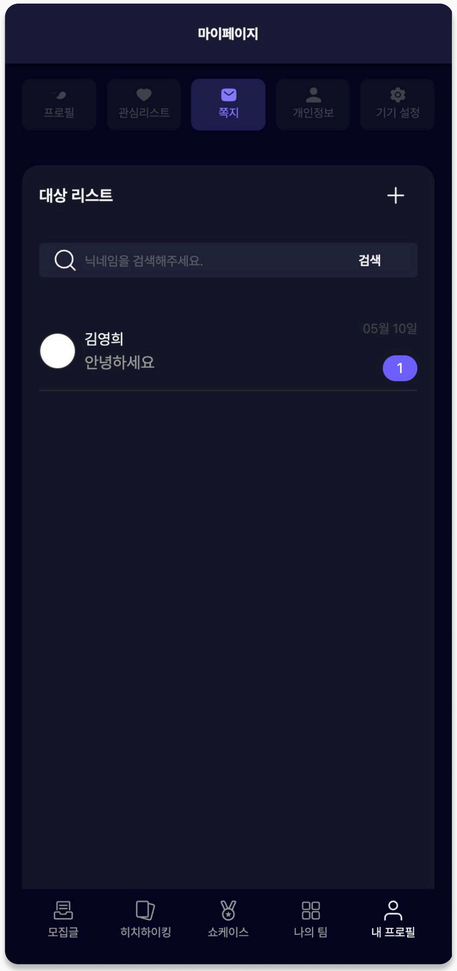
peer (피어)

42Seoul 학생들을 위한 팀 빌딩 플랫폼

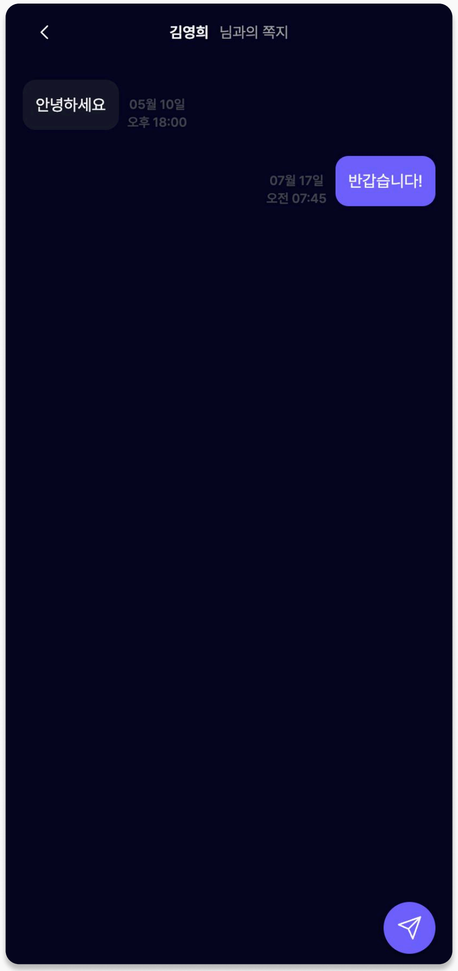
peer는 42 Seoul 커뮤니티원을 위한
프로젝트/스터디 팀을 찾고, 연결하고, 함께하기까지의 번거로운 과정을
단순하게 이어주는 팀 빌딩 플랫폼입니다.



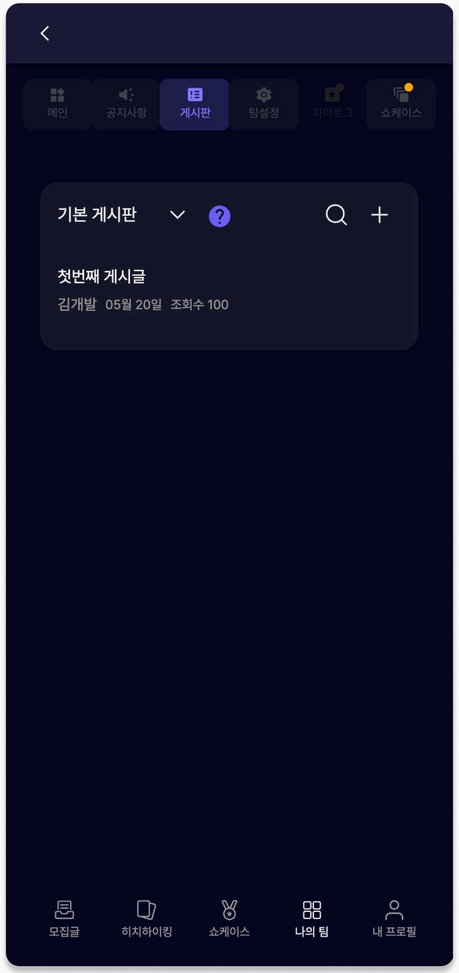
깃허브	https://github.com/yoouyeon/Peer-Frontend
팀 구성	프론트엔드 7명 / 백엔드 6명 / 디자이너 2명
개발 기간	2023.11 ~ 2024.03
기술 스택	TypeScript, Next.js, SWR, MUI
역할	· 프론트엔드 · 쪽지 기능과 팀게시판 기능 개발



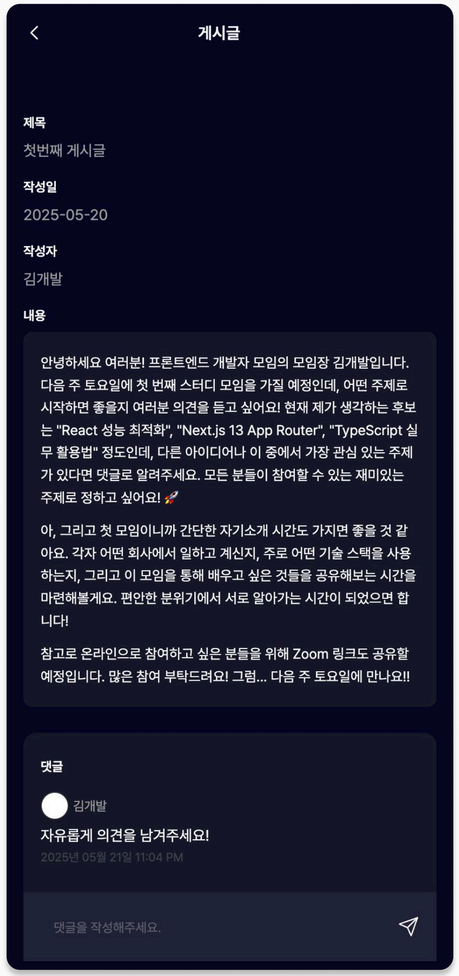
쪽지 목록 화면



쪽지 대화 화면



팀 게시판 화면



팀 게시글 화면

React-Testing-Library를 활용해 컴포넌트 테스트 환경을 구축했습니다.

Background

React-Testing-Library와 MSW를 활용해
백엔드 없이도 프론트엔드 개발과 테스트가 가능한 환경을 구축했습니다.

이 과정 중 zustand store의 초기값을 localStorage 값에 따라
설정하는 기능을 테스트해야 하는 상황이 있었습니다.

Problem

하지만 **localStorage 모킹이 테스트에 반영되지 않는 문제**가 있었습니다.

이 문제는 create 함수가 모듈 import 시점에 실행되어,
테스트 환경에서 모킹이 적용되기 전에 store가 먼저 생성되는 구조 때문이었습니다.

Solution

팩토리 함수를 구현해서 store 생성 시점을 제어하고, **생성 시점에 의존성 주입이 가능**하도록 리팩토링했습니다.

이를 통해서 모듈 import와 mock 적용 시점의 차이를 알게 되었고,
테스트 가능한 구조와 의존성 주입 설계의 중요성을 배울 수 있었습니다.

```
// 문제가 된 기존 코드
const useAuthStore = create<IAuthStore>((set) => {
  const authDataJSON = LocalStorage.getItem('authData')l
  // ...
});
```

▲ localStorage의 값을 모킹할 수 없었던 기존 코드

```
// 해결 : 팩토리 함수로 store 생성 시점 제어
export interface IDependencies {
  localStorage: typeof LocalStorage;
}

export const createAuthStore = (deps: IDependencies) => {
  const { localStorage } = deps;
  const authDataJSON = localStorage.getItem('authData');
  // ...
  return create<IAuthStore>(stateCreator);
}
```

▲ 팩토리 함수로 모킹한 의존성을 전달해 store를 생성하는 코드

▸ **관련해서 작성했던 블로그 글**
https://blog.yoouyeon.dev/zustand-store-initial-state-testing

감사합니다

복잡함을 단순하게 바꾸는 프론트엔드 개발자, 윤정연

yoouyeon.dev@gmail.com
010 - 2290 - 6150