

## 기술 스택

\* Next.js 16(App Router), Typescript, Zustand, Tanstack Query, Tailwind

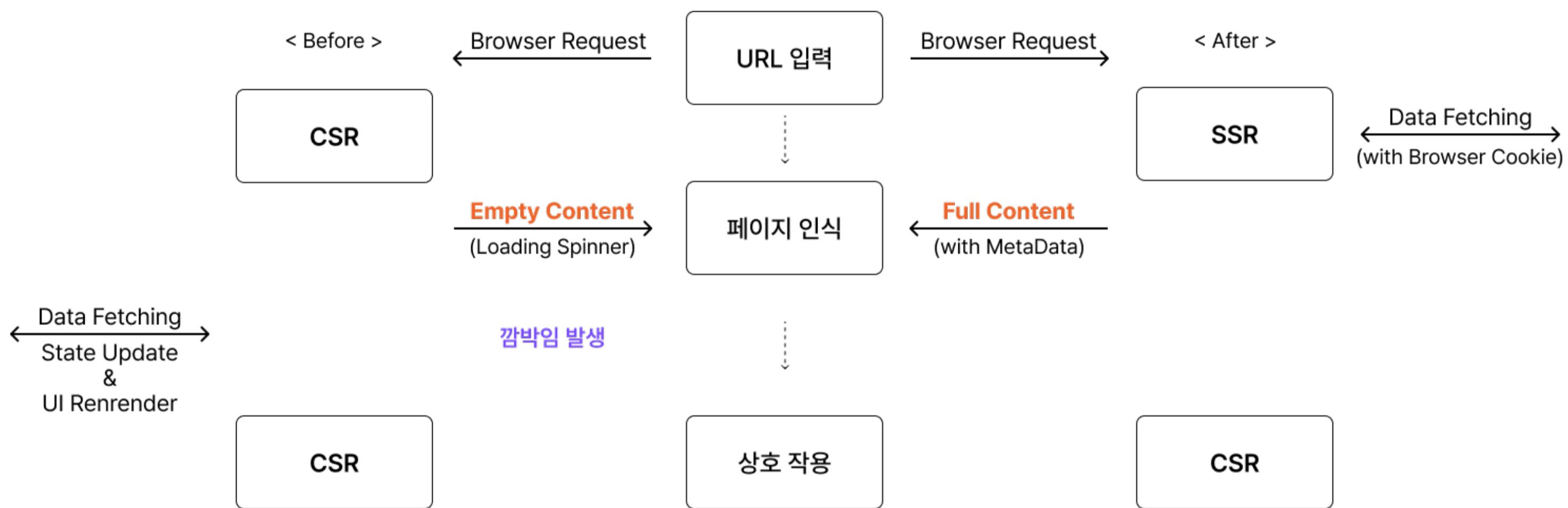
## 핵심 업무

- Next.js 하이브리드 렌더링 도입 및 SEO 최적화
- 프론트엔드 아키텍처 설계
- 기술적 제약 해결 및 생산성 향상

## 주요 인사이트

- 렌더링 방식은 단순한 기술 선택이 아니라 사용자 경험과 검색 노출 전략까지 결정하는 설계 요소임을 체감했습니다
- 서버 데이터는 캐시 전략으로 관리하고 UI 상호작용 상태는 전역 스토어로 분리함으로써 책임 경계를 명확히 하는 설계의 중요성을 체감했습니다.
- 프론트엔드로서 API 구조와 상태 흐름 설계에 적극적으로 참여해야 개발 생산성과 유지보수성이 향상된다는 점을 경험했습니다.

## 사용자 초기 경험과 검색 노출 문제를 해결하기 위한 Next.js 하이브리드 렌더링 설계



### [문제 상황]

- 페이지 초기 진입 시 빈 화면이 보이다가 데이터가 렌더링되는 UI 깜박임 발생으로 사용자 경험 저하
- 초기 렌더링 시 로딩 스피너가 필수적으로 노출되어 Layout Shift 발생 및 First Contentful Paint(FCP) 지표 악화
- 초기 HTML에 핵심 콘텐츠가 부족해 검색 엔진 봇이 콘텐츠를 수집하지 못하는 SEO 취약점이 존재하여 서비스 검색어 유입에 제한

### [해결 과정]

- 초기 HTML 단계에서 콘텐츠를 포함할 수 있도록 클라이언트 컴포넌트 대신 서버 컴포넌트를 사용하는 방식을 선택
- Next.js 서버 컴포넌트에서 초기 렌더에 필요한 데이터를 먼저 로드한 뒤 클라이언트에 초기 상태를 주입
- 사전 조회된 데이터를 활용해 각 페이지별 동적 메타데이터 생성, robots.txt 및 sitemap.xml 생성으로 검색 엔진 크롤링 효율화

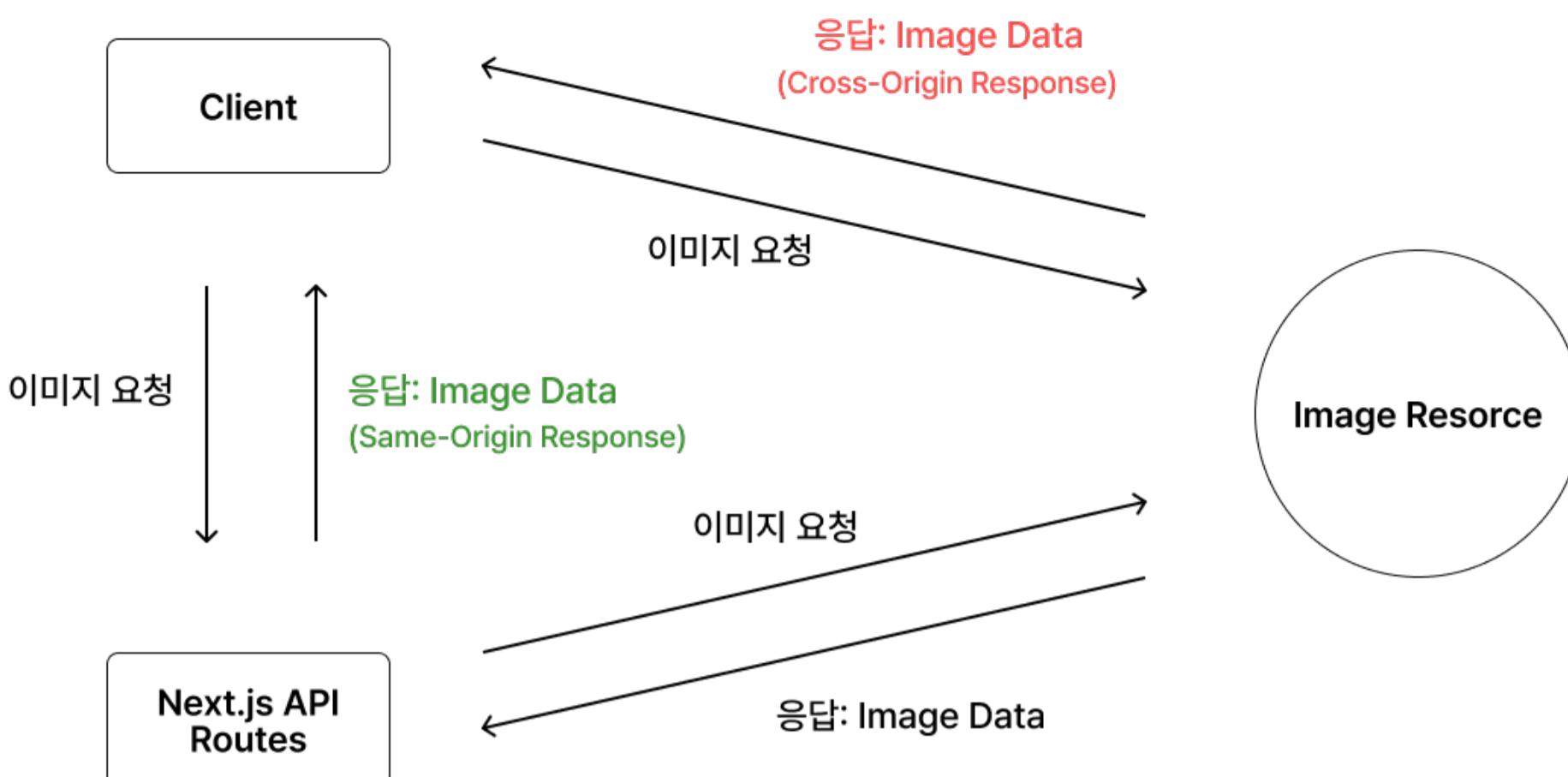
### [결과]

- 초기 렌더링 시 콘텐츠가 즉시 표시, 시각적 흔들림을 개선하여 사용자가 첫 화면에서 콘텐츠를 인지할 수 있도록 UX 개선
- 첫 렌더 HTML에 핵심 텍스트 콘텐츠가 포함되면서 검색 엔진이 페이지 주제를 더 명확히 해석할 수 있는 구조 확보
- 구글 검색 엔진에 "모두의 식물" 키워드로 1페이지 상위 노출 달성, 검색 유입 기반의 트래픽 확보에 기여

### [SSR 도입 과정에서 마주한 인증 상태 문제 해결]

- 서버 컴포넌트가 브라우저 메모리에 접근 불가한 제약으로 인해 인증 토큰을 쿠키 기반으로 재설계해 SSR 환경에 인증 상태 공유
- 서버 컴포넌트 호환을 위해 fetch 기반 공통 코어를 구축하고 Axios를 벤치마킹한 인스턴스 형태로 래핑
- `(server|client)ApiInstance.method<ResponseType>(endPoint, body, options)` 형태의 표준화된 인터페이스로 유지보수성과 생산성 향상

## ❷ CORS 정책 대응을 위한 서버 사이드 프록시 구현



### [ 문제 상황 ]

- 이미지 비중이 큰 화면에서 외부 도메인의 이미지 리소스를 직접 호출하면서 이미지가 깨지거나 로딩에 실패하는 현상이 발생
- 게시글 수정 시 이미지를 가공하는 과정(URL 리소스 → fetch → Blob 변환)에서 CORS 에러로 인한 이미지 재전송 불가능 문제 발생

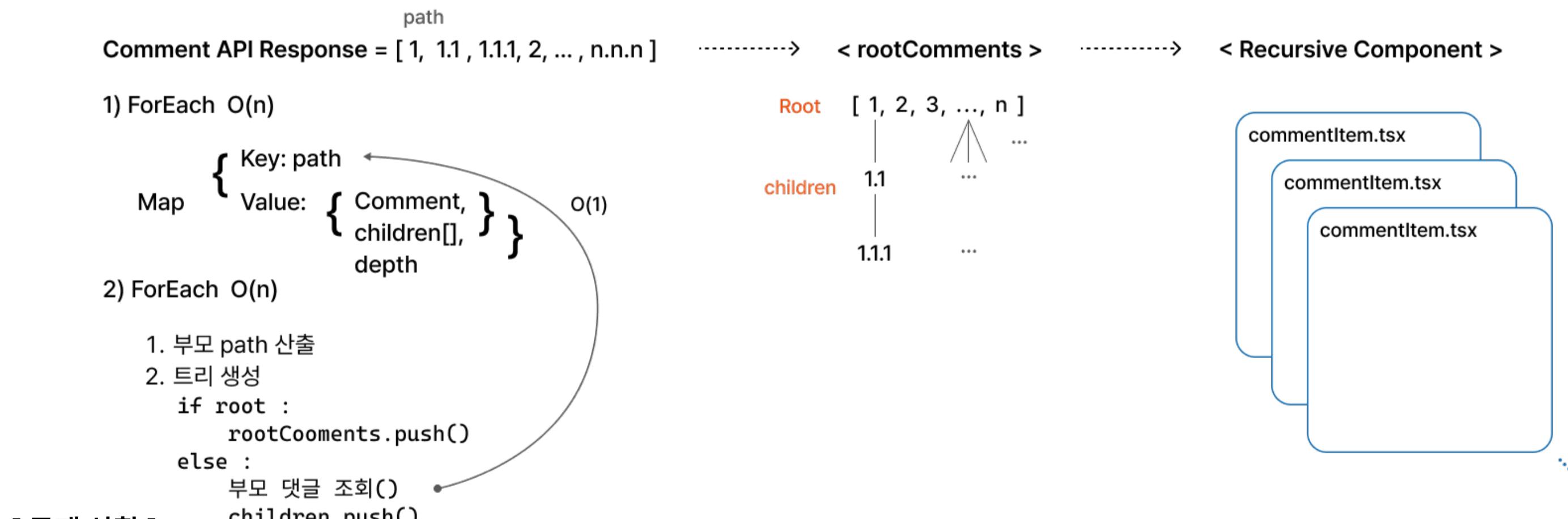
### [ 해결 과정 ]

- 별도의 프록시 서버를 구축하는 것은 인프라 비용 발생 및 배포/관리 복잡도가 증가한다고 판단
- 추가 인프라 없이 서비스처럼 동작하는 Next.js API Routes를 활용하기로 결정
- `/api/image-proxy?url={ImageURL}` 엔드포인트를 구현하여 클라이언트 요청을 서버 사이드에서 중계

### [ 결과 ]

- CORS로 인한 외부 이미지 로딩 실패 해소, 이미지가 필요한 화면에서 깨짐/빈칸 현상을 줄여 UX 안정화
- 게시글 수정 로직이 정상 작동하여 사용자가 이미지를 재업로드하지 않아도 기존 이미지를 유지하며 수정할 수 있는 UX 완성

## ❸ Path 기반 플랫 댓글 데이터를 트리 구조로 변환한 무한 Depth 댓글 설계



### [ 문제 상황 ]

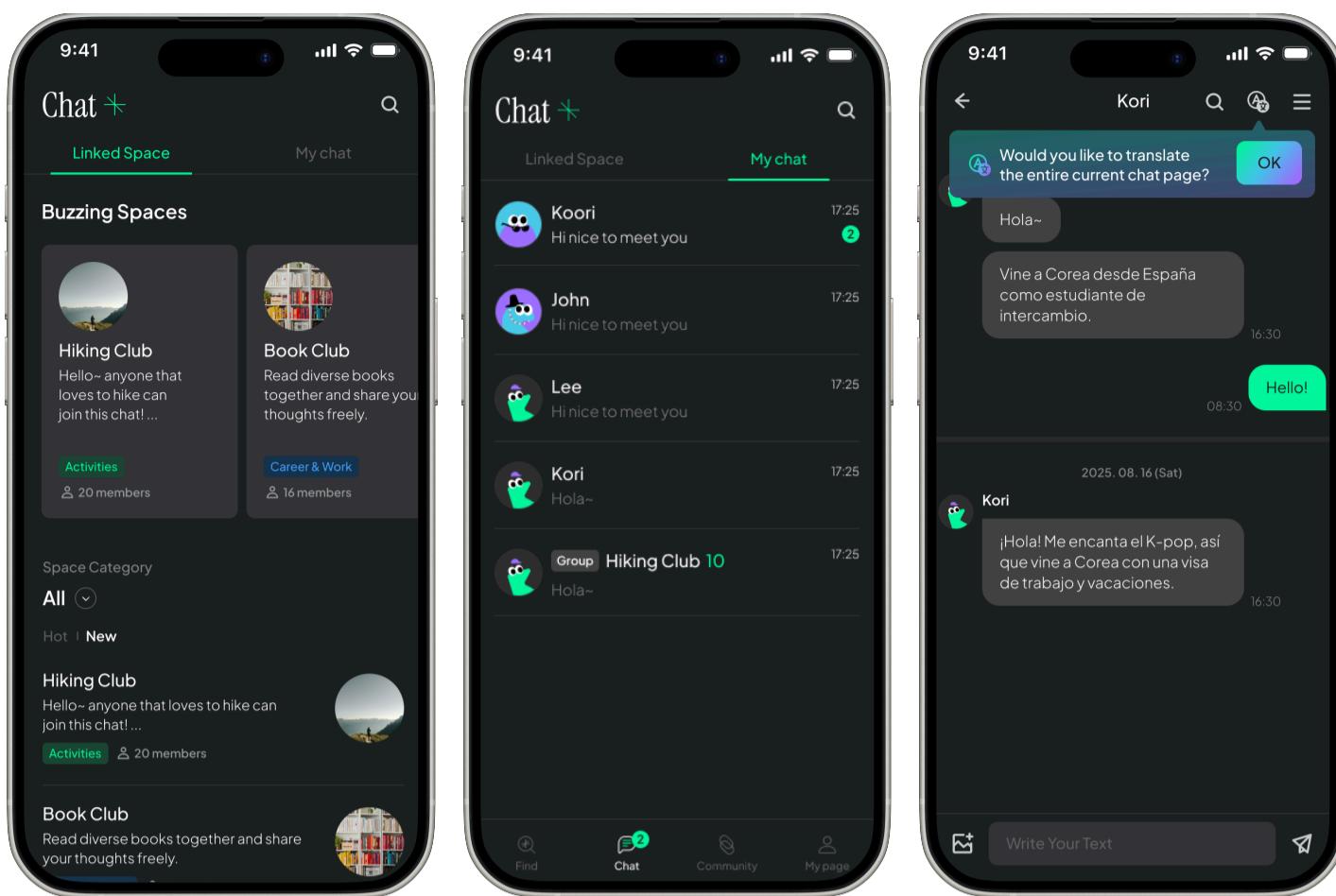
- 댓글 API가 path 기반의 플랫 배열로 응답하지만, UI는 들여쓰기와 담글의 답글을 표현하는 계층 구조(트리)로 렌더링해야 하는 데이터 구조 불일치 문제
- 각 댓글마다 부모를 선형 탐색하는 구조로 구현할 경우 전체 시간 복잡도가  $O(n^2)$ 로 증가해 댓글 수가 증가할수록 성능 저하가 발생하는 구조적 한계

### [ 해결 과정 ]

- 첫 번째 순회에서 path를 key로 하는 Map을 구성하여 부모 댓글을  $O(1)$ 로 조회할 수 있는 기반을 마련  
→ 두 번째 순회에서 부모 path를 계산한 뒤 Map을 통해 부모 노드를 즉시 조회하여 children에 연결  
→ Depth가 1인 rootComments 반환
- 렌더링 시 rootComments를 순회, children이 있는 경우 자기 자신을 호출하는 재귀 컴포넌트(Recursive Component) 패턴을 도입

### [ 결과 ]

- 깊이 제한 없는 댓글 시스템의 시간 복잡도를  $O(n^2)$ 이 아닌  $O(n)$ 으로 구현
- 재귀 컴포넌트 패턴을 통해 단 하나의 컴포넌트로 모든 Depth를 커버하여 Depth 증가에 따른 UI 로직 복잡도를 최소화



## 서비스

[ 외국인 대상 K-Culture 중심 커뮤니티 ]

## 구성원

총 7인 ( FE 3인 / BE 3인 )

## 기술 스택

\* React Native(Expo 53), Typescript, Zustand, Tanstack Query, WebSocket(STOMP.js)

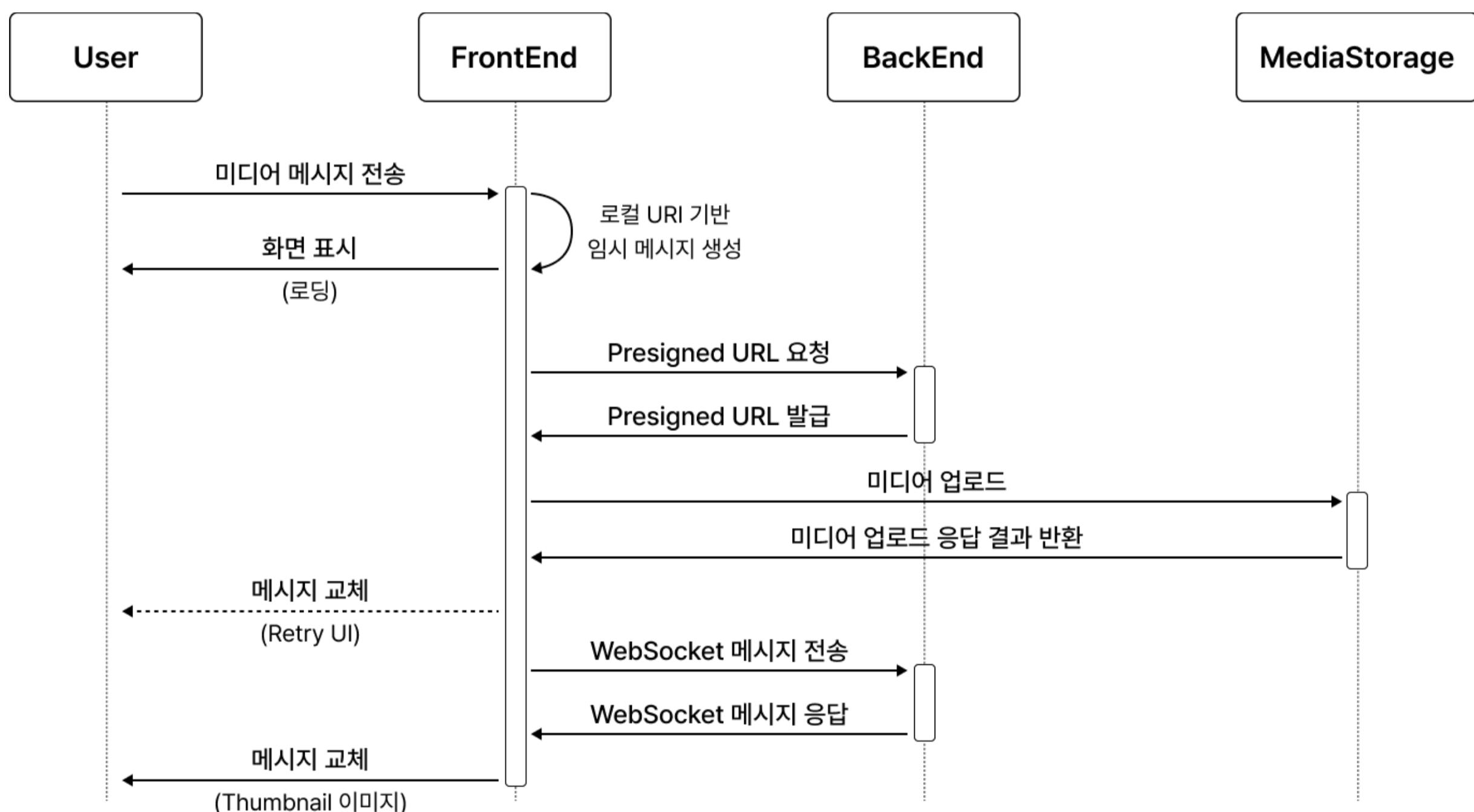
## 핵심 업무

- STOMP 기반 실시간 채팅 시스템 고도화
- 데이터 정합성 확보 및 성능 최적화
- 코드 품질 및 서비스 지표 개선

## 주요 인사이트

- 네트워크 지연을 줄이는 것보다 사용자가 지연을 인지하지 않도록 설계하는 것이 더 중요하다는 점을 경험했습니다.
- 기능이 늘어날수록 컴포넌트와 로직이 뒤엉기는 구조적 위험을 경험했고 이를 계층 분리와 구조 재설계로 해결하며 확장 가능한 프론트엔드 아키텍처의 중요성을 체감했습니다.
- 기술적 개선이 사용자 행동 변화와 서비스 지표 개선으로 이어질 때 비로소 의미를 가진다는 점을 경험했습니다.

## 즉각적 피드백을 제공하는 미디어 메시지 업로드 흐름 재설계



## [ 문제 상황 ]

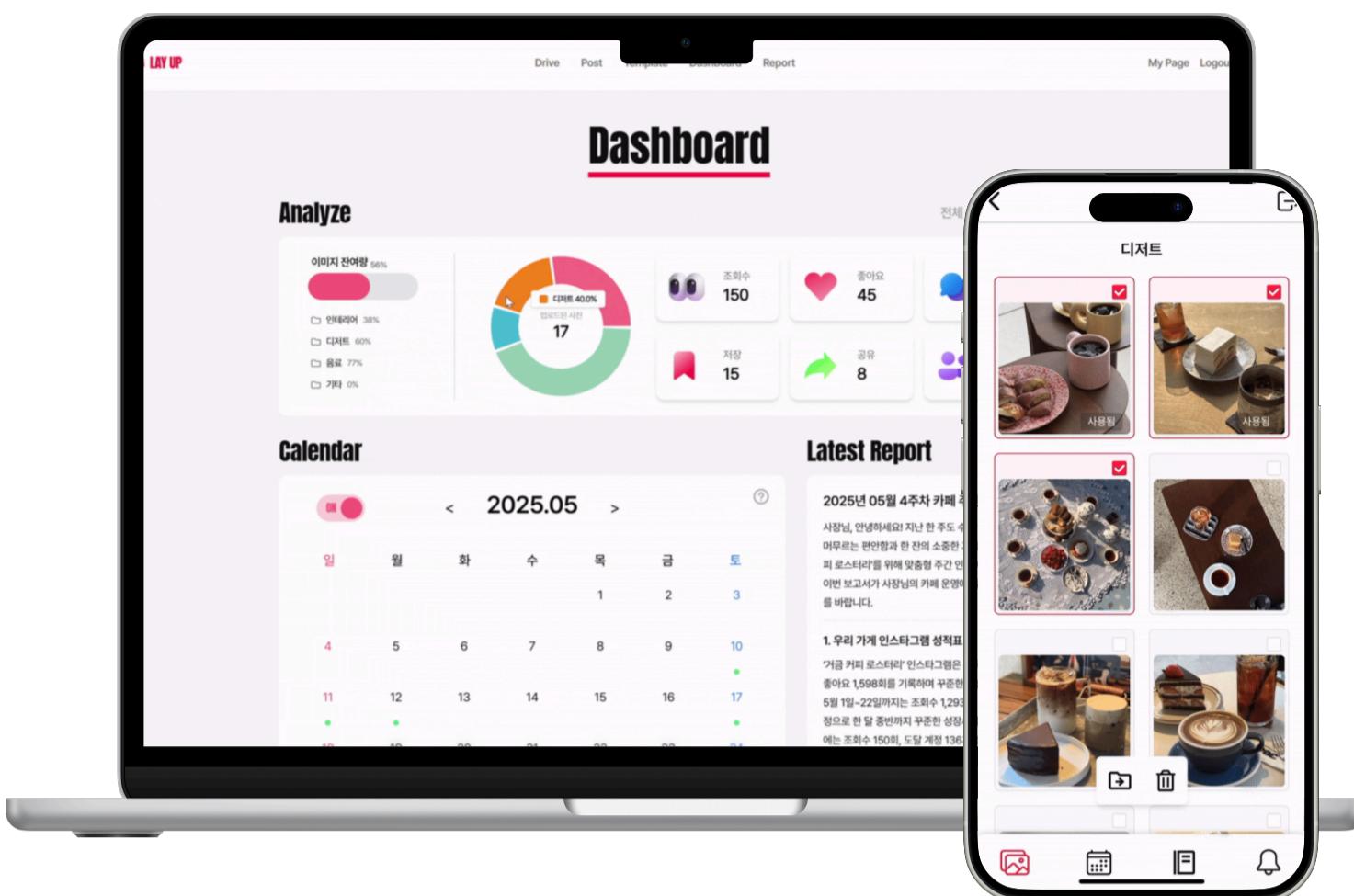
- 사용자 액션에 대한 즉각적인 시각적 피드백이 제공되지 않는 UX 공백 발생
- 미디어 파일이 서버를 경유하여 스토리지로 업로드되면서 서버 네트워크 대역폭 및 메모리 과다 소비

## [ 해결 과정 ]

- 사용자 니즈는 즉각 업로드가 아닌 즉각 피드백이라 판단하고 UI 응답성 개선을 위해 미디어 메시지 전송에 Optimistic Update 패턴을 적용
- 서버 부하 감소를 위해 Presigned URL 기반 직접 업로드 방식을 도입하여 클라이언트가 스토리지에 직접 업로드
- uploadStatus 필드를 통해 'pending' → 'uploading' → 'success/failed' 순으로 상태를 관리하여 각 단계마다 사용자에게 시각적 피드백을 제공
- 업로드 완료 이벤트를 실시간으로 반영하기 위해 WebSocket 기반 메시지 수신 시 tempId를 기준으로 임시 메시지를 실제 메시지로 교체

## [ 결과 ]

- 미디어 메시지 전송 직후 화면에 UI가 나타나는 "즉시성"을 제공하여 체감 지연을 제거
- 미디어 파일 업로드 중에도 다른 메시지 입력과 스크롤이 가능한 논블로킹 인터랙션 구현
- 백엔드 서버의 대용량 파일 중계 부담을 제거하여 미디어 트래픽이 많은 상황에서의 서버 리소스 사용과 응답성 악화 가능성 개선



## 서비스

[ 소상공인의 SNS 홍보를 돋기 위한 마케팅 서비스 ]

## 구성원

총 6인 ( FE 3인 / BE 3인 )

## 기술 스택

\* Next.js 13(App Router), Typescript, Redux, Tanstack Query, Tailwind, PWA

## 핵심 업무

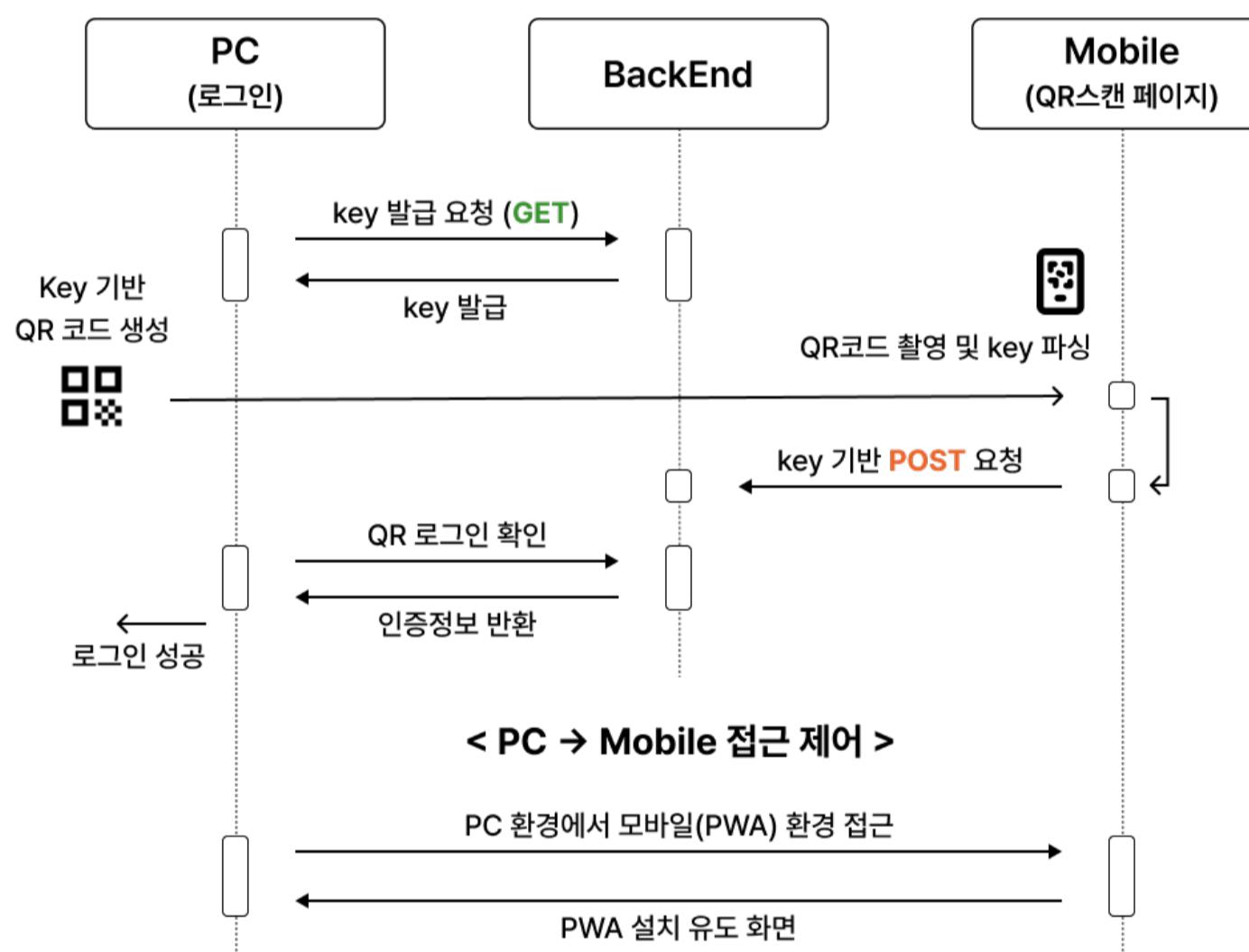
- 기기별 라우팅 및 서버 렌더링 최적화
- API 호출 패턴 및 QR 로그인 플로우 설계
- 데이터 시각화 및 기술 문서화 주도

## 주요 인사이트

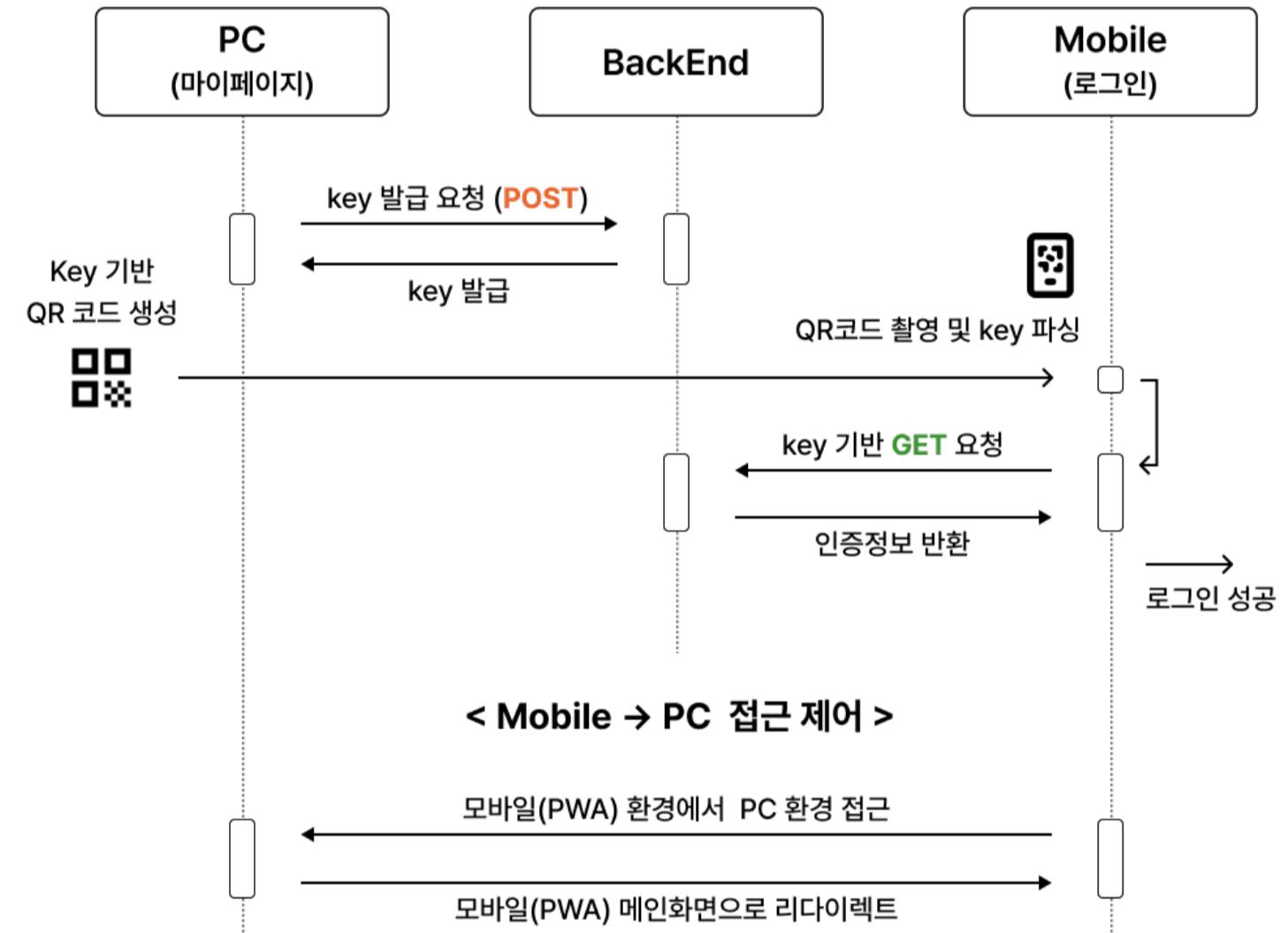
- 프론트엔드 또한 단순 UI 계층이 아니라 인증, 통신, 상태 흐름을 설계하는 아키텍처 영역임을 경험했습니다.
- 기술 구현 이전에 사용자 진입 경로와 사용 맥락을 고려하는 설계가 서비스 완성도를 좌우한다는 점을 학습했습니다.
- 문제-해결-대안 구조의 기술 문서화를 통해 구현 경험을 팀의 공통 자산으로 전환하는 과정의 중요성을 경험했습니다.

## 양방향 QR 로그인 &amp; User-Agent 기반 라우팅을 통한 PWA 사용성 개선

## &lt; PC 미로그인, Mobile 로그인 &gt;



## &lt; PC 로그인, Mobile 미로그인 &gt;



## [ 문제 상황 ]

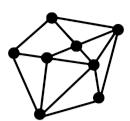
- PC와 모바일(PWA) 환경이 독립적으로 운영되면서 사용자가 매번 별도 로그인을 해야 하는 불편함 발생
- 사용자가 환경에 맞지 않는 URL로 접근할 경우 의도하지 않은 레이아웃이 노출되어 UX 혼란 및 서비스 신뢰도 저하 우려

## [ 해결 과정 ]

- 백엔드와 협업하여 양방향 QR 로그인 플로우 설계
  - QR 키 발급 → 간접 → 검증 3단계 API 스펙 직접 협의 및 문서화 (POST/GET 분기 처리, 키 TTL 설정 등)
  - PC → 모바일 로그인, 모바일 → PC 로그인 양방향 시나리오 모두 지원
- User-Agent 기반 라우팅 로직 구현
  - PC 환경에서 모바일 환경 접근 시, PWA 설치 유도 페이지 노출
  - 모바일 환경에서 PC 환경 접근 시, 모바일 홈 화면으로 리다이렉트

## [ 결과 ]

- 복잡한 아이디/비밀번호 입력 없이 QR 스캔만으로 기기 간 세션을 동기화
- 환경별 독립적인 라우팅 처리를 통해 PC/모바일 코드 베이스의 간섭을 줄이고 유지보수성을 확보



## 서비스

[ 실시간 알고리즘 코딩 배틀 플랫폼 서비스 ]

## 구성원

총 6인 ( FE 3인 / BE 3인 )

## 기술 스택

\* React 18, Typescript, Zustand, Tailwind, WebSocket(STOMP.js)

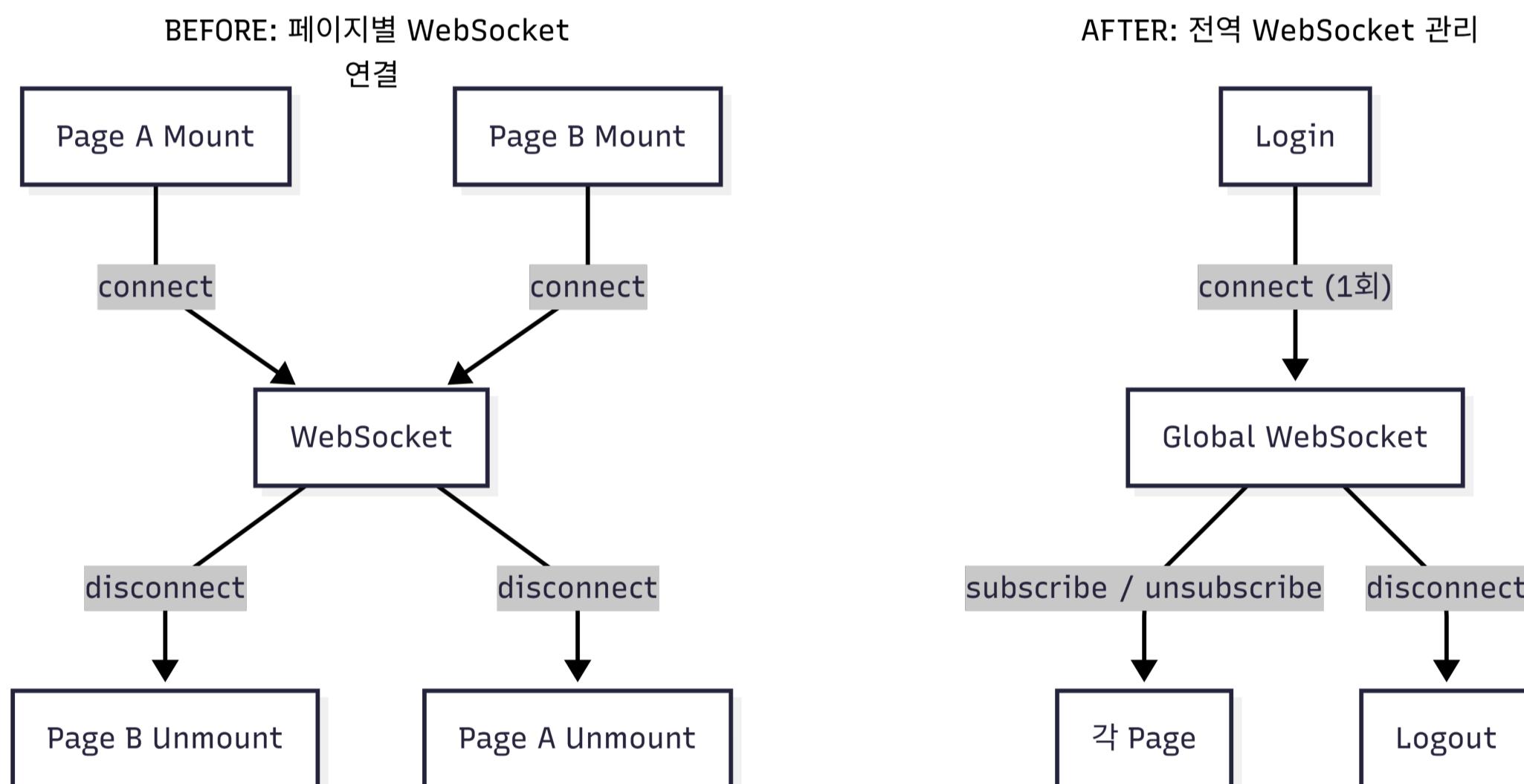
## 핵심 업무

- STOMP 기반 실시간 통신 및 리소스 최적화
- 데이터 흐름 일원화 및 동기화 정밀도 향상
- 복잡한 렌더링 로직의 추상화 및 효율화

## 주요 인사이트

- 실시간 서비스의 안정성은 클라이언트 상태와 서버 이벤트 간의 명확한 경계 설계에서 시작됨을 경험했습니다.
- UI 렌더링 레이어와 비즈니스 로직 레이어를 엄격히 분리하지 않으면 데이터 정합성과 사용자 경험 모두 무너질 수 있다는 점을 실감했습니다.

## WebSocket 연결 단일화 및 모듈화로 효율성과 안정성 확보



## [ 문제 상황 ]

- 페이지 전환 시마다 반복되는 WebSocket 핸드셰이크와 연결 해제로 인해 불필요한 네트워크 오버헤드가 발생
- 네트워크 일시 단절 시의 재연결 전략 부재 및 브라우저 새로고침 시 기존 데이터 구독 상태가 유실되는 휘발성 구조

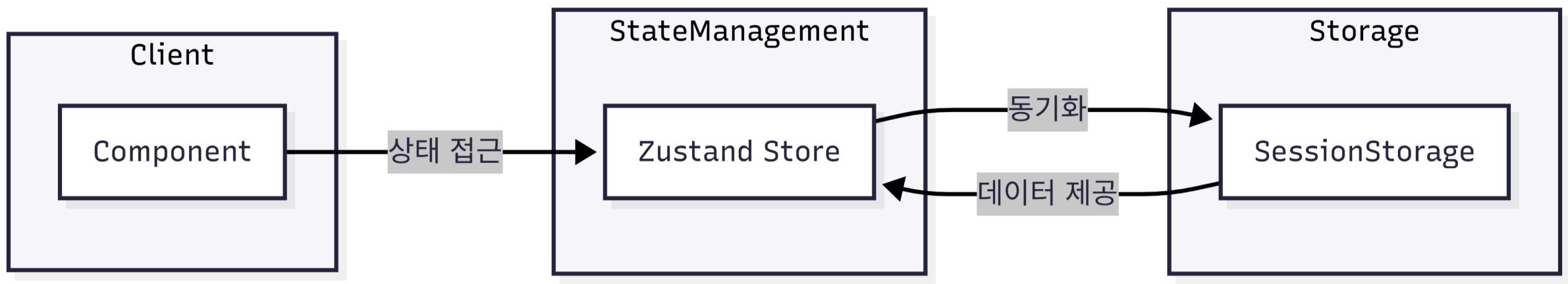
## [ 해결 과정 ]

- 로그인 시 단일 WebSocket을 생성하고 로그아웃 시 해제하는 전역 연결 구조로 전환
- 구독 등록·해제 로직을 모듈화하고 Map 자료구조를 활용해 중복 구독을 방지
- sessionStorage 기반의 상태 영속화로 새로고침 후에도 끊김 없는 데이터 흐름 유지
- 자동 재연결 및 최대 재시도 횟수 제한 정책을 도입해 네트워크 예외 상황 대응 체계 수립

## [ 결과 ]

- 중복 구독 방지와 중앙 핸들러 기반 메시지 통합 처리 구조 확립
- 불필요한 핸드셰이크 제거를 통한 연결 안정성 확보 및 리소스 효율성 개선
- 경로 기반 메시지 필터링과 자동 정리를 통한 메모리 누수 예방 및 유지보수성 향상

## 상태 접근 일원화로 실시간 데이터 안정성 확보



### [ 문제 상황 ]

- useEffect 기반의 비동기 로직과 WebSocket 수신 시점의 차이로 인해 매칭 인원 갱신 및 방장 권한 위임 기능의 오작동 발생
- Zustand 스토어와 개별 컴포넌트가 sessionStorage에 교차 접근하며 발생하는 데이터 정합성 결함 및 실시간 동기화 오류

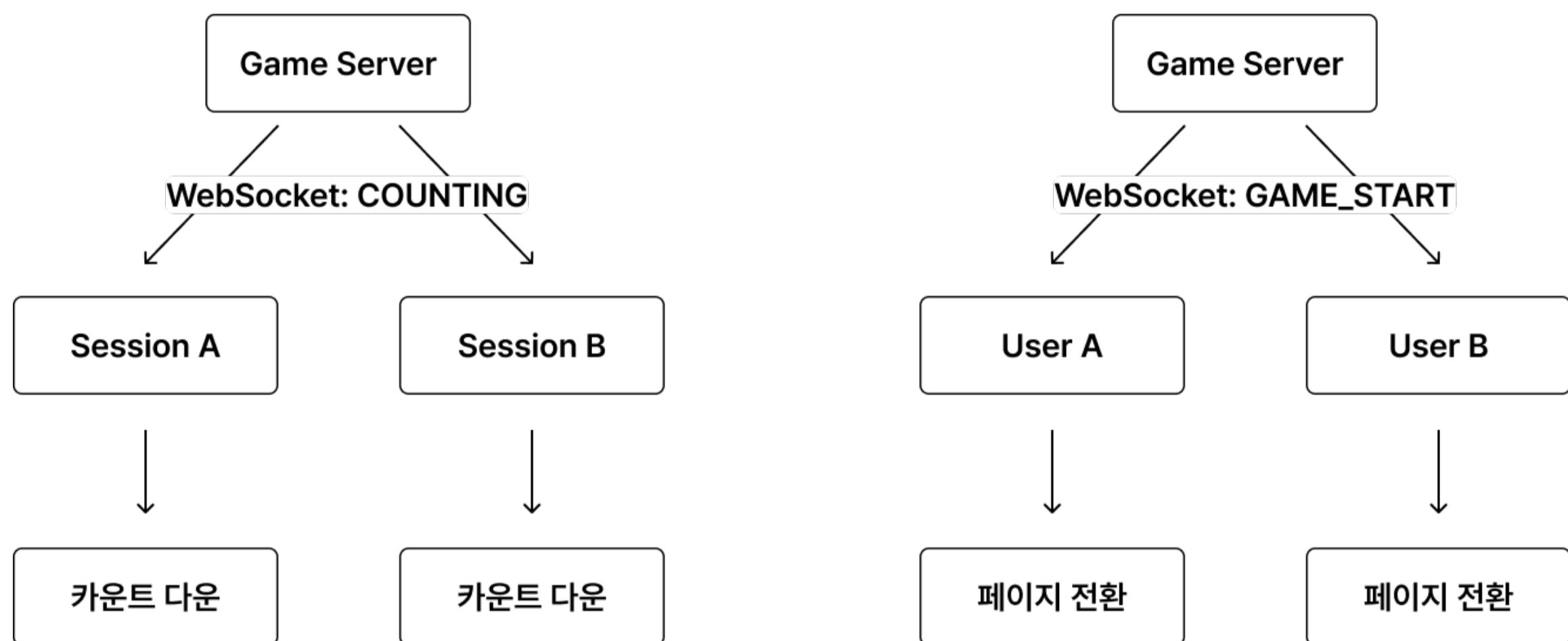
### [ 해결 과정 ]

- 단일 진실 공급원(SSOT) 구축
  - 모든 컴포넌트의 상태 접근 경로를 Zustand로 일원화
  - 외부 스토리지(SessionStorage)와의 동기화 로직을 스토어 내부로 캡슐화하여 데이터 흐름 단일화

### [ 결과 ]

- 상태 접근 로직 통합을 통해 매칭 인원 및 방장 상태 동기화 오류를 완전히 해결하고 실시간 데이터 처리 안정성 확보
- 상태 변경 추적 지점을 스토어 내부로 한정함으로써 문제 발생 시 원인 파악 및 코드 수정이 용이한 유지보수 환경 조성

## 서버 소켓 이벤트 기반 게임 시작 타이밍 동기화



### [ 문제 상황 ]

- 클라이언트 기반 타이머(setTimeout) 사용 시 개별 유저의 기기 성능 및 네트워크 지연에 따라 게임 진입 시점이 달라지는 불공정성 발생
- 탭 비활성화(Background Tab) 시 리소스 절약을 위해 타이머 정밀도가 떨어지는 브라우저 특성상 일관된 카운트다운 동기화 불가능

### [ 해결 과정 ]

- 카운트다운과 게임 시작 시점을 서버의 소켓 이벤트 수신 시점으로 동기화하여 클라이언트 간 편차 제거

### [ 결과 ]

- 모든 유저가 동일한 서버 이벤트 수신 시점에 게임에 진입하게 되어 시작 타이밍의 공정성을 보장
- 브라우저 탭 활성화 여부 등 외부 변수로부터 자유로운 안정적인 게임 전이 프로세스 확보

### [ 이벤트 동기화 설계의 개선 방향 도출 ]

- 매초 소켓 메시지를 전송하는 방식은 유저가 많아질수록 서버 네트워크 오버헤드가 증가할 수 있음을 인지
- 서버는 시작 시각만 전달하고 클라이언트가 이를 계산해 동기화하는 방식이 더 효율적인 설계임을 학습