강민재 프론트엔드 개발자

010-5519-7425 · towndev@naver.com · https://minjaekang.dev/

대학교를 휴학하고 창업에 도전하며 서비스를 직접 구현하기 위해 개발을 시작했습니다. 이후 오토바이 정비·라이딩 플랫폼 라이더타운과 협업 여행 플랫폼 저스트트립을 기획하고, React·Next.js·React Native 기반의 웹과 앱으로 개발해 출시·운영했습니다.

화면과 서버의 상태가 어긋나는 지점을 찾아 구조로 해결해 왔고, 증상을 고치기 전에 문제를 다시 정의하고 어떤 대안을 왜 버렸는지 설명할 수 있는 결정을 중요하게 생각합니다.

경력 신입

저스트트립 AI 기반 크로스플랫폼 협업 여행 일정·기록 서비스 2026.03 - 2026.10

서비스 기획 · 프론트엔드 개발 · 백엔드 개발자 1명과 2인 협업

- 여행 상세 화면 First Load JS 828kB → 452kB, 45% 절감, LCP 28%·TBT 32% 개선 및 마커 클릭 시 일정 목록 재렌더링 제거 (사례 1)
- 낙관적 삭제 도입과 삭제 상태 3단계 추적으로 지운 항목이 되살아나는 문제 해결, 7개 도메인에 공통 구조로 적용 (사례 2)

Next.js, React Native (Expo), TypeScript, TanStack Query, Redux Toolkit

얼리버드 알람 알람으로 광고 음원을 듣고 리워드 포인트를 받아 기프티콘으로 교환하는 알람 앱 2026.05 - 2026.10

기획 · 풀스택 개발

- iOS AlarmKit 제약 안에서 잠금 상태의 볼륨·정지 버튼에도 다시 울리는 알람 구조 재설계 (사례 3)

React Native (Expo), TypeScript, iOS AlarmKit, Firebase Cloud Functions, Firestore

라이더타운 V2 라이딩 기록 서비스 피벗 & 서버리스 전환 2024.06 - 2024.12

공동창업 · 풀스택 개발 · 2인 팀

- Firebase Analytics로 라이딩 화면의 리텐션이 정비 예약보다 10배 이상 높은 것을 확인하고 서비스 피벗
- 백그라운드에서도 끊기지 않는 실시간 크루 위치 공유 구축, 경로 전송량 70% 이상 절감 (사례 4)
- AWS MSA를 Firebase 서버리스로 전환해 월 고정 인프라 비용 90% 이상 절감

React Native, Firebase, react-native-background-geolocation, Mapbox

라이더타운 V1 오토바이 정비 플랫폼 창업 · 라이딩 기록 및 그룹 음성 통화 2022.11 - 2023.12

공동창업 · 풀스택 개발 · 3인 창업팀

- React SPA를 Next.js SSR로 전환해 초기 번들 2.5MB → 550KB, 웹뷰 첫 화면 3.5초 → 0.8초 개선
- 웹소켓 1초 주기 크루 주행 데이터 동기화와 Mediasoup SFU 기반 4인 음성 통화 구현

React, Next.js, React Native WebView, WebSocket, Mediasoup v3

기술 역량

- Frontend: React, Next.js, TypeScript, TanStack Query, Redux Toolkit, Tailwind CSS
- Mobile: React Native, Expo, React Native WebView
- Realtime & Backend: WebSocket, WebRTC, Firebase, Node.js, Spring Boot, Redis

수상 · 학력

청년창업사관학교 (창업성공패키지) 선정 및 수료 2024.03

예비창업패키지 선정 및 수료 2023.05

광운대학교 전자공학과 학사 졸업 2018.03 - 2026.08

사례 1 · 저스트트립

저스트트립

친구와 함께 여행을 계획하고 기록하는 협업 여행 서비스입니다. 여행, 바우처, 체크리스트, 항공, 숙소, 일정, 예산, 지출, 사진을 웹과 앱에서 함께 관리합니다.

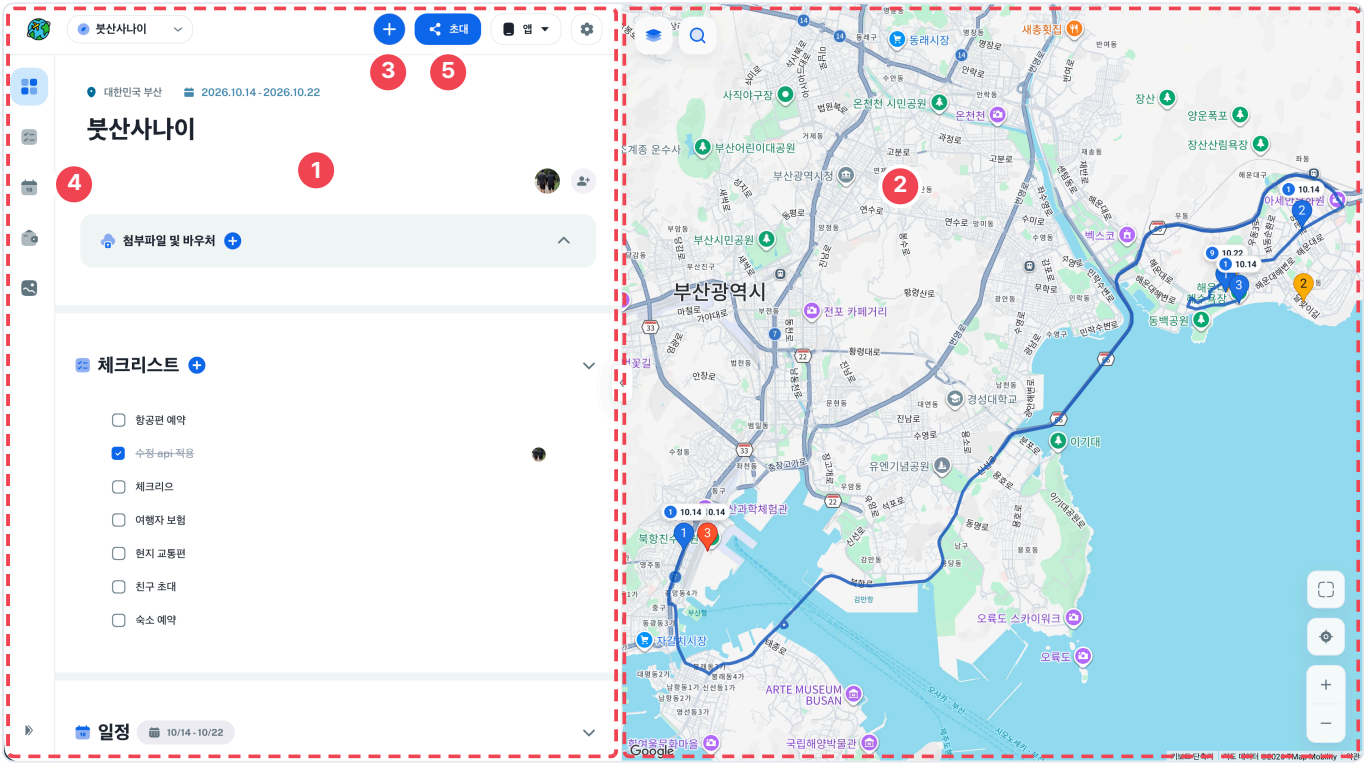
2026.03 - 2026.10 · 서비스 기획, 프론트엔드 개발 · 백엔드 개발자 1명과 2인 협업 · justtripplanner.com

여행 상세 화면의 First Load JS 45% 절감과 지도 조작 시 불필요한 재렌더링 제거

Next.js, next/dynamic, React.memo, requestIdleCallback

문제 정의

저스트트립 웹의 여행 상세 화면에서는 여행 기본 정보와 바우처, 체크리스트, 항공, 숙소, 일정, 예산, 지출, 사진을 왼쪽 패널과 오른쪽 지도에서 함께 관리합니다. 항공·일정 등의 상세보기와 등록을 위한 독립적인 Dialog 11개가 화면 위로 올라오고, 체크리스트·일정·바우처·예산·사진 섹션 5개와 지도 오버레이 2개가 상세 화면 안에 함께 배치된 구조입니다.



저스트트립 웹의 여행 상세 화면

1. **왼쪽 패널** 체크리스트, 일정, 바우처 등 여행 정보를 보여 줍니다.
2. **오른쪽 지도** 일정의 장소와 경로를 표시합니다. 마커 선택 등으로 지도를 조작하면 왼쪽 일정 목록까지 다시 렌더링되던 영역입니다.
3. **새로 등록** 항공·일정 등을 등록하는 11개의 독립적인 Dialog 중 하나입니다.
4. **탭 메뉴** 일정·지출·사진 등의 상세 섹션으로 이동합니다. 이 상세 섹션과 지도 오버레이 7개도 초기 번들에 포함돼 있었습니다.
5. **초대** 동행자를 초대하는 Dialog입니다. 이런 독립적인 Dialog 11개가 열림 여부와 관계없이 초기 렌더 트리에 있었습니다.

여행 상세 화면은 사용자가 일정을 짜면서 지도와 패널을 오가는 이 서비스의 핵심 화면입니다. 그런데 화면에 들어올 때 로딩이 길었고, 지도에서 마커를 선택하면 조작이 버벅였습니다. 일정을 짜는 동안 계속 반복되는 동작이라 그대로 둘 수 없었습니다.

원인을 확인해 보니 두 가지였습니다.

- 첫 로딩: 다이얼로그 11개와 상세 섹션·오버레이 7개, 총 18개 UI가 초기 번들에 포함돼 First Load JS가 828kB였습니다.
- 지도 버벅임: 지도 조작으로 부모 화면이 다시 렌더링될 때, 지도와 무관한 일정 목록까지 함께 다시 그려졌습니다.

그래서 "닫혀 있는 UI가 첫 화면 로딩에 비용을 내고 있다"와 "지도의 상태 변경이 무관한 목록까지 전파된다"는 두 문제로 나눠 접근했습니다.

판단

> **Dialog 구조와 마운트 시점을 어떻게 설계할까**
>
> **기각** 단일 Dialog에서 콘텐츠 교체: 중첩 흐름과 독립적인 입력 상태를 위해 별도의 화면 스택·복원 로직이 필요
>
> 여행 일정 편집에서는 항공권 입력 도중 바우처 분석 Dialog를 중첩해서 열거나, 바우처 분석 중 항공권 편집 Dialog를 호출하는 양방향 전환이 필요했습니다. 단일 Dialog에서 콘텐츠를 교체하면 각 흐름의 입력 상태를 독립적으로 유지하기 위해 화면 스택과 상태 복원 로직을 별도로 구현해야 했습니다.

```
// 중첩 전환마다 화면 스택과 각 입력 상태를 따로 보존해야 함
const [dialogStack, setDialogStack] = useState<DialogScreen[]>([]);
const current = dialogStack.at(-1);

<Dialog open={dialogStack.length > 0}>
  {current === 'flight' ? <FlightForm /> : <VoucherAnalysis />}
</Dialog>
```

> **기각** next/dynamic만 적용: 닫혀 있어도 렌더 트리에 있으면 청크를 내려받음
>
> dialog는 open 값과 관계없이 항상 렌더 트리에 있습니다. next/dynamic은 컴포넌트가 렌더링될 때 청크를 요청하므로, 코드는 분리되어도 사용자가 열지 않은 dialog까지 화면에 들어오자마자 받게 됩니다.

```
const FlightDialog = dynamic(loadFlightDialog, { ssr: false });

// open이 false여도 컴포넌트가 렌더링되어 청크를 요청
<FlightDialog open={flightOpen} onClose={handleClose} />
```

> **기각** 열 때 마운트·닫을 때 언마운트: 닫힘 애니메이션이 끊기고 입력 내용이 사라짐
>
> 청크는 필요할 때만 받습니다. 하지만 flightOpen이 false가 되는 순간 컴포넌트가 바로 제거되어 닫힘 애니메이션이 재생될 시간이 없고, 작성 중이던 입력값도 함께 사라집니다.

```
// 닫는 순간 컴포넌트가 트리에서 바로 사라짐
{flightOpen && <FlightDialog open onClose={handleClose} />}
```

> **선택** 처음 열릴 때 마운트하고 이후 유지하는 LazyMount: 초기 번들에서는 제외하고 상태와 애니메이션은 유지
>
> 양방향 중첩 흐름과 독립적인 입력 상태를 유지하기 위해 각 Dialog는 독립적인 컴포넌트로 유지했습니다. 초기 로딩 비용은 next/dynamic과 LazyMount를 조합해 줄였습니다.
>
> dialog의 상태를 "열림/닫힘" 둘이 아니라 "아직 안 열림 / 열림 / 닫힘" 셋으로 봤습니다. 아직 안 열린 dialog는 그릴 필요가 없지만, 한 번 열린 dialog는 닫힘 애니메이션과 입력 내용을 위해 남아 있어야 합니다.
>
> LazyMount는 이 차이를 hasMounted 플래그 하나로 표현합니다. 처음 열리기 전에는 null을 반환해 next/dynamic 컴포넌트가 렌더링되지 않고, 그래서 청크도 받지 않습니다. 한 번 연 dialog는 메모리에 남지만, 11개가 모두 쌓여도 기존보다 나빠지지 않아 감수했습니다.

```
export function LazyMount({ when, children }: Props) {
  const [hasMounted, setHasMounted] = useState(when);

  useEffect(() => {
    if (when) setHasMounted(true);
  }, [when]);

  return hasMounted || when ? <>{children}</> : null;
}

// 사용: 처음 열릴 때까지 FlightDialog 청크를 받지 않음
const FlightDialog = dynamic(loadFlightDialog, { ssr: false });

<LazyMount when={flightOpen}>
  <FlightDialog open={flightOpen} onClose={handleClose} />
</LazyMount>
```

분할로 생기는 첫 클릭 지연

기각 화면 마운트 직후 모든 Dialog 청크 프리페치: 초기 로딩 리소스와 네트워크 경쟁

화면이 마운트되자마자 모든 Dialog 청크를 받으면 첫 클릭 지연은 줄일 수 있습니다. 하지만 사용 여부와 관계없이 11개 청크 요청이 즉시 시작되어 초기 데이터와 이미지 다운로드에 필요한 네트워크를 함께 사용합니다. 번들을 나눴지만 다운로드 시점이 초기 진입 직후에 몰리기 때문에 첫 화면의 실제 로딩 부담을 충분히 분리하지 못했습니다.

```
// 화면이 마운트되자마자 모든 Dialog 청크를 요청
useEffect(() => {
  prefetchDialogs();
}, []);
```

기각 hover 시점 프리페치: 모바일에는 hover가 없음

마우스를 올리는 순간은 클릭 의도를 알 수 있는 좋은 신호입니다. 하지만 터치 기기에는 hover가 없어, 모바일 웹 사용자에게는 첫 클릭 지연이 그대로 남습니다.

```
// 터치 기기에서는 호출되지 않음
<Button onMouseEnter={() => loadFlightDialog()} onClick={handleAddFlight}>
  항공 추가
</Button>
```

선택 첫 로딩 후 requestIdleCallback 프리페치, 미지원 브라우저는 2초 지연 setTimeout으로 대체

초기 로딩 직후 바로 받으면 핵심 리소스와 경쟁하고, hover만 사용하면 모바일을 처리할 수 없습니다. 그래서 데이터 로딩이 끝난 뒤 브라우저가 한가해진 시점에 Dialog 청크를 프리페치했습니다.

next/dynamic과 프리페치가 같은 import 함수를 쓰게 해, 미리 받은 청크를 dialog가 그대로 재사용합니다. 총 다운로드 양을 줄인 것이 아니라 받는 시점을 첫 화면 뒤로 미룬 것입니다.

```
const loadFlightDialog = () =>
  import('../dialogs/flight-dialog').then((m) => m.FlightDialog);

const prefetchDialogs = () => {
  [loadFlightDialog, loadScheduleDialog /* ... */].forEach((load) => {
    load().catch(() => {});
  });
};

// 첫 화면 로딩이 끝나면 dialog 청크를 유휴 시간에 미리 받음
useEffect(() => {
  if (loading) return undefined;
  const idle = (window as any).requestIdleCallback;
  if (idle) {
    const handle = idle(prefetchDialogs);
    return () => (window as any).cancelIdleCallback?.(handle);
  }
  const timer = setTimeout(prefetchDialogs, 2000);
  return () => clearTimeout(timer);
}, [loading]);
```

지도 선택 상태가 바뀔 때 일정 목록의 재렌더링을 어떻게 막을까

지도 마커를 선택하면 handleSelectPlace가 선택된 일정 ID, 마커 종류, 장소 상세 정보를 변경합니다. 이 상태를 가진 부모 TravelDetailsView가 다시 렌더링되면서, 일정 데이터가 바뀌지 않았는데도 자식인 TravelSchedules까지 다시 실행됐습니다.

TravelSchedules는 일차별 일정 정렬, 드래그 앤 드롭, 항공·숙소와 구간 이동 정보까지 렌더링하는 무거운 목록이어서, 지도 선택 상태가 바뀔 때마다 함께 다시 그리는 비용이 컸습니다.

```
marker.addListener('click', () => handleSelectPlace(/* ... */));
setSelectedScheduleId(scheduleId); setSelectedMarkerType('schedule');
setSelectedPlaceDetails(place); // 부모 → TravelSchedules 재렌더링
```

기각 모든 컴포넌트에 memo: 비교 비용만 늘고 원인이 해결되지 않음

React.memo는 부모의 재렌더링을 막는 것이 아니라 이전과 다음 props가 같을 때 자식 렌더링을 건너뜁니다. 원인을 확인하지 않고 모든 컴포넌트를 감싸면 매번 props 비교 비용이 추가되고, 새 배열이나 새 함수를 받는 컴포넌트는 비교 후에도 결국 다시 렌더링됩니다.

```
// 원인을 확인하지 않고 모든 섹션을 감싸는 방식
export default React.memo(TravelChecklist);
export default React.memo(TravelBudget);
export default React.memo(TravelVouchers);
```

선택 지도 선택 상태와 무관한 TravelSchedules만 memo하고, 전달하는 props의 참조를 함께 고정

지도에서 변경된 selectedScheduleId, selectedMarkerType, selectedPlaceDetails는 일정 목록에 필요한 props가 아니었습니다. 따라서 TravelSchedules를 React.memo로 감싸고, 부모가 다시 렌더링돼도 일정 목록의 props가 같게 유지되도록 배열과 콜백의 참조를 고정했습니다.

getFullUserList 함수 자체는 useCallback이었지만 호출할 때마다 새 배열을 반환했고, 인라인 콜백도 렌더링마다 새 함수가 됐습니다. 이 상태에서는 memo를 적용해도 얕은 비교가 실패합니다.

그 결과 지도 마커나 장소 선택으로 부모만 다시 렌더링될 때는 일정 목록 렌더링을 건너뛰고, 일정 데이터나 경로 정보처럼 목록에 실제로 필요한 props가 변경될 때만 다시 렌더링하게 됐습니다.

```
// 변경 전: 부모가 렌더링될 때마다 새 배열과 새 함수 생성
<TravelSchedules
  userList={getFullUserList()}
  onEditFlight={(flight) => { /* ... */ }}
/>

// 변경 후: 일정 데이터가 같으면 props의 참조도 유지
const fullUserList = useMemo(() => getFullUserList(), [getFullUserList]);

const handleEditFlight = useCallback((flight: any) => {
  setSelectedFlight(flight);
  setFlightOpen(true);
}, []);

<TravelSchedules
  userList={fullUserList}
  onEditFlight={handleEditFlight}
/>

export default React.memo(TravelSchedules);
```

해결

초기 진입 번들에서 무거운 UI 분리

11개 Dialog와 상세 섹션 5개, 지도 오버레이 2개 등 총 18개 UI를 next/dynamic으로 분리했습니다. Dialog는 LazyMount로 감싸 처음 열리기 전에는 렌더 트리에 포함하지 않고, 한 번 열린 뒤에는 마운트 상태를 유지해 닫힘 애니메이션과 작성 중인 입력 상태를 보존했습니다.

첫 클릭 지연을 초기 로딩 이후로 이동

데이터 로딩이 끝난 뒤 requestIdleCallback으로 Dialog 청크를 프리페치했습니다. next/dynamic과 프리페치가 같은 import 함수를 사용하게 해 유휴 시간에 받은 청크를 그대로 재사용했고, 미지원 브라우저에서는 2초 지연 setTimeout으로 대체했습니다.

지도 선택 상태와 일정 목록의 렌더링 분리

TravelSchedules만 React.memo로 감싸고 전달하는 배열과 콜백을 useMemo, useCallback으로 고정했습니다. 지도 선택 상태만 변경될 때는 일정 목록 렌더링을 건너뛰고, 일정이나 경로 정보처럼 목록에 실제로 필요한 props가 변경될 때만 다시 렌더링하게 했습니다.

추가적으로, 날짜 유틸의 의존성 경계 정리

네이티브 Date만 필요한 parseSafeDate와 dayjs가 필요한 parseSafeDayjs가 같은 모듈에 있었습니다. parseSafeDayjs를 dayjs 기반 시간 유틸로 옮기고 trip-date 모듈은 네이티브 Date만 사용하게 해, dayjs가 필요 없는 코드가 해당 의존성을 간접적으로 가져오지 않도록 분리했습니다.

성과

45%↓
First Load JS 828kB → 452kB

28%↓
LCP 6.9초 → 5.0초

32%↓
TBT 960ms → 650ms

0회
마커 클릭 시 일정 목록 렌더링

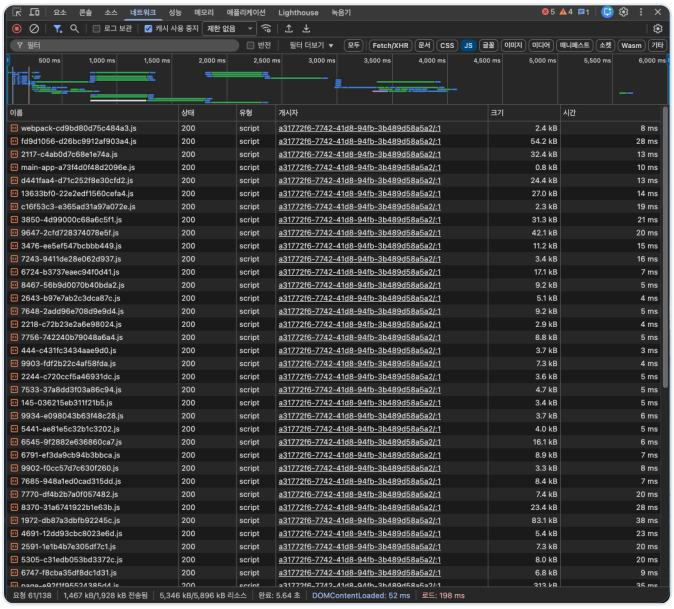
Next.js production build를 비교한 결과, 여행 상세 화면의 First Load JS가 828kB에서 452kB로 45% 감소했습니다. Lighthouse 3회 측정 중간값으로 LCP는 6.9초에서 5.0초로 28%, TBT는 960ms에서 650ms로 32% 개선됐습니다.

실제 로딩을 Network 탭으로 확인했습니다. 페이지 JS 파일은 313kB에서 51.3kB로 줄었고, 큰 Dialog 청크는 지도 스크립트 뒤에 받아집니다. JS 전송량은 1,467kB에서 1,496kB로 비슷하고, 요청 수는 61개에서 75개로 늘어 다운로드 시점만 옮겨졌다는 것을 확인했습니다.

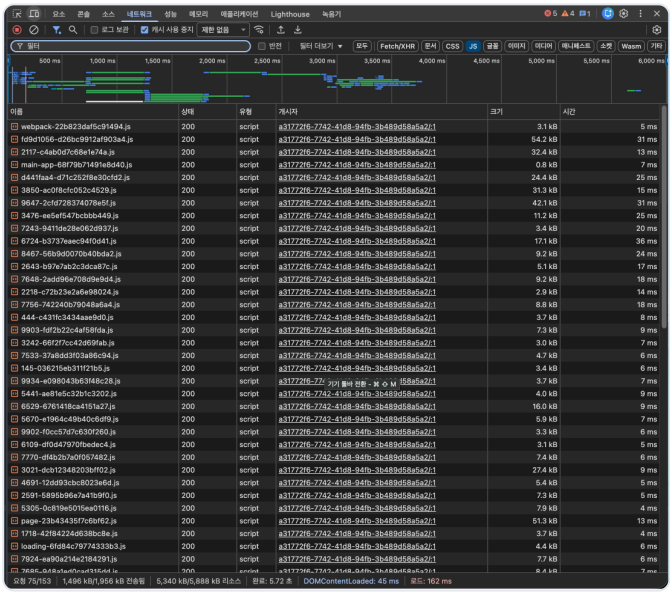
지도 선택 상태와 일정 목록의 렌더링을 분리해, 첫 마커 클릭 시 일정 목록 렌더링을 16회에서 0회로 줄였습니다.

지도 마커나 장소 선택으로 부모 화면만 다시 렌더링될 때는 일정 목록 렌더링을 건너뛰도록 했습니다. 일정 데이터나 경로 정보가 실제로 변경될 때는 정상적으로 다시 렌더링되므로, 필요한 업데이트는 유지하면서 지도 선택과 무관한 렌더링만 제거했습니다.

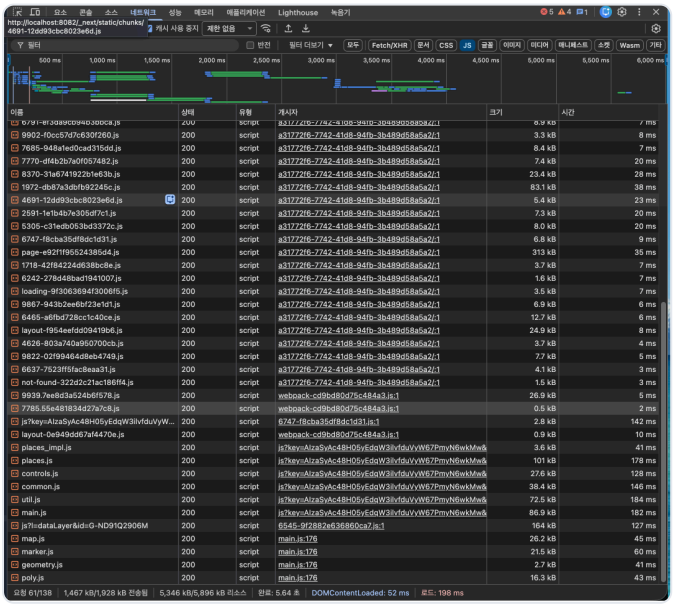
총 다운로드 양 자체를 줄인 것은 아닙니다. Dialog 청크를 초기 진입에서 제외한 뒤 브라우저 유휴 시간으로 다운로드 시점을 옮겼습니다. 또한 한 번 연 Dialog는 입력 상태와 닫힘 애니메이션을 보존하기 위해 메모리에 유지됩니다. 다만 11개가 모두 마운트돼도 기존 구조보다 나빠지지 않는다고 판단해 감수했습니다.



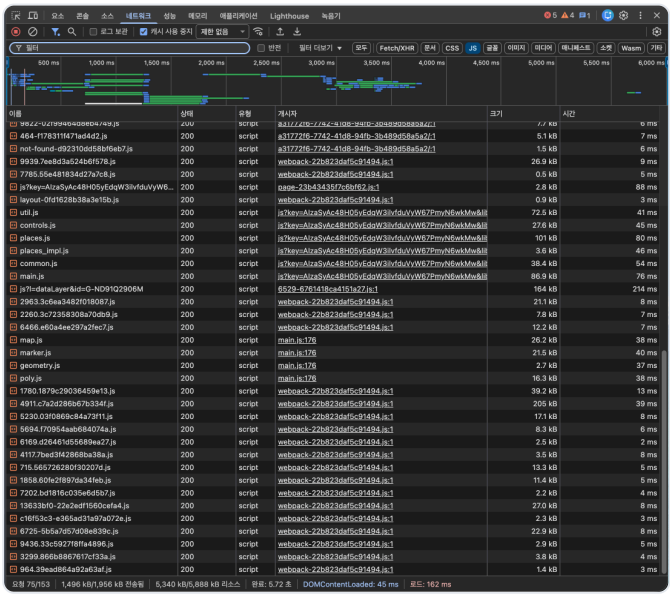
개선 전: page 파일 313kB



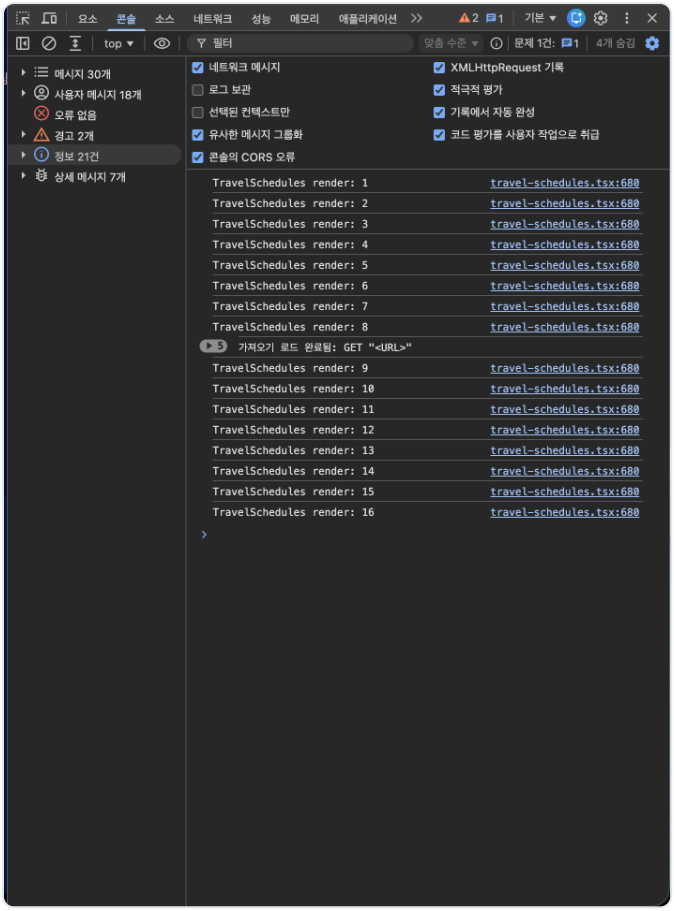
개선 후: page 파일 51.3kB



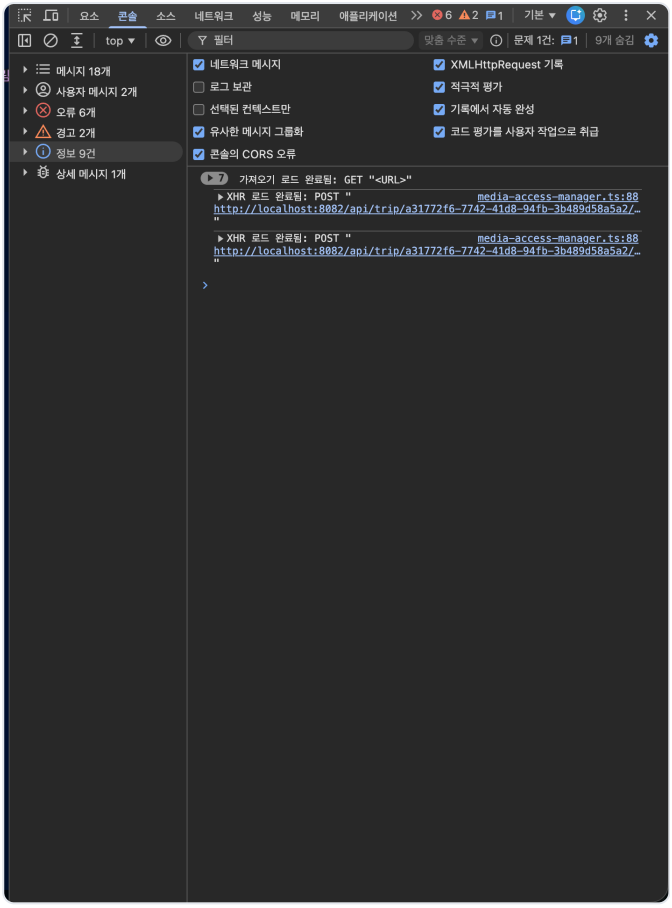
개선 전: 지도 스크립트 뒤에 추가 청크 없음



개선 후: 큰 청크가 지도 스크립트 뒤에 로드됨



개선 전: 첫 마커 클릭 시 렌더링 로그



개선 후: 첫 마커 클릭 시 렌더링 로그 없음

배운 점

로딩 최적화 — 무엇을 나눌지보다 언제 받게 할지가 중요했습니다

next/dynamic을 적용해도 동적 컴포넌트가 처음부터 렌더 트리에 포함되면 청크가 즉시 요청되므로 초기 로딩 비용을 충분히 늦출 수 없었습니다. LazyMount로 최초 마운트 시점을 제어하고, 첫 클릭 지연은 유휴 시간 프리페치로 보완하면서 코드 분할·마운트·프리페치를 함께 설계해야 한다는 것을 배웠습니다. 또한 First Load JS 감소는 총 다운로드 양 감소와 다르므로, 어떤 비용을 제거했고 어떤 비용은 뒤로 옮겼는지 구분해 측정해야 했습니다.

렌더링 최적화 — memo보다 상태와 props의 경계를 먼저 확인해야 했습니다

부모의 지도 선택 상태가 변경되면 자식도 다시 호출되며, React.memo를 적용해도 매번 새로 생성되는 배열과 콜백을 전달하면 얕은 비교가 실패했습니다. 일정 목록에 실제로 필요한 데이터와 지도에만 필요한 선택 상태를 구분하고, 배열과 콜백의 참조를 고정한 뒤 필요한 컴포넌트에만 memo를 적용했습니다. 이를 통해 모든 컴포넌트를 감싸기보다 어떤 상태 변경이 어디까지 전파되는지 추적하고 렌더링 경계를 좁히는 것이 중요하다는 것을 배웠습니다.

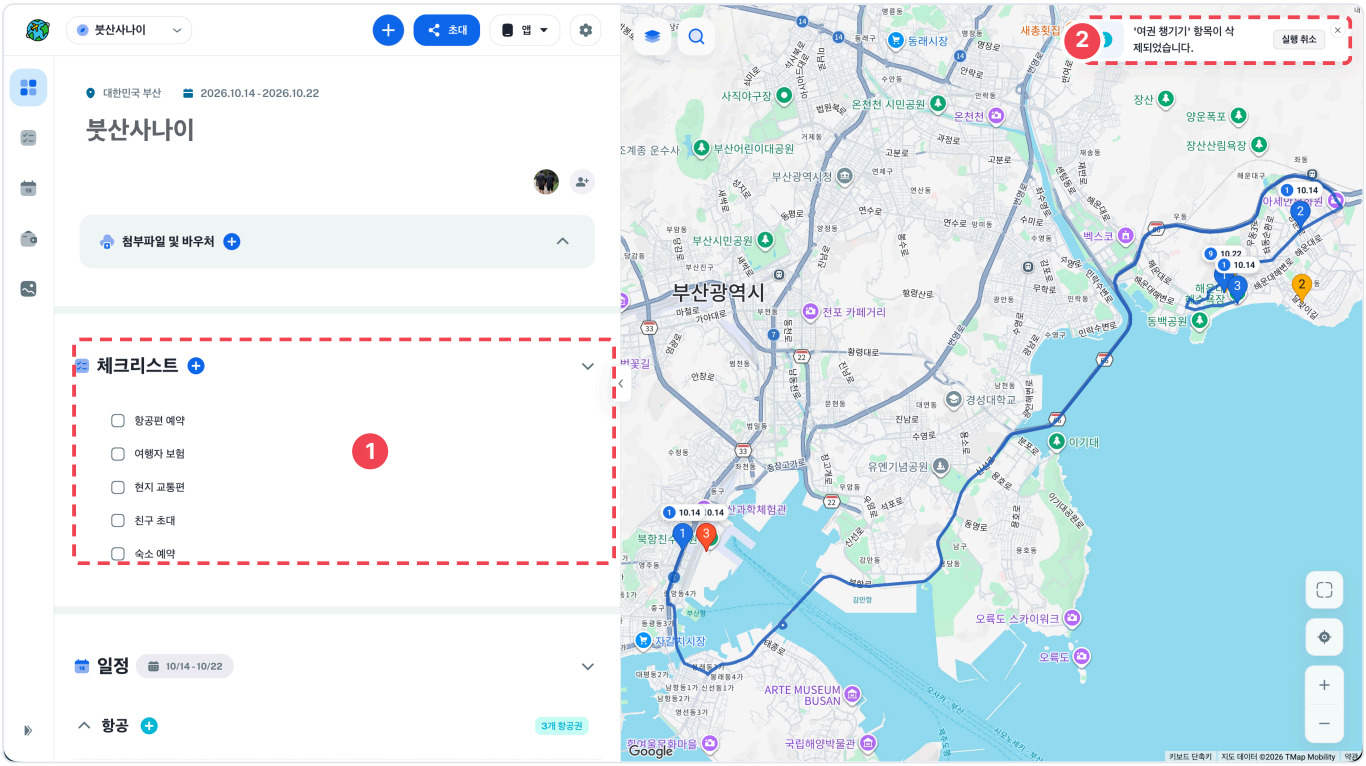
사례 2 · 저스트트립

7개 도메인에서 반복되던 삭제 문제를 실행 취소 가능한 공통 구조로 추상화

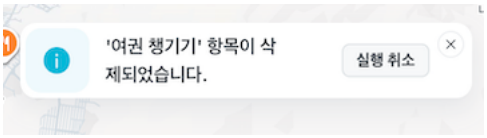
TypeScript, TanStack Query, fetch keepalive, Next.js

문제 정의

저스트트립은 동행자와 함께 여행을 편집하는 서비스라, 일정·숙소·항공·바우처·지출·체크리스트·사진 7개 도메인 어디서나 항목을 삭제합니다.



체크리스트 항목을 삭제한 직후의 여행 상세 화면



1 **낙관적 삭제** '여권 챙기기' 항목이 서버 응답을 기다리지 않고 목록에서 바로 사라집니다.

2 **실행 취소** 4초 동안 실행 취소할 수 있고, 이 시간이 지나야 서버에 삭제 요청을 보냅니다.

기존에는 삭제할 때마다 확인 dialog를 띄웠습니다. 여행 계획을 세우다 보면 항목을 자주 고치고 지우는데, 그때마다 확인을 눌러야 해서 사용이 너무 불편했습니다. 그래서 dialog 없이 바로 지우고, 잘못 지웠으면 되돌리는 흐름으로 바꾸기로 했습니다.

되돌리려면 서버에서 바로 지우면 안 됩니다. 다시 만들면 새 ID가 발급되어 항공·숙소·일정·지출과 바우처 사이의 연결이 끊어지기 때문입니다. 그래서 화면에서 먼저 지우고 서버에는 4초 뒤에 보내기로 했는데, 그동안 화면과 서버의 데이터가 달라지면서 도메인과 관계없이 같은 문제가 생겼습니다.

- 되살아나는 항목: 서버에는 아직 항목이 남아 있어서, 그 사이 목록을 다시 불러오면 지운 항목이 화면에 다시 나타났습니다.
- 반복되는 재조회: 사진 여러 장처럼 연속으로 지우면 삭제가 끝날 때마다 목록 전체를 다시 불러왔습니다.

그래서 삭제를 "요청 한 번"이 아니라 "대기, 전송 중, 최근 완료를 거치는 상태 변화"로 다시 정의하고, 이 흐름을 도메인마다 구현하지 않도록 공통 구조로 만들기로 했습니다.

판단

> **삭제 UX를 어떻게 설계할까**
>
> **기각** 기존 확인 dialog 유지: 사진 수십 장을 지울 때마다 확인을 눌러야 함
>
> 실수는 줄지만 정상적인 삭제까지 모두 느려집니다.
>
> **기각** 즉시 삭제 후 실행 취소 시 다시 생성: 새 ID가 발급되어 연결이 끊어짐
>
> 항공, 숙소, 일정, 지출은 바우처와 연결될 수 있습니다. 삭제한 항목을 다시 만들면 새 ID가 발급되어 서로 간의 연결이 끊어지고, 겉보기만 같은 아예 다른 항목이 됩니다.
>
> **선택** 화면에서 먼저 지우고 4초 뒤 서버로 보내는 지연 커밋
>
> 삭제해도 4초 동안은 서버에 요청을 보내지 않고, 클라이언트에서만 항목을 숨긴 채 원래 항목과 위치를 기억해 둡니다. 실행 취소하면 기억해 둔 항목을 다시 보여 주기만 하면 되므로, 서버 데이터가 그대로 남아 ID와 바우처 연결이 유지됩니다. 이후 4초 뒤에 서버에 삭제 요청을 보냅니다.

하지만 지연 커밋을 넣자 새로운 문제가 생겼습니다. 4초 동안 화면과 서버의 데이터가 달라, 그 사이 목록을 다시 불러오면 지운 항목이 돌아왔습니다.

```
0초  사용자가 '여권 챙기기' 삭제 → 화면에서 바로 사라짐
1초  다른 작업 때문에 목록 재조회 → 서버에는 아직 항목이 있음
1초  응답으로 목록 덮어쓰기 → 지운 항목이 다시 보임
4초  그제야 서버에 삭제 요청
```

> **되살아나는 항목을 어떻게 막을까**
>
> **기각** 대기 중에는 목록 재조회를 막기: 삭제와 무관한 변경까지 늦게 반영됨
>
> 삭제 대기 중 재조회를 멈추면 항목이 되살아나지는 않습니다. 하지만 동행자가 바꾼 내용이나 다른 작업의 결과까지 화면에 늦게 반영되고, 연속 삭제 중에는 멈추는 시간이 계속 길어집니다.
>
> **기각** 즉시 soft delete하고 실행 취소 시 복구: 가장 깔끔하지만 백엔드 변경 범위가 큼
>
> 삭제 즉시 서버에 표시하면 화면과 서버가 처음부터 일치합니다. 하지만 7개 도메인에 삭제 표시와 복구 API를 추가하고 모든 조회를 함께 고쳐야 해서, 당장 적용할 수 없었습니다.
>
> **선택** 삭제 중인 ID를 3단계로 기억하고 모든 목록 응답에서 걸러내기
>
> 목록을 다시 불러오면 서버 응답이 화면 데이터를 통째로 덮어씁니다. 그래서 응답을 화면에 넣기 전에, 지금 삭제 중인 항목을 한 번 더 걸러내도록 했습니다.
>
> ```
> 체크리스트 삭제 예시
>
> [서버에 삭제 요청을 보내기 전]
> 서버 응답: [항공권 예약, 여권 챙기기, 여행자 보험]
> 삭제 중인 항목: [여권 챙기기]
> 화면에 넣는 목록: [항공권 예약, 여행자 보험]
>
> [서버 삭제 완료 후]
> 서버 응답: [항공권 예약, 여행자 보험]
> 최근 완료 항목: [여권 챙기기] (20초 뒤 정리)
> 화면에 넣는 목록: [항공권 예약, 여행자 보험]
> ```
>
> 처음에는 4초 대기 중인 항목만 기억하면 될 것 같았습니다. 하지만 재조회와 삭제 요청은 서로 겹칠 수 있습니다. 대기가 끝나 추적을 멈춘 순간, 그보다 먼저 시작된 조회의 응답이 도착하면 항목이 되살아납니다.
>
> ```
> 3.9초  목록 재조회 시작 (서버에 아직 항목이 있음)
> 4.0초  대기 종료, 서버에 삭제 요청
> 4.2초  삭제 성공
> 4.5초  3.9초에 시작한 조회의 응답 도착 → 항목이 포함되어 있음
> ```
>
> 그래서 삭제가 끝난 뒤에도 잠시 기억하도록 세 단계로 나눴습니다.

단계	시점	왜 응답에 항목이 들어올까
대기	삭제 후 4초	서버에 삭제 요청을 아직 보내지 않음
전송 중	요청을 보내고 응답을 기다리는 동안	서버가 조회를 삭제보다 먼저 처리할 수 있음
최근 완료	삭제 성공 후 20초	삭제 전에 시작된 조회의 응답이 늦게 도착할 수 있음

이 필터는 목록을 받는 함수 한 곳에만 넣었습니다. 어떤 화면이 어떤 이유로 재조회하든 같은 결과가 나오고, 화면 코드는 삭제 상태를 몰라도 됩니다.

```
export async function fetchChecklists(tripId: string) {
  const res = await axiosInstance.get(endpoints.travel.checklist(tripId));
  const sorted = [...(res.data.data || [])].sort(bySortOrder);
  return filterPendingDeletes(sorted, (c) => c.id, 'checklist');
}
```

항목이 되살아나는 문제는 막았지만, 삭제를 연달아 하는 상황이 남아 있었습니다. 사진 여러 장을 지우면 삭제가 끝날 때마다 목록 전체를 다시 불러왔고, 탭을 닫으면 아직 보내지 않은 삭제가 사라질 수 있었습니다.

연속 삭제와 탭 종료는 어떻게 처리할까

기각 debounce만으로 재조회 묶기: 삭제 간격이 벌어지면 중간에 재조회가 생김

debounce는 짧은 시간에 몰린 호출을 마지막에 한 번만 실행합니다. 사진을 빠르게 지우면 잘 묶이지만, 삭제마다 4초 타이머가 따로 돌기 때문에 간격을 두고 지우면 삭제가 남아 있는데도 재조회가 실행됐습니다. debounce 시간을 늘리면 그만큼 화면 반영이 늦어지고, 얼마나 늘려야 하는지도 추측해야 합니다.

```
0초    사진 A 삭제 (4초 뒤 전송)
1초    사진 B 삭제 (5초에 전송)
4초    A 삭제 완료 → 재조회 예약
4.2초  추가 완료가 없어 재조회 실행 (B가 아직 남아 있음)
5초    B 삭제 완료 → 재조회 예약
5.2초  재조회 또 실행
```

기각 sendBeacon으로 남은 삭제 전송: POST만 가능해 DELETE를 보낼 수 없음

4초 대기 중에 탭을 닫으면, 아직 보내지 않은 삭제가 사라져 지운 항목이 그대로 남습니다. sendBeacon은 페이지가 닫혀도 전송을 마쳐 주는 브라우저 API지만 POST만 보낼 수 있어, DELETE로 만든 삭제 API를 그대로 호출할 수 없었습니다.

```
// 항상 POST로 전송되어 삭제 API를 호출할 수 없음
navigator.sendBeacon(url, body);
```

선택 그룹의 남은 작업이 0개일 때만 재조회하고, 탭을 닫으면 fetch keepalive로 전송

재조회 시점을 시간이 아니라 남은 삭제 개수로 판단했습니다. 위 예시에서 A가 끝났을 때는 B가 남아 있으므로 재조회하지 않고, B까지 끝나 남은 삭제가 0개가 되면 한 번만 실행합니다. 200ms debounce는 마지막에 몰리는 호출을 묶는 보조 역할로 남겼습니다.

탭 종료에는 fetch의 keepalive 옵션을 사용했습니다. 페이지가 닫혀도 요청이 유지되고 DELETE도 보낼 수 있어, 백엔드를 고치지 않고 해결할 수 있었습니다.

```
export function scheduleGroupRefresh(group: string, refreshFn: () => void, debounceMs = 200) {
  // ...
  const activeCount = countActiveDeletesInGroup(group);

  // 대기 중이거나 전송 중인 삭제가 남아 있으면 재조회하지 않음
  if (activeCount === 0) {
    const timer = setTimeout(/* 재조회 실행 */, debounceMs);
    groupRefreshTimers.set(group, timer);
  }
}
```

여기까지의 처리를 7개 도메인마다 구현할 수는 없었습니다. 데이터는 달라도 삭제의 흐름은 같았기 때문에, 이 흐름을 공통 구조로 분리했습니다.

> **어떻게 7개 도메인에 반복 없이 적용할까**
>
> 변하지 않는 흐름은 공통 모듈이 맡고, 변하는 부분만 도메인이 콜백으로 넘기도록 나눴습니다.

```
// 도메인은 "어떻게 지우는지"만 알려 주고, 타이밍과 경쟁 상태는 몰라도 됨
requestUndoableDelete({
  id: `checklist-${item.id}`,
  group: 'checklist',
  title: item.content || '체크리스트 항목',
  onOptimisticRemove: () => { /* 캐시에서 제거 */ },
  onRestore: () => { /* 원래 위치로 복구 */ },
  commitDelete: () => deleteChecklist(tripId, item.id),
  onSuccessRefresh: () => queryClient.invalidateQueries({ queryKey }),
  fallbackEndpoint: endpoints.travel.checklist_delete(tripId, item.id),
});
```

구분	공통 모듈이 맡는 것	도메인이 넘기는 것
삭제 흐름	4초 지연, 실행 취소 알림	항목 정보 (id, group, title)
화면 반영	언제 지우고 언제 복구할지	지우는 방법, 복구하는 방법
서버 요청	3단계 상태 추적, 실패 시 복구	삭제 API 호출
데이터 동기화	응답 필터링, 재조회 1회로 묶기	재조회 방법
탭 종료	남은 삭제 일괄 전송	삭제 엔드포인트

성과

7개
도메인에 공통 적용

1회
연속 삭제 시 목록 재조회

0건
백엔드 API 변경

삭제 경험 개선
삭제할 때마다 뜨던 확인 dialog를 없애고 낙관적 삭제로 바꿔, 사용자가 서버 응답을 기다리지 않고 항목을 바로 지울 수 있게 했습니다. 잘못 지운 항목은 4초 안에 실행 취소할 수 있고, 서버 데이터가 그대로 남아 있어 ID와 바우처 연결도 유지됩니다.

화면과 서버의 불일치 해결
대기, 전송 중, 최근 완료 3단계로 삭제를 추적해, 항목을 추가하거나 다른 작업으로 목록을 다시 불러와도 지운 항목이 되살아나지 않게 했습니다. 서버 삭제가 실패하면 항목을 복구하고 사용자에게 알려, 화면과 서버가 어긋난 채 남지 않도록 했습니다.

연속 삭제 요청 최적화
삭제가 끝날 때마다 실행되던 목록 재조회를, 남은 삭제가 없을 때 한 번만 실행하도록 줄였습니다. 탭을 닫아도 대기 중인 삭제를 keepalive 요청으로 전송하도록 했습니다.

반복 코드의 공통화
백엔드 API 변경 없이 7개 도메인에 같은 삭제 구조를 적용했습니다. 도메인은 지우는 방법과 복구하는 방법만 넘기면 되고, 타이밍과 경쟁 상태는 공통 모듈이 처리합니다. 삭제 흐름을 고칠 곳도 한 곳으로 모였습니다.

배운 점

반복되는 코드를 합치기 전에, 무엇이 반복되는지를 먼저 정의해야 한다는 것을 배웠습니다.
삭제를 요청 한 번으로 보면 일정 삭제와 사진 삭제는 서로 다른 기능처럼 보입니다. 하지만 시간에 따라 바뀌는 상태로 보니 대기, 전송 중, 최근 완료라는 같은 흐름이 드러났습니다. 그 뒤에야 변하는 부분과 변하지 않는 부분을 나눌 수 있었고, 공통 구조도 자연스럽게 정해졌습니다.

낙관적 업데이트는 화면을 먼저 바꾸는 기술이 아니라, 불일치를 관리하는 일이라는 것을 배웠습니다.
낙관적 삭제를 넣으니 사용자는 빨라졌지만, 그 대가로 화면과 서버가 달라지는 시간이 생겼습니다. 그동안 재조회, 실행 취소, 실패, 탭 종료가 모두 끼어들 수 있었습니다. 빠르게 보이게 만드는 것보다, 그 사이의 경우를 빠짐없이 처리하는 쪽이 훨씬 많은 고민이 필요했습니다.

백엔드를 바꿀 수 없어도 클라이언트가 책임질 범위는 나눌 수 있다는 것을 배웠습니다.
근본 해결은 soft delete라고 판단했지만, 범위가 커서 바로 적용할 수 없었습니다. 그래서 서버는 그대로 둔 채, 사용자가 겪는 문제를 클라이언트에서 막는 구조를 만들었습니다. 이상적인 해결책과 지금 가능한 해결책을 구분하는 것도 중요했습니다.

사례 3 · 얼리버드 알람 (React Native - iOS)

얼리버드 알람
설정한 시간에 광고 음원 알람이 울리고, 리워드 포인트를 받아 기프티콘으로 교환하는 알람 앱입니다.
2026.05 - 2026.10 · 기획, 풀스택 개발 · iOS·Android

무의식적으로 누른 볼륨 버튼이 기상 미션을 끊지 않도록

React Native (Expo), Swift, AlarmKit, App Intents, App Group

문제 정의

이 알람 앱은 울리는 것으로 끝나지 않습니다. 사용자가 앱에 들어와 광고 음원을 듣고 버튼을 눌러야 리워드를 받습니다. 그래서 OS가 알람을 끝내도, 앱에 들어오기 전이면 서비스 입장에서는 끝난 것이 아닙니다.

Android는 문제가 없었습니다. 다른 앱 위에 표시 권한으로 알람이 울릴 때 앱을 바로 실행할 수 있어, 알람 화면이 곧 앱 화면이었습니다. 사용자는 그 화면에서 다시 알림을 누르거나 리워드를 받고 알람을 끄면 됐습니다.

iOS는 달랐습니다. WWDC25에서 공개된 AlarmKit으로 알람을 구현했는데, AlarmKit 알람은 앱이 아니라 iOS 시스템 화면에서 울립니다. 알람이 울리는 곳과 리워드를 받는 앱이 분리되어 있어서, 사용자가 알람을 끈 뒤 앱으로 들어오게 만드는 것이 핵심이었습니다.



AlarmKit 시스템 알람 화면 (Apple WWDC25 예제)

구분	Android	iOS
알람을 그리는 주체	앱	iOS 시스템
리워드 받는 곳	알람 화면에서 바로 진행	앱에 따로 들어와야 함
사용자가 거치는 단계	알람 화면에서 미션 완료	알람 끄기 → 앱 진입 → 미션 완료

그런데 AlarmKit의 시스템 화면에서는 버튼이 두 가지 동작으로만 나뉩니다.

- A **화면의 Dismiss 버튼, 볼륨 버튼, 측면 버튼** 알람 정지 또는 다시 알림 (하나의 동작으로 묶임)
- B **화면의 Repeat 버튼** 다시 알림

A 버튼에는 AlarmKit의 stopIntent로 App Intent를 연결할 수 있습니다. 버튼을 누르면 지정한 코드가 실행되는 기능으로, 여기서 앱 실행과 다음 알람 등록을 함께 하려고 했습니다.

문제는 실행 순서였습니다. App Intent의 openAppWhenRun을 켜면 A 버튼을 눌러 시스템 알람이 종료될 때, 앱이 자동으로 켜진 뒤에 다음 다시 울림 알람을 등록하는 Intent 코드가 실행됩니다. 잠금 상태에서는 사용자가 잠금을 풀어야 앱이 열리므로, 다음 알람을 등록하는 코드도 그때까지 실행되지 않습니다.

화면에 다시 알림 버튼이 따로 있으니 문제가 없어 보일 수 있습니다. 하지만 iOS 기본 알람에서는 볼륨 버튼을 누르면 다시 알림을 설정한 시간 뒤에 알람이 다시 울립니다. 사용자는 같은 동작을 기대하고 볼륨 버튼을 누르는데, openAppWhenRun을 켜는 선택을 하면, 이 앱에서는 앱만 켜지고 알람은 다시 울리지 않습니다. 잠결에 눌렀다면 그대로 다시 잠들 수 있습니다.

1. openAppWhenRun을 켜면: 앱은 열리지만, 잠금 화면에서 볼륨 버튼으로 끈 사용자에게는 다음 알람이 등록되지 않아 다시 울리지 않습니다.
2. openAppWhenRun을 끄면: 잠금 상태에서도 다음 알람을 바로 등록할 수 있지만, 앱이 자동으로 열리지 않습니다.

```
openAppWhenRun = true
볼륨 버튼 누름 → 앱 실행 (잠금 상태면 해제 필요) → Intent 코드 실행 → 다음 알람 등록
                    ↑ 잠금을 안 풀면 다음 알람 등록 불가능

openAppWhenRun = false
볼륨 버튼 누름 → Intent 코드 바로 실행 → 다음 알람 등록 → 앱은 열리지 않음
```

시스템 알람을 사용자가 기대하는 동작으로 작동하게 하면서도 자연스럽게 앱에 들어와 리워드를 받게 하기 위해서 많은 고민을 했습니다.

판단

이 문제를 해결하기 위해, openAppWhenRun의 활성화 여부로 나눠서 생각했습니다.

> **openAppWhenRun을 활성화하고 다시 알림을 모두 예약해 두기**
>
> 기각 2시간 분량을 미리 예약하고 앱 열기: 알람 수가 크게 늘고 안전한 한도를 알 수 없음
>
> 앱에서 알람을 예약할 때 다시 울릴 알람을 2시간 분량 미리 예약합니다. A 버튼을 눌러서 알람을 해제하면, openAppWhenRun을 켜서 앱을 여는 방식으로 구성했습니다. A 버튼을 누르고 잠금 화면을 해제하지 않거나, 실수로 또는 더 자고 싶어서 볼륨 버튼을 눌러도, Intent 코드의 실행 없이 다음 알람이 이미 예약되어 있어 문제가 없습니다. 다음 회차가 되거나 앱에 들어오면 미리 예약한 알람을 취소합니다.
>
> 
>
> 다시 알림 버튼이 없는 화면 (Apple WWDC25 예제)
>
> A **화면의 Dismiss 버튼, 볼륨 버튼, 측면 버튼** 앱 실행. 다시 울림은 미리 예약해 둔 알람이 맡습니다.
>
> ```
> 볼륨 버튼 누름 → 앱 실행 (잠금 상태면 해제 필요)
>                     ↑ 잠금을 안 풀어도 미리 예약한 알람이 다시 울림
> ```
>
> 하지만 알람 수가 크게 늘었습니다. 예를 들어 1분 단위 다시 알림으로 매일 반복 알람을 2시간 분량 선예약하면, 사용자가 만든 알람은 1개인데도 1 × 60 × 2 × 7 = 840개의 알람이 예약됩니다. AlarmKit에는 한도 초과 에러(maximumLimitReached)가 있지만 공식 문서에도 몇 개인지 정확한 숫자가 나와 있지 않았습니다. 한도 숫자를 찾지 못해, 안전한 범위를 판단할 수 없었습니다.

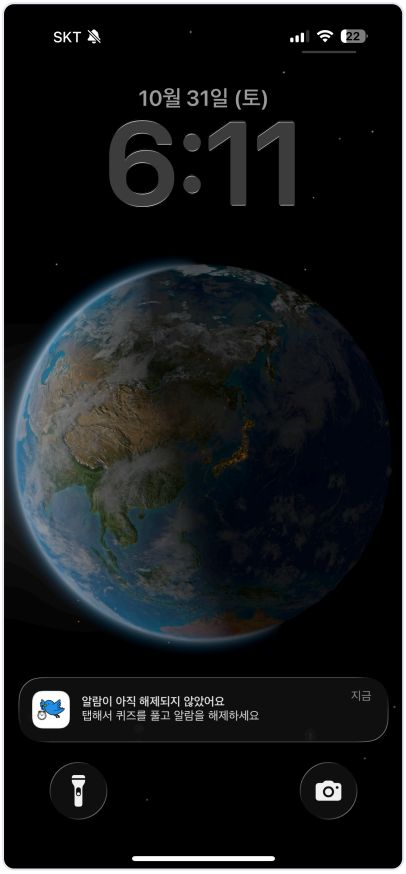
openAppWhenRun을 비활성화하고 다음 알람을 그때그때 등록하기

선택 알람을 하나만 예약하고 openAppWhenRun을 꺼서, 버튼이 눌릴 때마다 다음 알람 1개만 등록

앱에서 알람을 예약할 때 사용자가 설정한 시간에 알람을 하나만 예약하고, openAppWhenRun을 끕니다. 사용자가 A 버튼을 누르면 울리던 알람이 꺼지면서 Intent가 실행됩니다. 버튼을 누르는 순간 앱이 꺼져 있거나 화면이 잠겨 있을 수 있으므로, 다음 알람은 JS 없이 알람 종료 시 잠깐 실행되는 백그라운드 App Intent가 예약합니다.

Intent는 다시 알림을 켠 알람인지 확인한 뒤, 다음 알람 1개를 등록하고 알람이 아직 끝나지 않았다는 배너 알림을 보냅니다. 사용자가 배너를 누르거나 앱에 직접 들어와도 다음 알람은 유지됩니다. 광고를 보다가 앱을 꺼도 다시 울려야 하기 때문입니다. 광고 리워드를 받으면 남은 알람을 취소해 더는 울리지 않습니다.

알람을 등록할 때 설정을 앱과 Intent가 함께 읽는 저장소인 App Group에 미리 저장해 두고, 버튼이 눌리면 Intent가 다음 순서로 처리합니다.



알람을 끈 뒤 잠금 화면에 표시된 배너

```
볼륨·정지 버튼 누름 (앱은 열리지 않음)
  ↓
Intent 실행
  ↓
App Group에서 알람 설정 읽기
  ↓
진행 중 회차 기록 ("알람이 울렸지만 아직 광고를 보지 않음")
  ↓
다음 알람 1개 등록
  ↓
배너 알림 표시
  ↓
Intent 종료
  ↓
사용자가 배너 알림을 누르거나 직접 진입
  ↓
기록을 읽고 광고 화면으로 이동
```

앱이 꺼져 있는 동안의 상태를 네이티브가 저장해 두고, 앱이 켜지면 이어받는 구조입니다. 알람 수가 늘지 않고 사용자가 기대하는 동작도 유지되지만, 앱을 자동으로 켜지 못한다는 단점이 있습니다. 하지만 안정적으로 동작한다는 장점이 있고, 사용자가 기대하는 대로 시스템 알람 자체가 작동합니다.

전체 흐름

- 알람 등록: 설정을 App Group에 저장하고, 이번 울림(회차)을 앱과 네이티브가 함께 알 수 있도록 기록
- 다시 알림 버튼(B)을 누르면: OS가 설정한 간격 뒤에 다시 울림. 앱은 관여하지 않으며, 다시 울렸을 때 아래 흐름 중 하나로 이어짐
- A를 누르면: 다음 알람 1개 등록 + 앱 진입을 안내하는 배너 표시. 반복 일정은 그대로 유지
- 배너를 누르거나 앱을 직접 열면: 미해제 회차를 확인해 바로 광고 화면으로 이동. 알람이 울리는 중이면 멈추고, 광고를 보다가 앱을 꺼도 다시 울리도록 다음 알람을 유지
- 광고 리워드를 받으면: 남은 알람과 배너를 취소해 회차를 종료. 다음 날 알람은 유지
- 다시 알림을 끈 알람은 배너만 표시하고, 켠 알람도 다음 회차와 겹치지 않도록 최대 24시간까지만 이어서 울림

성과

사용자가 기대하는 알람 동작 유지
볼륨 버튼이나 측면 버튼으로 알람을 꺼도 다시 울리도록 구성해, iOS 기본 알람에 익숙한 사용자의 기대와 맞췄습니다. 앱을 열기 위해 알람의 기본 동작을 포기하지 않았습니다.

앱이 꺼진 상태까지 고려한 리워드 흐름
앱이 꺼져 있거나 화면이 잠겨 있어도 JS 없이 다음 알람을 등록하고, 진행 중 회차를 네이티브에 기록하도록 했습니다. 사용자가 나중에 앱에 들어와도 기록을 읽어 광고 화면으로 바로 이동합니다.

알람 수를 늘리지 않는 구조
미리 예약하는 방식에서는 알람 1개가 최대 840개까지 늘어날 수 있었습니다. 버튼이 눌릴 때마다 다음 알람 1개만 등록하도록 바꿔, 문서로 확인할 수 없는 한도에 의존하지 않게 했습니다.

감수한 대가
앱이 자동으로 열리지 않아 사용자가 배너를 한 번 눌러야 합니다. 하지만 실제 사용 시에는 알람을 끄고 1초 이내에 배너 알림이 화면 최상단에 올라옵니다.

배운 점

플랫폼 제약을 우회할 수 없을 때는 가장 피해가 큰 상황을 기준으로 정해야 한다는 것을 배웠습니다.
앱 실행과 다시 울림을 모두 얻는 방법을 찾았지만, 시도한 구성에서는 한쪽을 포기해야 했습니다. 이때 사용자가 앱에 늦게 들어오는 것보다, 알람을 끄고 그대로 잠드는 경우가 더 치명적이라고 판단했습니다. 기능 욕심보다 실패했을 때의 피해를 기준으로 삼으니 결정이 분명해졌습니다.

서비스의 완료 조건과 OS의 완료 조건은 다를 수 있다는 것을 배웠습니다.
OS에게 알람은 끄는 순간 끝나지만, 이 서비스에서는 광고를 보고 리워드를 받아야 끝납니다. 그래서 "알람이 울렸지만 아직 광고를 보지 않은 상태"를 따로 기록했습니다. 덕분에 앱이 꺼져 있는 동안 알람이 끝나도, 앱이 켜진 뒤 그 상태를 이어받아 광고 화면으로 보낼 수 있었습니다.

React Native 앱에서도 JS가 할 수 없는 영역을 먼저 나눠야 한다는 것을 배웠습니다.
앱이 꺼진 상태에서는 JS가 실행되지 않습니다. 그래서 무엇을 네이티브가 맡고 무엇을 JS가 맡을지 경계를 먼저 나눴고, 저장소를 사이에 두고 둘을 연결했습니다. 프론트엔드 코드만 보던 시야가 앱의 실행 상태까지 넓어졌습니다.

사례 4 · 라이더타운 V2

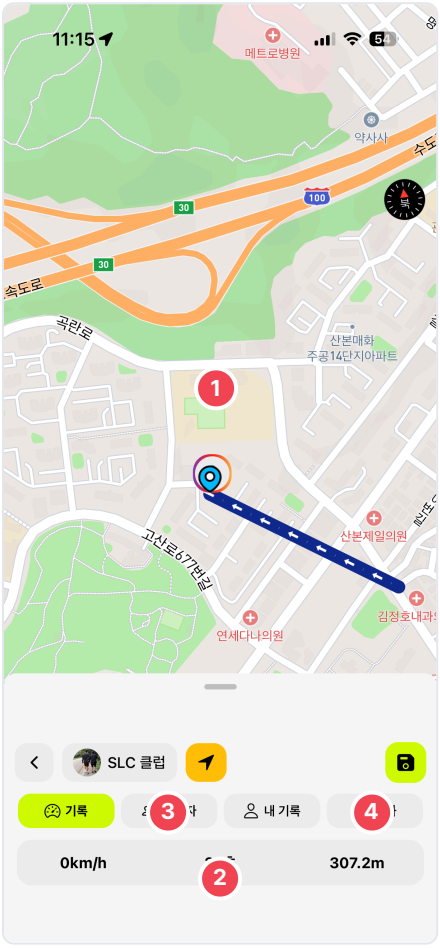
라이더타운 V2
오토바이 정비 견적·예약 플랫폼에서 라이딩 기록 측정과 크루간의 위치와 실시간 기록 공유 서비스로 피벗했습니다.
2024.06 - 2024.12 · 공동창업, 풀스택 개발 · 영업 담당 공동창업자와 2인 협업 · iOS·Android 출시

앱이 백그라운드에 있어도 끊기지 않는 실시간 라이딩

React Native, Firebase Realtime Database, react-native-background-geolocation, Mapbox

문제 정의

당시 백엔드 개발자가 이탈해 프론트엔드부터 백엔드까지 혼자 운영해야 했습니다. 그래서 운영 언어를 JavaScript로 통일하고 서버 비용을 줄이기 위해 AWS 인프라를 Firebase로 전환하고 있었습니다.



라이더타운의 실시간 라이딩 화면

라이딩 서비스는 자신의 라이딩 시간, 속력, 이동 거리, 최대 속력, 뱅킹각 등을 기록하고, 현재 위치와 기록을 크루원에게 실시간으로 공유하는 서비스였습니다. 주행 중에는 같은 크루원과 실시간 음성 통화도 할 수 있었습니다.

1. **실시간 위치와 경로** 지도에서 현재 위치와 이동 경로를 확인하고 크루원에게 공유합니다.
2. **라이딩 기록** 속력·시간·이동 거리를 실시간으로 보여 주고, 종료 후 최대 속력과 뱅킹각 등을 함께 기록합니다.
3. **크루 참가자** 함께 달리는 크루원의 위치와 라이딩 기록을 실시간으로 확인합니다.
4. **음성 통화** 이동 중에도 Node.js 기반 Mediasoup SFU 서버를 통해 크루원과 실시간으로 통화합니다.

위치·속력·거리 등의 라이딩 기록을 웹소켓으로 1초마다 공유하고, Node.js 기반 Mediasoup SFU 서버의 음성 채팅방 입장·퇴장과 참가자 상태도 웹소켓으로 관리했습니다. 앱이 포그라운드 상태에 있을 때는 문제없이 작동했습니다.

그런데 라이더가 내비게이션을 보거나 화면을 끄고 주행하면 앱은 대부분의 시간을 백그라운드에서 보냅니다. 이때 내 라이딩 기록과 크루 공유가 끊겼고, 처음에는 이를 두 가지 문제로 정의하고 시작했습니다.

- 백그라운드 실행: JS가 주기적으로 위치를 수집하고 전송하는 구조가 백그라운드에서 멈춘다.
- 웹소켓 연결: 백그라운드에서 웹소켓 연결이 유지되지 않아 기록 공유와 음성 채팅방의 참가자 상태 갱신이 중단된다.

이 두 문제가 각각 따로 발생하는 것인지, 둘 다 동작하지 않는 것인지 파악하기 위해 순차적으로 테스트했습니다.

판단

> **백그라운드 실행에 대한 검증**
>
> 기각 JS 타이머로 위치 수집: 백그라운드에서 콜백 실행을 보장할 수 없음
>
> 앱이 백그라운드로 이동하면 JS 타이머만으로는 수 초 단위의 위치 수집을 안정적으로 유지할 수 없었습니다. 위치를 받지 못하면 기록과 공유가 함께 끊겼습니다.
>
> 선택 네이티브 BackgroundGeolocation의 위치 이벤트를 기준으로 기록
>
> iOS의 Background Modes와 Android의 Location Foreground Service를 이용해 네이티브 계층에서 위치 추적을 유지했습니다. JS가 주기적으로 위치를 요청하는 대신 onLocation으로 전달된 네이티브 이벤트를 기준으로 기록했습니다.
>
> 즉 JS를 살려 두는 구조가 아니라, 네이티브 위치 세션이 이벤트를 전달한 시점에 JS 콜백이 실행되는 구조입니다. 이 구조가 보장하는 범위는 백그라운드와 화면 꺼짐까지이며, 앱을 강제 종료하면 기록은 이어지지 않습니다. 이때 남은 라이딩 상태는 onDisconnect로 서버에서 정리합니다.

위 테스트 결과, 실제 원인은 하나였습니다. 앱이 백그라운드로 가면 JS가 멈추는 것이었고, 웹소켓이 끊긴 것도 그 결과였습니다. 그래서 끊김을 해결하는 일은 웹소켓 교체가 아니라 백그라운드에서 앱이 계속 실행되도록 만드는 일이었습니다.

다만 실시간 통신은 웹소켓 대신 Realtime Database로 옮기기로 했습니다. 앱이 계속 실행돼도 라이딩 중에는 네트워크 단절이 생기므로, 웹소켓을 유지하면 재연결과 예외 처리를 직접 구현해야 했습니다. 또한 Firebase로 전환하던 중이었고, Realtime Database를 쓰면 위치 공유용 웹소켓 서버를 AWS에 유지할 필요가 없어 비용도 줄일 수 있었습니다.

```
iOS Core Location / Android Foreground Service
  ↓ 네이티브 위치 이벤트
react-native-background-geolocation (onLocation)
  ↓ 이벤트가 온 시점에 JS 콜백 실행
현재 시각과 마지막 전송 시각 비교 (백그라운드 5초)
  ↓ 조건을 만족하면 기록 전달
React Native Firebase 네이티브 SDK → Realtime Database
```

> **웹소켓과의 비교**
>
> 웹소켓에서 Realtime Database로 전환해 직접 구현과 비용 측면에서 이점이 많았습니다.

항목	웹소켓을 유지할 때	Realtime Database
실시간 전달	서버가 방 참가자에게 보내는 로직을 직접 구현	다른 크루원이 노드를 구독해 변경 사항만 수신
연결 복구	재연결 로직을 직접 구현	SDK가 재연결 처리
끊긴 동안의 쓰기	기기 버퍼링과 전송 순서를 직접 구현	SDK가 쓰기를 대기시켰다가 전송
정상 종료	종료 이벤트와 서버 상태 정리를 직접 구현	클라이언트가 라이딩 상태를 종료
비정상 종료	서버에서 연결 종료를 감지해 정리 로직 구현	기록 시작 시 onDisconnect를 등록해 Firebase 서버가 정리
정리 누락 보완	서버 또는 클라이언트에 만료 처리 구현	마지막 전송 후 2분이 지난 마커를 지도에서 제외
서버 운영	소켓 서버 배포·모니터링을 혼자 담당	소켓 서버를 운영하지 않음
비용	사용자가 없어도 서버 고정 비용 발생	서버 고정 비용 없이 사용량만큼 과금

Realtime Database로 전환하는 과정에서 데이터 전송의 효율도 개선했습니다.

> **위치 데이터를 얼마나, 언제 보낼까**
>
> 기각 1초 고정 전송: 신호 대기나 휴식 중에도 계속 보내고, 주행이 길어질수록 전송량이 늘어남
>
> 선택 상황별 전송 주기와 경로 압축
>
> - 전송 주기: 위치 이벤트마다 직전 전송 시각과 비교해 백그라운드 5초, 15초 이상 정지 시 30초, 주행 중 속도별 2~4초로 나눴습니다.
> - 경로 압축: 속도별 기준 거리(20/40/60m)를 넘을 때만 점을 추가하고, Google Encoded Polyline으로 인코딩했습니다.

성과

네이티브 위치 이벤트 기반으로 변경해 화면을 끄거나 내비게이션을 사용하는 동안에도 라이딩 기록과 크루 위치 공유가 이어지도록 개선했습니다.

onDisconnect와 2분 마커 만료 기준을 적용해 비정상 종료 후에도 오래된 라이더 위치가 지도에 계속 남는 문제를 방지했습니다.

상황별 전송 주기와 경로 압축으로 실시간 경로 데이터 전송량을 70% 이상 절감했습니다.

위치 공유용 WebSocket 서버를 RTDB로 대체해 서버 운영 부담을 줄이고, 크루별 이벤트와 재연결 처리를 SDK에 위임해 실시간 동기화 로직을 단순화했습니다.

배운 점

증상이 나타난 지점과 실제 원인, 기술을 교체한 이유를 구분해야 한다는 것을 배웠습니다.
처음에는 백그라운드에서 공유가 끊기는 현상을 WebSocket 문제로 생각했지만, 수집과 전송 과정을 분리해 확인하면서 JS 타이머에 의존한 위치 수집이 핵심 원인임을 알게 됐습니다. 이를 해결한 것은 네이티브 위치 추적이었고, RTDB 전환은 연결 관리와 운영 비용을 줄이기 위한 별도의 선택이었습니다.

또한 실시간 서비스라고 무조건 데이터를 자주 보내는 것이 좋은 건 아니었습니다. 주행 상태에 따라 전송 주기와 경로 정밀도를 조절하면서, 기록의 정확성을 유지하는 것과 데이터를 효율적으로 전달하는 것을 별개로 설계해야 한다는 점을 배웠습니다.