Minjae Kang Frontend Developer

+82 10-5519-7425 · towndev@naver.com · https://minjaekang.dev/en/

I took a leave of absence from university to found a startup and started coding to build the service myself. I then planned the motorcycle maintenance and riding platform RiderTown and the collaborative travel platform JustTrip, and built, launched, and operated both as web and mobile apps with React, Next.js, and React Native.

I find the points where the screen and the server fall out of sync and fix them structurally. Before fixing a symptom, I redefine the problem, and I value decisions where I can explain which alternatives I dropped and why.

Experience Entry level

JustTrip AI-powered cross-platform collaborative trip planning and journaling service 2026.03 - 2026.10

Product Planning · Frontend Development · Team of 2 with one backend developer

- Cut First Load JS on the trip detail page from 828kB to 452kB (45%), improved LCP by 28% and TBT by 32%, and removed schedule list re-renders on marker clicks (Case 1)
- Stopped deleted items from reappearing with optimistic deletion and three-stage delete tracking, applied as a shared structure across 7 domains (Case 2)

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

Early Bird Alarm Alarm app where users hear audio ads, earn reward points, and exchange them for gift cards 2026.05 - 2026.10

Product Planning · Full-Stack Development

- Redesigned the alarm structure within iOS AlarmKit constraints so alarms ring again even after the volume or stop button is pressed on the lock screen (Case 3)

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

RiderTown V2 Pivot to a ride-tracking service and serverless migration 2024.06 - 2024.12

Co-founder · Full-Stack Development · Team of 2

- Pivoted the service after Firebase Analytics showed ride screen retention over 10x higher than maintenance booking
- Built real-time crew location sharing that keeps working in the background and cut route data transfer by over 70% (Case 4)
- Migrated an AWS MSA to Firebase serverless, cutting fixed monthly infrastructure costs by over 90%

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

RiderTown V1 Motorcycle maintenance platform startup · Ride tracking and group voice calls 2022.11 - 2023.12

Co-founder · Full-Stack Development · Founding team of 3

- Migrated a React SPA to Next.js SSR, reducing the initial bundle from 2.5MB to 550KB and the first WebView screen from 3.5s to 0.8s
- Built 1-second WebSocket sync of crew ride data and 4-person voice calls on a Mediasoup SFU

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

Skills

- 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

Awards · Education

Selected for and completed the Youth Startup Academy (Startup Success Package) 2024.03

Selected for and completed the Pre-Startup Package 2023.05

B.S. in Electronic Engineering, Kwangwoon University 2018.03 - 2026.08

Case 1 · JustTrip

JustTrip

A collaborative service for planning and recording trips with friends. Users manage trips, vouchers, checklists, flights, stays, schedules, budgets, expenses, and photos together on web and mobile.

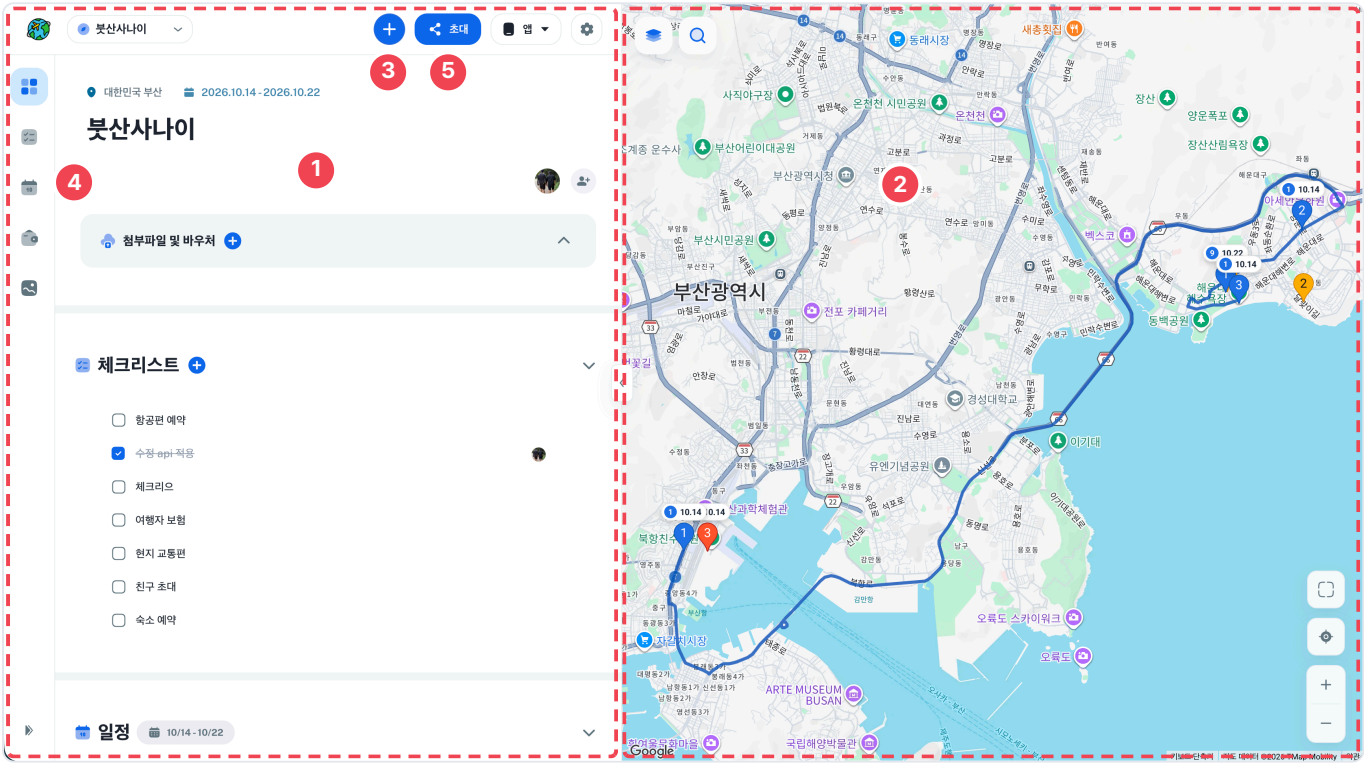
2026.03 - 2026.10 · Product Planning, Frontend Development · Team of 2 with one backend developer · justtripplanner.com

Cutting First Load JS by 45% and Removing Unnecessary Re-renders on the Trip Detail Page

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

Problem

On the JustTrip web trip detail page, users manage trip information, vouchers, checklists, flights, stays, schedules, budgets, expenses, and photos across a left panel and a right map. Eleven independent Dialogs for viewing and adding flights, schedules, and more open over the page, while five detail sections and two map overlays live inside it.



The JustTrip web trip detail page (Korean UI)

1. **Left panel** Shows trip information such as checklists, schedules, and vouchers.
2. **Right map** Shows schedule places and routes. Map interactions such as selecting a marker used to re-render the schedule list on the left as well.
3. **Add** One of the 11 independent Dialogs for adding flights, schedules, and more.
4. **Tab menu** Moves between detail sections such as schedules, expenses, and photos. These sections and map overlays, seven in total, were also in the initial bundle.
5. **Invite** A Dialog for inviting travel companions. All 11 independent Dialogs like this one stayed in the initial render tree whether they were open or not.

The trip detail page is the core screen of the service, where users move between the map and the panel while planning. But it took a long time to load, and interaction stuttered when selecting map markers. Because users repeat these actions throughout planning, this could not be left as it was.

Investigation pointed to two causes.

- First load: 18 UIs, made up of 11 Dialogs and 7 detail sections and overlays, were in the initial bundle, putting First Load JS at 828kB.
- Map stutter: when map interaction re-rendered the parent, the schedule list, which has nothing to do with the map, was redrawn too.

So I split the work into two problems: "closed UI is paying a cost on first load" and "map state changes are spreading to an unrelated list."

Decisions

> **How should the Dialog structure and mount timing work?**
>
> **Rejected** Swap content inside a single Dialog: nested flows and independent input state need a separate screen stack and restore logic
>
> Editing a trip needs two-way transitions, such as opening a voucher analysis Dialog while entering a flight, or opening the flight editor during voucher analysis. Swapping content inside one Dialog would require a separate screen stack and state restoration to keep each flow's input independent.
>
> ```
> // Every nested transition must preserve the screen stack and each input state
> const [dialogStack, setDialogStack] = useState<DialogScreen[]>([]);
> const current = dialogStack.at(-1);
>
> <Dialog open={dialogStack.length > 0}>
>   {current === 'flight' ? <FlightForm /> : <VoucherAnalysis />}
> </Dialog>
> ```
>
> **Rejected** Use next/dynamic alone: a closed Dialog still downloads its chunk if it is in the render tree
>
> A Dialog stays in the render tree regardless of its open value. next/dynamic requests the chunk when the component renders, so even with the code split, Dialogs the user never opens are downloaded as soon as the page loads.
>
> ```
> const FlightDialog = dynamic(loadFlightDialog, { ssr: false });
>
> // The component renders and requests its chunk even when open is false
> <FlightDialog open={flightOpen} onClose={handleClose} />
> ```
>
> **Rejected** Mount on open and unmount on close: the close animation is cut off and input is lost
>
> Chunks are only downloaded when needed. But the moment flightOpen becomes false, the component is removed immediately, leaving no time for the close animation and discarding anything the user was typing.
>
> ```
> // The component leaves the tree the moment it closes
> {flightOpen && <FlightDialog open onClose={handleClose} />}
> ```
>
> **Chosen** **LazyMount, which mounts on first open and then stays mounted: excluded from the initial bundle while keeping state and animation**
>
> I kept each Dialog as an independent component to preserve two-way nested flows and independent input state, and reduced the initial loading cost by combining next/dynamic with LazyMount.
>
> I treated Dialog state as three values, "not yet opened / open / closed," rather than just open and closed. A Dialog that has never opened does not need to render, but one that has opened must stay for its close animation and input.
>
> LazyMount expresses this difference with a single hasMounted flag. Before the first open it returns null, so the next/dynamic component never renders and its chunk is not requested. A Dialog that has been opened stays in memory, which I accepted because even all 11 mounted is no worse than before.

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

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

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

// Usage: the FlightDialog chunk is not downloaded until the first open
const FlightDialog = dynamic(loadFlightDialog, { ssr: false });

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

First-click delay caused by code splitting

Rejected Prefetch every Dialog chunk right after mount: competes with initial loading for network resources

Downloading all Dialog chunks as soon as the page mounts would reduce first-click delay. But 11 chunk requests would start immediately regardless of use, sharing the network with the initial data and images. The bundle would be split, yet downloads would still cluster right after entry, so the real load on the first screen would not be separated enough.

```
// Requests every Dialog chunk as soon as the page mounts
useEffect(() => {
  prefetchDialogs();
}, []);
```

Rejected Prefetch on hover: mobile has no hover

Hovering is a good signal of click intent. But touch devices have no hover, so mobile web users would keep the first-click delay.

```
// Never called on touch devices
<Button onMouseEnter={() => loadFlightDialog()} onClick={handleAddFlight}>
  Add flight
</Button>
```

Chosen **Prefetch with requestIdleCallback after the first load, falling back to a 2-second setTimeout in unsupported browsers**

Downloading right after the initial load competes with critical resources, and hover alone cannot cover mobile. So I prefetched Dialog chunks once data loading finished and the browser became idle.

next/dynamic and the prefetch share the same import function, so the Dialog reuses the chunk that was already downloaded. This did not reduce the total download size; it moved the download to after the first screen.

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

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

// After the first screen loads, fetch Dialog chunks during idle time
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]);
```

How can the schedule list avoid re-rendering when map selection changes?

Selecting a map marker makes handleSelectPlace update the selected schedule ID, marker type, and place details. The parent TravelDetailsView that owns this state then re-rendered, running its child TravelSchedules again even though the schedule data had not changed.

TravelSchedules is a heavy list that renders per-day sorting, drag and drop, flights, stays, and transit segments, so redrawing it on every map selection was costly.

```
marker.addListener('click', () => handleSelectPlace(/* ... */));
setSelectedScheduleId(scheduleId); setSelectedMarkerType('schedule');
setSelectedPlaceDetails(place); // Parent → TravelSchedules re-render
```

Rejected Wrap every component in memo: adds comparison cost without addressing the cause

React.memo does not stop the parent from re-rendering; it skips a child render when the previous and next props are equal. Wrapping everything without finding the cause adds a props comparison each time, and components that receive new arrays or functions re-render anyway.

```
// Wrapping every section without finding the cause
export default React.memo(TravelChecklist);
export default React.memo(TravelBudget);
export default React.memo(TravelVouchers);
```

Chosen Memoize only TravelSchedules, which does not depend on map selection, and stabilize the references of its props

The selectedScheduleId, selectedMarkerType, and selectedPlaceDetails changed by the map were not props the schedule list needed. So I wrapped TravelSchedules in React.memo and stabilized array and callback references so its props stayed equal when the parent re-rendered.

getFullUserList itself was a useCallback, but it returned a new array on every call, and inline callbacks became new functions on every render. In that state, the shallow comparison fails even with memo.

As a result, when only the parent re-renders from a map marker or place selection, the schedule list render is skipped, and it re-renders only when props it actually needs, such as schedules or routes, change.

```
// Before: a new array and a new function on every parent render
<TravelSchedules
  userList={getFullUserList()}
  onEditFlight={(flight) => { /* ... */ }}
/>

// After: prop references stay the same while the schedule data is unchanged
const fullUserList = useMemo(() => getFullUserList(), [getFullUserList]);

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

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

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

Solution

Moving heavy UI out of the initial bundle
I split 18 UIs, made up of 11 Dialogs, 5 detail sections, and 2 map overlays, with next/dynamic. Dialogs are wrapped in LazyMount so they stay out of the render tree until first opened, then stay mounted to preserve the close animation and in-progress input.

Moving first-click delay to after the initial load
After data loading, Dialog chunks are prefetched with requestIdleCallback. next/dynamic and the prefetch share the same import function so idle-time downloads are reused directly, with a 2-second setTimeout fallback in unsupported browsers.

Separating map selection from schedule list rendering
I wrapped only TravelSchedules in React.memo and stabilized its arrays and callbacks with useMemo and useCallback. When only the map selection changes, the list render is skipped; it re-renders only when props it needs, such as schedules or routes, change.

Also: cleaning up the date utility dependency boundary
parseSafeDate, which only needs native Date, and parseSafeDayjs, which needs dayjs, lived in the same module. I moved parseSafeDayjs to a dayjs-based time utility and kept the trip-date module on native Date, so code that does not need dayjs no longer pulls it in indirectly.

Results

45%↓	28%↓	32%↓	0
First Load JS 828kB → 452kB	LCP 6.9s → 5.0s	TBT 960ms → 650ms	List renders on marker click

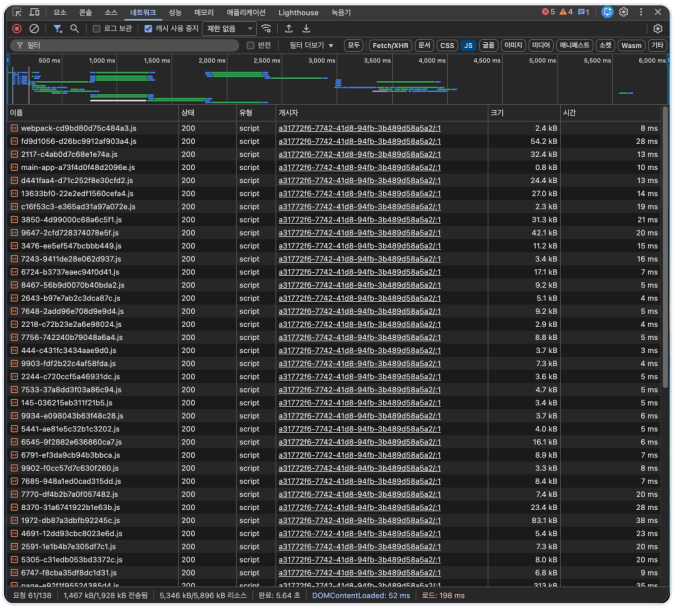
Comparing Next.js production builds, First Load JS on the trip detail page fell from 828kB to 452kB, a 45% reduction. By the median of 3 Lighthouse runs, LCP improved 28% from 6.9s to 5.0s and TBT improved 32% from 960ms to 650ms.

I checked real loading in the Network tab. The page JS file fell from 313kB to 51.3kB, and large Dialog chunks now download after the map scripts. JS transfer stayed similar at 1,467kB to 1,496kB, while requests rose from 61 to 75, confirming that only the download timing moved.

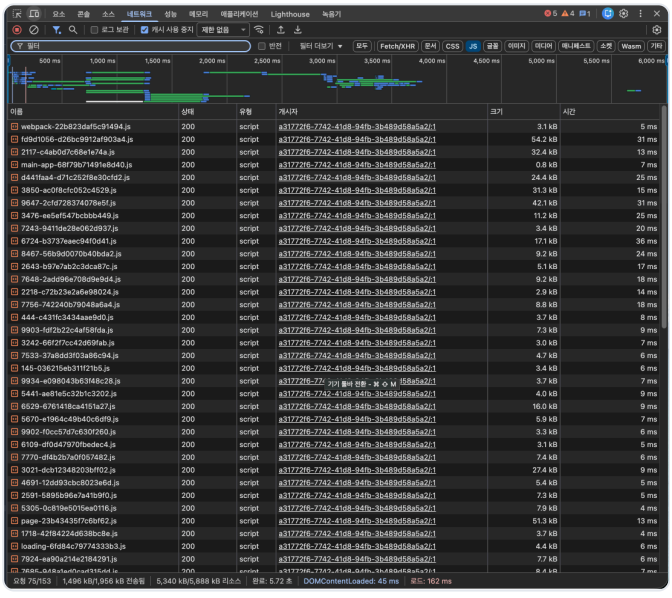
Separating map selection from schedule list rendering cut list renders on the first marker click from 16 to 0.

When only the parent re-renders from a map marker or place selection, the schedule list render is skipped. It still re-renders normally when schedules or routes actually change, so necessary updates stay while selection-related renders are gone.

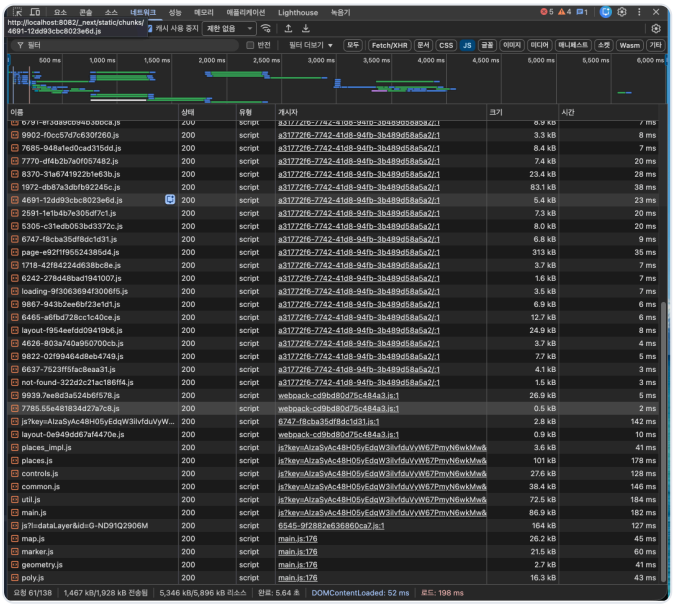
This did not reduce the total download size. Dialog chunks were excluded from initial entry and their download moved to browser idle time. A Dialog that has been opened also stays in memory to preserve input and the close animation, which I accepted because even all 11 mounted is no worse than the previous structure.



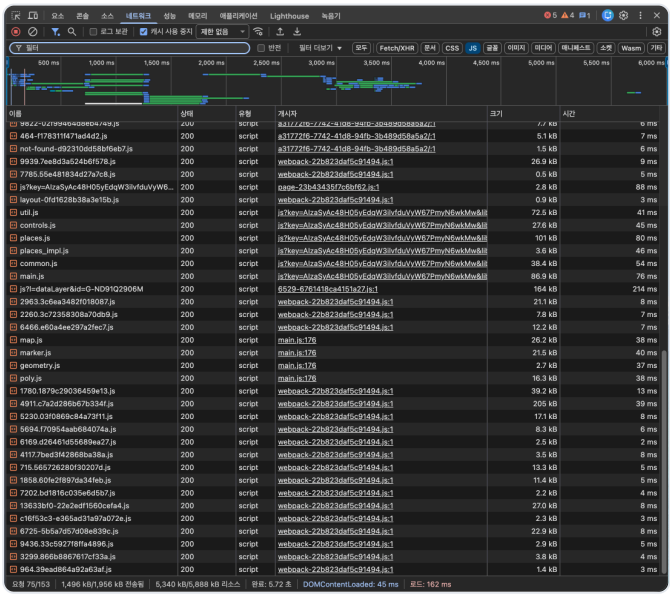
Before: page file 313kB



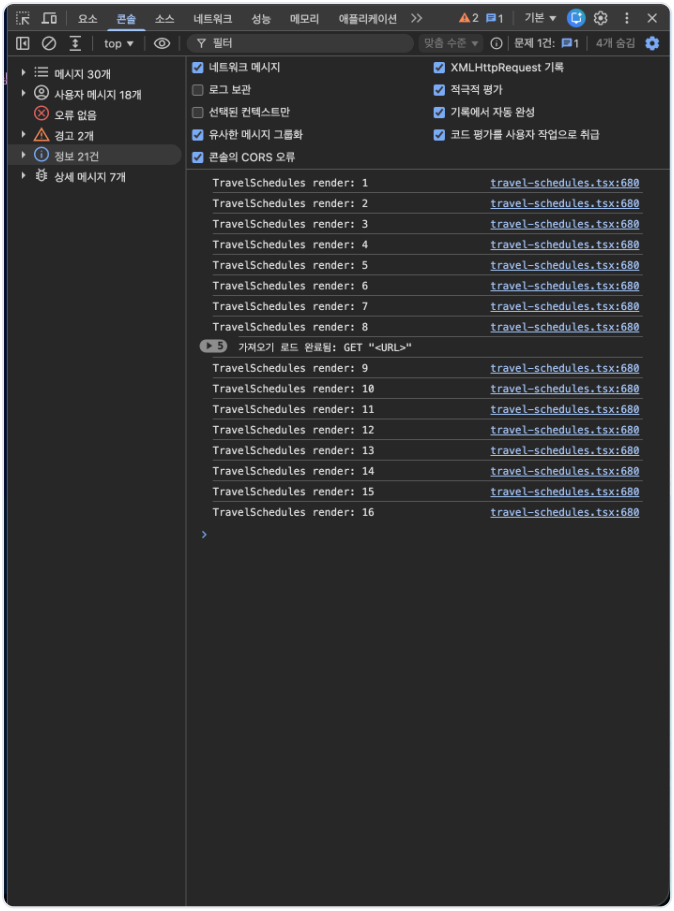
After: page file 51.3kB



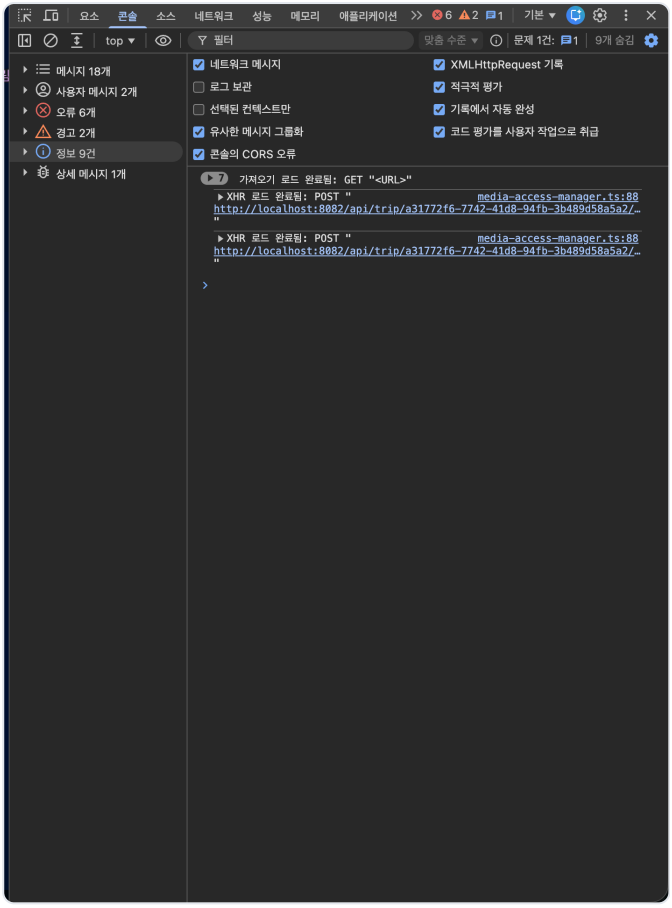
Before: no extra chunks after the map scripts



After: large chunks load after the map scripts



Before: render log on first marker click



After: no render log on first marker click

Learnings

Loading optimization: when code is downloaded mattered more than what was split

Even with next/dynamic, a dynamic component in the render tree from the start requests its chunk immediately, so the initial cost could not be delayed enough. Controlling first mount with LazyMount and covering first-click delay with idle prefetching taught me to design code splitting, mounting, and prefetching together. I also learned that a smaller First Load JS is not a smaller total download, so I need to measure which costs were removed and which were only moved.

Rendering optimization: check state and prop boundaries before reaching for memo

When the parent's map selection changes, children are called again, and even React.memo fails its shallow comparison if new arrays and callbacks are passed each time. I separated the data the list actually needs from selection state only the map needs, stabilized references, and applied memo only where needed. This taught me that tracing how far a state change spreads and narrowing the render boundary matters more than wrapping everything.

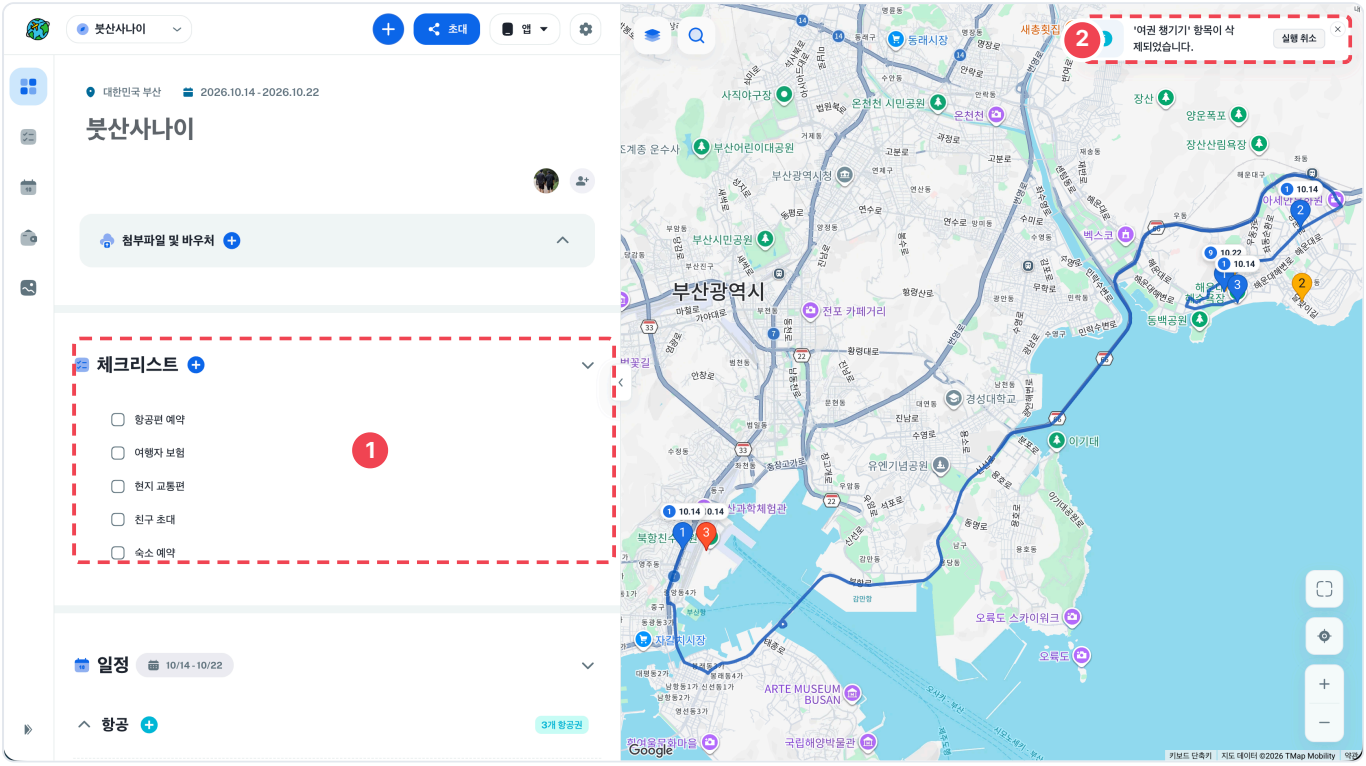
Case 2 · JustTrip

Abstracting a Delete Problem Repeated Across 7 Domains into a Shared, Undoable Structure

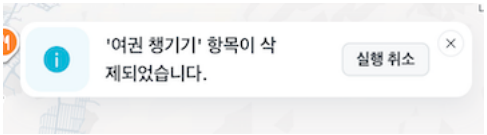
TypeScript, TanStack Query, fetch keepalive, Next.js

Problem

JustTrip lets travel companions edit a trip together, so items can be deleted across 7 domains: schedules, stays, flights, vouchers, expenses, checklists, and photos.



The trip detail page right after deleting a checklist item (Korean UI)



1 **Optimistic delete** The "Pack passport" item disappears from the list immediately, without waiting for the server.

2 **Undo** Users can undo for 4 seconds, and the delete request is sent to the server only after that.

Deleting used to open a confirmation Dialog every time. People edit and remove items constantly while planning a trip, so confirming each one was too inconvenient. I decided to remove the Dialog, delete immediately, and let users undo mistakes.

To support undo, the server cannot delete right away. Recreating an item issues a new ID, which breaks the links between flights, stays, schedules, or expenses and their vouchers. So I removed items from the screen first and sent the request 4 seconds later. During that window the screen and the server disagreed, and the same problems appeared in every domain.

- Reappearing items: the item still existed on the server, so refetching the list in that window brought the deleted item back on screen.
- Repeated refetching: deleting several items in a row, such as photos, refetched the entire list after every delete.

So I redefined deletion from "one request" to "a state change through pending, in-flight, and recently committed," and decided to build a shared structure so each domain would not have to implement this flow.

Decisions

> **How should the delete UX work?**
>
> **Rejected** Keep the confirmation Dialog: users must confirm every time when deleting dozens of photos
>
> It reduces mistakes but slows down every normal delete.
>
> **Rejected** Delete immediately and recreate on undo: a new ID is issued and the links break
>
> Flights, stays, schedules, and expenses can be linked to vouchers. Recreating a deleted item issues a new ID, breaking those links and producing a different item that only looks the same.
>
> **Chosen** **Deferred commit: remove from the screen first and send to the server 4 seconds later**
>
> After a delete, no request is sent for 4 seconds. The client only hides the item and remembers the original item and position. Undo simply shows the remembered item again, and because the server data is untouched, the ID and voucher links remain. If not undone, the delete request is sent after 4 seconds.

But deferred commit created a new problem. For 4 seconds the screen and the server disagreed, so refetching the list in that window brought the deleted item back.

```
0s  User deletes "Pack passport" → it disappears from the screen
1s  Another action refetches the list → the item still exists on the server
1s  The response overwrites the list → the deleted item reappears
4s  Only now is the delete request sent to the server
```

> **How can deleted items be kept from reappearing?**
>
> **Rejected** Block list refetches during the pending window: unrelated changes are delayed too
>
> Stopping refetches during the pending window keeps items from reappearing. But changes from companions and results of other actions also appear late, and the pause keeps growing during consecutive deletes.
>
> **Rejected** Soft delete immediately and restore on undo: the cleanest option, but the backend change is large
>
> Marking the item on the server right away keeps the screen and server in sync from the start. But it needs delete flags and restore APIs across 7 domains along with changes to every query, so it could not be applied right away.
>
> **Chosen** **Remember deleting IDs in three stages and filter them from every list response**
>
> A refetch overwrites the on-screen data with the server response. So before the response reaches the screen, items currently being deleted are filtered out once more.
>
> ```
> Checklist delete example
>
> [Before the delete request is sent]
> Server response: [Book flights, Pack passport, Travel insurance]
> Deleting: [Pack passport]
> List shown on screen: [Book flights, Travel insurance]
>
> [After the server delete completes]
> Server response: [Book flights, Travel insurance]
> Recently committed: [Pack passport] (cleared after 20s)
> List shown on screen: [Book flights, Travel insurance]
> ```
>
> At first it seemed enough to remember only items in the 4-second pending window. But refetches and delete requests can overlap. Once pending ends and tracking stops, the response to a query started earlier can arrive and bring the item back.
>
> ```
> 3.9s  List refetch starts (the item still exists on the server)
> 4.0s  Pending ends, delete request sent
> 4.2s  Delete succeeds
> 4.5s  Response from the 3.9s query arrives → it still includes the item
> ```
>
> So I split tracking into three stages, remembering the item briefly even after the delete finishes.

Stage	Timing	Why the item can appear in a response
Pending	4s after delete	The delete request has not been sent yet
In-flight	While waiting for the response	The server may process the query before the delete
Recent	20s after success	A query started before the delete may respond late

This filter lives in one place: the function that receives the list. Whichever screen refetches, and for whatever reason, the result is the same, and screen code does not need to know about delete state.

```
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');
}
```

That stopped items from reappearing, but consecutive deletes were still a problem. Deleting several photos refetched the entire list after each one, and closing the tab could lose deletes that had not been sent.

How should consecutive deletes and tab closing be handled?

Rejected Group refetches with debounce alone: a refetch runs in the middle when deletes are spaced out

Debounce runs the last of several calls made within a short time. It groups quick photo deletes well, but each delete has its own 4-second timer, so spaced-out deletes trigger a refetch while others are still pending. A longer debounce delays screen updates, and the right length has to be guessed.

```
0s    Delete photo A (sent after 4s)
1s    Delete photo B (sent at 5s)
4s    A delete completes → refetch scheduled
4.2s  No more completions, so refetch runs (B is still pending)
5s    B delete completes → refetch scheduled
5.2s  Refetch runs again
```

Rejected Send remaining deletes with sendBeacon: POST only, so it cannot send DELETE

Closing the tab during the 4-second window loses the unsent delete, leaving the item in place. sendBeacon is a browser API that finishes sending after the page closes, but it only sends POST, so it could not call the DELETE API as it was.

```
// Always sent as POST, so it cannot call the delete API
navigator.sendBeacon(url, body);
```

Chosen **Refetch only when no work remains in the group, and send with fetch keepalive when the tab closes**

I based refetch timing on the number of remaining deletes rather than time. In the example above, no refetch runs when A finishes because B remains, and it runs once when B finishes and none remain. A 200ms debounce stays as a helper that groups calls clustering at the end.

For tab closing, I used the fetch keepalive option. The request survives page close and can use DELETE, so this was solved without backend changes.

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

  // Do not refetch while pending or in-flight deletes remain
  if (activeCount === 0) {
    const timer = setTimeout(/* run refetch */, debounceMs);
    groupRefreshTimers.set(group, timer);
  }
}
```

None of this could be implemented separately in each of 7 domains. The data differed, but the delete flow was the same, so I extracted the flow into a shared structure.

How can this apply to 7 domains without repetition?

The shared module owns the flow that never changes, and each domain passes only the parts that differ as callbacks.

```
// Domains only say "how to delete"; they need not know about timing or races
requestUndoableDelete({
  id: `checklist-${item.id}`,
  group: 'checklist',
  title: item.content || 'Checklist item',
  onOptimisticRemove: () => { /* remove from cache */ },
  onRestore: () => { /* restore to original position */ },
  commitDelete: () => deleteChecklist(tripId, item.id),
  onSuccessRefresh: () => queryClient.invalidateQueries({ queryKey }),
  fallbackEndpoint: endpoints.travel.checklist_delete(tripId, item.id),
});
```

Area	Handled by the shared module	Passed by the domain
Delete flow	4-second delay, undo notification	Item info (id, group, title)
Screen updates	When to remove and restore	How to remove and restore
Server request	Three-stage tracking, restore on failure	Delete API call
Data sync	Response filtering, single refetch	How to refetch
Tab close	Sending remaining deletes	Delete endpoint

Results

7
Domains using the shared structure

1
List refetch for consecutive deletes

0
Backend API changes

A better delete experience

I removed the confirmation Dialog and moved to optimistic deletion, so users delete items instantly without waiting for the server. Mistakes can be undone within 4 seconds, and since the server data stays intact, IDs and voucher links are preserved.

Resolving mismatches between screen and server

Tracking deletes through pending, in-flight, and recently committed stages keeps deleted items from reappearing even when adding an item or another action refetches the list. If a server delete fails, the item is restored and the user is told, so the screen and server do not stay out of sync.

Optimizing consecutive delete requests

List refetches that ran after every delete now run once, when no deletes remain. Pending deletes are also sent with a keepalive request when the tab closes.

Unifying repeated code

The same delete structure applies to 7 domains with no backend API changes. Domains pass only how to delete and restore, while the shared module handles timing and race conditions. There is now one place to fix the delete flow.

Learnings

Before merging repeated code, define what is actually being repeated.

Seen as a single request, deleting a schedule and deleting a photo look like different features. Seen as state changing over time, the same flow of pending, in-flight, and recently committed emerged. Only then could I separate what changes from what does not, and the shared structure followed naturally.

Optimistic updates are not about changing the screen first. They are about managing inconsistency.

Optimistic deletion made things faster for users, but at the cost of a window where the screen and server differ. Refetches, undo, failures, and tab closing can all happen in that window. Handling every case in between took far more thought than making it feel fast.

Even when the backend cannot change, the client's responsibilities can still be defined.
I judged soft delete to be the root fix, but its scope was too large to apply right away. So I left the server as it was and built a client structure that prevents the problems users experience. Distinguishing the ideal solution from what is possible now also mattered.

Case 3 · Early Bird Alarm (React Native - iOS)



Early Bird Alarm

An alarm app that rings with an audio ad at the set time and gives reward points users exchange for gift cards.

2026.05 - 2026.10 · Product Planning, Full-Stack Development · iOS, Android

Keeping an Accidental Volume Button Press from Ending the Wake-up Mission

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

Problem

This alarm app does not end with the alarm ringing. Users must enter the app, listen to the audio ad, and press a button to get the reward. So even after the OS ends the alarm, it is not finished from the service's point of view until the user enters the app.

Android had no problem. The display-over-other-apps permission lets the app launch immediately when the alarm rings, so the alarm screen is the app screen. Users could snooze there, or take the reward and turn the alarm off.

iOS was different. I implemented the alarm with AlarmKit, announced at WWDC25, and AlarmKit alarms ring on the iOS system screen rather than in the app. Because the place the alarm rings and the app that gives the reward are separate, the key was getting users into the app after they dismissed the alarm.



AlarmKit system alarm screen (Apple WWDC25 sample)

	Android	iOS
Who draws the alarm	App	iOS system
Where the reward is	On the alarm screen	Requires entering the app
User steps	Finish mission on alarm screen	Dismiss → Enter app → Finish mission

However, AlarmKit's system screen divides buttons into only two actions.

- A **On-screen Dismiss, volume, and side buttons** Stop or snooze the alarm (grouped as one action)
- B **On-screen Repeat button** Snooze

An App Intent can be connected to button A through AlarmKit's stopIntent. Pressing the button runs the specified code, and I wanted to open the app and register the next alarm there.

The problem was execution order. With openAppWhenRun on in the App Intent, when button A ends the system alarm, the app launches first and only then does the Intent code that registers the next alarm run. On a locked phone, the app opens only after the user unlocks, so the registration code does not run until then either.

With a separate snooze button on screen, this may not look like a problem. But in the default iOS alarm, pressing a volume button makes the alarm ring again after the snooze interval. Users press the volume button expecting the same behavior, but with openAppWhenRun on, this app only opens and the alarm never rings again. If they pressed it half asleep, they can simply go back to sleep.

1. With openAppWhenRun on: the app opens, but for users who dismiss with the volume button on the lock screen, the next alarm is not registered and it does not ring again.
2. With openAppWhenRun off: the next alarm can be registered immediately even while locked, but the app does not open automatically.

```
openAppWhenRun = true
Volume press → App launches (unlock needed) → Intent code runs → Next alarm registered
                ↑ Next alarm cannot be registered without unlocking

openAppWhenRun = false
Volume press → Intent code runs immediately → Next alarm registered → App stays closed
```

I put a lot of thought into keeping the system alarm working the way users expect while still bringing them naturally into the app for the reward.

Decisions

To solve this, I considered the two options of openAppWhenRun being on or off.

> **Turn openAppWhenRun on and schedule every repeat in advance**
>
> **Rejected** Schedule 2 hours of alarms in advance and open the app: the alarm count grows sharply and the safe limit is unknown
>
> When the app schedules an alarm, it also schedules 2 hours of repeat alarms in advance. Pressing button A dismisses the alarm and opens the app with openAppWhenRun on. Even if users do not unlock after pressing A, or press the volume button by mistake or to sleep longer, the next alarm is already scheduled with no Intent code needed, so nothing breaks. The scheduled alarms are canceled at the next occurrence or when the user enters the app.
>
> 
>
> Screen with no snooze button (Apple WWDC25 sample)
>
> A **On-screen Dismiss, volume, and side buttons** Opens the app. Ringing again is left to the alarms scheduled in advance.

```
Volume press → App launches (unlock needed)
                ↑ Scheduled alarms still ring without unlocking
```

But the number of alarms grew sharply. For example, scheduling 2 hours ahead for a daily repeating alarm with 1-minute snooze means 1 × 60 × 2 × 7 = 840 alarms for one alarm the user created. AlarmKit has a limit error (maximumLimitReached), but the official documentation does not state the exact number. Without knowing the limit, I could not judge a safe range.

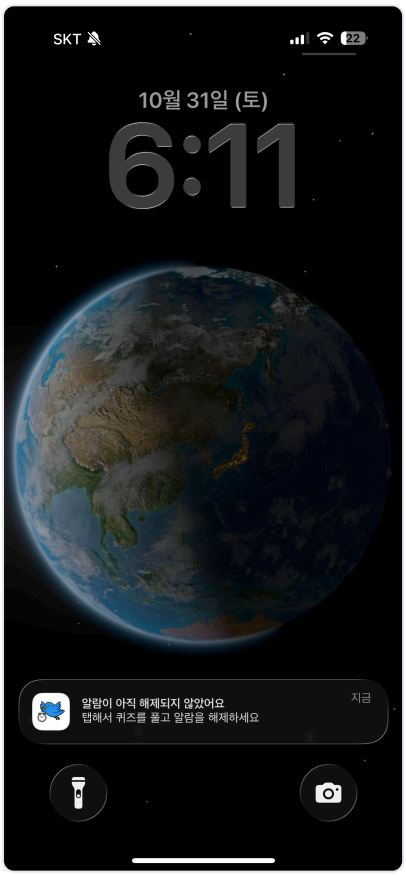
Turn openAppWhenRun off and register the next alarm as needed

Chosen Schedule one alarm with openAppWhenRun off, and register only the next alarm each time a button is pressed

When the app schedules an alarm, it schedules a single alarm at the time the user set and turns openAppWhenRun off. Pressing button A turns off the ringing alarm and runs the Intent. Because the app may be closed or the screen locked at that moment, the next alarm is scheduled without JS by a background App Intent that runs briefly when the alarm ends.

The Intent checks that snooze is enabled for the alarm, registers one next alarm, and sends a banner notification saying the alarm is not finished. The next alarm stays even if the user taps the banner or opens the app directly, because it must still ring if they close the app while watching the ad. Once the user gets the ad reward, the remaining alarm is canceled and it stops ringing.

When an alarm is registered, its settings are saved in App Group, storage both the app and the Intent can read. When the button is pressed, the Intent works through the following steps.



Banner on the lock screen after dismissing the alarm (Korean UI)

```
Volume or stop button pressed (app stays closed)
  ↓
Intent starts
  ↓
Read alarm settings from App Group
  ↓
Record the active occurrence
("alarm rang but the ad has not been watched")
  ↓
Register one next alarm
  ↓
Show banner notification
  ↓
Intent ends
  ↓
User taps the banner or opens the app directly
  ↓
App reads the record and moves to the ad screen
```

Native code saves state while the app is closed, and the app picks it up when it opens. The alarm count does not grow and the behavior users expect is preserved, but the app cannot open automatically. In exchange it runs reliably, and the system alarm itself works the way users expect.

Full flow

- Registering an alarm: save settings to App Group and record this ring (occurrence) so both the app and native code know about it

- Pressing the snooze button (B): the OS rings again after the set interval. The app is not involved, and the next ring continues into one of the flows below
- Pressing A: register one next alarm and show a banner prompting the user to enter the app. The repeat schedule stays as it is
- Tapping the banner or opening the app: check for an unfinished occurrence and go straight to the ad screen. A ringing alarm stops, and the next alarm stays so it rings again if the user closes the app during the ad
- Getting the ad reward: cancel the remaining alarm and banner to end the occurrence. The next day's alarm stays
- Alarms with snooze off show only the banner, and alarms with snooze on continue for at most 24 hours so they do not overlap the next occurrence

Results

Keeping the alarm behavior users expect
The alarm is set up to ring again even when dismissed with the volume or side button, matching the expectations of users familiar with the default iOS alarm. The basic alarm behavior was not given up in order to open the app.

A reward flow that accounts for the app being closed
Even with the app closed or the screen locked, the next alarm is registered without JS and the active occurrence is recorded natively. When the user enters the app later, it reads the record and goes straight to the ad screen.

A structure that does not multiply alarms
Scheduling in advance could turn one alarm into as many as 840. Registering only the next alarm on each button press removed the dependence on a limit that could not be confirmed from the documentation.

The trade-off
The app does not open automatically, so users have to tap the banner once. In actual use, though, the banner appears at the top of the screen within a second of dismissing the alarm.

Learnings

When a platform constraint cannot be worked around, decide based on the most damaging case.
I looked for a way to get both app launch and ringing again, but in the configurations I tried, one had to give. I judged that a user dismissing the alarm and going back to sleep was far worse than entering the app late. Using the damage of failure as the standard, rather than wanting every feature, made the decision clear.

A service's definition of done can differ from the OS's.
To the OS, an alarm ends when it is dismissed, but in this service it ends when the user watches the ad and gets the reward. So I recorded "the alarm rang but the ad has not been watched" as its own state. Because of this, even if the alarm ends while the app is closed, the app can pick up that state when it opens and send the user to the ad screen.

Even in a React Native app, first separate what JS cannot do.
JS does not run while the app is closed. So I first divided what native code and JS would each handle, and connected them through shared storage. My perspective widened from frontend code alone to the execution state of the app.

Case 4 · RiderTown V2

RiderTown V2

Pivoted from a motorcycle maintenance quote and booking platform to a service for measuring rides and sharing location and records with a crew in real time.

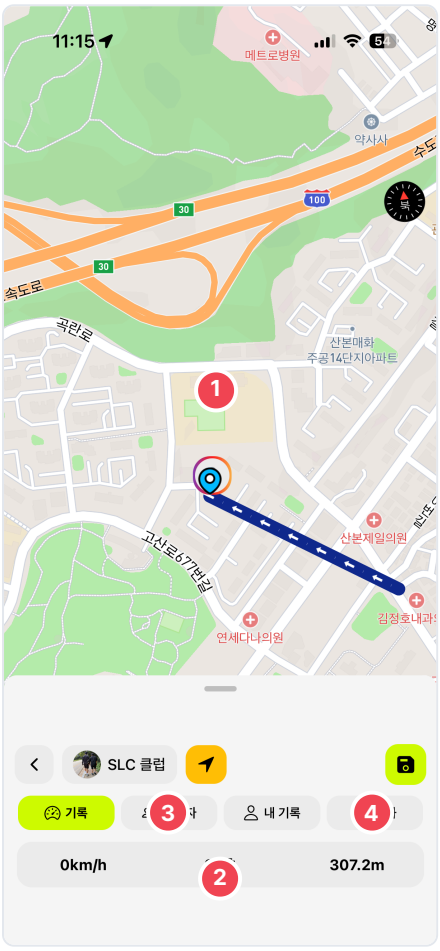
2024.06 - 2024.12 · Co-founder, Full-Stack Development · Team of 2 with a sales co-founder · Released on iOS and Android

Real-Time Riding That Keeps Working While the App Is in the Background

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

Problem

When the backend developer left, I had to run everything from frontend to backend alone. To unify operations on JavaScript and reduce server costs, I was migrating the AWS infrastructure to Firebase.



RiderTown real-time riding screen (Korean UI)

The riding service recorded ride time, speed, distance, top speed, and banking angle, and shared current location and records with the crew in real time. Riders could also talk to their crew in live voice calls while riding.

1. **Live location and route** Shows current location and route on the map and shares them with the crew.
2. **Ride record** Shows speed, time, and distance in real time, and records top speed, banking angle, and more after the ride.
3. **Crew members** Shows the location and ride records of crew members riding together in real time.
4. **Voice call** Riders talk to their crew in real time through a Node.js-based Mediasoup SFU server.

Ride records such as location, speed, and distance were shared every second over WebSocket, and joining, leaving, and participant state for voice rooms on the Node.js Mediasoup SFU server were also managed over WebSocket. Everything worked fine while the app was in the foreground.

But riders using navigation or riding with the screen off keep the app in the background most of the time. In that state, ride records and crew sharing cut out, and I initially defined this as two problems.

- Background execution: the structure where JS periodically collects and sends location stops in the background.
- WebSocket connection: the WebSocket connection is not maintained in the background, stopping record sharing and voice room participant updates.

To find out whether these were separate problems or both failing together, I tested them one after another.

Decisions

> **Verifying background execution**
>
> **Rejected** Collect location with JS timers: callback execution cannot be guaranteed in the background
>
> Once the app moves to the background, JS timers alone cannot reliably keep up location collection every few seconds. Without location, records and sharing cut out together.
>
> **Chosen** Record based on location events from native BackgroundGeolocation
>
> Location tracking stayed alive in the native layer through iOS Background Modes and Android's Location Foreground Service. Instead of JS requesting location periodically, records were based on native events delivered through onLocation.
>
> In other words, this does not keep JS alive; the JS callback runs when the native location session delivers an event. This covers the background and screen-off states. If the app is force-quit, recording does not continue, and the remaining ride state is cleaned up on the server through onDisconnect.

The test showed there was really one cause: JS stops when the app goes to the background, and the WebSocket disconnect was a result of that. So fixing the cutouts was not about replacing WebSocket but about keeping the app running in the background.

Even so, I decided to move real-time communication from WebSocket to Realtime Database. Network drops still happen during rides even with the app running, so keeping WebSocket meant implementing reconnection and exception handling myself. The Firebase migration was also underway, and with Realtime Database there was no need to keep a WebSocket server for location sharing on AWS, which reduced costs too.

```
iOS Core Location / Android Foreground Service
  ↓ Native location event
react-native-background-geolocation (onLocation)
  ↓ JS callback runs when the event arrives
Compare current time with last send time (5s in background)
  ↓ Pass the record if the condition is met
React Native Firebase native SDK → Realtime Database
```

> **Comparison with WebSocket**
>
> Moving from WebSocket to Realtime Database brought many advantages in implementation effort and cost.

Item	Keeping WebSocket	Realtime Database
Real-time delivery	Implement sending to room members	Crew members subscribe to a node and get changes
Reconnection	Implement reconnection logic	Handled by the SDK
Writes while offline	Implement local buffering and ordering	SDK queues and sends writes
Normal end	Implement end event and server cleanup	Client ends the ride state
Abnormal end	Detect disconnects and implement cleanup	Register onDisconnect; Firebase cleans up
Missed cleanup	Implement expiry on server or client	Exclude markers 2 minutes after last update
Server operations	Deploy and monitor socket servers alone	No socket server to run
Cost	Fixed server cost even with no users	Usage-based with no fixed server cost

During the move to Realtime Database, I also improved data transfer efficiency.

> **How much location data should be sent, and when?**
>
> **Rejected** Send every second: data keeps flowing at red lights and rest stops, and volume grows with ride length
>
> **Chosen** Situation-based send intervals and route compression

- Send interval: on each location event, compare with the last send time and use 5s in the background, 30s when stopped for over 15s, and 2-4s by speed while riding.
- Route compression: add a point only past a speed-based distance threshold (20/40/60m), then encode with Google Encoded Polyline.

Results

Moving to native location events kept ride records and crew location sharing working with the screen off or during navigation.

Applying onDisconnect and a 2-minute marker expiry keeps stale rider locations from staying on the map after an abnormal exit.

Situation-based send intervals and route compression cut real-time route data transfer by over 70%.

Replacing the location-sharing WebSocket server with RTDB reduced server operations, and delegating crew events and reconnection to the SDK simplified the real-time sync logic.

Learnings

Separate where a symptom appears, what actually causes it, and why a technology was replaced.
I first thought the background sharing cutouts were a WebSocket problem, but checking collection and transmission separately showed the core cause was location collection that depended on JS timers. Native location tracking fixed that, and moving to RTDB was a separate choice to reduce connection management and operating costs.

I also learned that sending data often is not always better for a real-time service. Adjusting send intervals and route precision by riding state taught me to design record accuracy and efficient delivery as separate concerns.