

데이터로 사용자 경험을 입증하고
AI로 개발 효율을 최적화하는
프론트엔드 개발자입니다.

About

Name 박철현

Birth 2000.10.16

Phone 010-3180-7113

Email play3step@naver.com

Github <https://github.com/play3step>

blog <https://velog.io/@play3step/posts>

Education

Education 한국공학대학교 소프트웨어학과 (2019.03 ~ 2026.02)

Certification 정보처리기사 (2025.02)

Experience

실무급 코드 퀄리티와 협업 프로세스 체득

2025.05 ~ 2025.10 | 프로그래머스 데브코스 (프론트엔드)

- 최우수 프로젝트 선정: 4회 프로젝트 중 2회 최우수 팀 선정
- 기술 심화: React, Next.js, TanStack Query 기반의 심화 학습 및 성능 최적화 연구
- 지식 공유: 21명 규모의 기술 스터디 운영 및 주 1회 기술 세션 주도

Jira 기반의 애자일 프로세스 및 팀 리딩 경험

2024.04 ~ 2024.10 | 2024 한이음 ICT 멘토링 (팀장)

- 팀 리딩: 멘티 팀장으로서 WBS 수립 및 일정 관리, 현업 멘토와의 커뮤니케이션 주도
- 성과: 프로젝트 'MoreView', 'PlanDing' 완주 및 최종 수료

레거시 코드 리팩토링 및 성능 최적화 경험

2024.01 ~ 2024.02 | 헬퍼로보틱스 (Frontend Intern)

- 성능 개선: React.lazy 및 Suspense 도입으로 초기 로딩 속도 단축 (Code Splitting)
- 디자인 시스템: 하드코딩 스타일을 디자인 토큰으로 상수화하여 유지보수 효율 증대
- UI/UX 개편: 디자이너와 협업하여 서빙 로봇 관제 시스템 UI 고도화

웹 기초 역량 확보 및 동료 학습 문화 주도

2023.09 ~ 2024.01 | UMC 5기 (대학생 개발 연합 동아리)

- 피어 러닝: 주 1회 정기 스터디 및 코드 리뷰 주도로 동반 성장 문화 정착



MapleLink

프로젝트 소개

Nexon API 기반 길드 데이터 분석 및 시각화 플랫폼

"데이터 관리의 비효율을 자동화로 해결하다."

[[GitHub](#)] [[Live Demo](#)]

프로젝트 기간 2025.03 ~ 2025.06

문제점 인게임 정보와 외부 커뮤니티(Discord 등)로 파편화된 길드 운영 데이터로 인해 관리 효율 저하.

해결책 Nexon Open API를 기반으로 길드원 정보를 실시간 동기화하고 시각화하는 온라인 대시보드 구축.

인원 2명(FrontEnd 1, BackEnd 1)

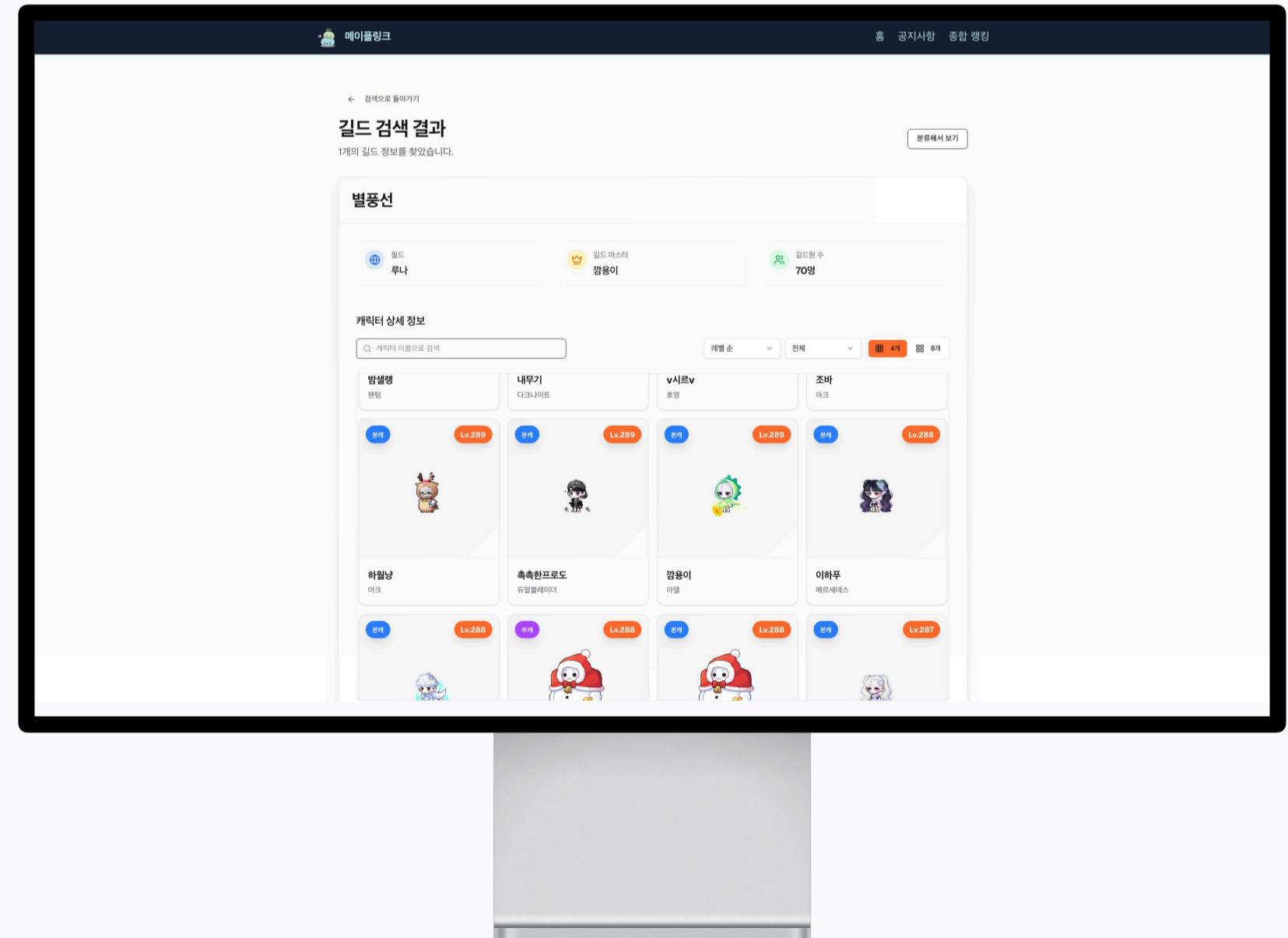
기술스택

Core Next.js, React, TypeScript

Styling Tailwind CSS, shadcn/ui (Radix UI)

State TanStack Query, Zustand

Infra & DB Supabase, Vercel





Feature 1. 실시간 데이터 동기화

기술 TanStack Query + Nexion Open API

설명 API 폴링 및 캐싱 전략을 통해 게임 내 변동 사항을 대시보드에 즉각 반영하여 데이터 최신성 유지.

Feature 2. 데이터 무결성 전략

기술 Partial Cache Update (부분 갱신)

설명 멤버 1명 정보 수정 시 전체 리스트를 다시 불러오는 비효율을 개선. 부분 캐시 업데이트로 네트워크 비용 절감 및 깜빡임 없는 UX 구현.

Feature 3. 길드 캘린더

기술 FullCalendar + dayjs + Zustand

설명 공식 이벤트 일정과 길드 자체 일정을 통합 관리. 복잡한 날짜 연산을 경량 라이브러리로 최적화.



무료 인프라로 150만 데이터 파이프라인 구축하기

Phase 1. 환경 설정

“10초 안에 끝내거나, 돈을 내거나.”

Trade-off

- Vercel Serverless: 무료지만 10초 타임아웃 제한으로 150만 건 크롤링 불가능.
- AWS EC2: 안정적이지만, 학생 개발자로서 매달 발생하는 서버 비용 부담.

결론

- 무료이면서 최대 6시간 제공을 해주는 GitHub Actions의 Matrix 기능 활용

Phase 2. 데이터 생성

“매일 200만 건을 쓰는데 500MB를 넘지 말 것.”

Trade-off

- UPSERT (수정): 정석적인 방법이지만, PostgreSQL 특성상 삭제 흔적이 누적되어 일주일 만에 용량 초과.
- TRUNCATE & INSERT: 지우고 다시 채우는 짧은 시간 동안 조회 시 데이터가 없을 수 있음.

결론

- 데이터 연속성을 일부 포기하더라도, 매일 테이블을 비우는 'Full Refresh' 전략으로 전환하여 삭제 흔적을 차단

Scheduler (매일 새벽 4시)



Job 1: DB Truncate (RPC)



Job 2: Matrix 생성 (30개 청크)
[1-250] [251-500] ... [7251-7500]



Job 3: 병렬 실행 (max 10 동시)



Phase 3. 데이터 모델링

“텍스트 몇 글자가 DB 용량을 위협하다.”

Trade-off

- DB 저장: 긴 직렬화 문자열 저장 시 DB 용량 60% 잠식.
- 지연 로딩: 저장 없이, 상세 페이지 진입 시 API 호출로 대체.

결론

- 저장 비용을 네트워크 비용으로 치환하여, 이미지 데이터의 스토리지 점유율 0MB를 달성했습니다.

결과

- 소요 시간: 약 1시간 47분 (150만 건 수집 기준)

Distributed Maple Ranking Update
Distributed Maple Ranking Update #9: Scheduled
main
Today at 4:24 AM
1h 47m 20s

- 데이터베이스 용량: 최종 417MB (500MB 한도 내 안착)

NAME	DESCRIPTION	ROWS (ESTIMATED)	SIZE (ESTIMATED)	REALTIME ENABLED
maple_characters	No description	1,457,504	417 MB	X

+ New table
8 columns



데이터 무결성 전략: 부분 캐시 갱신

문제 상황

- 멤버 1명의 정보를 수정하기 위해 리스트 전체(50명+)를 refetch 하는 것은 비효율적
- 전체 로딩 스피너가 돌면서 사용자 흐름이 끊김.

상태 관리 비교

Full Refetch (전체 재호출)

"무식하지만 가장 확실한 방법"

- 데이터 변경 시 전체 목록을 다시 가져옴.
- 사용자 빈도가 낮은 초기 로딩이나 간헐적 업데이트.
- 예: 쇼핑몰 상품 목록 필터링.
- 네트워크 낭비 & 깜빡임 발생

Optimistic Update (낙관적 업데이트)

"성공을 확신하고 미리 그리는 방법"

- 서버 요청과 동시에 UI를 먼저 바꾸고, 실패 시 롤백.
- 사용자의 즉각적인 반응이 필수적인 경우.
- 예: 인스타그램 '좋아요', 카카오톡 채팅 전송.
- 데이터 불일치 위험 & 롤백 복잡

Partial Cache Update (부분 갱신)

"확인 후 정밀 타격하는 방법"

- 서버의 성공 응답을 확인한 후, 캐시 내부의 특정 데이터만 교체.
- CRUD 후 즉각적인 반영이 필요한 대시보드/폼.
- 예: 엑셀 스타일의 데이터 수정, 노션(Notion)의 블록 편집.
- 검증된 신뢰성(Reliability) + 자원 효율(Efficiency)

```
queryClient.setQueryData(['guildsInfo', guildList, server], (oldData) =>
{ return oldData.map(guild => ({
...guild,
guildMember: guild.guildMember.map(member => {
const match = response.find(res => res.memberName === member.name)
return { ...member, ...match } // 즉시 병합
}))
}))})
```

결과

- API 호출 횟수 대폭 감소 - UI 변경 시 즉시 반영, 검증만 서버 통신
- 실시간 업데이트 느낌 - 사용자는 즉각적인 반응 체험 (5~10초 → 0초)
- 데이터 일관성 유지 - onError로 실패 시 자동 롤백
- 네트워크 최적화 - 변경된 데이터만 처리



인터랙티브 AI 챗봇 이력서

"보여주는 포트폴리오가 아닌, 경험하는 포트폴리오."

[[GitHub](#)] [[Live Demo](#)]

프로젝트 기간 2025.12 ~ 진행중

문제점 일방향 소통: 지원자는 많은 정보를 담고 싶지만, 채용 담당자는 필요한 정보만 빠르게 찾길 원함.

해결책 AI 채용 어시스턴트: LLM 기반 챗봇을 도입하여, 채용 담당자가 궁금한 기술 스택과 프로젝트 경험을 대화형으로 즉시 답변.

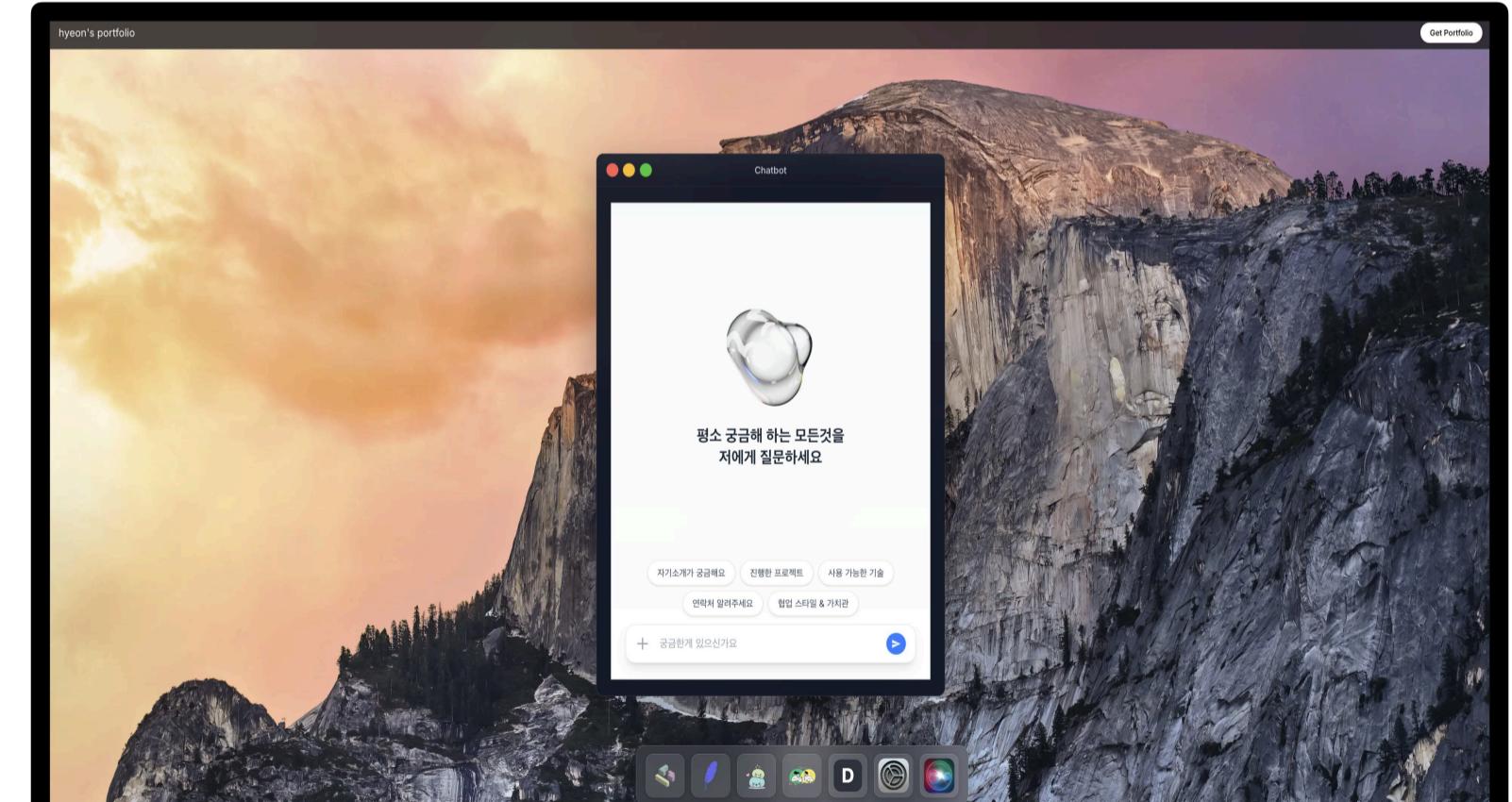
인원 1명(FrontEnd 1)

기술스택

Core Next.js, Framer Motion

AI Infra Gemini API, Upstash Redis (Rate Limiting)

State Zustand





Feature 1. 비용 0원 AI 아키텍처

기술 Upstash Redis + Gemini Flash

설명 트래픽 제어(Rate Limiting), 정적 필터링, 멀티 모델 폴백 시스템을 구축하여 유지비 0원의 지속 가능한 AI 서비스 구현.

Feature 2. 지능형 스크롤 UX

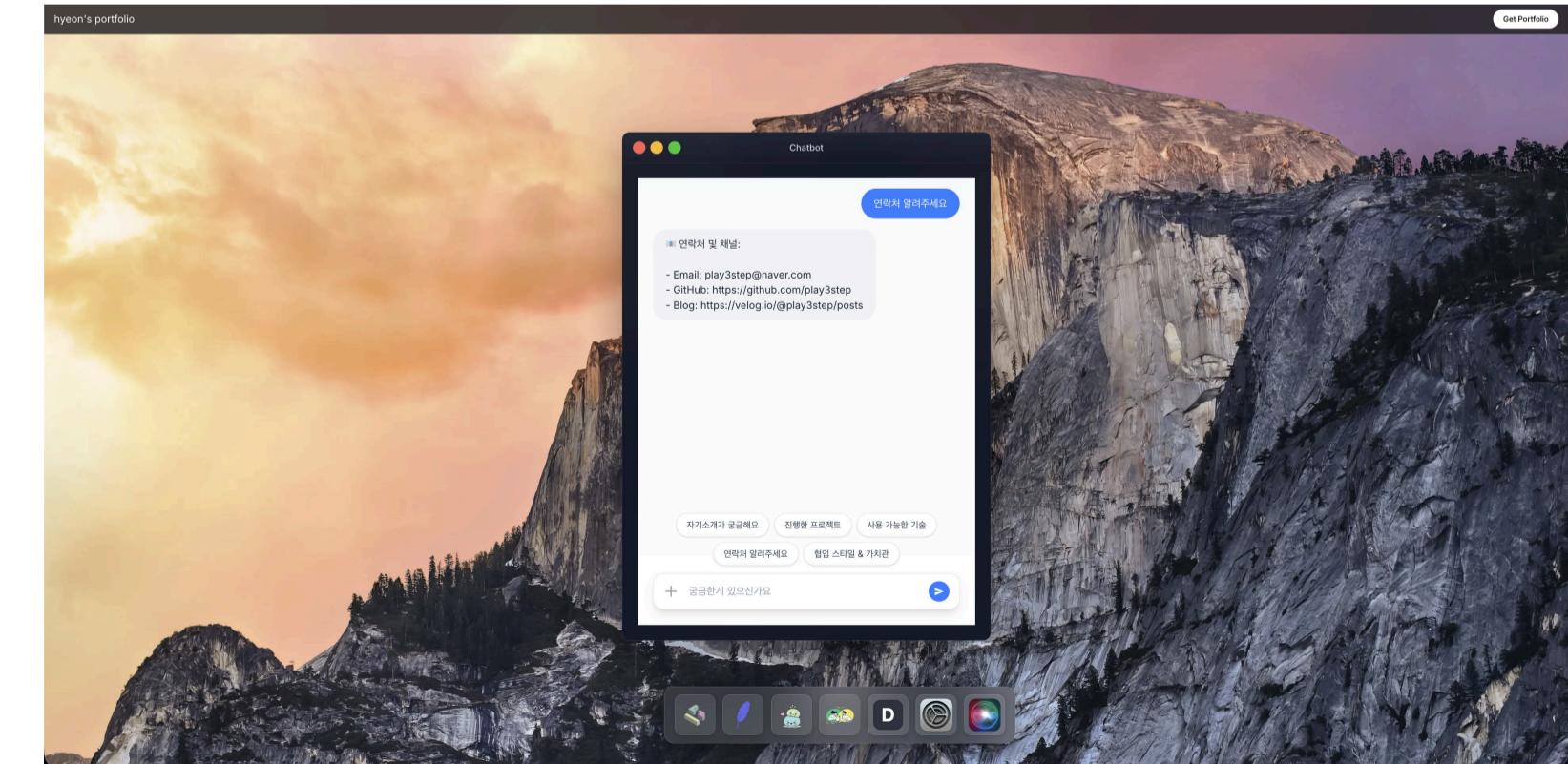
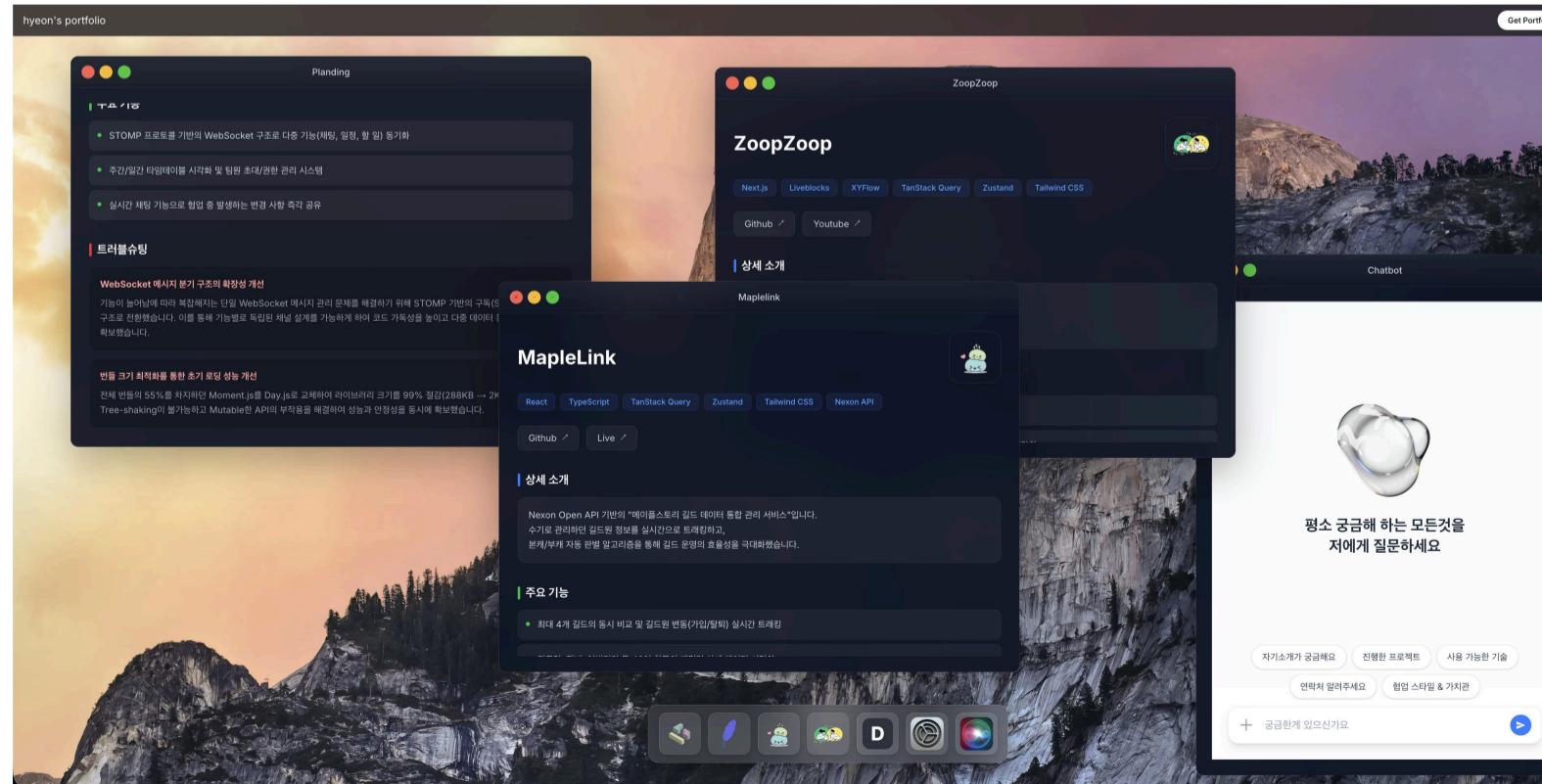
기술 ResizeObserver

설명 AI 스트리밍 답변 시 발생하는 화면 밀림(Layout Shift) 현상을 해결하는 앱커 기반 스크롤 시스템 개발.

Feature 3. 데스크탑 경험 구현

기술 Framer Motion

설명 창 이동, 최소화, 독(Dock) 애니메이션 등 네이티브 OS의 경험을 구현



지속 가능한 AI 서비스: 3단계 방어 아키텍처

무분별한 호출과 비용 폭증을 막고, 서비스 가용성을 확보

Phase 1. 트래픽 제어

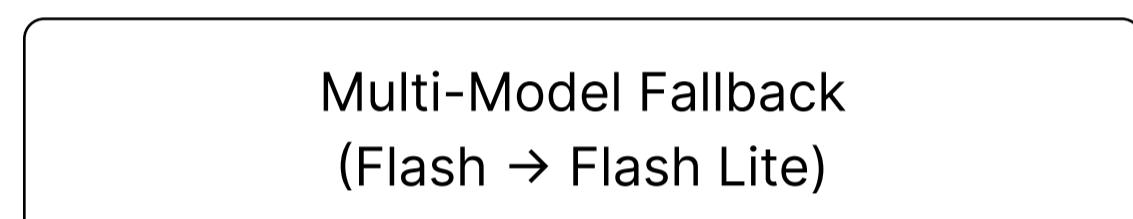
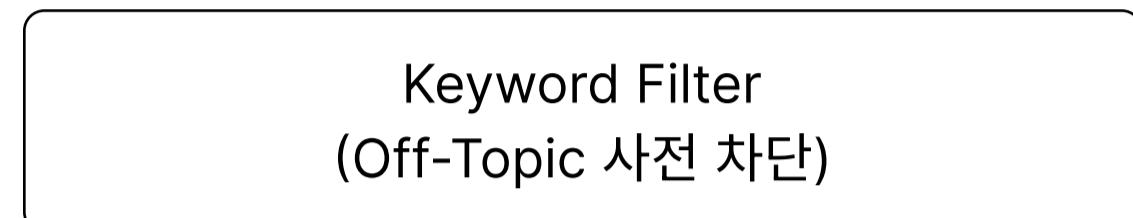
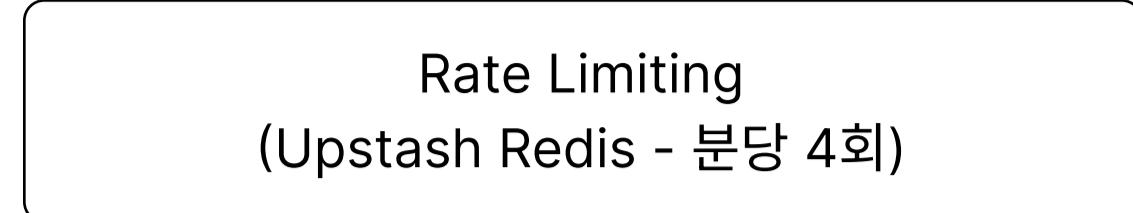
“서버는 꺼져도, 차단은 계속되어야 한다.”

Trade-off

- In-Memory Map: 구현은 쉽지만, 서비스(Vercel) 인스턴스 재시작 시 차단 기록 증발.
- Database: 기록은 확실하지만, 단순 카운팅을 위해 DB를 호출하는 건 오버헤드.

결론

- 서버 상태와 무관하게 글로벌 상태 공유가 가능한 Upstash Redis를 도입하여 영구적 차단 환경 구축.



Phase 2. 서비스 가용성

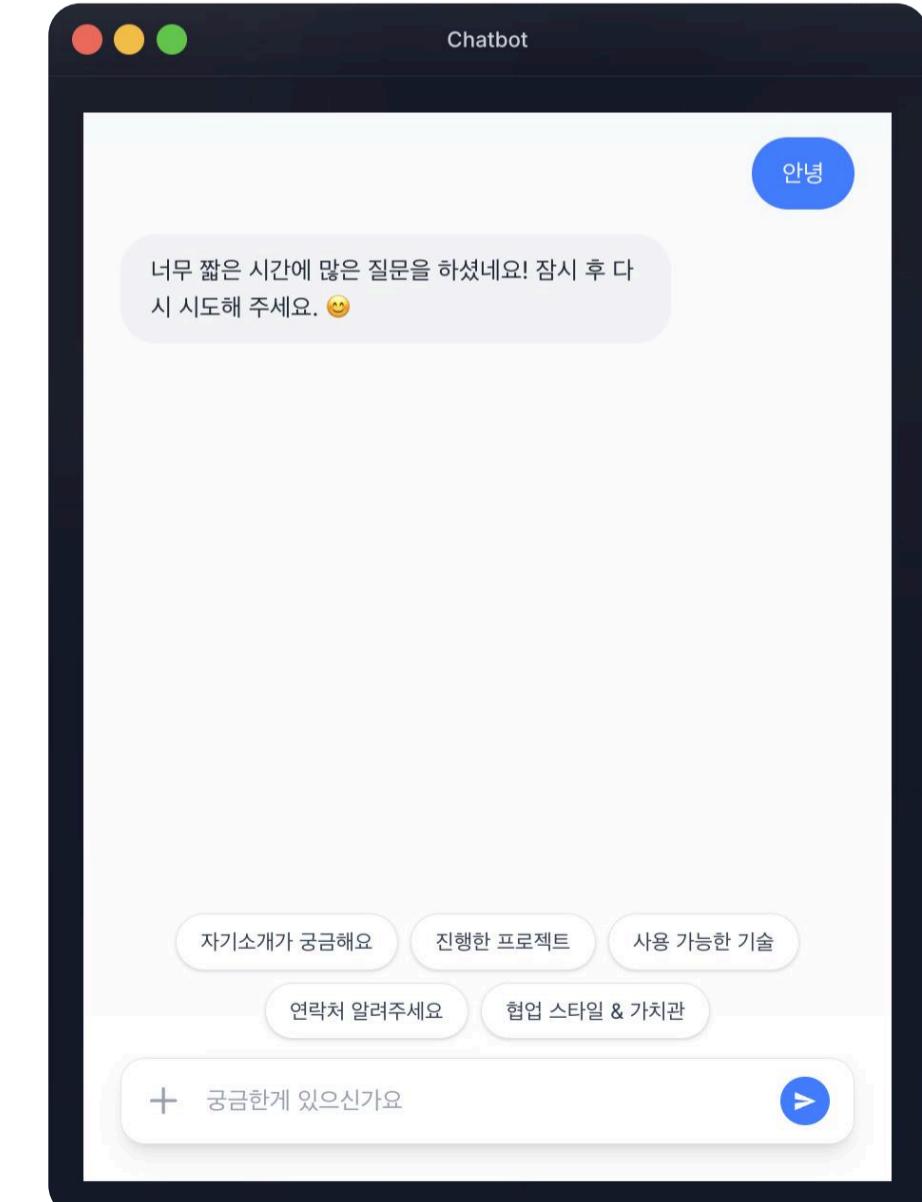
“하나가 막혀도, 서비스는 멈추지 않는다.”

Trade-off

- Single Model: 메인 모델 할당량 초과(429 Error) 시 서비스 전체가 멈추는 SPOF(단일 실패 지점).
- Premium Plan: 돈을 내면 해결되지만, 비용 문제.

결론

- 메인 모델 실패 시 즉시 경량 모델로 전환하는 Multi-Model Fallback(플백) 로직으로 가동률 99.9% 방어.





Phase 3. 비용 최적화

“잡담에 토큰을 낭비하지 않는다.”

Trade-off

- AI 분류: 질문 분류에도 AI를 쓰면, 무관한 질문에도 토큰 비용 발생.
- 정적 필터: 유연성은 떨어지지만, 비용이 완전 무료.

결론

- 날씨, 주식 등 무관한 키워드를 Rule-based로 사전 차단하여 불필요한 호출 0건 달성.

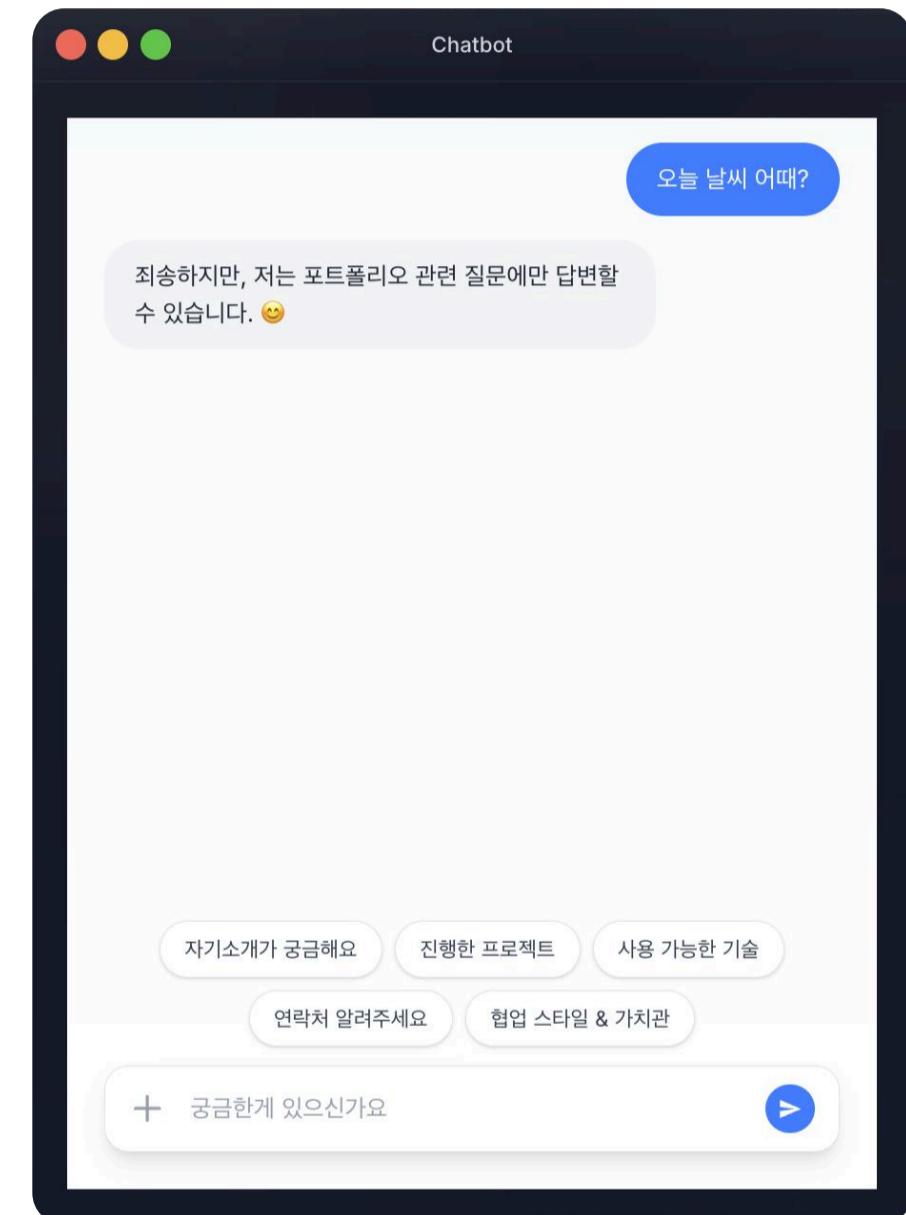
그 외

“질문을 고민하게 만들지 않는다.”

'자기소개', '기술 스택' 등 핵심 질문을 버튼으로 미리 제공하여, 클릭 한 번으로 자연스럽게 포트폴리오 탐색 유도.

결과

- 가용성 극대화: 비정상 요청을 사전에 차단하여 실제 방문자를 위한 API 할당량을 상시 확보.
- 안정적 운영: 별도의 인프라 비용 지출 없이 무료 티어만으로 중단 없는 챗봇 서비스 운영 성공.





AI 스트리밍을 위한 채팅 UX의 재정의

스크롤 UX

"AI 답변은 '대화'가 아니라 '읽기'다."

Trade-off

Bottom-Up (일반 채팅 방식):

- 무조건 최하단 강제 스크롤. 답변이 길어지면 질문(맥락)이 화면 위로 밀려남.

Anchor System (시선 고정):

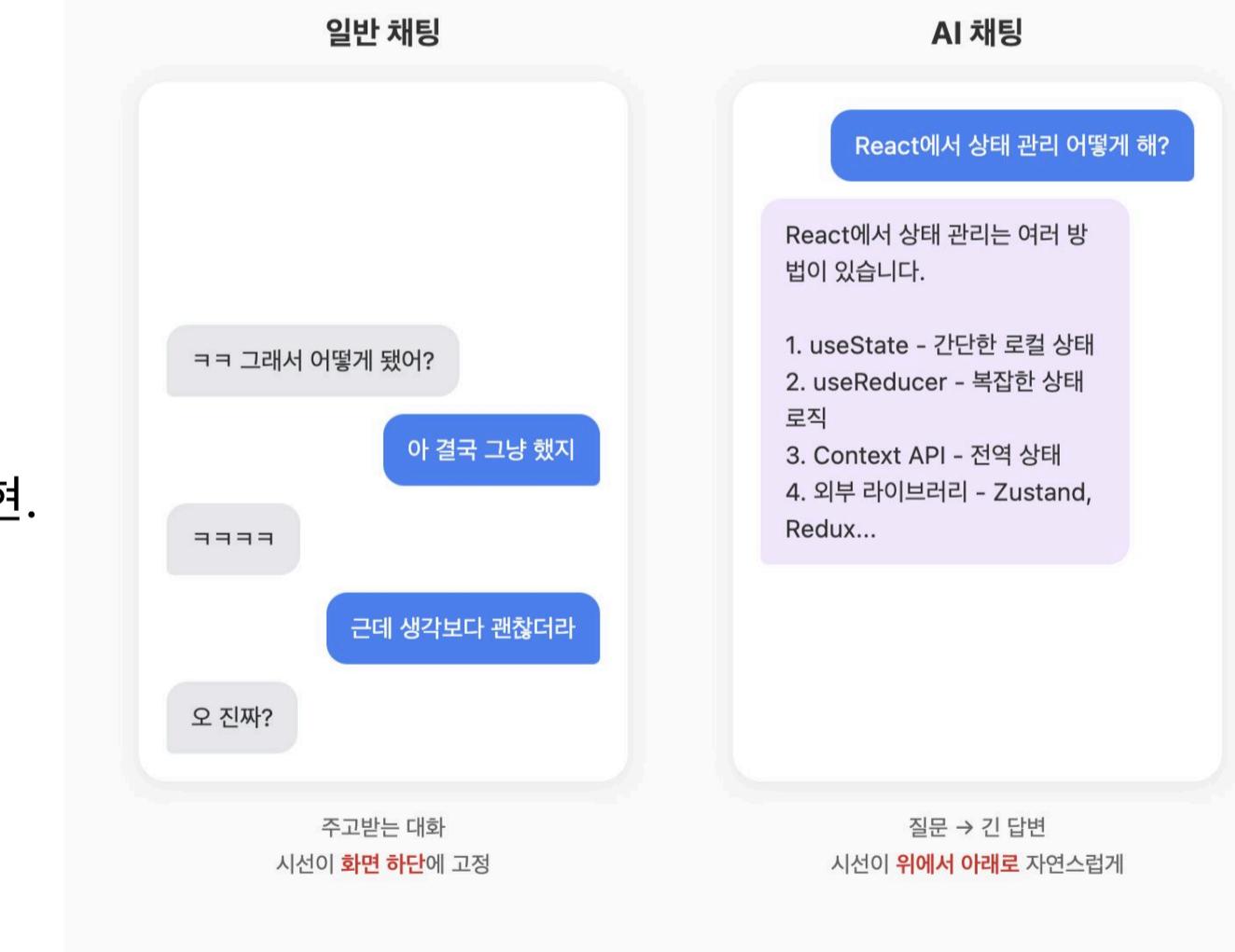
- 마지막 질문을 앵커로 고정. 답변이 아무리 길어져도 사용자의 시선 시작점 유지.

결론

- ResizeObserver로 높이 변화를 실시간 감지하여, Layout Shift가 없는 지능형 스크롤 구현.

결과

- 맥락 유지: 답변이 길어져도 질문이 항상 상단에 노출되어 대화의 맥락을 놓치지 않음.
- 자연스러운 UX: 사용자가 읽기 시작하는 시점과 AI 답변의 시작점을 일치시켜 정보 습득 효율을 증가시킴.





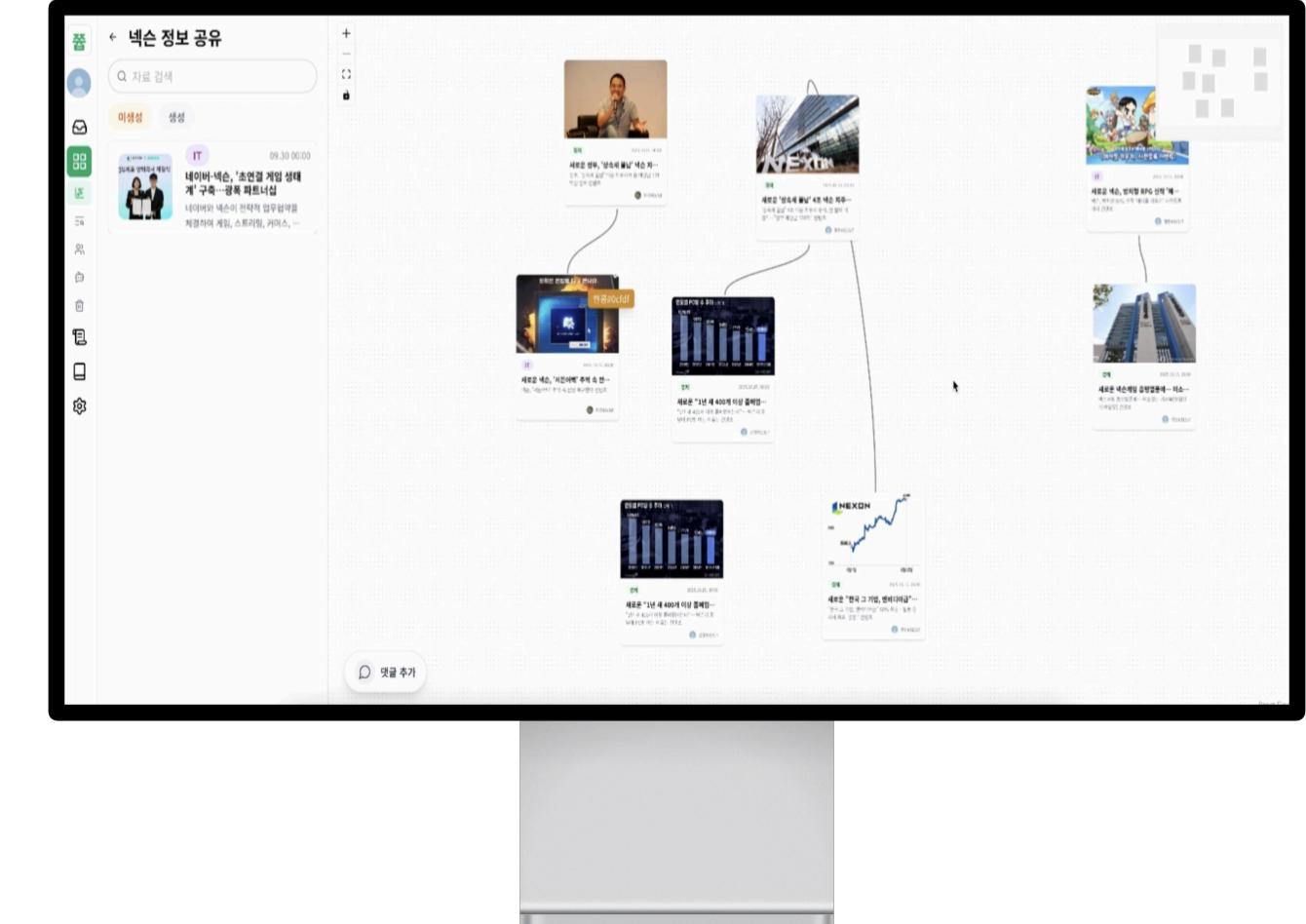
정보 수집부터 정리까지, 끊김 없는 워크플로우.

"데이터 관리의 비효율을 자동화로 해결하다."

[[GitHub](#)] [[Notion](#)] [[데모 영상](#)]

프로젝트 기간 2025.09 ~ 2025.11

문제점	<ul style="list-style-type: none">맥락의 단절: 자료 조사(Web)와 정리(Notion/Slack) 툴이 분리되어, 팀원 간 정보 공유 시 맥락이 끊기는 현상 발생.휘발되는 인사이트: 웹 서핑 중 발견한 유용한 정보들이 체계적으로 아카이빙되지 않고 개인 메모장에 방치됨.
해결책	<ul style="list-style-type: none">원클릭 아카이빙: 자체 개발한 Chrome Extension으로 웹 문서를 클릭 한 번에 요약 및 수집.실시간 협업 캔버스: 수집된 카드를 캔버스에 드래그하여 팀원들과 실시간으로 분류하고 아이디어 확장.
인원	8명(FrontEnd 3, BackEnd 5)
기술스택	
Core	Next.js, TypeScript
Real-time	Liveblocks (WebSocket)
Canvas	XYFlow (React Flow)





Feature 1. 실시간 협업 캔버스

기술 Liveblocks + React Flow

설명 다수의 사용자가 동시에 노드를 편집해도 충돌 없는 CRDT 기반 동시 편집 환경 구축.

Feature 2. 렌더링 성능 최적화

기술 Layered Architecture (상태 계층 분리)

설명 마우스 커서, 물체 이동 등 휘발성 데이터와 영구 저장 데이터를 분리하여, 다중 접속 시에도 부드러운 인터랙션 유지

Feature 3. AI 지식 요약

기술 Chrome Extension + LLM

설명 긴 웹 문서를 클릭 한 번으로 요약하여 캔버스에 카드로 생성. 정보 습득 시간을 단축시키는 워크플로우 자동화.

The image contains two side-by-side screenshots of a web application interface, likely the ZoopZoop platform, demonstrating its AI summarization features.

Left Screenshot: The interface has a header with "ZOOPZOOP" and "AI 추천". On the left, there's a sidebar with user profile, "AI 추천", "스페이스", and "뉴스" (News). The main area is titled "AI 추천 뉴스" and shows cards for news articles. One card for "ORACLE + AI" has a timestamp of "2025.06.20. 12:16". Another card for "일론 머스크, 오라클과 AI 인프라 통...". A third card for "[인터뷰] 마이크 히초와 오라클 부사..." has a timestamp of "2024.09.12. 01:00". At the bottom, large blue text reads "42MΛ 42MΛ".

Right Screenshot: This is a screenshot of a web browser window showing a news article from "n.news.naver.com". The article is about "李대통령 출연 '냉부해' 댓글 1만2000개 사라져...구글 "정부 요청 없었다!"". The browser has several tabs open, including "ZoopZoop", "여행의 관계 확장에도 추가 학습 모드", "마스터 편스러... 온전한 전...", and "최대한 출연 '냉부해' 댓글 1만2...". The ZoopZoop extension icon is visible in the top right corner of the browser window.



끊김 없는 상태 동기화 전략

Phase 1. 데이터 동기화

"사용자는 로딩 스피너를 보고 싶어 하지 않는다."

Trade-off

Client-Side Fetching (CSR):

- 페이지 진입 후 데이터를 가져옴. HTML 로드 후 로딩 스피너가 돌면서 '깜빡임' 발생.

SSR Prefetching (Dehydration):

- 선택: 서버에서 데이터를 미리 채워(Prefetch) HTML과 함께 전송. 진입 즉시 완성된 화면 노출

결론

- HydrationBoundary를 통해 서버의 상태(State)를 클라이언트로 그대로 이식하여, 초기 렌더링 속도 향상 및 네트워크 요청 절감.

```
export default async function Page() {
  const queryClient = new QueryClient()

  await queryClient.prefetchQuery({
    queryKey: QUERY_KEYS.USER.me(),
    queryFn: fetchUserProfile
  })

  return (
    <HydrationBoundary state={dehydrate(queryClient)}>
      <Component />
    </HydrationBoundary>
  )
}
```

Phase 2. 캐시 키 관리

"오타 하나가 데이터 불일치를 만든다."

Trade-off

하드코딩:

- 개발자가 문자열을 직접 입력. 오타 발생 시 캐시 무효화 실패로 데이터 갱신 누락 위험.

중앙 관리:

- 모든 쿼리 키를 객체로 중앙 관리. 자동완성 지원으로 오타 가능성 0% 및 유지보수성 확보.

결론

- 키 관리 시스템을 구축하여, 협업 시 발생할 수 있는 휴먼 에러를 구조적으로 차단.

```
export const QUERY_KEYS = {
  USER: {
    me: () => ['user', 'me'],
    byName: (name: string) => ['user', name]
  },
  SPACE: {
    all: () => ['spaces'],
    pagination: (page: number, state?: SpaceStatus) => ['spaces', page, state]
  },
  // 8개 도메인으로 체계화
}
```



실시간 협업 성능 최적화 전략(Liveblocks)

Phase 1. 상태 계층 분리

"내 마우스 커서 위치까지 DB에 저장해야 할까?"

Trade-off

Shared Storage (단일 저장소):

- 단순 선택(Select)이나 드래그 좌표 같은 임시 상태까지 DB(Storage)에 기록. → 불필요한 리렌더링 유발 & 15fps로 성능 저하.

3-Layer Architecture (계층 분리):

- 서버에서 데이터를 미리 채워(Prefetch) HTML과 함께 전송. 진입 즉시 완성된 화면 노출
- Local: 내 선택 박스 / Presence: 드래그 중 좌표 / Storage: 드래그 끝난 최종 위치.

결론

- "휘발성 데이터"와 "영구 데이터"를 철저히 격리하여, 타인의 작업이 내 화면을 멈추게 하는(Blocking) 현상 원천 차단.

Local State(개인) Presence(실시간, 임시) Storage(영구)

- | | | |
|------------|----------------|------------|
| • 각 사용자 독립 | • 빠른 업데이트 | • 영구 저장 |
| • 공유 불필요 | • 연결 끊기면 사라짐 | • 최종 노드 위치 |
| • 선택 상태 | • 드래그 중 위치, 커서 | |



실시간 협업 성능 최적화 전략(Liveblocks)

Phase 2. 업데이트 빈도 제어

“1초에 60번 저장하는 것은 공격이다.”

Trade-off

Real-time Sync (실시간 동기화):

- mousemove 이벤트(60hz)마다 네트워크 요청. → 10초간 600회 트래픽 발생으로 서버 부하 폭증.

Event Batching (이벤트 배치):

- 선택: 드래그 '중'에는 가벼운 Presence(좌표)만 공유하고, 드래그가 '끝난 순간(onPointerUp)'에만 Storage(DB) 업데이트.

결과

- 10초간 Storage 업데이트 600회 ⇒ 10회
- devtool(프레임 렌더링) 20fps ⇒ 60fps
- 동시 작업 가능인원 증가