ChatGPT

기술적 챌린지 도입 제안서

Real-Time Weather API 통합 및 최적화

주요 화면에 **실시간 날씨 API 데이터**를 표시하여 API 최적화 기법을 연습할 수 있습니다. Next.js의 서버사이드 렌더링(SSR)을 활용해 초기 페이지 로드 시 날씨 데이터를 받아오고, **React Query**의 폴링 기능(`refetchInterval`)으로 클라이언트에서 주기적으로 갱신하면 구현 가능합니다. 날씨 API 호출 간격은 적절히 조절하여 (예: 5분마다) 불필요한 트래픽을 방지합니다. 또한 **stale-time** 등을 설정해 **캐싱**하면 이미 받아온 날씨 정보를 일정 시간 재사용하고, 변경 시에만 재-fetch하게 되어 성능을 높일 수 있습니다.

이렇게 하면 사용자가 페이지를 열 때 최신 날씨 정보가 즉시 보이고, **API 응답이 변경되지 않으면 추가 다운로드 없이 캐시를 재사용**하여 네트워크 부하를 줄일 수 있습니다 [1]. 팀 프로젝트가 **모의 주식투자**라면, 이 기법을 주식 시세 API에 응용하여 **실시간 주가 표시**를 구현할 때 도움이 됩니다 (예: 초기 SSR로 종목 시세 표시 후 일정 주기로 갱신).

- **구현 방안:** Next.js **SSR**로 초기 데이터 패칭 → React Query `useQuery` + `refetchInterval`로 클라이언트 실시간 갱신 + `staleTime`으로 캐싱.
- **도입 이유:** 외부 API 데이터를 **최소 지연**으로 보여주고, 불필요한 중복 호출을 줄여 **고성능** UI를 제공하기 위함.
- **기대 효과:** 초기 로딩 속도 향상, 실시간 데이터 업데이트로 **동적인 UX** 제공, 네트워크 트래픽 감소 (캐시 활용).

Optimistic UI를 활용한 댓글 기능 (즉각적인 피드백)

낙관적 UI(Optimistic UI) 패턴은 서버 응답을 기다리지 않고 **사용자 액션에 즉각 반응**하는 UI를 만드는 기법입니다. 댓글을 작성할 때, 서버에 저장되기 전에 UI에 바로 댓글이 추가된 것처럼 표시하고, 실패 시 롤백합니다. **React Query**의 `useMutation` 훅에서 `onMutate`나 `optimisticResponse`를 사용하여 **로컬 캐시를 즉시 업데이트**하면 구현할 수 있습니다 [2]. 예를 들어 새 댓글 객체를 미리 생성해 화면에 추가하고, `onError`에서 에러 시 제거하는 식입니다. 또는 React 18+에서 제공하는 `useOptimistic` 훅을 활용할 수도 있습니다.

이 접근은 사용자에게 **빠른 피드백**을 주어 앱이 매우 빠르게 느껴지게 합니다. 실제로 TanStack React Query는 **뮤테이션 완료 전에 UI를 미리 업데이트**하는 두 가지 방식을 제공합니다: `onMutate`로 캐시를 직접 변경하거나, 뮤테이션 변수로 임시 UI를 그리는 방법이 있습니다 [2]. 구현 시 두 방법 중 하나를 택하여, 댓글 목록에 새로운 댓글을 **투명도 낮춘 상태**로 추가해두고 서버 응답 후 정상표시/제거하면 됩니다 [3] [4].

- **구현 방안:** React Query `useMutation` 활용 – `onMutate`에서 **댓글 캐시 업데이트**, `onError`에서 롤백. 혹은 `isPending` 상태로 임시 댓글 렌더링.
- **도입 이유: 네트워크 지연 없이 즉각적인 인터랙션**을 제공하여 댓글 작성 경험을 향상시키기 위함입니다.
- **기대 효과:** 사용자 만족도 상승 (버튼 클릭 후 바로 댓글이 보임), 앱이 **더 빠르고 반응적으로** 느껴짐. 향후 주식 앱에서도 주문 처리나 거래내역 업데이트 시 **빠른 피드백**을 주는 데 활용할 수 있습니다.

부드러운 Infinite Scroll 구현 (무한 스크롤)

데이터를 여러 페이지로 나누고 **스크롤 위치에 따라 자동으로 추가 로드** 하는 **Infinite Scrolling**은 사용자 경험과 성능 모두에 이점이 있습니다. **React Query**의 `useInfiniteQuery` 훅을 사용하면 페이징된 API를 쉽게 다룰 수 있습니다. 예를 들어 게시글 목록 API에 한 번에 10개씩 불러오는 식으로 구성하고, **IntersectionObserver**로 화면 하단에 배치한 sentinel 요소가 보일 때마다 `fetchNextPage()`를 호출합니다 [5] [6]. 이렇게 하면 스크롤이 밑으로 진행될 때 자동으로 다음 데이터를 가져와 끊김 없이 리스트가 이어집니다.

이 기법은 **초기 로딩을 가볍게** 하고, 사용자에게는 마치 데이터가 끝없이 이어지는 듯한 자연스러운 경험을 줍니다. Medium의 사례에서도 "처음에 필요한 일부만 불러오고, 사용자가 스크롤할 때 다음 부분을 가져온다"고 설명합니다 [7]. 한 번에 모든 데이터를 불러오는 경우의 **로딩 지연**이나 메모리 낭비를 피하고, **필요한 순간에만 추가 요청**함으로써 앱이 더 경쾌해집니다.

- **구현 방안:** `useInfiniteQuery`로 페이지네이션 API 연동 → **IntersectionObserver**로 하단 요소 관찰하여 `fetchNextPage` 호출 [6]. 로딩 상태 시 중복 호출 방지를 위해 `isFetching` 체크.
- **도입 이유: 한꺼번에 너무 많은 데이터를 불러오는 문제**를 해결하고, 사용자에게 **끊김 없는 스크롤 경험**을 제공하기 위함입니다 [8].
- **기대 효과:** 초기 페이지 로드 시간 단축, 스크롤 시 **자연스러운 콘텐츠 로드**로 UX 향상. 팀프로젝트에서도 방대한 거래 내역이나 뉴스 피드 등을 **효율적으로 표시**하는 데 활용 가능합니다.

React Query 프리패치와 깜빡임 제거

사용자에게 **순간적인 페이지 전환** 경험을 주기 위해 **데이터 프리패칭(prefetching)**을 도입합니다. 이는 사용자가 특정 콘텐츠를 보기 직전에 데이터를 미리 받아놓는 전략입니다. 예를 들어 게시글 제목에 **마우스 오버** 시 해당 게시글의 상세 데이터를 `queryClient.prefetchQuery()`로 미리 가져와 캐시에 저장해둘 수 있습니다 [9] [10]. 이렇게 해두면 사용자가 실제로 게시글을 클릭해 열 때, **이미 받아둔 데이터**를 즉시 보여줄 수 있어 로딩 스피너 없이 콘텐츠가 보입니다 [11] [12]. "로딩 스피너도, 지연도 없이, 오직 속도만 있다(No loading spinner. No delay. Just speed.)"는 표현처럼, 사전에 캐싱된 데이터를 활용해 **UI 깜빡임(flicker)**을 없앨 수 있습니다 [11].

또한 **React Query**의 옵션을 조정해 데이터가 갱신될 때 **이전 데이터 유지(keepPreviousData)**나 **staleTime**을 늘리는 방식으로, 짧은 페이지 전환 시 새로 로딩 화면이 나타나지 않도록 할 수 있습니다. 예를 들어 페이지네이션이나 필터 변경 시 `keepPreviousData: true`를 사용하면 데이터가 바뀌는 동안 이전 데이터를 잠시 표시하여 빈 화면 깜빡임을 방지합니다.

- **구현 방안:** React Query의 `queryClient.prefetchQuery`를 사용해 **페이지 이동 전에 데이터 미리 로드** [9]. 또한 `staleTime` / `keepPreviousData` 설정으로 캐시 데이터 활용.
- **도입 이유: 사용자 경험 극대화** – 빠른 내비게이션과 전환 시 콘텐츠 깜빡거림 제거를 통해 앱을 **네이티브 수준**으로 부드럽게 만들기 위함입니다.
- **기대 효과:** 거의 **즉각적인 페이지 전환** (사용자는 로딩을 느끼지 못함), 화면 전환 시 **깜빡임 없는** 매끄러운 UI. 향후 주식 앱에서 종목 상세 정보 미리불러오기나 차트 데이터 prefetch 등에 적용해 **UX 향상**을 기대할 수 있습니다.

효율적인 검색 기능 구현

게시판이나 주식 종목 검색 등 **검색 기능**은 최적화가 필요합니다. 우선 프론트엔드에서는 **디바운싱(debouncing)** 기법을 적용하여 사용자가 검색어를 입력할 때 **일정 시간 멈춘 후에만** 실제 검색 요청을 보내도록 합니다. 예를 들어 300ms 디바운

스로 구현하면, 불필요한 다수의 API 호출을 줄이고 서버 부하를 완화할 수 있습니다. 또한 React Query를 사용할 경우 `enabled` 옵션과 검색 키워드를 결합하여, 키워드가 있을 때만 쿼리를 활성화하거나, `refetchOnWindowFocus` 등을 꺼서 입력 중 불필요한 재요청을 막을 수 있습니다.

백엔드 측면에서는 **데이터베이스 인덱스**를 활용하여 검색 쿼리를 최적화해야 합니다 (이 부분은 아래 Indexing 항목과 연계). 예를 들어 제목이나 종목명 컬럼에 `INDEX`를 생성하면 해당 컬럼으로 `WHERE` 검색 시 **전체 테이블 스캔 없이** 빠르게 결과를 찾을 수 있습니다. **Prisma**를 사용한다면 스키마에 @@index 지시자를 통해 쉽게 인덱스를 추가할 수 있습니다 [13] [14]. 인덱스가 없으면 데이터가 커질수록 검색 속도가 **선형적으로 악화**되지만, 인덱스를 걸면 수백만 건이 있어도 검색 시간이 거의 **상수 수준으로 유지**됩니다 [15] [16].

- **구현 방안: 프론트에서 디바운스**로 API 호출 빈도 조절, **백엔드에서 인덱스/적절한 쿼리** 사용. 필요 시 PostgreSQL의 Full-text search 등도 고려.
- **도입 이유:** 검색은 자칫하면 **성능 병목**이 되기 쉬우므로, 이를 최적화하여 **빠른 응답성과 낮은 서버 부하**를 달성하기 위함입니다.
- **기대 효과:** 입력 도중 앱이 느려지지 않고 부드러운 사용자 경험 제공, 많은 데이터에서도 **신속한 검색 결과** 반환. 주식 종목 검색 기능에 적용하면 사용자에게 **즉각적인 종목 필터링**을 제공하면서도 서버는 효율적으로 동작합니다.

Zod를 활용한 입력값 검증 (Runtime Validation)

Zod 라이브러리는 *"TypeScript-first schema validation"* 도구로, 런타임에 데이터 구조와 규칙을 검증할 수 있습니다 [17]. 프론트엔드에서 폼 입력값을 처리할 때 Zod 스키마를 정의해두고 `parse()`나 `safeParse()`로 검사하면 사용자의 잘못된 입력을 서버에 보내기 전에 잡아낼 수 있습니다. 예를 들어 댓글 작성 폼에 대해 `z.object({ content: z.string().min(1) })` 스키마를 만들고 submit 시 검증하면, 빈 내용 등 오류를 즉시 표시할 수 있습니다. **React Hook Form**과 Zod를 함께 쓰면 form 상태관리와 검증을 손쉽게 결합할 수 있고, Nest.js 백엔드에서도 동일한 Zod 스키마로 요청을 검증하면 **프론트-백 타입 일관성**을 유지할 수 있습니다.

TypeScript의 타입체크는 컴파일 타임에만 유효하므로, **런타임의 실제 데이터 보호를 위해 Zod와 같은 도구가 필수적** 입니다. Zod를 통해 **컴파일타임-런타임 간 격차**를 메워 더욱 신뢰성 있는 코드를 작성할 수 있습니다 [18]. 잘못된 입력이 초기에 걸러지면 예상치 못한 서버 오류나 DB 오류를 예방하고, 사용자에게는 무엇이 잘못됐는지 명확히 피드백을 줄 수 있습니다 (예: "이메일 형식이 올바르지 않습니다" 등의 메시지).

- **구현 방안:** 각 입력 폼/데이터에 대한 **Zod 스키마 정의**, 제출 이벤트에서 `schema.parse()`로 검증 후 에러 시 UI 표시. (React Hook Form 사용할 경우 Zod용 Resolver 활용 가능).
- **도입 이유: 런타임에 입력 데이터의 신뢰성**을 보장하여 버그를 줄이고, 사용자에게 **즉각적이고 정확한 검증 피드백**을 주기 위함입니다.
- **기대 효과:** 클라이언트 단계에서 잘못된 입력을 걸러 서버 부하 감소, 입력 실수에 대한 **친절한 안내**로 UX 향상. 팀 프로젝트의 거래 입력 폼 등에 적용하면 **비정상 데이터가 시스템에 들어오는 것을 방지**하여 안정성이 높아집니다.

데이터베이스 인덱스 및 정렬 최적화

백엔드 성능에도 도전해보고 싶다면 **DB 인덱싱**과 **정렬 최적화**를 실습할 수 있습니다. PostgreSQL + Prisma 환경에서, 자주 검색하거나 조인/정렬에 사용되는 필드에 인덱스를 추가하면 조회 속도가 비약적으로 빨라집니다 [19] [16]. 예를 들어 게시판의 `createdAt` 필드로 최신순 정렬을 자주 한다면 해당 컬럼에 인덱스를 걸어두면 정렬 연산이 효율화됩니다. Prisma 스키마에서는 모델 정의에 `@@index([필드])` 혹은 각 필드에 `@index` 등을 명시하여 쉽게 인덱스를 설정할 수 있습니다 [13].

인덱스가 없는 경우, DB는 **순차 스캔(full table scan)**으로 모든 레코드를 훑어야 하므로 데이터가 많아질수록 **쿼리가 기하급수적으로 느려집니다** [15]. 반면 인덱스가 있으면 수십만 건 이상에서도 **ms 단위** 응답을 유지할 수 있습니다 [20]. 단, 인덱스는 쓰기 성능에 약간 영향을 주고 너무 많으면 오히려 과부하가 될 수 있으므로, **자주 필터링/정렬하는 필드 위주로 전략적으로 적용**해야 합니다 [21].

- **구현 방안:** Prisma 모델에 **인덱스 지시자 추가** (예: `@@index([title, createdAt])` 복합인덱스 등). 대용량 데이터를 가정해 `EXPLAIN ANALYZE`로 쿼리 계획을 확인하면서 병목 지점을 찾아 최적화.
- **도입 이유: 대용량 데이터에서의 성능 문제**를 선제적으로 해결하여, 응답시간을 낮추고 애플리케이션이 **확장성**을 갖도록 준비하기 위함입니다.
- **기대 효과:** 데이터가 늘어나도 **일관된 쿼리 성능** 확보, 페이지네이션이나 검색 시 **응답 지연 최소화**. 주식 프로젝트에서 거래내역이 방대해져도 인덱싱을 통해 **실시간에 가까운 데이터 조회**가 가능해집니다.

SSE를 활용한 실시간 댓글 업데이트

Server-Sent Events (SSE)를 이용하면 서버에서 **실시간으로 데이터를 푸시**하여 클라이언트 UI를 업데이트할 수 있습니다. Nest.js에서는 `@Sse()` 데코레이터를 사용해 SSE 엔드포인트를 쉽게 생성할 수 있으며, 해당 엔드포인트에서 `Observable` 스트림을 반환하면 자동으로 **텍스트 이벤트 스트림**을 형성합니다 [22]. 클라이언트에서는 **EventSource API**로 SSE 스트림에 연결하여 메시지를 수신할 수 있습니다 [23]. 예를 들어 `/comments/sse`로 새로운 댓글이 발생할 때마다 이벤트를 보내도록 하고, React에서는 EventSource로 연결해 오는 데이터마다 상태를 업데이트해주면, 사용자들이 별도 새로고침 없이도 **댓글 리스트가 실시간 갱신**됩니다.

SSE는 **HTTP 기반의 지속 연결**을 사용하기 때문에 방화벽 등 통과에 유리하고, 자동 재연결 등을 기본 지원하며, **한 방향 통신**만 필요한 상황에 적합합니다 (댓글 알림, 주가 변경 등). WebSocket에 비해 구현 난이도가 낮고, 서버 부하도 이벤트 발생시에만 데이터 전송하므로 효율적입니다 [24] [25]. 특히 **모의 주식투자** 프로젝트에서는 SSE를 활용해 **실시간 주가**나 **거래 체결 알림**을 모든 사용자에게 푸시할 수 있어 현실감 있는 업데이트를 구현할 수 있습니다.

- **구현 방안:** Nest.js `@Sse()`로 **이벤트 스트림 API** 구축 [22] → React에서 `new EventSource()`로 연결하여 메시지 수신시 상태 갱신. 필요에 따라 이벤트명 (`event:`)과 데이터 포맷(JSON) 정의.
- **도입 이유: 실시간 성**이 요구되는 기능(댓글, 알림, 주가 변동 등)에 폴링보다 효율적이고 신속한 업데이트를 제공하기 위함입니다.
- **기대 효과:** 사용자들이 **새로고침 없이도 최신 상태**를 바로 확인 (예: 다른 사용자가 단 댓글이 자동 등장), 앱의 **실시간 인터랙티브** 수준 향상. 주식 앱의 시세 변동, 알림 시스템 등에 적용하여 **시장 상황을 실시간 반영**할 수 있습니다.

조건부 GET과 HTTP 304 캐싱

API 요청 시 **조건부 GET**을 활용하면, 변경되지 않은 리소스에 대해 **304 Not Modified** 응답을 받아 **네트워크 트래픽을 절감**할 수 있습니다. 구현 방법은 다음과 같습니다: 서버에서 응답 헤더로 ETag(엔터티 태그) 혹은 Last-Modified 값을 보내주고, 클라이언트는 이후 같은 리소스 요청시 If-None-Match 또는 If-Modified-Since 헤더를 포함합니다. 서버는 데이터 변경 여부를 검사하여 **변경되지 않았다면 304 응답**을 보내고 내용 본문은 전송하지 않습니다 [1]. 브라우저는 이전에 캐시된 내용을 그대로 활용하게 되므로, **왕복 시간은 소비되지만 전체 리소스를 재전송하는 것보다 효율적**입니다 [1].

Nest.js에서 구현한다면 `Response.header()`를 통해 ETag를 설정하거나 Nest 미들웨어로 `etag` 패키지를 활용할 수 있습니다. React Query로 데이터를 가져올 때 fetch API는 브라우저 캐시를 기본 활용하므로, 서버가 적절히 304를 반

환하기만 하면 자동으로 캐시를 쓰게 됩니다. 다만 API가 `Cache-Control: no-cache`로 되어 있다면, 클라이언트 코드에서 수동으로 ETag를 관리해줄 수도 있습니다.

- **구현 방안: NestJS 응답에 ETag/Last-Modified 헤더 추가**, 변경 검출 로직 구현. 클라이언트는 기본 fetch 캐시 또는 Axios 인터셉터 등을 통해 If-None-Match 헤더 전송.
- **도입 이유:** 빈번한 GET 요청에 대해 **불필요한 데이터 전송을 방지**하고, 네트워크 지연을 줄이기 위함입니다. 특히 목록이나 정적 데이터가 자주 조회되는 경우 효과가 큽니다.
- **기대 효과:** 업데이트가 없을 시 **응답 바이트 수와 처리 시간 감소**, 서버 부하 완화. 팀 프로젝트에서는 예를 들어 **기업 정보나 사용자 프로필처럼 거의 안 바뀌는 데이터**에 적용하여, 여러 사용자가 조회해도 서버는 한 번만 보내고 이후엔 304로 응답하게 할 수 있습니다.

가시성 기반 스마트 폴링

리소스 폴링(polling)을 해야 한다면, **페이지나 컴포넌트의 가시성**을 고려한 스마트 폴링으로 최적화를 할 수 있습니다. **Page Visibility API**를 사용하면 사용자가 현재 탭을 보고 있는지 확인할 수 있는데, 이를 활용해 **페이지가 백그라운드로 넘어가면 폴링 중지** 또는 주기 연장, 다시 활성화되면 재개하는 로직을 구현합니다 [26] [27]. React에서는 `document.addEventListener('visibilitychange', ...)`로 페이지 visibility 변화를 감지하거나, 커스텀 훅(`usePageVisibility`)을 만들어 `document.hidden` 상태를 추적할 수 있습니다 [27] [28].

또한 **IntersectionObserver**를 사용하면 특정 컴포넌트 (예: 대시보드 위젯)가 화면에 보일 때만 폴링을 수행하도록 제어 가능합니다. 예를 들어 날씨 위젯이 화면 밖으로 스크롤되면 해당 컴포넌트 업데이트를 중단했다가, 다시 보일 때 업데이트를 재개할 수 있습니다. 이렇게 하면 사용자에게 보이지 않을 때 불필요한 네트워크 요청과 연산을 피할 수 있습니다.

스마트 폴링을 적용하면 **리소스 절약과 성능 향상** 측면에서 크게 도움이 됩니다. Atul Banwar의 글에서도 "사용자가 페이지를 볼 때 폴링을 시작하고, 떠나거나 탭 전환 시 중단함으로써 최적화한다"고 설명합니다 [26]. 또한 폴링 시 **에러 발생 시 일정 횟수 실패하면 중지**하는 등의 추가 로직도 넣어 서버에 부담을 주지 않도록 할 수 있습니다 [29] [30].

- **구현 방안: Page Visibility API**로 전역 폴링 제어 (예: `visibilitychange` 이벤트로 `isPageVisible` 상태 관리) [27]. + **IntersectionObserver**로 개별 컴포넌트 노출 여부 체크하여 폴링 `enabled` 토글.
- **도입 이유:** 사용자에게 보이지 않는 정보 업데이트를 **일시 중지하여 자원 낭비를 막고**, 필요한 순간에만 폴링해서 효율을 극대화하기 위함입니다.
- **기대 효과:** 불필요한 API 호출 감소로 **배터리와 데이터 절약**, 화면 전환 시에도 앱이 **원활하게 동작**. 주식 프로젝트에서는 사용자가 차트 탭을 보고 있을 때만 가격 갱신 폴링을 하고, 다른 탭으로 가면 중지하는 식으로 **효율적인 실시간 데이터 관리**가 가능해집니다.

이상으로 제안된 여러 프론트엔드 기술 챌린지들은 **Next.js + TypeScript + TailwindCSS + React Query + Zustand** 스택을 활용하여 구현할 수 있는 사항들입니다. 각각 **구현 방법**, **도입 이유**, **기대 효과**를 정리하였으며, 5주 팀프로젝트(모의 주식투자 앱 가정)에 응용할 수 있는 시사점도 함께 언급했습니다. 이러한 도전 과제들을 이번 사이드 프로젝트에 미리 적용해 봄으로써, 추후 본격적인 팀프로젝트 진행 시 한층 향상된 **성능 최적화**와 **우수한 사용자 경험**을 제공할 수 있을 것입니다.

---

[1] HTTP conditional requests - HTTP | MDN
https://developer.mozilla.org/en-US/docs/Web/HTTP/Guides/Conditional_requests

2 3 4 Optimistic Updates | TanStack Query React Docs
https://tanstack.com/query/v5/docs/framework/react/guides/optimistic-updates

5 6 7 8 Seamless infinite scrolling: TanStack Query and Intersection Observer | by Sanjiv Jangid | Aug, 2025 | Medium
https://medium.com/@sanjivjangid/seamless-infinite-scrolling-tanstack-query-and-intersection-observer-c7ec8a544c83

9 10 11 12 How to Prefetch Data with TanStack Query (React Query)
https://jsdev.space/howto/react-query-prefetch/

13 14 15 16 19 20 21 Boosting Query Performance in Prisma ORM: The Impact of Indexing on Large Datasets | by Manoj Shrestha | Mar, 2025 | Medium
https://medium.com/@manojbicte/boosting-query-performance-in-prisma-orm-the-impact-of-indexing-on-large-datasets-a55b1972ca72

17 colinhacks/zod: TypeScript-first schema validation with ... - GitHub
https://github.com/colinhacks/zod

18 Validating TypeScript Types in Runtime using Zod - Wisp CMS
https://www.wisp.blog/blog/validating-typescript-types-in-runtime-using-zod

22 23 25 Server-Sent Events | NestJS - A progressive Node.js framework
https://docs.nestjs.com/techniques/server-sent-events

24 NestJS Server-Sent Events (SSE) and Its Use Cases - Medium
https://medium.com/@kumar.gowtham/nestjs-server-sent-events-sse-and-its-use-cases-9f7316e78fa0

26 27 28 29 30 Efficient Polling in React. Handling Visibility Changes for Optimal... | by Atul Banwar | Medium
https://medium.com/@atulbanwar/efficient-polling-in-react-5f8c51c8fb1a