React-Query에 관하여

React-Query

React Query는 React Application에서 **서버 상태를 불러오고, 캐싱하며, 지속적으로 동기화하고 업데이트**하는 작업을 도와주는 라이브러리입니다.

React Query에 관한 흔한 오해

1. 비동기 통신 라이브러리다? ❌

```
const { data } = useQuery(['myData'], () =>
  axios.get('https://api.example.com/1').then((res) => res.da
);
```

위 코드에서 비동기 통신 라이브러리는 `axios` 이고, useQuery는 비동기 통신을 호출한 결과값을 **스토어에 캐싱**해주는 역할을 한다.

```
const { data, isLoading, error } = useQuery(['fetchText'], (
    return new Promise(resolve => {
      resolve("Hello, World!");
    });
  });
```

꼭 API 호출이 아니더라도 Promise 를 반환하는 비동기 함수에도 사용할 수 있다.

2. 클라이언트 상태관리를하는 상태관리 라이브러리다? ❌

Don't use the queryCache as a local state manager

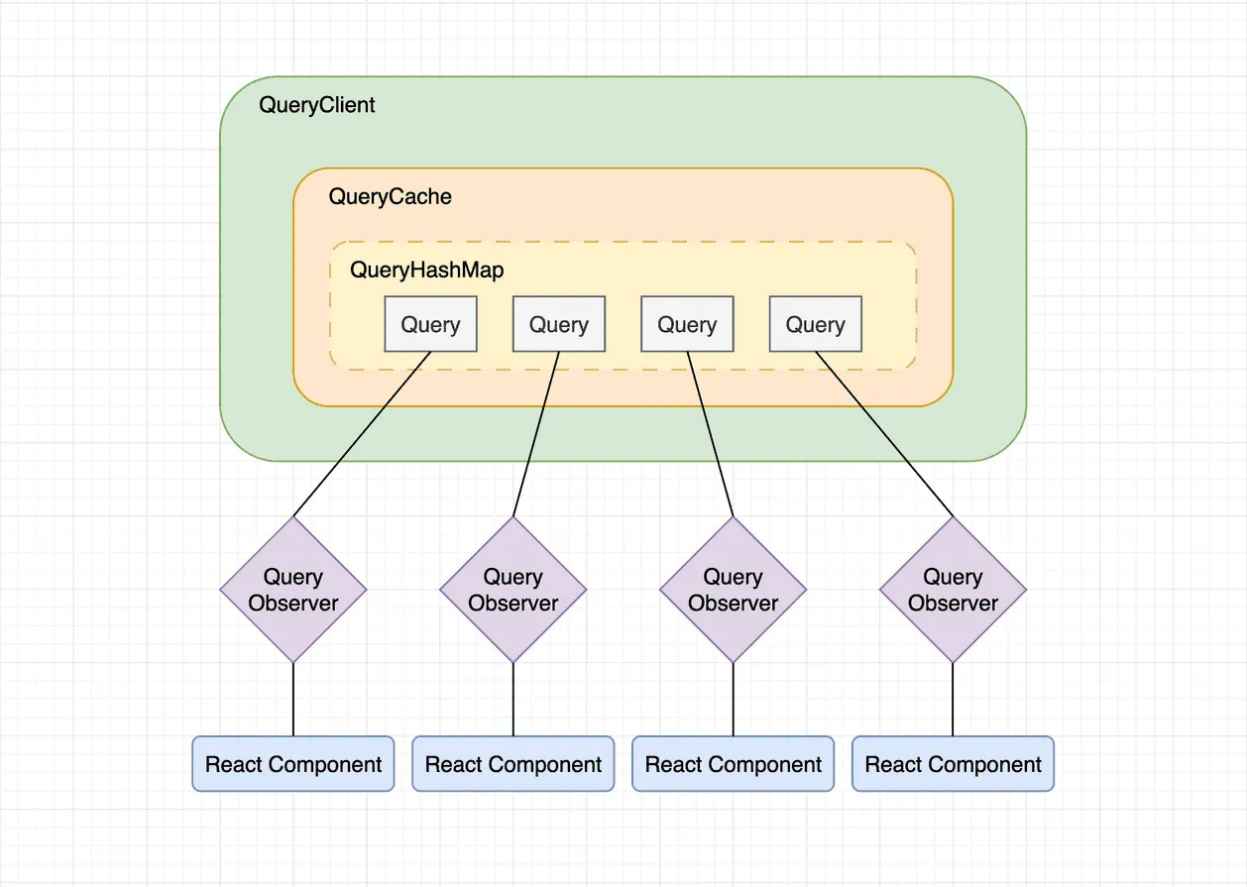
If you tamper with the queryCache (`queryClient.setQueryData`), it should only be for optimistic updates or for writing data that you receive from the backend after a mutation. Remember that every background refetch might override that data, so use something else for local state.

queryCache(*queryClient.setQueryData*)를 수정하는 경우는 낙관적 업데이트 또는 변환 후 백엔드로부터 수신한 쓰기 전용의 데이터여야 합니다. 모든 background fetch는 해당 데이터를 재정의할 수 있으므로 지역 상태에는 사용하지 마세요.

즉 웬만하면 쿼리 캐시를 직접 수정하면서 상태관리 라이브러리처럼 사용하지 말아라. 왜냐하면 백그라운드에서 fetch 하면 보존되지 않고 사라지는 데이터이다.

React-Query의 내부 동작에 대해 알아보자

추상적으로 표현한 전체 구조의 모습



QueryClient

흔히 아는 queryClient 생성 방식

```
import { QueryClient, QueryClientProvider } from '@tanstack/r
eact-query'

// ⬇️ client 생성
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 5 * 60 * 1000,
      retryDelay: (attemptIndex) => 600 * 2 ** attemptIndex,
    },
  },
})

function App() {
  return (
    // ⬇️ client 분배
    <QueryClientProvider client={queryClient}>
      <RestOfYourApp />
    </QueryClientProvider>
  )
}
```

모든 것은 `QueryClient`에서 시작합니다.

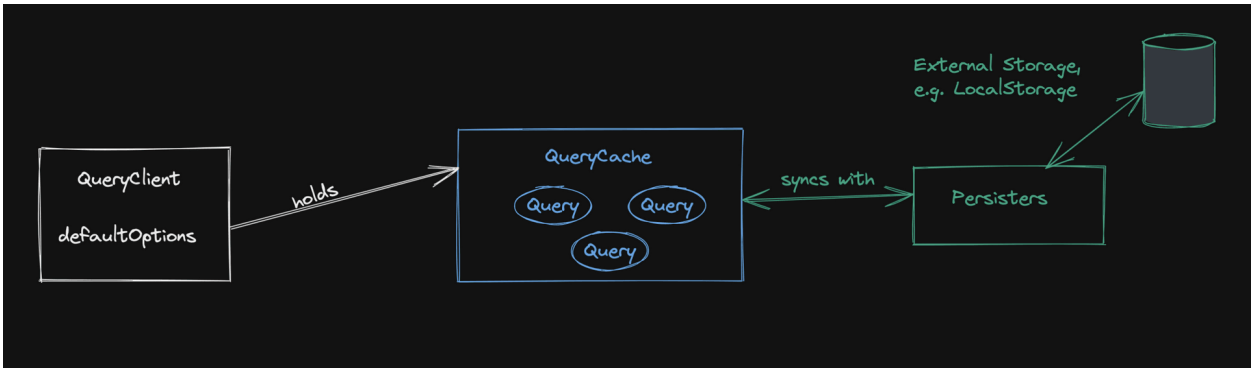
queryClient 는 React Context 를 사용하여 만든 QueryClientProvider를 통해 전체 애플리케이션에 분배됩니다. 자식 컴포넌트에선 `useQueryClient`를 통해 이 client에 대해 접근이 가능합니다.

`QueryClient` 자체는 실제로 많은 일을 하지 않습니다. `QueryCache` 및 `MutationCache`의 컨테이너 역할을 합니다. query 및 mutation에 대해 설정할 수 있는 몇 가지 **기본 옵션값**을 보유합니다. 대부분의 경우 캐시와 직접 상호작용하지 않고 `QueryClient` 에서 제공하는 methods를 통해 캐시에 접근합니다.

QueryClient 내부 코드

```
export class QueryClient {
  private queryCache: QueryCache

  constructor(config: QueryClientConfig = {}) {
    this.queryCache = config.queryCache || new QueryCache()
} // queryCache 들고 있음 주목!

  setQueryData() {}
  getQueryData() {}
  invalidateQueries() {}
  getQueryCache() {}
  // methods...
}
```

QueryCache



`QueryCache`는 여러 개의 `Query` 인스턴스들을 자신의 내부 프로퍼티 필드인 `queries`, `queriesMap`에 등록하여 관리하는 역할을 합니다.

React Query의 캐시는 메모리에만 데이터를 저장하고 다른 곳에는 저장하지 않습니다. 브라우저 페이지를 새로고침하면 캐시가 사라집니다. localstorage와 같은 외부 저장소에 캐시를 쓰려면 persisters를 살펴보세요.

QueryCache 내부 코드

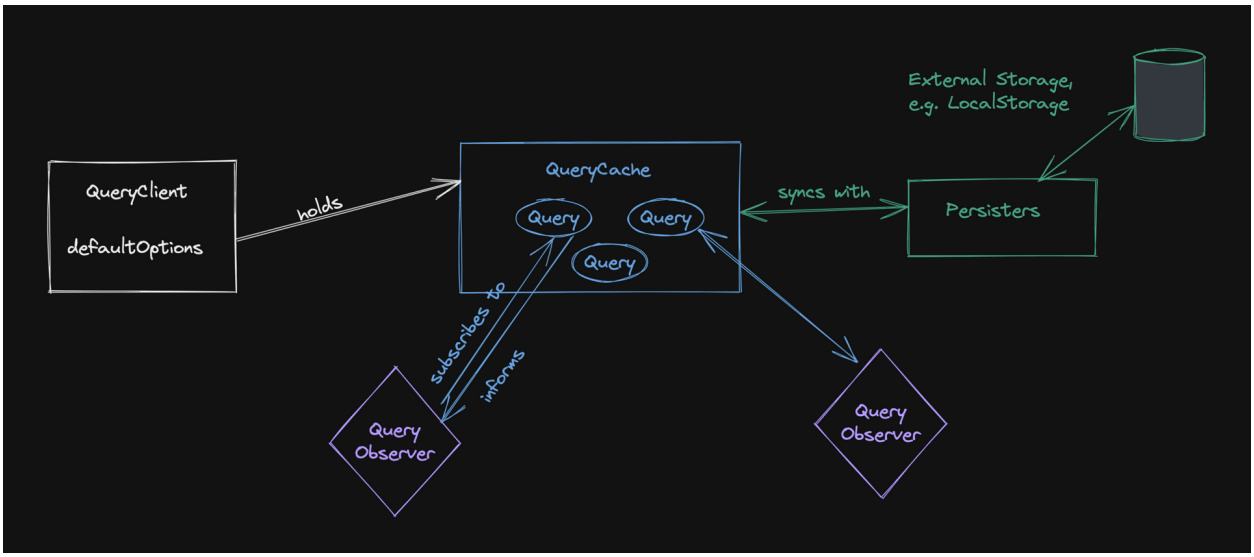
```
export class QueryCache extends Subscribable<> {
  private queries: Query<any, any, any, any>[]
  private queriesMap: QueryHashMap

  constructor(config?: QueryCacheConfig) {
    this.queries = []    // 여기에 등록
    this.queriesMap = {}   // 여기에 등록
  }

  build() {}
  add() {}
  remove() {}
  // ...
}
```

build 내부 코드 분기문 참고

queryKey 로 queriesMap을 참조하여 유효한 쿼리가 있으면 재사용하고 없으면 새로 생성해서 캐시에 등록 후 반환합니다.

이 덕분에 코드에서는 중복 호출을 해도 유효시간(staleTime)동안은 중복된 네트워크 요청을 하지 않게 됩니다.

Query



`Query`는 `QueryCache`가 가진 `queries`, `queriesMap`에 저장되는 인스턴스입니다. 쿼리에서 대부분의 로직이 실행됩니다.

- 자신을 가지고 있는 `QueryCache`
- 자신의 상태가 변경되었을 때 호출할 `this.observers`
- • `useQuery`로 전달한 `queryFn`의 호출 결과 값 데이터 `this.state.data`

Query 내부 코드

케이스 별로 dispatch 호출하고, dispatch 호출시 action타입에 따라 쿼리 상태를 업데이트 하면서 onQueryUpdate를 호출해서 현재 Query를 구독중인 `QueryObserver`를 호출합니다. (내부 코드)

중요한 건 **쿼리가 누가 쿼리 데이터에 관심있는지를 알고 해당 관찰자에게 모든 변경 사항을 알릴 수 있다는 것입니다.**

QueryObserver

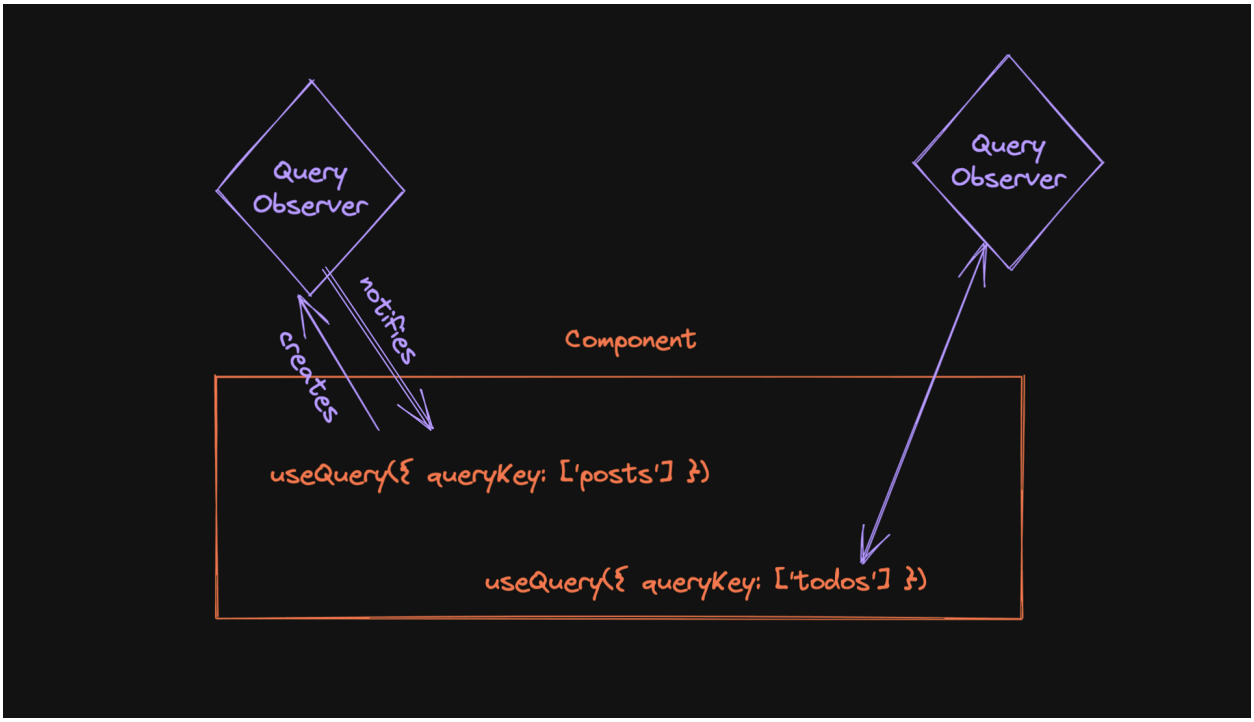
`QueryObserver`는 `Query`의 변화를 관찰하고 리스너(컴포넌트)에게 알리는 역할을 합니다.

`Observer`는 useQuery를 호출할 때 생성되며 항상 정확히 하나의 쿼리를 구독합니다. 그렇기 때문에 `useQuery`에 `queryKey`를 전달해야 합니다.

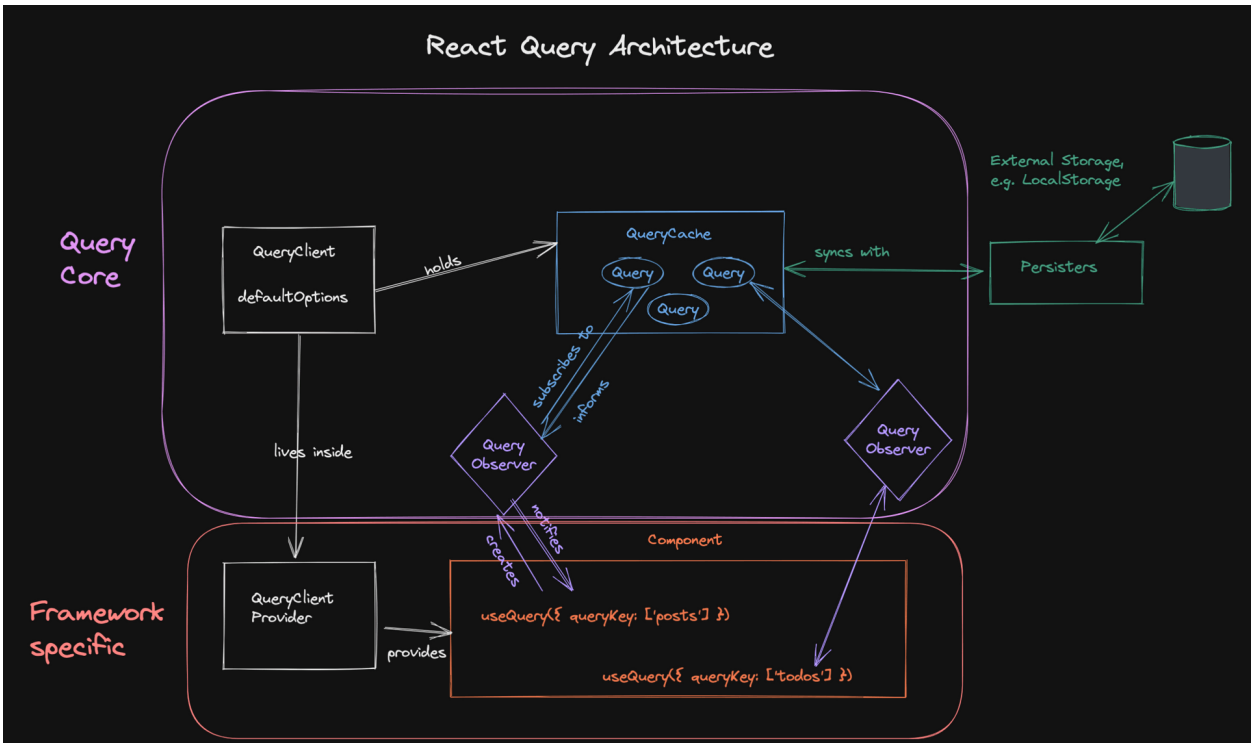
`Observer`는 대부분의 최적화가 이루어지는 곳입니다. `Observer`는 컴포넌트가 사용 중인 쿼리의 속성을 알고 있으므로 관련 없는 변경 사항을 알릴 필요가 없습니다. 예를 들어 데이터 필드만 사용하는 경우 백그라운드 refetch에서 *isFetching*이 변경되는 경우 컴포넌트를 다시 렌더링할 필요가 없습니다.

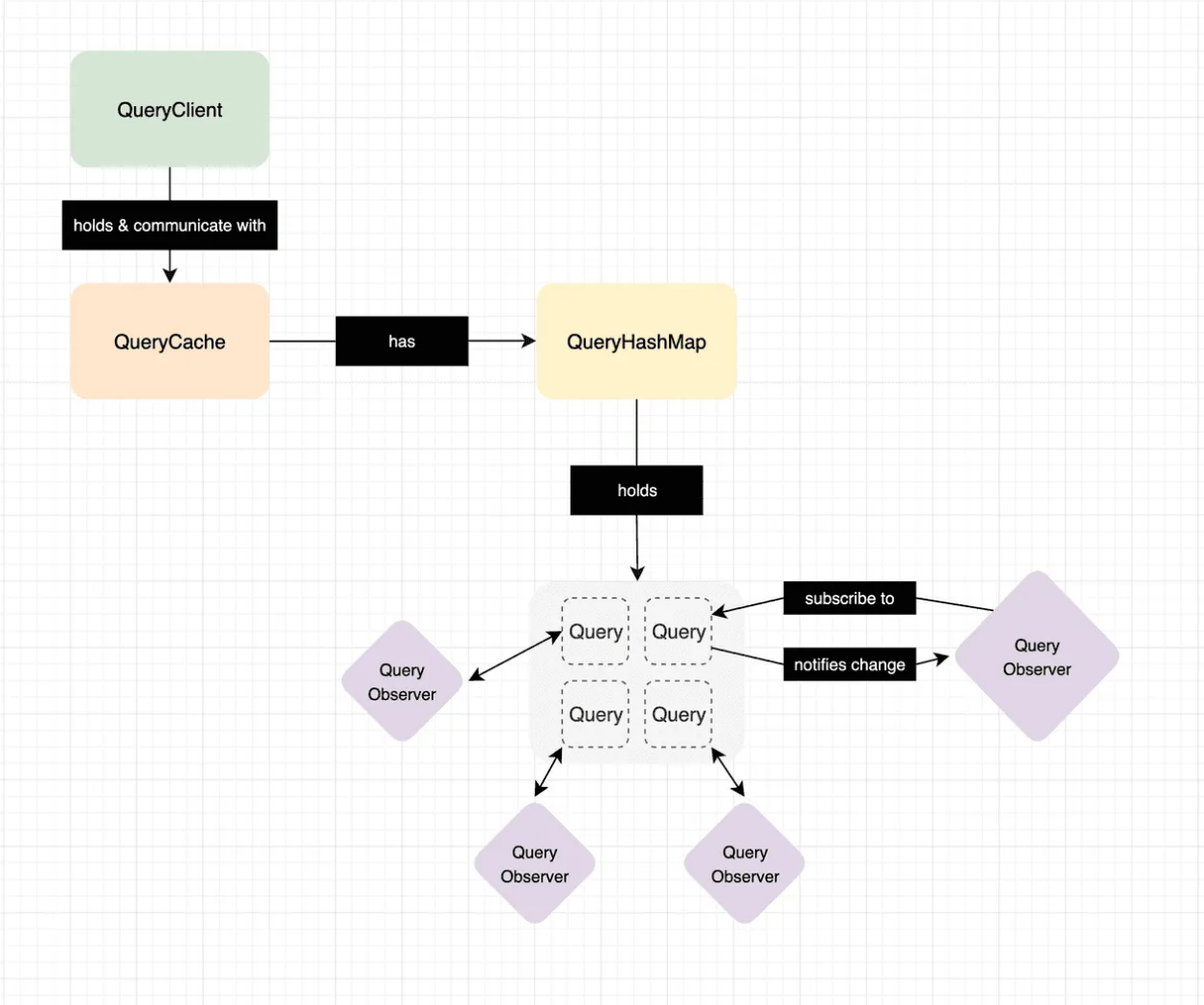
QueryObserver의 onQueryUpdate 내부 코드

onQueryUpdate ⇒ updateResult ⇒ notify listener



전체 합친 그림


React Query Architecture
Query Core
QueryClient
defaultOptions
holds
QueryCache
Query
Query
Query
syncs with
External Storage, e.g. LocalStorage
Persisters
subscribes to
informs
lives inside
Query Observer
Query Observer
notifies
creates
Component
Framework specific
QueryClient Provider
provides
useQuery({ queryKey: ['posts'] })
useQuery({ queryKey: ['todos'] })



queryClient 관련 장애 경험 공유

CSR에서 SSR 환경으로 이관하다가 SSR에서 queryClient가 공유된 아찔한… 장애 경험

Next.js SSR 환경에서는 QueryClient 를 다른 사용자와 공유하지 않기 위해선 전역에 선언하는것이 아니라 컴포넌트 내부에서 선언하라는 안내가 문서에 있음

SSR | TanStack Query Docs

> Create a new QueryClient instance inside of your app, and on an instance ref (or in React state). This ensures that data is not shared between different users and requests, while still only creating the QueryClient once per component lifecycle.
>
> 앱 내부와 인스턴스 참조(또는 React 상태)에서 새 QueryClient 인스턴스를 만듭니다. 이렇게 하면 서로 다른 사용자와 요청 간에 데이터를 공유하지 않고 컴포넌트 생명 주기당 한 번만 QueryClient를 생성할 수 있습니다.

SSR(Server Side Rendering) 환경(getServerSideProps 함수 내부)에서는 아래처럼 queryClient를 전역 선언으로 사용할시 queryClient 공유가 서버 단위로 일어남.

```
// _app.tsx
export const queryClient = new QueryClient({
  defaultOptions: {
    queries: {
      staleTime: 5 * 60 * 1000,
      retryDelay: (attemptIndex) => 600 * 2 ** attemptIndex,
    },
  },
});

// 전역 설정
export default function App({ Component, pageProps }: AppProp
s) {
  return (
    <QueryClientProvider client={queryClient}>
      <Hydrate state={pageProps.dehydratedState}>
        <Component {...pageProps} />
      </Hydrate>
```

```
    </QueryClientProvider>
  );
}
```

```
// detail.tsx
import { queryClient } from '~/apis/reactQuery';

// Client
export default function DetailPage() {
  const { data } = useQuery({
    queryKey: ['getDetail'],
    queryFn: async () => {
      const { data } = await commonClient.get('/api.example.com');

      return data;
    },
  });

  return <div>DetailPage Render...</div>;
}

// Server
export async function getServerSideProps(context: GetServerSidePropsContext) {
  await queryClient.prefetchQuery({ // 전역 queryClient 에 값이 업데이트 되는 시점
    queryKey: ['getDetail'],
    queryFn: async () => {
      const { data } = await commonClient.get(
        '/api.example.com',
        { headers: { Authorization: `Bearer ${getSnowdonJwt(context)}` } },
      );
```

```
      return data;
    },
    staleTime: 0,
  });

  return { props: { dehydrateState: dehydrate(queryClient) }
};
}
```

`queryClient`가 전달되는 경로

1. `getServerSideProps` 에서 전역의 `queryClient.prefetchQuery` 를 호출하며 전역의 `queryClient` 가 업데이트가 일어남
2. dehydrate를 통해 업데이트된 `queryClient`가 전역의 `App` 컴포넌트에 설정되어있는 `Hydrate` 에 `dehydrateState` 값으로 전달됨.
3. `QueryClientProvider` 에 등록되어있는 CSR의 `queryClient` 를 `dehydrateState` 를 통해 받은 SSR `queryClient` 으로 업데이트 함.
4. `DetailPage` 컴포넌트에서 `useQuery` 를 호출할시 내부적으로 `useQueryClient` 를 통해 `QueryClientProvider` 에 등록된 `queryClient` 값을 참조하여 반환함.

`staleTime: 0` 으로 줬기 때문에 일반적인 상황에서는 문제가 되지 않았음.

하지만 알람톡 전송 후 동시접속자가 대량으로 많아졌을 경우에 `queryClient` 값이 공유되는 문제가 발생함.

`staleTime: 0` 임에도 불구하고 문제가 발생 가능한 케이스

1. 해당 문제 발생 하는 코드에서 staleTime이 0으로 설정되어 있었으므로 ms단위 까지 동일한 시간에 요청한 경우
2. `queryClient.prefetchQuery` 호출 후 `dehydrate` 가 호출되기 전에 다른 사용자가 queryClient에 저장되어있는 cache를 업데이트 하는 경우

번외. StaleTime & CacheTime

StaleTime은 쿼리가 '실행된 후' 얼마나 빨리 '오래된(stale)' 상태가 되는지를 정의. StaleTime이 지나면 캐시는 남아있되, 데이터는 '오래된' 것으로 간주되고, 해당 쿼리가 다시 호출될 때 데이터를 새로 가져오려고 시도.

CacheTime 쿼리 데이터가 캐시에 얼마나 오래 남아있을지를 정의. CacheTime이 지나면 캐시는 **제거됨**

주로 CacheTime을 StaleTime 보다 길게 하는것이 일반적.

만약에 StaleTime 10초, CacheTime 20초로 설정한다면?

1. 쿼리 호출된 후 5초 뒤 호출시 ⇒ 쿼리 재사용.
2. 쿼리 호출된 후 15초 뒤 호출시 ⇒ StaleTime over 되어서 쿼리 재 호출. 로딩 상태에서 Cache 데이터 보여줌.
3. 쿼리 호출된 후 25초 뒤 호출시 ⇒ CacheTime over 되어서 쿼리 재 호출. 로딩 상태에서 보여줄 Cache 데이터 없음.

+ 번외 Apllo Client & React Query 비교…

참고자료

https://tkdodo.eu/blog/inside-react-query

https://fe-developers.kakaoent.com/2023/230720-react-query/

https://www.timegambit.com/blog/digging/react-query/01