12. react-query 가이드

LGES Web Application Platform 구축

Exported on 2024-04-02 13:29:14

Table of Contents

1 데이터 가져오기(fetch) - useQuery 4

2 데이터 업데이트 - useMutation 7

3 캐싱 전략 이해하기 9

3.1 캐시 설정 커스터마이징하기 9

3.2 캐시 무효화 및 업데이트 9

React Query는 서버 상태 관리를 위한 라이브러리로, 데이터를 가져오기(fetching), 캐싱(caching), 동기화(synchronizing), 업데이트(update)하는 과정을 쉽게 만들어줍니다.

이 라이브러리는 리액트 애플리케이션에서 비동기 데이터를 다룰 때 발생할 수 있는 많은 복잡성을 추상화합니다.

# 데이터 가져오기(fetch) - useQuery

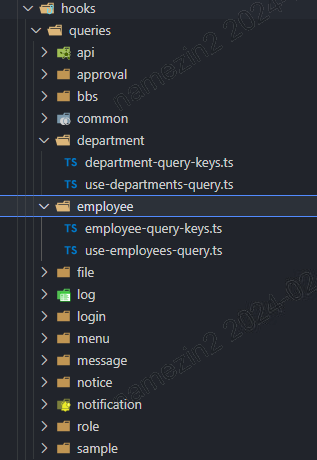
React Query를 사용하여 API에서 데이터를 가져오려면 useQuery 훅을 사용합니다. 이 훅은 데이터를 비동기적으로 가져오고, 로딩 상태와 에러를 처리하는 기능을 제공합니다.

표준 FW에서는 useReactQuery라는 훅으로 한 번 더 감싸서 필요한 설정을 미리 설정하여 사용할 수 있게 추상화하여 사용합니다.

|  |
| --- |
| ...    import {  // QueryFunction,  QueryKey,  UseQueryOptions,  UseQueryResult,  useQuery,  } from '@tanstack/react-query';    // React Query Custom Hook으로 전역으로 설정할 옵션을 정의 한다.  export const useReactQuery = <  TQueryFnData = unknown,  TError = ErrorResponse,  TData = TQueryFnData,  TQueryKey extends QueryKey = QueryKey  >(  options: UseQueryOptions<TQueryFnData, TError, TData, TQueryKey>  ): UseQueryResult<TData, TError> => {  const result = useQuery(options);    ...    **return** result;  }; |

따라서 이 useReactQuery를 사용하여 react-query 데이터 가져오기(fetch)를 api에 맞게 사용합니다.

실제로 api 호출하기 위한 react-query 훅들은 /hooks/queries 하위에 생성하여 사용합니다. [폴더 별 역할](https://wire.lgcns.com/confluence/pages/viewpage.action?pageId=827625763)을 확인하면 좀 더 명확하게 이해할 수 있습니다.



useQuery를 사용할 시에는 2가지가 기본적으로 필요합니다.

* query key: 로컬 캐싱을 위한 query에 대한 key 값
* query function: Fetch를 수행하는 비동기 함수
* 옵션들...

query key 같은 경우는 아래와 같이 함수 속성을 가진 객체 형태로 관리합니다.

|  |
| --- |
| export const EmployeeQueryKeys = {  key: ['employee'] as const,  employees: (params: { searchItem: string; deptCd: string; deptNm: string; empNm: string }) =>  [...EmployeeQueryKeys.key, { ...params }] as const,  }; |

query function은 실제 fetch를 수행하는 비동기 함수입니다. (HTTP Method가 'GET'인 경우)

표준 FW는 axios를 이용하여 API 레이어를 추상화하여 사용하고 있습니다. [폴더 별 역할](https://wire.lgcns.com/confluence/pages/viewpage.action?pageId=827625763) 에서 apis 영역이 그 레이어를 담당합니다.

따라서 실제로 사용될 useQuery 훅은 다음과 같이 나옵니다.

|  |
| --- |
| import { useReactQuery } from '@/hooks/use-react-query';  import { EmployeeQueryKeys } from './employee-query-keys';  import { getEmployeeBySearchCondition } from '@/apis/admin/Employee';    type UseEmployeesQueryProps = {  searchItem?: string;  deptCd?: string;  deptNm?: string;  empNm?: string;  };    **function** useEmployeesQuery({  searchItem = '',  deptCd = '',  deptNm = '',  empNm = '',  }: UseEmployeesQueryProps) {  **return** useReactQuery({  queryKey: EmployeeQueryKeys.employees({ searchItem, deptCd, deptNm, empNm }),  queryFn: () => getEmployeeBySearchCondition(searchItem, deptCd, deptNm, empNm),  enabled: Boolean(deptCd),  });  }    export { useEmployeesQuery }; |

이를 만들었다면 실제 컴포넌트에서는 다음과 같이 손쉽게 데이터와 fetch 상태에 대한 값들을 확인할 수 있습니다.

|  |
| --- |
| const { data: employeesData, isLoading, isError } = useEmployeesQuery({  searchItem: employeeSearchCondition?.searchItem,  deptCd: employeeSearchCondition?.deptCd,  deptNm: employeeSearchCondition?.deptNm,  empNm: employeeSearchCondition?.empNm,  });    ...    <Box mt="4px">  <EmployeeGrid  employeeList={employeesData ?? []}  onSelectedRows={(employees) => {  setSelectedEmployees(employees);  }}  />  </Box> |

# 데이터 업데이트 - useMutation

서버의 데이터를 업데이트하고, 이 변경사항을 클라이언트에 반영하기 위해 useMutation 훅을 사용할 수 있습니다.

이 훅은 데이터를 생성(create), 업데이트(update), 삭제(delete)하는 등의 비동기 작업을 처리하는 데 사용됩니다.

표준 FW에서는 useReactMutation라는 훅으로 한 번 더 감싸서 필요한 설정을 미리 설정하여 사용할 수 있게 추상화하여 사용합니다.

|  |
| --- |
| ...    import {  MutationFunction,  // QueryFunction,  UseMutationOptions,  UseMutationResult,  useMutation,  } from '@tanstack/react-query';    export const useReactMutation = <  TData = unknown,  TError = ErrorResponse,  TVariables = void,  TContext = unknown  >(  mutationFn: MutationFunction<TData, TVariables>,  options?: Omit<  UseMutationOptions<TData, TError, TVariables, TContext>,  'mutationKey' | 'mutationFn'  >,  isLoading = **true** //loading이 필요 없을 경우에 해당 옵션 false  ): UseMutationResult<TData, TError, TVariables, TContext | void> => {  ...    **return** useMutation({  mutationFn,  ...{  ...tOptions,  onMutate(variables) {  ...  },  onSuccess(data, variables, context) {  ...  },  onError(error, variables, context) {  ...  },  onSettled(data, error, variables, context) {  ...  },  },  });  }; |

mutationFn은 실제 fetch를 수행하는 비동기 함수입니다. (HTTP Method가 'POST', 'PUT', 'PATCH', 'DELETE'... 인 경우)

따라서 실제로 사용될 useMutation 훅은 다음과 같이 나옵니다.

|  |
| --- |
| import { devLogin } from '@/apis/Session';  import { useReactMutation } from '@/hooks/use-react-mutation';    type LoginParams = {  userId: string;  langCd: string;  };    **function** useLoginMutation() {  **return** useReactMutation(({ userId, langCd }: LoginParams) => {  **return** devLogin(userId, langCd);  });  }    export { useLoginMutation }; |

이를 만들었다면 실제 컴포넌트에서는 다음과 같이 손쉽게 데이터를 mutate 할 수 있습니다.

|  |
| --- |
| const { mutateAsync: loginAsync } = useLoginMutation();    const onSubmit = handleSubmit(async ({ userId, langCd }) => {  **try** {  openLoading(**true**);    const response = await loginAsync({ userId, langCd });    **if** (response.successOrNot !== 'Y' || !response?.data) {  **throw** **new** Error('로그인에 실패하였습니다.');  }    const session = response.data;    changeLanguage(session.langCd || 'ko');  setSession(session);    navigate('/system/code', { replace: **true** });  } **catch** (e) {  console.log({ e });  openMessageBar({  content: '로그인에 실패하였습니다.',  messageBarType: 'error',  });  } finally {  openLoading(**false**);  }  }); |

useMutation에는 mutate와 같은 동기 함수(callback 을 통한 결과 확인)와 mutateAsync 비동기 함수 2개를 제공합니다.

편한 방식으로 사용하되 프로젝트에서는 한가지 방식으로 사용하는 것을 권장합니다.

# 캐싱 전략 이해하기

React Query는 기본적으로 성공적으로 가져온 모든 쿼리 결과를 캐싱합니다.

각 쿼리는 고유한 쿼리 키에 의해 식별되며, 이 키는 캐싱 및 데이터 재사용의 기준이 됩니다.

캐싱 전략은 크게 두 가지 측면에서 조정될 수 있습니다:

* 캐시 시간 (Cache Time):  
  캐시된 데이터가 비활성 상태(inactive)일 때 얼마나 오래 저장될지를 정의합니다.  
  데이터가 이 시간 동안 비활성 상태로 남아 있으면, 캐시에서 제거됩니다.
* 스테일 타임 (Stale Time):  
  캐시된 데이터가 "신선(fresh)"로 간주되는 시간을 정의합니다.  
  이 시간이 지나면, 데이터는 "오래됨(stale)"으로 간주되고, 자동으로 새로고침될 수 있습니다.

## 캐시 설정 커스터마이징하기

React Query의 QueryClient 설정을 통해 전역적으로 또는 개별 쿼리 수준에서 캐시 정책을 조정할 수 있습니다. 예를 들어, 전역 캐시 시간을 설정하려면 다음과 같이 QueryClient를 구성할 수 있습니다:

|  |
| --- |
| const queryClient = **new** QueryClient({    defaultOptions: {      queries: {        cacheTime: 1000 \* 60 \* 5, // 5분      },    },  }); |

개별 쿼리에 대해 캐시 시간을 조정하려면, useQuery의 옵션으로 cacheTime을 설정할 수 있습니다:

|  |
| --- |
| const { data } = useQuery('projects', fetchProjects, {    cacheTime: 1000 \* 60 \* 5, // 5분  }); |

## 캐시 무효화 및 업데이트

데이터가 변경되었을 때 캐시를 적절히 업데이트하거나 무효화하는 것은 중요합니다.

React Query는 몇 가지 방법으로 이를 지원합니다:

* 무효화 (Invalidation): 특정 쿼리의 캐시를 무효화하려면, queryClient.invalidateQueries 메소드를 사용할 수 있습니다. 이 방법은 해당 쿼리를 다시 가져오도록 강제합니다.
* 옵티미스틱 업데이트 (Optimistic Updates): useMutation 훅과 함께 onMutate 옵션을 사용하여 데이터 변경이 성공하기를 기대하고 미리 UI를 업데이트할 수 있습니다.
* 캐시 데이터 직접 조작: 때로는 캐시된 데이터를 직접 조작해야 할 필요가 있을 수 있습니다. 이를 위해 queryClient.setQueryData 메소드를 사용하여 캐시된 데이터를 직접 업데이트할 수 있습니다.