React Query

A popular library for managing server state in React applications. It provides a caching mechanism that can help reduce the number of network requests made by your application.

To Inject react query in the App, we use (QueryClient, QueryClientProvider)

- QueryClient is a class that manages the cache and state of queries in your application. It provides a way to store and manage data fetched from APIs or other sources, and it can be used to manage the state of your application's data.
- QueryClientProvider is a component that provides the QueryClient instance to your application. It is used to wrap your application and make the QueryClient instance available to all components that use the useQuery hook.

```
import { QueryClientProvider, QueryClient } from "react-query";
import { ReactQueryDevtools } from "react-query/devtools";
const queryClient = new QueryClient();
function App() {
  return (
    <QueryClientProvider client={queryClient}>
      <Router>
        {/* routes */}
      </Router>
      <ReactQueryDevtools initialIsOpen={false} position="bottom-right" />
    </QueryClientProvider>
  );
}
export default App;
```

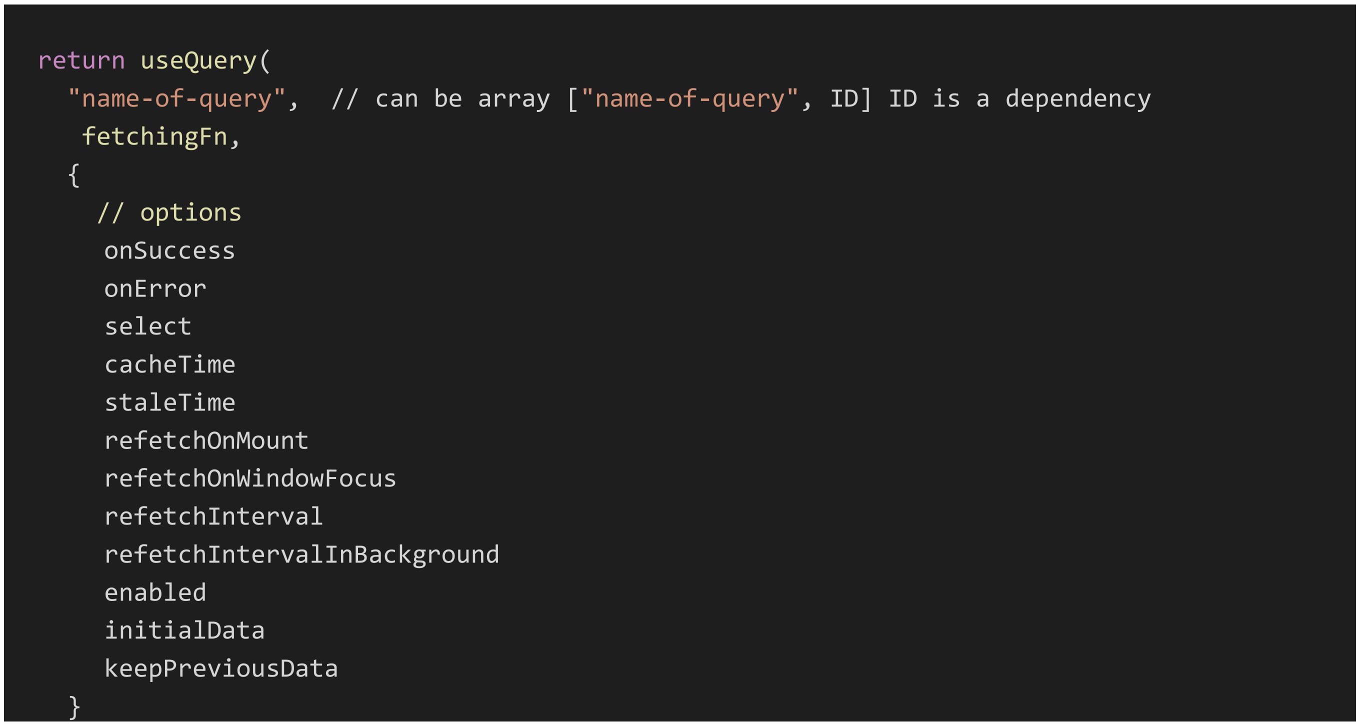
properties and methods of the QueryClient class:

- getQueryData: This method is used to retrieve the cached data for a given query key.
- setQueryData: This method is used to update the cached data for a given query key.
- removeQueries: This method is used to remove one or more queries from the cache.
- clear: This method is used to clear the entire cache.
- cancelQueries: This method is used to cancel one or more queries.
- invalidateQueries: This method is used to invalidate one or more queries (refetching).
- isFetching: This property is used to determine whether any queries are currently fetching data.
- isMutating: This property is used to determine whether any mutations are currently in progress.

Hooks: useQuery, useQueries, useQueryClient, useMutation, useIsFetching, useInfiniteQuery

1. useQuery

useQuery is a custom hook provided by the TanStack Query library that is used to fetch data in a React application. It requires two arguments: a queryKey and a queryFn 1. The queryKey is a unique identifier for the query, and the queryFn is a function that returns a promise that resolves to the data you want to fetch.



```
return useQuery(
  "name-of-query",  // can be array ["name-of-query", ID] ID is a dependency
   fetchingFn,
  {
    // options
     onSuccess
     onError
     select
     cacheTime
     staleTime
     refetchOnMount
     refetchOnWindowFocus
     refetchInterval
     refetchIntervalInBackground
     enabled
     initialData
     keepPreviousData
  }
```

select: is an option that allows you to select or transform parts of the query result.

staleTime: This property determines how long a query can be considered fresh before it becomes stale. When a query is fresh, data will always be read from the cache only - no network request will happen. If the query is stale, you will still get data from the cache, but a background refetch can happen under certain conditions .

// time to remove the whole query after unmounting
// time to wait(keep cached data) after query became inactive(unmounted)
// specifies the duration for which data should be kept in the cache before it can be garbage collected.

cacheTime: This property determines how long data should be kept in the cache before it can be garbage collected. This is only relevant for unused queries, because active queries can per definition not be garbage collected. The default value is 5 minutes.

// time to keep fetched data without any fetching (within navigation)
// navigation in this time, will not refetch the query
// time to keep fresh flag (no fetching)
// time to switch to stale flag

// specifies the duration until a query transitions from fresh to stale. As long as the query is fresh, data will always be read from the cache only - no network request will happen! If the query is stale, you will still get data from the cache, but a background refetch can happen under certain conditions.

```
const query = useQuery(["myQueryKey"], () => fetchMyData(), {
  staleTime: 1000 * 60 * 5, // 5 minutes
  cacheTime: 1000 * 60 * 60, // 1 hour
});
```

In this example, the query will be considered stale after 5 minutes and will trigger a background refetch if it's accessed again. Data will be kept in the cache for 1 hour before it can be garbage collected.

refetchOnMount: When set to true, the query will be refetched when the component mounts.

refetchOnWindowFocus: When set to true, the query will be refetched when the window regains focus.

refetchInterval is an optional property that sets the interval (in milliseconds) at which the query will be refetched.

refetchIntervalInBackground is another optional property that, if set to true, allows the query to continue refetching in the background while the tab/window is not in focus.

enabled is an optional boolean property that, if set to false, disables the query from automatically running. This can be useful for dependent queries refetch is a function that manually refetches the query.

```
function Example() {
  const { data, isLoading, isError, refetch } = useQuery(
    "example",
    fetchExampleData,
    {
      enabled: false, // Disables the query from automatically running
    }
  );
  if (isLoading) {
    return <div>Loading...</div>;
  }
  if (isError) {
    return <div>Error fetching data</div>;
  }
  return (
    <div>
      <h1>{data.title}</h1>
      <p>{data.description}</p>
      <button onClick={() => refetch()}>Refetch data</button>
    </div>
  );
}
```

initialData is a property in React Query that allows you to provide initial data to a query. This data is used to prepopulate the cache if it is empty, and is returned immediately to the component without waiting for the query to complete.

```
export const useSuperHeroData = (heroId: { heroId: string }) => {
  const queryClient = useQueryClient();
  return useQuery(["super-hero", heroId], fetchSuperHero, {
    initialData: () => {
      const hero = queryClient
        .getQueryData(super-heroes")
        ?.data?.find((hero: SuperHero) => hero.id === parseInt(heroId));
      return hero ? { data: hero } : undefined;
    },
  });
};
```

2. useQueryClient

useQueryClient is a hook that returns the current QueryClient instance. It is used to access the QueryClient instance from within a component. The QueryClient instance is used to manage the cache and state of queries in the application .

```
export const useSuperHeroData = (heroId: { heroId: string }) => {
  const queryClient = useQueryClient();
  return useQuery(["super-hero", heroId], fetchSuperHero, {
    initialData: () => {
      const hero = queryClient
        .getQueryData<{ data: SuperHero[] }>("super-heroes")
        ?.data?.find((hero: SuperHero) => hero.id === parseInt(heroId));
      return hero ? { data: hero } : undefined;
    },
  });
};
```

3. useQueries

useQueries is a custom hook that allows you to fetch a variable number of queries. It accepts an options object with a `queries` key whose value is an array with query option objects identical to the `useQuery` hook (excluding the `queryClient` option - because the `QueryClient` can be passed in on the top level). The `useQueries` hook can be used to fetch a variable number of queries.

```
const queryResults = useQueries([
  {
    queryKey: "super-heroes",
    queryFn: fetchSuperHeroes,
  },
  {
    queryKey: "friends",
    queryFn: fetchFriends,
  },
]);
```

4. useInfiniteQuery

useInfiniteQuery This hook is used to fetch paginated data from an API. It returns an object with the data, loading state, error state, and other metadata about the query.

```
// data here contains a list of pages
const {
  isLoading,
  isError,
  error,
  data,
  fetchNextPage,
  hasNextPage,
  isFetching,
  isFetchingNextPage,
} = useInfiniteQuery(["colors"], fetchColors, {
  getNextPageParam: (_lastPage, pages) =>
    pages.length < 4 ? pages.length + 1 : undefined,
});
```

5. useMutation

useMutation is used to perform a mutation on the server, such as creating or updating data. It returns a function that can be called to perform the mutation, as well as an object with the loading state, error state, and other metadata about the mutation.

```
const useAddSuperHeroData = () => {
  const queryClient = useQueryClient();
  return useMutation(addSuperHero, {
    onSuccess: (data) => {},
    onMutate: async (newHero) => {},               // optimistic update
    onError: (_error, _hero, context) => {},
    onSettled: () => {},
  });
};
---------------------------------------------------------------------------------------------
const { mutate: addHero } = useAddSuperHeroData();

const handleAddHeroClick = () => {
  const hero = { name, alterEgo };
  addHero(hero);
};
```

We can use `queryClient.invalidateQueries("name-of-query");` with (onSuccess / onSettled) to refetch / get the new results.

We can use `queryClient.setQueryData ("name-of-query", (oldQueryData) => {});` with (onSuccess / onError / onMutate) to optimisticlly update the UI or return old data in error case.

We can use `queryClient.cancelQueries ("name-of-query);` with onMutate to cancel any refetches could happens, avoiding overwriting the optimistic update.

Optimistic updates:

A way to update the UI before a mutation has completed by using the onMutate option to update your cache directly. When we optimistically update your state before performing a mutation, there is a chance that the mutation will fail. In most of these failure cases, you can just trigger a refetch for your optimistic queries to revert them to their true server state.

```
const useAddSuperHeroData = () => {
 const queryClient = useQueryClient();
 return useMutation(addSuperHero, {
 // get the data that passed to the addSuperHero async fn, to optimisticlly update the UI
   onMutate: async (newHero: SuperHero) => {
     // cancel any refetches could happens, to avoid overwriting the optimistic update
     await queryClient.cancelQueries("super-heroes");
     const previousHeroData = queryClient.getQueryData("super-heroes");
     queryClient.setQueryData("super-heroes", (oldQueryData) => {
       return {
         ...oldQueryData,
         data: [
           ...oldQueryData.data,
           { id: oldQueryData.data.length + 1, ...newHero },
         ],
       };
     });
     return {
       previousHeroData,
     };
   },
   onError: (_error, _hero, context) => {
     queryClient.setQueryData("super-heroes", context.previousHeroData);
   },
   onSettled: () => {
     queryClient.invalidateQueries("super-heroes");  // refetch the api
   },
 });
};
```

In React Query, **isLoading** and **isFetching** are both properties of the useQuery hook that let you know when a query is in progress. However, they have different meanings:

- **isLoading** is true when the query is being fetched for the first time and there is no data in the cache yet. It’s equivalent to status === "loading".
- **isFetching** is true whenever the query is fetching, including the first fetch and any subsequent fetches. It’s useful for showing a loading spinner or other UI element while the query is in progress.