First To Use React Query We Must Install it: `npm i @tanstack/react-query`

****

Then We Import QueryClient And QueryClientProvider:

```
import { QueryClient, QueryClientProvider } from "@tanstack/react-query";
```

****

Then We Create Object From QueryClient:

```
const queryClient = new QueryClient(); // this must be outside of App-Component.
Ex:
const queryClient = new QueryClient();
function App() {
  return (
    <>
      <Demo />
    </>
  )
}
```

****

Then We Define Our Provider:

```
function App() {
  return (
    <QueryClientProvider client={queryClient}> // in this way we can access all
// hooks that Query Client Provide To Us.
      <Demo />
    </QueryClientProvider>
  )
}
```

****

Then inside The Component That We Want To Use React-Query, We Import:

```
import { useQuery } from "@tanstack/react-query"
```

****

Then We Define Our Query That Contain The Query Function That Do The Operations, And Query Key That Make Us Work With Our Queries, Caches, ...etc.

```
const query = useQuery({
    queryFn: () => axios.get("https://jsonplaceholder.typicode.com/todos/"),
    queryKey: ["users"],
  });
**********************************************************
```

The Best Way To use useQuery is to Destruct it to Their Attributes:

```
const {
    data: todos,
    isLoading,
    isError,
    error,
  } = useQuery({
    queryFn: () =>
axios.get<ITodo[]>("https://jsonplaceholder.typicode.com/todos/"),
    queryKey: ["users"],
  });
**********************************************************
```

Like GraphQL, We Can Use Mutation To Make Changes To Our Data Like Adding, Updating, Deleting, ...etc.

Also, Here We Define The Mutation Function:

```
const mutation = useMutation({
    mutationFn: () =>
      axios.post("https://jsonplaceholder.typicode.com/todos", {
        id: todos?.data?.length,
        title: "Jafar Loka Todo",
        completed: true,
      }),

      mutationKey: ["todos"],
  });
**********************************************************
```

Also, useMutation Can Be Destructed to Available Attributes, Functions, ...etc.

Noe: mutate, and mutateAsync Are The Same But mutateAsync Can Be Awaited.

```
const { mutate, mutateAsync } = useMutation({
    mutationFn: () =>
      axios.post("https://jsonplaceholder.typicode.com/todos", {
        id: todos?.data?.length,
        title: "Jafar Loka Todo",
        completed: true,
      }),

      mutationKey: ["todos"],
  });
```

****

In This Way We Can Alias The Attributes in TypeScript:

```
const { mutateAsync: addTodoMutation } = useMutation({...});
```

****

The `addTodoMutation` Add The Value To Our DB But Not Update Our Data, So We Must Use The Query Key That Are Used To Cache Our Data.

****

When The Mutation Complete The Work We Can use onSuccess-Function To Re-fetch The Data:

```
const { mutateAsync: addTodoMutation } = useMutation({
    mutationFn: (title: string) =>
      axios.post("https://jsonplaceholder.typicode.com/todos", {
        id: todos?.data != undefined ? todos?.data?.length + 1 : 500,
        title: title,
        completed: true,
      }),
      mutationKey: ["todos_mutation"],
      onSuccess: () => {
      }
  });
**********************************************************
```

Then We Must Use New Hook From React-Query, To Re-Fetch The Data Again:

```
const queryClient = useQueryClient();
**********************************************************
```

Then Inside The on-Success:

```
onSuccess: () => {
        queryClient.invalidateQueries({ queryKey: ['todos'] });
      }
**********************************************************
```

```
const { mutateAsync: addTodoMutation } = useMutation({
    mutationFn: (title: string) =>
      axios.post("https://jsonplaceholder.typicode.com/todos", {
        id: todos?.data != undefined ? todos?.data?.length + 1 : 500,
        title: title,
        completed: true,
      }),

      mutationKey: ["todos_mutation"],

      onSuccess: () => {
        queryClient.invalidateQueries({ queryKey: ['todos'] });
      }
  });
**********************************************************
```

If We Pass Parameters To QueryFn Of useQuery, We Must Set This Parameters As Query Key As Object.

```
const [tId, setTId] = useState<number>(1);
const {
    data: todo,
    isLoading: dataLoading,
    isError: errSt,
  } = useQuery({
    queryFn: () =>
      axios.get<ITodo>("https://jsonplaceholder.typicode.com/todos/" + tId),
    queryKey: ["todos", { tId }],
  });

<div>
        { dataLoading && <div> Data For Parameter is Loading...</div>}

        <div>
          <input type="number" value={tId} onChange={(e) =>
setTId(parseInt(e.target.value))} />
        </div>

        <div>
          <p>The Todo Data is: </p>
          <p>The Id is: {todo?.data.id}</p>
          <p>The title is: {todo?.data.title}</p>
          <p>{todo?.data.completed == true ? "Todo Completed" : "Todo Not
Completed"}</p>
          <p>{errSt && <div>We Have Error</div>}</p>
        </div>
      </div>

**********************************************************
```

The Important Thing is to understand the Caching Of React Query, it will not execute the query again if nothing changed or we don't call:

```
queryClient.invalidateQueries({ queryKey: ['todos'] });
```

```
But In Background It Will Fetch The Data And Make The Changes.

That Mean If We Have A Cache The React Query Will Display The Cache, Then Do
Background Fetch, Then Display The Data.

**********************************************************
```

The time in milliseconds after data is considered stale. If set to Infinity, the data will never be considered stale. If set to a function, the function will be executed with the query to compute a staleTime.

```
**********************************************************

const {
    data: todos,
    isLoading: todosLoading,
    isError,
    isFetching,
  } = useQuery({
    queryFn: () => {
      console.log("Fetched Todos Again")
      return axios.get<ITodo[]>("https://jsonplaceholder.typicode.com/todos/")
    },
    queryKey: ["todos"],
    staleTime: 2000,
  });
{todosLoading || isFetching && <div>Loading Data Begin...</div>}
**********************************************************
```

(property) gcTime?: number | undefined

The time in milliseconds that unused/inactive cache data remains in memory. When a query's cache becomes unused or inactive, that cache data will be garbage collected after this duration. When different garbage collection times are specified, the longest one will be used. Setting it to Infinity will disable garbage collection.

****

```
const {
    data: todos,
    isLoading: todosLoading,
    isError,
    isFetching,
  } = useQuery({
    queryFn: () => {
      console.log("Fetched Todos Again")
      return axios.get<ITodo[]>("https://jsonplaceholder.typicode.com/todos/")
    },
    queryKey: ["todos"],
    staleTime: 2000,
    gcTime: 2000,
  });
```

****

Here We Can Define Multiple Options For Global React Query Configuration:

```
const queryClient = new QueryClient({
  defaultOptions: {
    queries: {

    },

  },
});
```

****