RTK Query Come With Redux-Toolkit.

****

To Install It We Write: **npm i @reduxjs/toolkit react-redux**

****

Then We Create The Store.

****

Then We Create The Slice File And Import:

```
import { createApi, fetchBaseQuery } from "@reduxjs/toolkit/query/react";
```

****

Then To Create Our Slice Using createApi:

```
export const productsApi = createApi({
  reducerPath: "productsApi",
  baseQuery: fetchBaseQuery({
    baseUrl: 'https://dummyjson.com/'
  }),
  endpoints: (builder) => ({
    getAllProducts: builder.query({
      query: () => "/products",


    })
  }),
})
```

****

Note: In Previous Example For Getting All The Data We Use builder.query, But For Add, Update, And Delete We Use: builder.mutation

****

The Redux Query Create A Hook For Every Endpoint We Create:

```
endpoints: (builder) => ({
    getAllProducts: builder.query({
      query: () => "/products",
    }),


  }),
```

****

Then We Must Import ApiProvider In The Main File:

```
import { ApiProvider } from '@reduxjs/toolkit/query/react'
```

****

RTK Query is a robust data fetching and caching tool within the Redux Toolkit package that simplifies loading data into your application. TypeScript, a strongly typed superset of JavaScript, enhances the development experience by providing type safety and reducing runtime errors. When combined, RTK Query and TypeScript offer a robust solution for managing the state of your application's data in a predictable and type-safe manner.

****

With RTK Query, you no longer need to write action creators, reducers, or custom middleware for data fetching. RTK Query's functionality includes auto-generated hooks that handle the loading data state, caching logic, and more.

****

Query Endpoints and TypeScript

Query endpoints are the specific functions within an API slice that define how to fetch data for a particular resource.

****

Handling Query Parameters and Arguments

When defining a query endpoint, you can specify query parameters or arguments to customize the request:

```
export const apiSlice = createApi({
 // ...other slice properties
 endpoints: (builder) => ({
   getPostsByCategory: builder.query<Post[], string>({
     query: (category) => `posts?category=${category}`,
   }),
 }),
});
```

****

To Avoid Errors when Using Hooks:

```
const { data: products, isLoading } = productsApi.useGetAllProductsQuery();
```

****

Difference between Mutations and Queries

Queries are used to fetch data, while mutations change or update data. In RTK Query, you define mutations using the builder.mutation method.

```
export const apiSlice = createApi({
 // ...other slice properties
 endpoints: (builder) => ({
   addPost: builder.mutation<Post, Partial<Post>>({
     query: (newPost) => ({
       url: 'posts',
       method: 'POST',
       body: newPost,
     }),
   }),
 }),
});
```

****

Customizing Caching Behavior with TypeScript

You can customize the caching behavior by specifying options in the API slice:

```
export const apiSlice = createApi({
  // ...other slice properties
  endpoints: (builder) => ({
    // ...endpoints
  }),
  keepUnusedDataFor: 60, // time in seconds
});
```

****

Cache Invalidation and Cache Entry Lifecycle

RTK Query provides mechanisms to invalidate cached data to ensure the UI displays the most current information. You can specify conditions under which the cache should be invalidated:

```
export const apiSlice = createApi({
addPost: builder.mutation<Post, Partial<Post>>({
     query: (newPost) => ({
       url: 'posts',
       method: 'POST',
       body: newPost,
     }),
     invalidatesTags: ['Post'],
   }),
 }),
});
```

****

```
export const productsApi = createApi({
    reducerPath: "productsApi",
    baseQuery: fetchBaseQuery({
        baseUrl: 'https://dummyjson.com/'
    }),
    tagTypes: ["Products"], // The Right Solution
    endpoints: (builder) => ({
        getAllProducts: builder.query<IProduct[], void>({
            query: () => "/products",

            providesTags: ["Products"],
        }),

    }),

    keepUnusedDataFor: 120,
});
```
****

```
baseQuery: fetchBaseQuery({

    baseUrl: '/api', prepareHeaders:  (headers) => {

      headers.set('Accept',  'plain/text, application/json');

      return headers;

    }

  }),
```

****

Data Fetching with Multiple Parameters

You can define endpoints that accept multiple parameters to fetch data based on various query arguments:

```
export const apiSlice = createApi({
 // ...other slice properties
 endpoints: (builder) => ({
   getPostsByAuthorAndCategory: builder.query<Post[], { authorId: number; category: string }>({
     query: ({ authorId, category }) => `posts?authorId=${authorId}&category=${category}`,
   }),
 }),
});
```

****

Optimistic Updates and Caching Logic (Test It)

RTK Query supports optimistic updates, allowing the UI to react to changes before the server confirms them:

```
export const apiSlice = createApi({
 // ...other slice properties
 endpoints: (builder) => ({
   updatePost: builder.mutation<Post, Partial<Post>>({
     query: (updatedPost) => ({
},
  url: `posts/${updatedPost.id}`,
  method: 'PUT',
  body: updatedPost,
}),
// Optimistically update the cache
onQueryStarted: async (updatedPost, { dispatch, queryFulfilled }) => {
  const patchResult = dispatch(
    apiSlice.util.updateQueryData('getPosts', undefined, (draft) => {
      const post = draft.find((p) => p.id === updatedPost.id);
      if (post) {
        Object.assign(post, updatedPost);
      }
    })
  );
  try {
    await queryFulfilled;
  } catch {
    patchResult.undo();
  }
},
```

   }),

 }),

});

****

Then To Configure The Store To Work With RTK Query:

```
import { configureStore } from "@reduxjs/toolkit";
import { productsApi } from "../slice/ApiSlice";
export const store = configureStore({
    reducer: {
        [productsApi.reducerPath]: productsApi.reducer,
    }
});
```

****

Accessing Cached Data from the Store

You can access cached data directly from the Redux store using the selectors provided by RTK Query:

import { useSelector } from 'react-redux';

import { apiSlice } from './apiSlice';

const selectPostsResult = apiSlice.endpoints.getPosts.select();

const { data: posts } = useSelector(selectPostsResult);

****

Troubleshooting Common Issues with RTK Query and TypeScript

Common issues include type mismatches and cache invalidation problems. Ensure that your TypeScript types align with your API responses and that cache tags are used correctly to invalidate and re-fetch data as needed.

****

```
export const store = configureStore({
    reducer: {
        [productsApi.reducerPath]: productsApi.reducer,
    },

    middleware: (getDefaultMiddleware) =>
        getDefaultMiddleware().concat(productsApi.middleware),
});

export const productsApi = createApi({
    reducerPath: "productsApi",
    baseQuery: fetchBaseQuery({
        baseUrl: 'https://dummyjson.com/'
    }),
    tagTypes: ["Products"],
    endpoints: (builder) => ({
        getAllProducts: builder.query<IFinalData, void>({
            query: () => "/products",
            providesTags: ["Products"],
        }),
    }),
    keepUnusedDataFor: 120,
});
```

****

```
<React.StrictMode>
      <ApiProvider api={productsApi}>
        <App />
      </ApiProvider>
</React.StrictMode>
```

****