RTK-Query

- Addis-Software Course

Overview

- Introduction
- Cache Behavior
- Queries
- Mutations
- Code Splitting
- Comparison With Saga
- Conditional Fetching
- Polling, Streaming Update
- Code Generation
- Error Handling

Introduction

RTK Query

- Is a powerful **data fetching** and **caching tool**.
- It is designed to simplify common cases for **loading data in a web application**, **eliminating the need to hand-write data fetching** & **caching logic yourself**.
- It is an optional addon included in the **Redux Toolkit package,** and its functionality is built on top of the other APIs in Redux Toolkit, This mean no need to add any package if you have redux-toolkit already installed.

Motivation

As we all know web applications normally need to do the following

- Fetch data from a server in order to display it.
- They also usually need to make updates to that data,
- Keep the cached data on the client in sync with the data on the server.
    - This is made more complicated by the need to implement other behaviors used in today's applications:

Intro...

Where RTK-Query Shines The Most

- Tracking loading state in order to show UI spinners
- Avoiding duplicate requests for the same data
- Optimistic updates to make the UI feel faster
- Managing cache lifetimes as the user interacts with the UI
- Streaming Updates
- Code Generation
- Code Splitting

Intro...

We have to realize that **"data fetching and caching"** is really a **different set of concerns** than **"state management".**

Intro...

While you can use a state management library like **Redux to cache data**, but the use cases are different enough that it's worth using tools that are **purpose-built for the data fetching use case.**

What's included

API

```
import { createApi } from '@reduxjs/toolkit/query'


/* React-specific entry point that automatically generates
   hooks corresponding to the defined endpoints */
import { createApi } from '@reduxjs/toolkit/query/react'
```

What's included

- **createApi()**: The core of RTK Query's functionality. It allows you to define a set of "endpoints" that describe how to retrieve data from backend APIs and other async sources, including the configuration of how to fetch and transform that data.
    - In most cases, you should use this once per app
- **fetchBaseQuery()**: A small wrapper around **fetch** that aims to simplify requests. Intended as the recommended baseQuery to be used in **createApi** for the majority of users.
- **<ApiProvider />**: Can be used as a Provider if you do not already have a Redux store.
- **setupListeners()**: A utility used to enable refetchOnMount and refetchOnReconnect behaviors.

Cache Behavior

When data is fetched from the server, RTK Query will store the data in the **Redux store as a 'cache'**. When an additional request is performed for the same data, **RTK Query will provide the existing cached data** rather than sending an additional request to the server.

Cache Behavior

Default Cache Behavior

With RTK Query, caching is based on:

- API endpoint definitions
- The query parameters used when components subscribe to data from an endpoint
  - When a subscription is started, the parameters used with the endpoint are serialized and stored internally as a **`queryCacheKey`** for the request
- Active subscription reference counts

Cache lifetime & subscription example

- 60 sec is the default life time for the cache
- Active Subscription Count
- Go to Example

Root Service, And Code Splitting

Query endpoints are defined by returning an object inside the **`endpoints`** section of **`createApi`**, and defining the fields using the `builder.query()` method.

```
import { createApi, fetchBaseQuery } from
'@reduxjs/toolkit/query/react';
import { API_ROUTE } from 'utils/constants';

// initialize an empty api service that we'll inject
endpoints into later as needed
export const rootService = createApi({
  baseQuery: fetchBaseQuery({ baseUrl: API_ROUTE }),
  endpoints: () => ({}),
  tagTypes: ['Carts', 'Cart'],
});
```

Adding It To Reducer Config

Adding The Root Reducer To The Store

```
import { rootService } from './service';

export function createReducer(injectedReducers:
InjectedReducersType = {}) {
if (Object.keys(injectedReducers).length === 0) {
return (state: any) => state;
}
return combineReducers({
...injectedReducers,
[rootService.reducerPath]: rootService.reducer,
});
}
```

Adding It To Store Config

Adding The Root Reducer To The Store

```
const store = configureStore({
reducer: createReducer(),
middleware: [
...getDefaultMiddleware({
serializableCheck: false,
}),
...middlewares,
].concat(rootService.middleware),
devTools: import.meta.env.NODE_ENV !==
'production',
enhancers,
});
```

Queries

Query endpoints are defined by returning an object inside the **`endpoints`** section of **`createApi`** `Or` **`Injecting It to The RootService`**, and defining the fields using the `builder.query()` method.

```
import { rootService } from 'store/service';

const cartApi = rootService.injectEndpoints({
 endpoints: build => ({
   getCart: build.query({
   query: () => '/carts',
  }),
}),
 overrideExisting: false,
});

export const { useGetCartQuery } = cartApi;
```

Queries Usage

```
import { rootService } from 'store/service';

const cartApi = rootService.injectEndpoints({
 endpoints: build => ({
  getCart: build.query<ICartModel, void>({
   query: () => '/carts',
  }),
 }),
 overrideExisting: false,
});

export const { useGetCartQuery } = cartApi;
```

Queries Usage

```
import { rootService } from 'store/service';

const cartApi = rootService.injectEndpoints({
 endpoints: build => ({
  getCart: build.query<ICartModel, void>({
   query: () => '/carts',
   transformResponse: (response: { data: ICartModel
},meta, arg) =>
   response.data,
   providesTags: ['Carts'],
  }),
}),
 overrideExisting: false,
});

export const { useGetCartQuery } = cartApi;
```

Queries Usage

```
const {
data, // Type Is ICartModel
error,
isFetching,
isError,
isSuccess,
isLoading,
refetch,
originalArgs,
fulfilledTimeStamp,
startedTimeStamp,
} = useGetCartQuery();
```

Avoiding unnecessary requests

By default, if you add a component that makes **the same query as an existing one**, no request will be performed.

In some cases, **you may want to skip this behavior and force a refetch -** in that case, you can call **refetch** that is returned by the hook.

Selecting Data

- **Using createSelector**

```
const rootCart = cartApi.endpoints.getCart.select();
export const selectCartApiData = createSelector(
 [rootCart],state => state.data);
```

- **Use transformResponse**
    - All consumers of the endpoint want a specific format, such as normalizing the response to enable faster lookups by ID
- **Or use useMemo**
    - when only some specific components need to transform the cached data

Mutation

```
const cartApi = rootService.injectEndpoints({
 endpoints: build => ({
  getCart: build.query<ICartModel, void>({
   query: () => routes.carts.get,
   providesTags: ['Carts'],
  }),
  addToCart: build.mutation<ISampleModel, string>({
   query: sample => ({
   url: routes.carts.get,
   method: 'POST',
   body: sample,
 }),
  invalidatesTags: ['Carts'],
  }),
 }),
 overrideExisting: false,
});
```

Conditional Fetching, Polling

Query hooks **automatically begin fetching data as soon as the component is mounted**. But, there are use cases where you may want to **delay fetching data until some condition becomes true**. RTK Query supports conditional fetching to enable that behavior.

If you want to prevent a query from automatically running, you can use the **`skip`** parameter in a hook.

```
1.Conditional
const {data, isFetching} = useGetCartQuery(undefined, {
 skip:true
});

2.Polling
const {data, isFetching} = useGetCartQuery(undefined, {
 pollingInterval:true
});
```

Streaming Updates

`Go to VSCode`

Refetching

- refetch()
  - `const { data, refetch } = useGetCartQuery(3);`
- Re-fetching on window focus with **refetchOnFocus**
- Re-fetching on network reconnection with **refetchOnReconnect**

```
Note: Just Add:-  setupListeners(store.dispatch);
```

END

Thanks for your Time