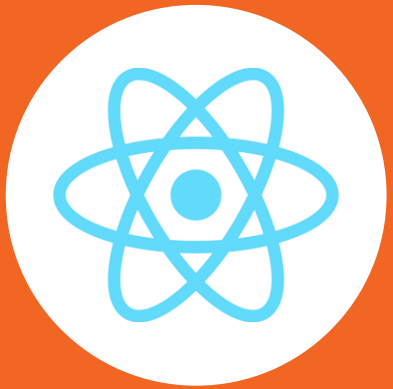
FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

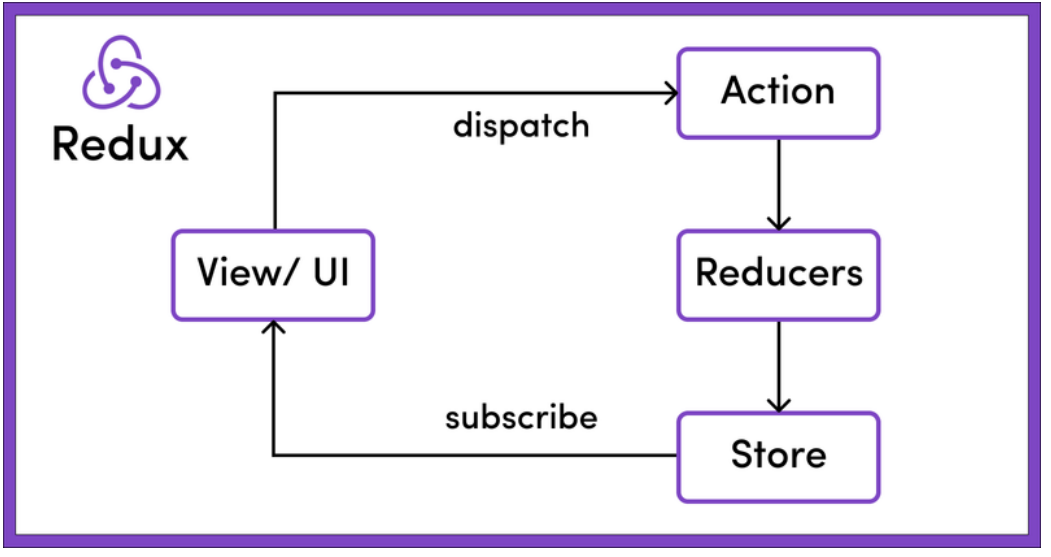
BÀI 6: QUẢN LÝ STATE VÀ XỬ LÝ NETWORK BẰNG REDUX TOOLKIT

PHẦN 1: QUẢN LÝ STATE BẰNG REDUCER VÀ DISPATCH ACTION TRONG RTK

Mục tiêu

- Xây dựng reducer trong Redux Toolkit
- Tìm hiểu về createSlice, createAction.
- Dispatch action, hiển thị dữ liệu trong reducer với useDispatch và useSelector

- Cùng ôn lại một chút kiến thức về Redux, như những gì bạn đã được học. Reducer là nơi sẽ chứa các hàm xử lý nhận các action được dispatch để cập nhật dữ liệu trong store.



Reducer

- Để tạo reducer trong Redux Toolkit có 2 cách:
  - Cách 1: sử dụng createSlice
  - Cách 2: sử dụng createReducer

2 cách tạo reducer trên cơ bản về cách hoạt động chúng đều giống nhau. Chúng chỉ khác nhau cách tạo reducer. Ở function createSlice sẽ dễ tiếp cận hơn cho các bạn mới học RTK, code sẽ được ngắn gọn hơn, nhờ chức năng tự tạo ra tên action của reducer. Bởi vì phạm vi bài học này không thể đủ, nên chúng ta sẽ chỉ tìm hiểu createSlice, chúng ta tiếp tục bài học ở phần dưới nhé.

createSlice

createSlice là gì?

Một hàm chấp nhận state ban đầu, một đối tượng của các hàm reducer và 'tên slice' và tự động tạo các action và các action type tương ứng với các reducer và state.

API này là cách tiếp cận tiêu chuẩn để viết logic Redux.

API này là cách tiếp cận tiêu chuẩn để viết logic Redux.

createSlice

- Tạo mới một file counter.ts để chứa một reducer, reducer này có chức năng thay đổi các giá trị liên quan đến biến đếm. Đầu tiên chúng ta import các thư viện createSlice và PayloadAction.

DaNenTang2 - counter.ts

```
1  import {createSlice} from '@reduxjs/toolkit';
2  import type {PayloadAction} from '@reduxjs/toolkit';
```

createSlice

- Tiếp theo, chúng ta khai báo các state ban đầu của reducer. Đây sẽ là các dữ liệu mà reducer sẽ chứa. Chúng ta có một state tên là value giá trị ban đầu là 0. Lưu ý bạn có thể lưu trữ nhiều kiểu dữ liệu khác nhau. Nhưng không nên lưu quá nhiều dữ liệu, object lồng nhau vào reducer, điều này sẽ ảnh hưởng đến performance ứng dụng của bạn.

DaNenTang2 - counter.ts

```
1  export interface CounterState {
2    value: number;
3  }
4
5  const initialState: CounterState = {
6    value: 0,
7  };
```

createSlice

- Chúng ta gọi hàm createSlcie để tạo reducer

DaNenTang2 - counter.ts

```ts
export const counterSlice = createSlice({
  name: 'counter',
  initialState,
  reducers: {
    increment: state => {
      state.value += 1;
    },
    decrement: state => {
      state.value -= 1;
    },
    incrementByAmount: (state, action: PayloadAction<number>) => {
      state.value += action.payload;
    },
  },
});
```

createSlice

- name: là tên của reducer, tên này được sử dụng trong action types
- initialState: các giá trị state ban đầu reducer xử lý.
- reducers: các hàm xử lý giá trị state trong reducer. Tên key sẽ được dùng để tạo action. Hàm 'builder callback' được sử dụng để thêm nhiều bộ reducer hoặc một object bổ sung của 'case reducers', trong đó các key phải là các action type khác

Parameters

- Các params của createSlice:
  - initialState

    Giá trị state ban đầu cho slice state này

    Đây cũng có thể là một hàm "lazy initializer", hàm này sẽ trả về giá trị state ban đầu khi được gọi. Điều này sẽ được sử dụng bất cứ khi nào reducer được gọi với undefined là giá trị state của nó và chủ yếu hữu ích cho các trường hợp như đọc state ban đầu từ localStorage.

  - name

    Tên chuỗi cho slice state này. Hằng số action type được tạo ra sẽ sử dụng tên này làm tiền tố.

Parameters

- reducers

  Một đối tượng chứa các hàm 'case reducer' Redux (các hàm nhằm xử lý một action type cụ thể, tương đương với một câu lệnh trường hợp duy nhất trong một switch).

  Các key trong object sẽ được sử dụng để tạo hằng số chuỗi action type và chúng sẽ hiển thị trong Redux DevTools Extension khi chúng được dispatched. Ngoài ra, nếu bất kỳ phần nào khác của ứng dụng, dispatch một hành động với chuỗi cùng type chính xác, reducer tương ứng sẽ được chạy. Do đó, bạn nên đặt tên mô tả cho các hàm.

  Đối tượng này sẽ được truyền vào createReducer, vì vậy các reducer có thể 'mutate' state mà chúng được cung cấp một cách an toàn.

Parameters

- Tiếp theo một param nữa mà chưa được demo trong code của chúng ta, đó là params extraReducers

  Về mặt khái niệm, mỗi slice reducer 'sở hữu' slice state của nó. Ngoài ra còn có sự tương ứng tự nhiên giữa logic cập nhật được xác định bên trong reducer và các action type được tạo dựa trên chúng.

  Tuy nhiên, có nhiều lần Redux slice cũng có thể cần cập nhật state của chính nó để đáp ứng với các action được xác định ở nơi khác trong ứng dụng (chẳng hạn như xóa nhiều loại dữ liệu khác nhau khi action 'Đăng xuất' được dispatch). Điều này có thể bao gồm các action type được xác định bởi một lệnh gọi createSlice khác, các action được tạo bởi trình khớp điểm cuối createAsyncThunk, RTK Query hoặc bất kỳ action nào

Parameters

extraReducers cho phép createSlice phản hồi và cập nhật state của chính nó để đáp ứng với các action type khác bên cạnh các type mà nó đã tạo.

Như với trường reducers, mỗi case reducer trong extraReducers được bọc trong Immer và có thể sử dụng cú pháp 'mutating' để cập nhật state bên trong một cách an toàn.

Tuy nhiên, không giống như reducer, mỗi case reducer bên trong extraReducers sẽ không tạo ra một action type hoặc reducer mới.

Nếu hai trường từ reducer và extraReducers tình cờ kết thúc với cùng một action type, hàm từ reducer sẽ được sử dụng để xử lý action type đó.

Parameters

- Ký hiệu "builder callback" extraReducers

  Cách sử dụng extraReducers được khuyến nghị là sử dụng callback nhận phiên bản ActionReducerMapBuilder.

  Ký hiệu trình tạo này cũng là cách duy nhất để thêm matcher reducers và default case reducers vào slice của bạn.

Ví dụ extraReducers

DaNenTang2 - extra.ts

```ts
import {createAction, createSlice, Action, AnyAction} from '@reduxjs/toolkit';
const incrementBy = createAction<number>('incrementBy');
const decrement = createAction('decrement');

interface RejectedAction extends Action {
  error: Error;
}

function isRejectedAction(action: AnyAction): action is RejectedAction {
  return action.type.endsWith('rejected');
}
```

- incrementBy và decrement là 2 action được tạo từ hàm createAction. Trong createSlice khi tạo reducer, nó sẽ tự tạo action kiểu như này, tên của action sẽ dựa vào key.
- isRejectedAcion là một hàm để bắt type action nếu đó bị “rejected”

Ví dụ extraReducers

DaNenTang2 - extra.ts

```ts
createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {},
  extraReducers: builder => {
    builder
      .addCase(incrementBy, (state, action) => {
        // hành động được suy ra chính xác ở đây nếu sử dụng TS
      })
      .addCase(decrement, (state, action) => {})
      .addMatcher(
        isRejectedAction,
        // 'action' sẽ được suy ra là RejectedAction do isRejectedAction được định nghĩa là type guard
        (state, action) => {},
      )
      // và cung cấp trường hợp mặc định nếu không có case nào khác khớp
      .addDefaultCase((state, action) => {});
  },
});

```

Ví dụ extraReducers

Bạn nên sử dụng API này vì nó hỗ trợ TypeScript tốt hơn, vì nó sẽ suy ra chính xác action type trong reducer dựa trên trình tạo action được cung cấp. Nó đặc biệt hữu ích khi làm việc với các action do createAction và createAsyncThunk tạo ra.

Giá trị trả về của createSlice

- createSlice sẽ trả về một object trông như sau:

```
{
   name : string,
   reducer : ReducerFunction,
   actions : Record<string, ActionCreator>,
   caseReducers: Record<string, CaseReducer>.
   getInitialState: () => State
}
```

Mỗi hàm được xác định trong đối số reducers sẽ có một trình tạo action tương ứng được tạo bằng createAction và bao gồm trong trường action của kết quả bằng cách sử dụng cùng một tên hàm.

Giá trị trả về của createSlice

Chức năng reducer được tạo ra phù hợp để chuyển sang chức năng Redux combineReducers như một 'slice reducer'.

Bạn có thể muốn xem xét việc phá hủy các trình tạo action và xuất chúng riêng lẻ, để dễ dàng tìm kiếm các tham chiếu trong một cơ sở mã lớn hơn.

Các hàm được truyền đến tham số reducers có thể được truy cập thông qua trường trả về caseReducers. Điều này có thể đặc biệt hữu ích cho việc kiểm tra hoặc truy cập trực tiếp vào các reducer được tạo nội tuyến.

Hàm của kết quả getInitialState cung cấp quyền truy cập vào giá trị state ban đầu được cung cấp cho slice. Nếu một trình khởi tạo state lazy được cung cấp, nó sẽ được gọi và trả về một giá trị mới.

Ví dụ reducer

- Chúng ta sẽ tạo một ứng dụng counter. Tiếp tục từ phần bài createSlice. Hoàn thành ví dụ, chúng ta có ứng dụng như sau:

Ví dụ reducer

- Thêm hàm multiply trong reducer counter của chúng ta. Hàm này có chức năng nhân state value trong store, nếu user không truyền payload action, thì mặc định sẽ nhân 2.

DaNenTang2 - counter.ts

```
reducers: {
  multiply: {
    reducer: (state, action: PayloadAction<number>) => {
      state.value = state.value * action.payload;
    },
    prepare: (value?: number) => ({payload: value || 2}),
  },
```

Ví dụ reducer

- Bổ sung thêm hàm addMatcher trong extraReducer để bắt action được dispatch. Nếu user gửi action RESET_COUNTER sẽ reset lại trở về state ban đầu

DaNenTang2 - counter.ts

```
extraReducers: builder => {
  builder.addMatcher(
    (action: AnyAction) => action.type === RESET_COUNTER.type,
    () => {
      return initialState;
    },
  );
},
```

Ví dụ reducer

- Export các reducer, các bạn dùng các action type này để dispatch action cho reducer nhận sự kiện.

DaNenTang2 - counter.ts

```
1  export const {increment, decrement, multiply} = counterSlice.actions;
2
```

Ví dụ reducer

- Tiếp theo, các bạn viết hook useAppDispatch dùng để dispatch các action. Hook useAppSelector dùng để lấy giá trị từ store

DaNenTang2 - useRedux.ts

```
1  import {useDispatch, TypedUseSelectorHook, useSelector} from 'react-redux';
2  import {store} from './redux/store';
3
4  export type AppDispatch = typeof store.dispatch;
5  export type RootState = ReturnType<typeof store.getState>;
6
7  export const useAppDispatch = () => useDispatch<AppDispatch>();
8  export const useAppSelector: TypedUseSelectorHook<RootState> = useSelector;
```

Ví dụ reducer

- Gọi hook useAppSelector để gọi state lên UI, counter là tên của reducer

DaNenTang2 - HomeScreen.tsx

```
1   const counter = useAppSelector(state => state.counter);
2
```

Ví dụ reducer

- Tiếp theo bạn gọi hook useDispatch, để dispatch các action cho reducer.

DaNenTang2 - HomeScreen.tsx

```
1  const dispatch = useAppDispatch();
2
3  const onIncreaseCounter = () => dispatch(increment());
4  const onDecrementCounter = () => dispatch(decrement());
5  const onMultiplyCounter = () => dispatch(multiply(3));
6  const onResetCounter = () => dispatch(RESET_COUNTER());
```

increment, decrement, multiply là các tên action tự động tạo trong reducer của createSlice. RESET_COUNTER là tên action được tạo từ createAction

Ví dụ reducer

- Hiển thị giá trị state, và gắn function vào giao diện.

DaNenTang2 - HomeScreen.tsx

```
<Text style={styles.counterText}>{counter?.value}</Text>
<Pressable onPress={onIncreaseCounter} style={styles.btn}>
  <Text>Tăng biến đếm</Text>
</Pressable>

<Pressable onPress={onDecrementCounter} style={styles.btn}>
  <Text>Giảm biến đếm</Text>
</Pressable>
```

Kết luận

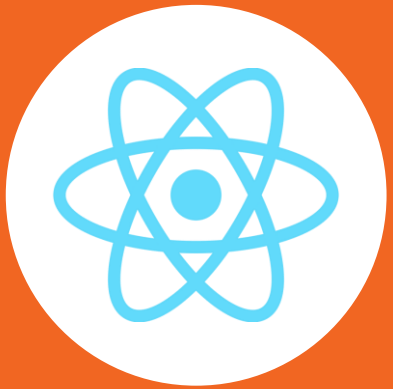
Tổng kết

Ở bài học trên các bạn đã học được cách, tạo reducer, dispatch action và hiển hiển thị dữ liệu từ store, thông qua một số API sau:

- **createSlice:** Tạo reducer và tự động tạo ra action cho reducer
- **createAction**: tạo action
- **useDispatch:** hook dùng để bắn action dựa vào action cung cấp
- **useSelector:** dùng để truy cập dữ liệu từ redux store

FPT Education

FPT POLYTECHNIC



LẬP TRÌNH ĐA NỀN TẢNG VỚI REACT NATIVE

BÀI 6: QUẢN LÝ STATE VÀ XỬ LÝ NETWORK BẰNG REDUX TOOLKIT

PHẦN 2: XỬ LÝ NETWORK BẰNG REDUX TOOLKIT

Mục tiêu

- Giới thiệu về Redux Toolkit query
- Sử dụng hàm createApi, fetchBaseQuery để tạo một request
- Cách sử dụng queries, mutation và transformResponse trong endpoints

createApi là gì?

createApi tự động tạo các hook React cho từng query & mutation endpoints.

Trước kia, để các bạn có thể thao tác với api, bạn phải sử dụng thư viện thứ 3, như axios, để gọi api. Đôi khi bạn còn phải sử dụng thêm thư viện react-query để xử lý thêm cho các query. Công việc setup ban đầu các thư viện này khá vất khả. Này điều cũng gây khó khăn cho các bạn mới tiếp cận React Native, bởi vì nó sẽ chia code ra nhiều phần khác nhau, khiến bạn khó nắm hết.

Đó là lý do createApi ra đời để giải quyết câu chuyện thao tác với network một cách trực quan, ngắn gọn và dễ hiểu.

- Để sử dụng Query trong RTK, bạn import createApi, fetchBaseQuery

DaNenTang2 - pokemon.ts

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

  - fetchBaseQuery(): Một wrapper nhỏ xung quanh fetch nhằm mục đích đơn giản hóa các request. Dự định là baseQuery được đề xuất sẽ được sử dụng trong createApi cho phần lớn người dùng.

- Tiếp theo chúng ta sẽ tạo một hàm xử lý api tên là pokemonApi để gọi api

DaNenTang2 - pokemon.ts

```
export const pokemonApi = createApi({
  reducerPath: 'pokemonApi',
  baseQuery: fetchBaseQuery({baseUrl: 'https://pokeapi.co/api/v2/'}),
  endpoints: builder => ({
    getPokemonByName: builder.query<PokemonType, string>({
      query: name => `pokemon/${name}`,
    }),
  }),
});
```

- reducerPath: là khóa duy nhất mà service của bạn sẽ được gắn vào store của bạn. Nếu bạn gọi createApi nhiều lần trong ứng dụng của mình, bạn sẽ cần cung cấp một giá trị duy nhất mỗi lần. Mặc định là 'api'.

- baseQuery: sử dụng kết hợp với fetchBaseQuery, chứa api gốc, bạn có thể truyền thêm header vào query và nhiều thứ khác vào query của mình.
- endpoints: chứa các function để gọi api, ở ví dụ trên chúng ta có hàm getPokemonByName. PokemonType là giá trị mà query sẽ nhận được, string là chuỗi tên pokemon, để thêm vào params cho query của chúng ta. Chúng ta sẽ truyền string này khi gọi hàm getPokemonByName.

- Chúng ta export function query được sử dụng. Từ hàm getPokemonByName create api trong endpoint sẽ tự động tạo ra 2 hook useGetPokemonByNameQuery, useLazyGetPokemonByNameQuery

DaNenTang2 - pokemon.ts

```
1  export const {useGetPokemonByNameQuery, useLazyGetPokemonByNameQuery} =
2    pokemonApi;
```

- useGetPokemonByNameQuery: sẽ gọi query ngay tại screen khi mout
- useLazyGetPokemonByNameQuery: chỉ gọi query khi chúng ta gọi function trong useLazyGetPokemonByNameQuery

- useGetPokemonByNameQuery: sẽ tự động query khi screen được mount hoặc khi prop truyền vào payload được thay đổi giá trị. Kết quả trả về cho chúng ta các prop liên quan đến query.

DaNenTang2 - PokemonScreen.tsx

```
1  const {data, refetch, isLoading} = useGetPokemonByNameQuery(name);
```

- data: dữ liệu trả về khi query api
- refetch: function gọi lại api để update dữ liệu
- isLoading: trạng thái gọi dữ liệu (boolean)

Ngoài ra, còn có thêm isError, isFetching, isSuccess, status, currentData, endpointName, ... Tuỳ vào yêu cầu từ ứng dụng mà chúng ta sẽ sử dụng các prop cụ thể.

- useLazyGetPokemonByNameQuery: hook cho phép chúng ta gọi query khi muốn

DaNenTang2 - PokemonScreen.tsx

```
1  const [getPokemonByName, result] = useLazyGetPokemonByNameQuery();
2  const {data, isFetching} = result || {};
```

- getPokemonByName: function gọi query của endpoint getPokemonByName.
- result: kết quả của query, cũng tương tự như kết quả trả về từ useGetPokemonByNameQuery. Chúng ta có data, dữ liệu trả về, isFetching trạng thái gọi query

RTK Mutation

- Mutations được sử dụng để gửi cập nhật dữ liệu đến máy chủ và áp dụng các thay đổi cho bộ đệm cục bộ. Mutations cũng có thể làm invalidate dữ liệu được lưu trong bộ nhớ cache và buộc tìm nạp lại dữ liệu.

- Nếu muốn request POST, PUT, DELETE, PATCH,... sử dụng mutation theo hướng dẫn dưới đây:

DaNenTang2 - pokemon.ts

```
updatePokemon: builder.mutation<PokemonResonseType, PokemonDetaiQueryType>({
      query: ({name, body}) => ({
        url: `pokemon/${name}`,
        method: 'PATCH',
        body,
      }),
    }),
```

RTK Mutation

- PokemonResonseType là type dữ liệu trả về của lệnh query,
- PokemonDetaiQuerylType là type dữ liệu body truyền lên query khi gọi hàm updatePokemon
- method là phương thức truyền lên query, có thể đặt POST, PUT, GET, DELETE, ...

RTK Mutation

- Cách gọi hàm gọi hook lệnh mutation vừa tạo. updatePokemon là function bắt đầu gọi lệnh query, resultUpdatePokemon là giá trị trả về của lệnh query

DaNenTang2 - PokemonScreen.tsx

```tsx
const [updatePokemon, resultUpdatePokemon] = useUpdatePokemonMutation();

const onUpdatePokemon = () => {
  updatePokemon({name: 'bulbasaur', body: {name: 'bulbasaur22'}});
};
```

Khi gọi function updatePokemon chúng ta truyền thêm body vào lệnh query. Tuỳ vào api, mà chúng ta sẽ truyền theo body tương ứng.

transformResponse

- endpoints trên createApi chấp nhận thuộc tính transformResponse cho phép thao tác dữ liệu được trả về bởi một truy vấn hoặc đột biến trước khi nó truy cập vào bộ nhớ cache.

- transformResponse được gọi với dữ liệu mà baseQuery thành công trả về cho endpoint tương ứng và giá trị trả về của transformResponse được sử dụng làm dữ liệu được lưu trong bộ nhớ đệm được liên kết với lệnh gọi endpoint đó.

```
transformResponse: (response, meta, arg) =>
  response.some.deeply.nested.collection
```

transformResponse

- Chúng ta sẽ transform lại response của chúng ta lại như sau: Nếu genderId giá trị từ api trả về bằng 1, chúng ta sẽ thêm field gender trong response trả về bằng male, ngược lại băng female

DaNenTang2 - pokemon.ts

```
PokemonDetaiQueryType>({
      query: ({name, body}) => ({
        url: `pokemon/${name}`,
        method: 'PATCH',
        body,
      }),
      transformResponse: (response: PokemonResonseFormatType, meta, arg) => {
        return {
          ...response,
          gender: response.genderId === 1 ? 'male' : 'female',
        };
      },
    }),
```

Header Query

- Để truyền header vào request của bạn, bạn có thể sử dụng prepareHeaders trong fetchBaseQuery

  Cho phép bạn thêm headers vào mọi yêu cầu. Bạn có thể chỉ định headers ở endpoint cuối, nhưng thông thường bạn sẽ muốn đặt các headers phổ biến như authorization tại đây. Là một cơ chế tiện lợi, đối số thứ hai cho phép bạn sử dụng getState để truy cập store redux của bạn trong trường hợp bạn lưu trữ thông tin bạn sẽ cần ở đó, chẳng hạn như auth token. Ngoài ra, nó cung cấp quyền truy cập extra, endpoint, type và forced để mở khóa các hành vi có điều kiện chi tiết hơn.

  Bạn có thể mutate headers trực tiếp và trả về nó là tùy chọn.

Header Query

- Để thêm token vào header bạn có thể sử dụng prop headers để set header cho request cách sau đây:

DaNenTang2 - pokemon.ts

```
baseQuery: fetchBaseQuery({
  baseUrl: 'https://api.restful-api.dev',
  prepareHeaders: (headers, {getState}) => {
    const token = (getState() as RootState).auth.token;

    // If we have a token set in state, let's assume that we should be passing it.
    if (token) {
      headers.set('authorization', `Bearer ${token}`);
    }

    return headers;
  },
}),
```

Tổng kết

- Ở chương query này, chúng ta đã được học cách gọi api từ createApi, cách gọi lệnh GET bằng query, lệnh POST, DELETE, PUT, PATCH, ... qua mutation
- Các bạn cũng đã được học cách truyền header thông qua prop prepareHeaders của fetchBaseQuery
- Cách sử dụng transformResponse để format lại dữ liệu trước khi được trả về và cache lại trong bộ nhớ

Tổng kết

- Xây dựng reducer trong Redux Toolkit
- Tìm hiểu về createSlice, createAction.
- Dispatch action, hiển thị dữ liệu trong reducer với useDispatch và useSelector
- Giới thiệu về Redux Toolkit query
- Sử dụng hàm createApi, fetchBaseQuery để tạo một request
- Cách sử dụng queries, mutation và transformResponse trong endpoints

Kết thúc