**ReactNative – JavaScript**

# **REDUX**

1. **Giới thiệu về cách hoạt động của Redux**

- **Redux** ra đời lấy cảm hứng từ tư tưởng của ngôn ngữ **Elm** và kiến trúc **Flux**của Facebook

-Công cụ dùng để quản lí trạng thái (state)

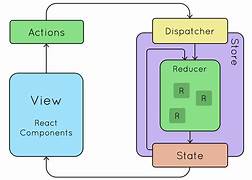
**Giải thích đơn giản:** Khi ứng dụng của bạn có nhiều thành phần cần truy cập và chia sẻ dữ liệu chung, Redux sẽ giúp bạn lưu trữ dữ liệu này ở một nơi gọi là **store**. Bất cứ khi nào dữ liệu thay đổi, Redux sẽ cập nhật cho tất cả các phần của ứng dụng cần dữ liệu đó, giúp ứng dụng dễ quản lý và tránh lỗi.

**Lợi ích của Redux:**

* Dữ liệu nằm trong một nơi duy nhất, dễ quản lý hơn.
* Mọi thay đổi dữ liệu đều qua một quy trình rõ ràng, dễ theo dõi và sửa lỗi.

**Ví dụ dễ hiểu:** Giả sử bạn có một ứng dụng quản lý giỏ hàng. Mỗi khi thêm sản phẩm, giỏ hàng phải được cập nhật ở nhiều nơi (trang giỏ hàng, tổng số sản phẩm ở góc màn hình). Redux giúp lưu trữ giỏ hàng ở một nơi và mỗi khi thêm sản phẩm, nó sẽ tự động cập nhật mọi nơi cần đến giỏ hàng đó.

1



1. **Tìm hiểu store, reducer, action, state**

## **Store (Kho lưu trữ)**

npm reudux --save

* **Là gì?**: Store là nơi chứa tất cả dữ liệu của ứng dụng trong Redux. Nó giống như một nơi lưu trữ trung tâm mà mọi phần của ứng dụng có thể truy cập.
* **Vai trò**: Giữ trạng thái (state) ứng dụng và cung cấp cách để truy cập và cập nhật trạng thái này.
* **Ví dụ**: Nếu ứng dụng có danh sách người dùng, giỏ hàng, và thông tin đăng nhập, store sẽ lưu trữ toàn bộ dữ liệu này ở một chỗ.

## **State (Trạng thái)**

* **Là gì?**: State là dữ liệu hiện tại của ứng dụng, hay chính là dữ liệu đang được lưu trữ trong store.
* **Vai trò**: Lưu trữ dữ liệu hiện tại của ứng dụng và tự động thay đổi khi có hành động mới.
* **Ví dụ**: Nếu người dùng đang đăng nhập và giỏ hàng có ba sản phẩm, state sẽ lưu trạng thái đăng nhập và danh sách sản phẩm đó.

## **Action (Hành động)**

* **Là gì?**: Action là một đối tượng mô tả hành động muốn thực hiện, chẳng hạn như thêm sản phẩm vào giỏ hàng hay đăng xuất người dùng.
* **Vai trò**: Nói cho Redux biết loại thay đổi nào cần được thực hiện trên state.
* **Ví dụ**: Khi người dùng nhấn "Thêm vào giỏ hàng," một action có tên ADD\_TO\_CART sẽ được tạo ra, bao gồm loại hành động (ADD\_TO\_CART) và dữ liệu cần thêm.

## **Reducer (Bộ xử lý)**

* **Là gì?**: Reducer là một hàm nhận vào state hiện tại và một action, sau đó quyết định cách cập nhật state dựa trên action đó.
* **Vai trò**: Xử lý các thay đổi state một cách có tổ chức và không trực tiếp thay đổi state, mà luôn tạo ra một bản sao mới của state với các thay đổi.
* **Ví dụ**: Khi action ADD\_TO\_CART được gọi, reducer sẽ thêm sản phẩm vào giỏ hàng trong state và trả về state mới.

**Cách chúng hoạt động với nhau**

1. Người dùng thực hiện một hành động (action), ví dụ: "Thêm sản phẩm vào giỏ hàng."
2. Hành động này được gửi đến reducer.
3. Reducer xử lý hành động và tạo ra state mới (giỏ hàng với sản phẩm mới).
4. Store lưu trữ state mới, cập nhật tất cả các thành phần cần biết về thay đổi này.

Tóm lại, **store** là nơi chứa dữ liệu, **state** là dữ liệu hiện tại, **action** là yêu cầu thay đổi dữ liệu, và **reducer** là nơi quyết định thay đổi nào được thực hiện cho state.

# **Middleware**

Hãy nghĩ về **middleware** như một "cầu nối" nằm giữa việc bạn gửi một hành động (**dispatch action**) và việc **reducer** xử lý hành động đó.

**Ví dụ thực tế về middleware**

Giả sử bạn có một nhân viên làm nhiệm vụ giao thư từ (giống như Redux **dispatch** action). Khi nhân viên giao thư, thư sẽ đi qua bộ phận kiểm tra an ninh (middleware), nơi thư có thể:

* Bị kiểm tra nội dung, xác minh tính hợp lệ.
* Được phép gửi tiếp đến người nhận hoặc bị dừng lại nếu có vấn đề.

Trong Redux, middleware có vai trò như **bộ phận an ninh**. Nó giúp kiểm tra hoặc thực hiện các thay đổi đối với hành động trước khi hành động đến reducer.

**Lợi ích của Middleware**

* **Xử lý bất đồng bộ**: Redux thường chỉ xử lý các hành động đồng bộ, nhưng middleware cho phép bạn làm việc với các tác vụ bất đồng bộ (như gọi API).
* **Theo dõi và log**: Bạn có thể dùng middleware để ghi lại toàn bộ hành động (logging) giúp dễ dàng kiểm tra lỗi.

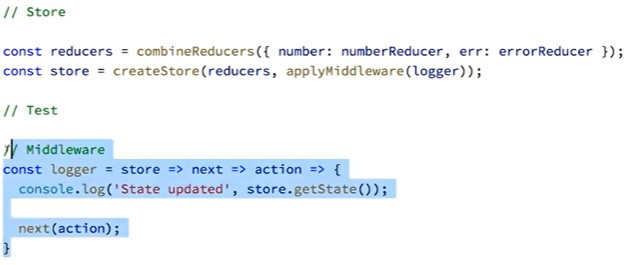
**Cách middleware hoạt động trong Redux**

Khi bạn gửi một hành động (ví dụ: tăng số đếm), Redux sẽ chuyển hành động qua middleware:

1. Middleware **kiểm tra** hành động này, và có thể chỉnh sửa hoặc thực hiện tác vụ nào đó.
2. Sau đó, middleware sẽ **gửi hành động đó đến reducer** để cập nhật state.

**Cách làm :**

Import {createStore, combineReducers, **applyMiddleware**} from ‘redux’



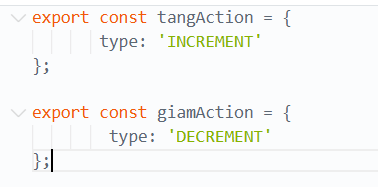
Trong store mỗi middleware cách nhau bởi dấu ,

# **4.Cách làm redux (Ví dụ như tăng và giảm)**

## **B1: Tạo trang chính (couter.js)**

## **B2:Tạo folder redux có các file như action.js, middleware.js, reducer.js, store.js**

## **B3: Action.js : định nghĩa các sự kiện**



**Lưu ý :**

* Phải export
* Action là đối tượng phải có **key** là **type**.Và có thể thêm các key khác
* Có thể có nhiều action

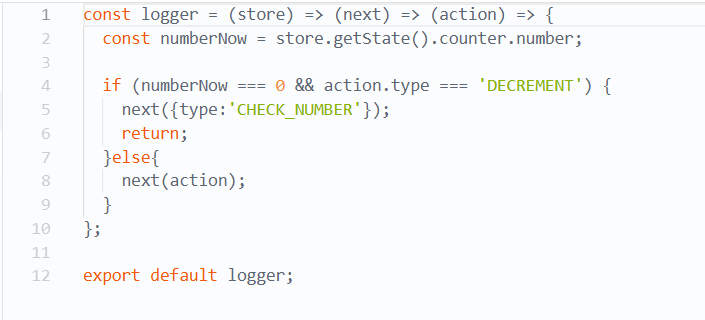
## **B4: Middleware.js (nếu có)**

Đây là giai đoạn trung gian giữa action và reducer .Middleware nó sẽ kiểm tra là cái action đó có nên được thông qua.Và nếu không được thông qua nó có thể bỏ hoặc là chọn action khác.

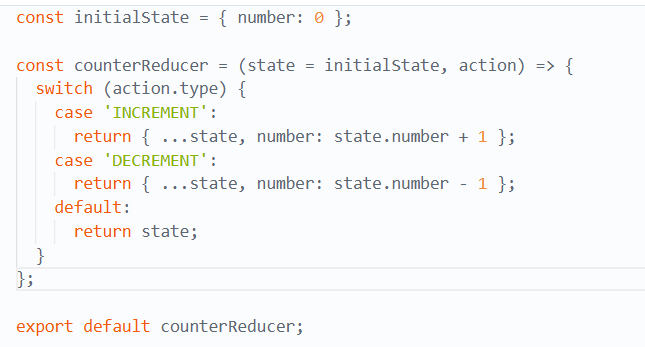
Ví dụ ta có action đăng nhập nhưng mà người dùng không chịu nhập tên đăng nhập mà bấm đăng nhập thì middleware nó sẽ không chp reducer làm đăng nhập mà nó tự động chuyển sang đăng kí.

**Lưu ý :**

* Dòng đầu giữ nguyên (chỉ thay đổi tên hàm)
* Phải export
* Phải là hàm "curry function”



## **B5: Reducer.js : thực thi theo cái action (code)**



**Lưu ý :**

* Phải export
* Là arrow function .Nhận vào 2 tham số là state và action
* Có thể có nhiều reducer
* Bắt buộc có return **state**

## **B6: store.js : Lưu trữ các trạng thái (number)**

**Lưu ý :**

* Phải export
* Chỉ có 1 store
* Import createStore truyền vào reducer và middleware
* Nếu k có middleware thì khỏi truyền
* Nếu có middleware thì phải import applyMiddleware.



## **B7: App.js**

* Import Provider, store, các pages / screens
* Bao bọc toàn bộ App.js bằng Provider



## **B8: Couter.js (1 trong các screen/page)**

**Lưu ý:**

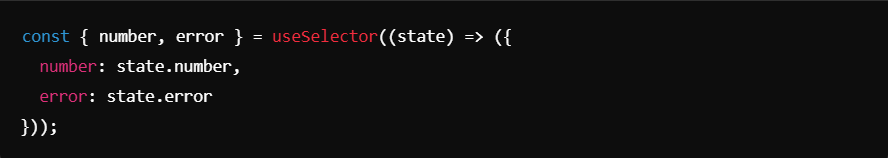
* Import useDispatch, useSelector
* Import các cái đối tượng trong action mà muốn sử dụng
* Nếu là đối tượng thì bao {} và cách nhau dấu ,

**Sử dụng :**

B1 : const dispatch = useDispatch(); (giữ nguyên)

B2: const number = useSelector((state) => state.number); //tùy theo state muốn lấy

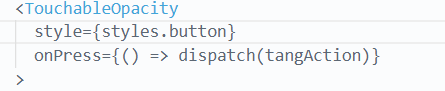
Ví dụ nếu state của mình có nhiều key hơn :



Hoặc lấy toàn bộ

****

B3: Gắn sự kiện



# **5.Cập nhật**

## **Có nhiều Reducer**

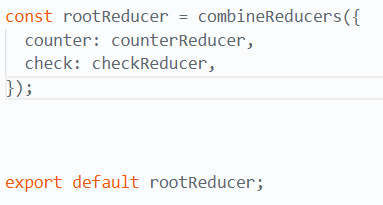
### B1: Tạo action

### B2: Viết thêm hàm reducer

### B3: Import trong reducer.js

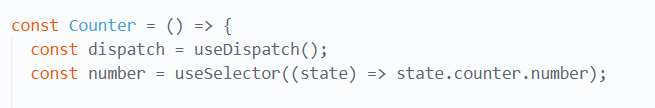
import { combineReducers } from 'redux';

### B4: Sử dụng



Giải thích **: counter và check là tên tự đặt, couterReducer và checkReducer là tên hàm Reducer.**

### B5: Update lại page/screen



# **REDUX TOOLKIT**

Redux Toolkit giúp việc phát triển ứng dụng với Redux trở nên dễ dàng hơn, nhanh chóng hơn và ít lỗi hơn. Nó rất phù hợp cho những người mới bắt đầu làm quen với Redux cũng như cho các dự án lớn cần sự quản lý trạng thái phức tạp.

**-**Phiên bản sau của Redux

**-Công dụng :** y chang redux

**- Ưu điểm :**

+ Hạn chế code

+ Hạn chế lỗi

+ Nhanh

## **Cách làm redux-toolkit (quản lí State)**

### B1: Tạo slice.js và store.js

### B2: slice.js : kết hợp của action và reducer thông qua createSlice

* Import

import { createSlice } from '@reduxjs/toolkit';

* Code

|  |
| --- |
| import { createSlice } from '@reduxjs/toolkit';  const counterSlice = createSlice({    name: 'counter', //đặt tên mà mình muốn    initialState: { number: 0 }, // state    reducers: { //các hàm của reducer      increment: (state,action) => {        state.number += 1; *// Tăng giá trị lên 1*      },      decrement: (state,action) => {        state.number -= 1; *// Giảm giá trị xuống 1*      },    },  });  export const { increment, decrement } = counterSlice.actions; // các tên hàm của reducer  export default counterSlice.reducer; |

### B3: store.js

* Import : configureStore và slice
* Code :



### B4: Tạo app.js

|  |
| --- |
| import React from 'react';  import { Provider } from 'react-redux';  import  store  from './reduxtoolkit/store'; //không phải là {store}  import Counter from './page/couter';  export default function App() {    return (      <Provider store={store}>        <Counter />      </Provider>    );  } |

### B5: page/screen ( có vài chỗ khác)

|  |
| --- |
| import { useDispatch, useSelector } from 'react-redux';  import {increment, decrement} from '../reduxtoolkit/counterSlice';  const Counter = () => {    const dispatch = useDispatch();    const count = useSelector((state) => state.counter.number);  //counter ở store    return (      <SafeAreaView style={styles.container}>        <TouchableOpacity          style={styles.button}          onPress={() => dispatch(increment())} //có dấu () sau hàm        >          <Text style={styles.buttonText}> + </Text>        </TouchableOpacity>        <Text style={styles.counterText}>{count}</Text>        <TouchableOpacity          style={styles.button}          onPress={() => dispatch(decrement())}//có dấu () sau hàm        >          <Text style={styles.buttonText}> - </Text>        </TouchableOpacity>       </SafeAreaView>    );  }; |

## **Cách làm redux\_toolkit**

### B1: Tạo slice.js và store.js

### B2: Code slice.js

|  |
| --- |
| import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';  import axios from 'axios';  *// Tạo một async thunk để gọi API*  export const fetchBikes = createAsyncThunk('bike/fetchBike', async () => {    const response = await axios.get('https://67219f8298bbb4d93ca9023e.mockapi.io/bike');    return response.data;  });  *// Hàm async thunk để thêm xe đạp mới*  export const addBike = createAsyncThunk('bike/addBike', async (newBike, { dispatch }) => {    const response = await axios.post('https://67219f8298bbb4d93ca9023e.mockapi.io/bike', newBike);  *// Refetch the bike list after adding a new bike*    dispatch(fetchBikes());    return response.data;  });  *// Tạo slice*  const bikeSlice = createSlice({    name: 'bike',    initialState: {      bikes: [],      status: 'idle', *// idle | loading | succeeded | failed*      error: null,    },    reducers: {},    extraReducers: (builder) => { // Đây là nơi xử lý các trạng thái của async thunks      builder  *//Lay danh sach xe dap*        .addCase(fetchBikes.pending, (state) => { // pending đang chạy          state.status = 'loading';        })        .addCase(fetchBikes.fulfilled, (state, action) => { // fulfilled thành công          state.status = 'succeeded';          state.bikes = action.payload;        })        .addCase(fetchBikes.rejected, (state, action) => { // rejected thất bại          state.status = 'failed';          state.error = action.error.message;        })  *// Xử lý khi thêm xe đạp*        .addCase(addBike.pending, (state) => {          state.status = 'loading';        })        .addCase(addBike.fulfilled, (state, action) => {          state.status = 'succeeded';        })        .addCase(addBike.rejected, (state, action) => {          state.status = 'failed';          state.error = action.error.message;        });    },  });  export default bikeSlice.reducer; |

createAsyncThunk là một API của Redux Toolkit cho phép bạn tạo ra các action bất đồng bộ (async action).

Reducers dành cho các action đồng bộ (synchronous).

ExtraReducers dành cho các action bất đồng bộ (asynchronous), đặc biệt là những action được tạo ra từ createAsyncThunk.

### B3: Code store.js

|  |
| --- |
| import { configureStore } from '@reduxjs/toolkit';  import bikeReducer from './slice';  const store = configureStore({    reducer: {      bikes: bikeReducer,    },  });  export default store; |

### B4:Code app.js (tương tự)

### B5: Page/screen

|  |
| --- |
| import { useDispatch, useSelector } from 'react-redux';  import { fetchBikes } from '../redux\_toolkit/slice';  *// import action ays trong slice*  import React, { useState, useEffect } from 'react'; |

|  |
| --- |
| const dispatch = useDispatch();  const bike = useSelector((state) => state.bikes.bikes); //state.store.slice  const status = useSelector((state) => state.bikes.status);  const error = useSelector((state) => state.bikes.error);   useEffect(() => {      if (status === 'idle') {        dispatch(fetchBikes());      }    }, [status, dispatch]);    if (status === 'loading') return <Text>Loading...</Text>;    if (status === 'failed') return <Text>Error: {error}</Text>; |

## **Cập nhật**

### Nếu slice.js có thêm state

|  |
| --- |
| import { createSlice } from '@reduxjs/toolkit';  const counterSlice = createSlice({    name: 'counter',    initialState: {      number: 0,   *// Trạng thái số đếm*      error: null, //ta thêm 1 state là error để lưu lỗi    },    reducers: {      increment: (state) => {        state.number += 1;        state.error = null; *// Đặt lại lỗi khi tăng số*      },      decrement: (state) => {        state.number -= 1;        state.error = null; *// Đặt lại lỗi khi giảm số*      },      reset: (state) => {        state.number = 0;        state.error = null; *// Đặt lại lỗi khi đặt lại số*      },      setError: (state, action) => { //hàm này để cập nhật lỗi        state.error = action.payload;  //payload là redux toolkit tự có (nếu ở cái trang xử lí ví dụ như counter.js truyền giá trị gì thì payload sẽ là giá trị đó      },      clearError: (state) => {        state.error = null; *// Xóa lỗi*      },    },  });  *// Xuất các action*  export const { increment, decrement, reset, setError, clearError } = counterSlice.actions;  *// Xuất reducer*  export default counterSlice.reducer; |

Thì ở trang page/screen

|  |
| --- |
| const { number, error } = useSelector((state) => state.counter);  // Nếu có nhiều state thì làm như trên . counter được định nghĩa ở store  //gọi hàm cập nhật lỗi  const handleError = () => {      dispatch(setError('Đã xảy ra lỗi!')); *// Thiết lập lỗi*  };  //'Đã xảy ra lỗi!' sẽ được gắn cho payload đã nói ở trên  const handleClearError = () => {   dispatch(clearError()); *// Xóa lỗi*  }; |

### Nếu slice.js có 2 hàm luôn

|  |
| --- |
| import { createSlice } from '@reduxjs/toolkit';  *// Slice cho counter*  const counterSlice = createSlice({    name: 'counter',    initialState: {      number: 0,   *// Trạng thái số đếm*      error: null, *// Trạng thái lỗi*    },    reducers: {      increment: (state) => {        state.number += 1;        state.error = null; *// Đặt lại lỗi khi tăng số*      },      decrement: (state) => {        state.number -= 1;        state.error = null; *// Đặt lại lỗi khi giảm số*      },      reset: (state) => {        state.number = 0;        state.error = null; *// Đặt lại lỗi khi đặt lại số*      },      setError: (state, action) => {        state.error = action.payload; *// Cập nhật lỗi*      },      clearError: (state) => {        state.error = null; *// Xóa lỗi*      },    },  });  *// Slice cho tasks (đây là hàm slice mới)*  const tasksSlice = createSlice({    name: 'tasks',    initialState: {      list: [],  *// Danh sách nhiệm vụ*    },    reducers: {      addTask: (state, action) => {        state.list.push(action.payload); *// Thêm nhiệm vụ mới vào danh sách*      },      removeTask: (state, action) => {        state.list = state.list.filter(task => task.id !== action.payload.id); *// Xóa nhiệm vụ theo id*      },      clearTasks: (state) => {        state.list = []; *// Xóa tất cả nhiệm vụ*      },    },  });  *// Xuất các action từ counter slice*  export const { increment, decrement, reset, setError, clearError } = counterSlice.actions;  *// bỏ sung thêm hàm sau*  export const { addTask, removeTask, clearTasks } = tasksSlice.actions;  *// Chỉnh sửa*  export default {    counter: counterSlice.reducer,    tasks: tasksSlice.reducer,  }; |

Nhưng vẫn khuyến khích là mỗi hàm nên tách nhau ra file.js thì tốt hơn.

### Nếu muốn dùng với middleware

B1: Viết middleware (tương tự chỉ khác cái type)

|  |
| --- |
| *// customMiddleware.js*  const checkNumberMiddleware = (store) => (next) => (action) => {    if (action.type === 'counter/decrement') {      const currentState = store.getState().counter.number; *// Lấy số hiện tại*      if (currentState === 0) {  *// Nếu số hiện tại là 0, dispatch hành động increment*        store.dispatch({ type: 'counter/increment' });      }    }    return next(action); *// Tiếp tục với hành động*  };  export default checkNumberMiddleware; |

Giải thích : do dùng reduxtoolkit ta không có tạo action nên type sẽ làm hơi khác

* 'counter/increment' : counter chính là cái tên slice và increment là tên hàm trong slice đó. Chỉ rõ ở bên dưới

B2: Update store

|  |
| --- |
| import { configureStore } from '@reduxjs/toolkit';  import counterReducer from './path/to/counterSlice';  import checkNumberMiddleware from './path/to/customMiddleware'; *// Đường dẫn tới tệp chứa middleware*  const store = configureStore({    reducer: {      counter: counterReducer,    },    middleware: (getDefaultMiddleware) =>      getDefaultMiddleware().concat(checkNumberMiddleware), *// Thêm middleware tùy chọn*  });  export default store; |

**//Hãy làm thêm các bài như đọc dữ liệu, CRDU bằng redux toolkit**

# **REDUX SAGA**

## **Redux saga là gì ?**

**Middleware trong Redux**

Middleware trong Redux là một đoạn mã giúp bạn can thiệp vào quá trình gửi (dispatch) và xử lý (process) các action. Điều này cho phép bạn thực hiện một số hành động trước hoặc sau khi một action được gửi tới reducer. Middleware có thể được sử dụng cho nhiều mục đích khác nhau, như:

* **Ghi log**: Theo dõi các action và trạng thái ứng dụng.
* **Gọi API**: Thực hiện các cuộc gọi tới API và xử lý phản hồi.
* **Xử lý tác vụ bất đồng bộ**: Quản lý các tác vụ không đồng bộ, như làm việc với Promise.

**Redux Saga**

Redux Saga là một thư viện middleware được thiết kế để quản lý các tác vụ bất đồng bộ một cách hiệu quả hơn. Saga sử dụng các generator functions(function \*) để quản lý các side effects (tác động bên ngoài) trong ứng dụng.

#### Cách Hoạt Động Của Redux Saga

1. **Lắng Nghe Action**:
   * Middleware saga sẽ "lắng nghe" các action mà bạn phát đi (dispatch). Khi một action được gửi, middleware sẽ kiểm tra xem action đó có phải là một action mà saga cần xử lý hay không.
2. **Chạy Saga Tương Ứng**:
   * Nếu action đó phù hợp với một trong những saga mà bạn đã định nghĩa (ví dụ: FETCH\_SONGS\_REQUEST), middleware sẽ chạy saga tương ứng.
   * Saga này có thể thực hiện các tác vụ như gọi API, xử lý dữ liệu và phát đi các action khác (như FETCH\_SONGS\_SUCCESS hoặc FETCH\_SONGS\_FAILURE) dựa trên kết quả của tác vụ.
3. **Quản Lý Tác Vụ Bất Đồng Bộ**:
   * Các saga có thể dễ dàng quản lý các tác vụ bất đồng bộ bằng cách sử dụng yield để chờ đợi các Promise (như gọi API) hoàn thành.
   * Điều này giúp cho mã nguồn trở nên dễ hiểu hơn và giúp bạn quản lý logic phức tạp mà không làm phức tạp hóa component React.

## **Redux saga dùng để làm gì ?**

## **Khi nào sử dụng ?**

1.Khi Có Nhiều Tác Vụ Bất Đồng Bộ

* Nếu ứng dụng của bạn có nhiều cuộc gọi API hoặc các tác vụ bất đồng bộ khác, Redux Saga sẽ giúp bạn quản lý và tổ chức chúng một cách rõ ràng và dễ dàng hơn.

2. Khi Cần Quản Lý Trạng Thái Phức Tạp

* Nếu bạn cần thực hiện nhiều hành động (actions) hoặc điều kiện (conditions) trước khi gửi một yêu cầu, Redux Saga sẽ giúp bạn dễ dàng quản lý các tình huống phức tạp này.

3. Khi Cần Thao Tác Với Các Tác Vụ Không Đồng Bộ

* Nếu bạn cần thực hiện các tác vụ như debounce, throttle, hoặc xử lý các hành động lặp lại (retries), Redux Saga cung cấp các hiệu ứng (effects) hữu ích để giúp bạn thực hiện những điều này.

4. Khi Cần Xử Lý Mạch Chạy Tác Vụ

* Redux Saga giúp bạn dễ dàng quản lý thứ tự các tác vụ. Bạn có thể dễ dàng xác định mạch chạy của các saga, giúp việc điều phối các tác vụ phức tạp trở nên đơn giản hơn.

## **Cách sử dụng redux saga ?**

Cần có các class .js như store (chỉ có 1) , sagas, reducer

### B1: Tạo store

|  |
| --- |
| import { createStore, applyMiddleware } from 'redux';import createSagaMiddleware from 'redux-saga';import rootReducer from './reducers'; //reducerimport rootSaga from './sagas'; //saga--------------------------------------------------------------------------------------------const sagaMiddleware = createSagaMiddleware(); //Gọi hàm này để tạo một middleware saga----------------------------------------------------------------------------------------------const store = createStore( //hàm tạo store :nhận 2 tham số là reducer và saga  rootReducer, //reducer  applyMiddleware(sagaMiddleware) //hàm này cho phép saga hoạt động);sagaMiddleware.run(rootSaga); //Gọi hàm này để khởi động saga gốc (rootSaga). Điều này cho phép middleware bắt đầu lắng nghe các action và thực hiện các tác vụ bất đồng bộ khi cần thiết.export default store; |

### B2: Tạo reducer

**-Đầu tiền cần có giá trị khởi tạo ban đầu**

|  |
| --- |
| const initialState = {songs: [], //mảng bài hát trốngartists: [],loading: false, //trang thái load lấy dữ liệu falseerror: null, //lỗi chưa có}; |

-Viết hàm reducer :nhận vào 2 tham số là state (trạng thái ban đầu) và action

|  |
| --- |
| const rootReducer = (state = initialState, action) => {    switch (action.type) { //kiểm tra nó là action gì      case 'FETCH\_SONGS\_REQUEST': //bắt đầu lấy      case 'FETCH\_ARTISTS\_REQUEST': //bắt đầu lấy        return { ...state, loading: true, error: null };      case 'FETCH\_SONGS\_SUCCESS': //lấy bài hát thành công        return { ...state, songs: action.payload, loading: false }; // reducer sẽ cập nhật mảng songs với dữ liệu từ action.payload      case 'FETCH\_ARTISTS\_SUCCESS':        return { ...state, artists: action.payload, loading: false }; //tương tự      case 'FETCH\_SONGS\_FAILURE':      case 'FETCH\_ARTISTS\_FAILURE':        return { ...state, loading: false, error: action.payload };      default:        return state; //lưu ý phải return state    }  }; |

### B3: Tạo các sagas

**-Import các thư viện cần**

|  |
| --- |
| import { call, put, takeLatest } from 'redux-saga/effects'; //của saga  import axios from 'axios'; //dùng để đọc api link |

**-Viết các hàm như đọc dữ liệu ,…..**

|  |
| --- |
| // Hàm gọi API lấy danh sách bài hát  **function\*** fetchSongs() {    try {      const accessToken = yield localStorage.getItem('spotifyAccessToken'); // Lấy access token đã lưu      const response = yield call(axios.get, 'https://api.spotify.com/v1/tracks', {        headers: {          Authorization: `Bearer ${accessToken}`,        },      });      yield put({ type: 'FETCH\_SONGS\_SUCCESS', payload: response.data.items });    } catch (error) {      yield put({ type: 'FETCH\_SONGS\_FAILURE', payload: error.message });    }  } |

Đây là một hàm generator. Trong Redux Saga, hàm generator cho phép bạn viết các hàm bất đồng bộ mà vẫn giữ được sự dễ đọc và quản lý.

**-Viết hàm gốc**

|  |
| --- |
| function\* rootSaga() {  yield takeLatest('FETCH\_SONGS\_REQUEST', fetchSongs);  yield takeLatest('FETCH\_ARTISTS\_REQUEST', fetchArtists);  } |

Giải thích : Khi action FETCH\_SONGS\_REQUEST được phát đi, saga sẽ gọi hàm fetchSongs để lấy bài hát.

### B4: Sử dụng trong các trang muốn hiển thị

**-Import thư viện**

import { useDispatch, useSelector } from 'react-redux';

**-Lấy kết quả**

const dispatch = useDispatch();

const { songs, artists, loading, error } = useSelector(state => state);

**-Viết hàm đọc kết quả**

Dùng flatlist

**-Nhưng nếu data gốc bị sửa thì cũng nên cập nhật ở đây nên viết thêm hàm cập nhật mỗi khi data gốc bị thay đổi**

|  |
| --- |
| useEffect(() => {  dispatch({ type: 'FETCH\_SONGS\_REQUEST' });  dispatch({ type: 'FETCH\_ARTISTS\_REQUEST' });  }, [dispatch]);  if (loading) { return <Text>Loading...</Text>; }  if (error) { return <Text>Error: {error}</Text>; } |

## **Cập nhật ?**

# **HOOKS**

**CallAPI**

|  |
| --- |
| import React, { useState, useEffect } from 'react';  const FetchDataWithHook = () => {  const [data, setData] = useState(null); // Dùng useState để lưu dữ liệu  const [loading, setLoading] = useState(true); // Dùng useState để theo dõi trạng thái tải  const [error, setError] = useState(null); // Dùng useState để lưu thông báo lỗi  // Sử dụng useEffect để gọi API khi component được render  useEffect(() => {  fetch('https://api.example.com/data')  .then((response) => response.json()) // Chuyển đổi dữ liệu sang JSON  .then((data) => {  setData(data); // Lưu dữ liệu vào state  setLoading(false); // Đặt loading là false khi có dữ liệu  })  .catch((err) => {  setError(err.message); // Lưu lỗi nếu có  setLoading(false);  });  }, []); // [] để gọi API chỉ một lần khi component được render lần đầu tiên  if (loading) return <div>Loading...</div>;  if (error) return <div>Error: {error}</div>;  return (  <div>  <h1>Data from API</h1>  <pre>{JSON.stringify(data, null, 2)}</pre>  </div>  );  };  export default FetchDataWithHook; |

# **REDUX THUNK**

## **Cách làm**

### B1:Code action

|  |
| --- |
| */ src/redux/actions/bikeActions.js*  import axios from 'axios';  *// Action để lấy danh sách xe đạp*  export const fetchBikes = () => async (dispatch) => {    dispatch({ type: 'bike/fetchBikes/pending' }); *// Trạng thái loading*    try {      const response = await axios.get('https://67219f8298bbb4d93ca9023e.mockapi.io/bike');      dispatch({        type: 'bike/fetchBikes/fulfilled',        payload: response.data,  *// Dữ liệu trả về từ API*      });    } catch (error) {      dispatch({        type: 'bike/fetchBikes/rejected',        error: error.message,  *// Thông báo lỗi*      });    }  };  *// Action để thêm xe đạp*  export const addBike = (newBike) => async (dispatch) => {    dispatch({ type: 'bike/addBike/pending' }); *// Trạng thái loading*    try {      const response = await axios.post('https://67219f8298bbb4d93ca9023e.mockapi.io/bike', newBike);      dispatch({        type: 'bike/addBike/fulfilled',        payload: response.data,  *// Dữ liệu trả về từ API*      });  *// Sau khi thêm xe đạp mới, chúng ta sẽ gọi lại fetchBikes để làm mới danh sách*      dispatch(fetchBikes());    } catch (error) {      dispatch({        type: 'bike/addBike/rejected',        error: error.message,  *// Thông báo lỗi*      });    }  }; |

### B2: Code Reducer

|  |
| --- |
| *// src/redux/reducers/bikeReducer.js*  const initialState = {    bikes: [],    status: 'idle',  *// idle | loading | succeeded | failed*    error: null,  };  *// Reducer xử lý các action*  const bikeReducer = (state = initialState, action) => {    switch (action.type) {      case 'bike/fetchBikes/pending':        return {          ...state,          status: 'loading',        };      case 'bike/fetchBikes/fulfilled':        return {          ...state,          status: 'succeeded',          bikes: action.payload,        };      case 'bike/fetchBikes/rejected':        return {          ...state,          status: 'failed',          error: action.error,        };      case 'bike/addBike/pending':        return {          ...state,          status: 'loading',        };      case 'bike/addBike/fulfilled':        return {          ...state,          status: 'succeeded',        };      case 'bike/addBike/rejected':        return {          ...state,          status: 'failed',          error: action.error,        };      default:        return state;    }  };  export default bikeReducer; |

### B3:Code store

|  |
| --- |
| import { createStore, applyMiddleware } from 'redux';  import thunk from 'redux-thunk';  import bikeReducer from './reducers/bikeReducer';  const store = createStore(bikeReducer, applyMiddleware(thunk));  export default store; |

# **TỔNG HỢP**

## **Redux**

* **Redux** là một thư viện quản lý state (trạng thái) cho ứng dụng. Nó lưu trữ trạng thái ở một nơi trung tâm gọi là "store", giúp dữ liệu của bạn nhất quán và dễ kiểm soát, đặc biệt khi bạn có nhiều components chia sẻ cùng một trạng thái.
* **Khi nào nên dùng:** Khi ứng dụng của bạn có nhiều thành phần cần chia sẻ dữ liệu, hoặc khi cần quản lý trạng thái phức tạp.

## **Redux Toolkit**

* **Redux Toolkit** là bộ công cụ chính thức từ Redux để đơn giản hóa việc viết mã Redux. Nó cung cấp các tiện ích như createSlice, configureStore, và createAsyncThunk giúp giảm boilerplate code.
* **Khi nào nên dùng:** Khi bạn sử dụng Redux và muốn giảm thiểu mã code phức tạp, đồng thời tích hợp các công cụ mạnh mẽ như Thunk hoặc Saga.

## **Redux Saga**

* **Redux Saga** là một middleware (lớp trung gian) cho Redux, cho phép xử lý side effects (hiệu ứng phụ) như gọi API không đồng bộ, sử dụng generator functions. Điều này giúp các tác vụ không đồng bộ dễ quản lý và dễ đọc hơn.
* **Khi nào nên dùng:** Khi ứng dụng của bạn có nhiều logic không đồng bộ và bạn muốn kiểm soát chặt chẽ các hiệu ứng phụ, đặc biệt khi có nhiều luồng dữ liệu song song.

## **Thunk**

* **Thunk** cho phép các action creators (hàm tạo action) trả về một hàm thay vì một đối tượng. Thunk rất hữu ích cho các thao tác không đồng bộ như gọi API.
* **Thunk** là một middleware của **Redux** và không thuộc riêng **Redux Toolkit**. Nhưng vì **Redux Toolkit** được thiết kế để đơn giản hóa Redux, nên nó tích hợp sẵn **Thunk**. Điều này có nghĩa là khi bạn tạo **store** bằng **Redux Toolkit**, **Thunk** đã được thêm vào mặc định, và bạn không cần cài đặt hoặc cấu hình thêm.
* **Khi nào nên dùng:** Khi bạn chỉ cần xử lý các thao tác không đồng bộ đơn giản, hoặc không cần các tính năng mạnh mẽ của Redux Saga.

## **Hooks**

* **Hooks** là một API được giới thiệu từ React 16.8, giúp bạn sử dụng state và lifecycle methods trong functional components mà không cần chuyển sang class components.
* Một số **hooks cơ bản**:
  + useState: Quản lý trạng thái.
  + useEffect: Xử lý các hiệu ứng như fetch dữ liệu, lắng nghe các sự kiện, hoặc đồng bộ hóa với DOM. Mặc dù useEffect không hỗ trợ async functions trực tiếp, bạn có thể sử dụng một async function bên trong nó.
  + useContext: Sử dụng dữ liệu từ Context API.
* **Khi nào nên dùng:** Dùng cho các chức năng phổ biến trong components, chẳng hạn như quản lý state và các hiệu ứng.

## **Custom Hooks**

* **Custom Hooks** cho phép bạn tái sử dụng logic trong nhiều components. Bạn có thể tạo ra một hook của riêng mình bằng cách kết hợp các hooks cơ bản.
* **Khi nào nên dùng:** Khi bạn có logic phức tạp và muốn tái sử dụng nó trong nhiều components khác nhau.

## **Async/Await**

* **Async/Await** là cú pháp trong JavaScript giúp viết mã không đồng bộ trông giống như mã đồng bộ. Bạn có thể sử dụng async/await để xử lý các tác vụ như fetch dữ liệu từ server.
* **Khi nào nên dùng:** Khi bạn gọi API hoặc thực hiện các tác vụ không đồng bộ khác và muốn viết mã dễ đọc hơn.

|  |  |  |
| --- | --- | --- |
| **Công cụ/Kỹ thuật** | **Khi nào nên sử dụng** | **Chức năng chính** |
| Hook (useState, useEffect, v.v.) | Khi bạn cần quản lý trạng thái hoặc xử lý các tác vụ đồng bộ trong một component cụ thể. | Quản lý trạng thái và xử lý hiệu ứng trong component chức năng. |
| Custom Hook | Khi bạn muốn tái sử dụng logic hoặc xử lý phức tạp trong nhiều component. | Đóng gói logic tái sử dụng (như gọi API, xử lý dữ liệu) để dùng lại trong nhiều component. |
| Redux | Khi ứng dụng của bạn phức tạp và có nhiều component cần dùng chung một loại dữ liệu. | Tạo một store trung tâm để quản lý trạng thái ứng dụng, giúp chia sẻ dữ liệu dễ dàng giữa các component. |
| Redux Toolkit | Khi bạn dùng Redux và muốn giảm mã cần viết, giúp setup Redux nhanh và dễ dàng hơn. | Tích hợp các công cụ mạnh mẽ như Thunk, đồng thời giúp tự động cấu hình store và viết action dễ dàng hơn. |
| Redux think   |  | | --- | |  | | Khi bạn cần gọi API hoặc xử lý các tác vụ không đồng bộ đơn giản, và không cần quy trình phức tạp. | Xử lý các thao tác không đồng bộ trong Redux bằng cách cho phép action creator trả về hàm (thay vì object). |
| Redux Saga | Khi bạn cần quản lý nhiều luồng dữ liệu không đồng bộ phức tạp hoặc thực hiện các tác vụ đồng bộ phức tạp. | Sử dụng generator function để quản lý các side effects, giúp cấu trúc các tác vụ không đồng bộ tốt hơn. |
| Context API | Khi bạn chỉ cần truyền dữ liệu đơn giản qua nhiều cấp component, nhưng không quá phức tạp. | Truyền dữ liệu toàn cục trong ứng dụng mà không cần sử dụng Redux. |
| Async/Await | Khi bạn gọi API hoặc cần xử lý các tác vụ không đồng bộ mà không muốn dùng middleware như Thunk hay Saga. | Viết mã không đồng bộ trông giống mã đồng bộ, giúp dễ đọc hơn trong các tác vụ như gọi API. |

**Tóm lại :**

-Hooks dùng để quản lý dữ liệu và hiệu ứng trong component đơn giản.

-Custom Hooks để tái sử dụng logic.

-Redux và Redux Toolkit để quản lý trạng thái khi ứng dụng phức tạp.

-Thunk và Saga để xử lý tác vụ không đồng bộ trong Redux (Thunk cho tác vụ đơn giản, Saga cho tác vụ phức tạp).

-Context API có thể thay thế Redux khi dữ liệu cần chia sẻ đơn giản.

## **Khi nào dùng Hook và khi nào dùng Redux?**

Hook:

-Khi bạn chỉ cần quản lý dữ liệu trong một component hoặc một số ít component.

-Ứng dụng đơn giản hoặc các dữ liệu không cần chia sẻ giữa nhiều component.

Redux:

-Khi ứng dụng lớn, phức tạp, có nhiều component cần dùng chung dữ liệu.

-Khi bạn cần quản lý các tác vụ không đồng bộ phức tạp (dùng thêm middleware như Thunk hoặc Saga).

## **Sự khác nhau giữa createAsyncThunk và redux thunk**

**Redux Thunk** là cách để bạn viết các hành động bất đồng bộ, còn **extraReducers** trong Redux Toolkit chỉ đơn giản là một cách để xử lý những hành động bất đồng bộ đó một cách dễ dàng và gọn gàng.

**createAsyncThunk** và **Redux Thunk** đều là công cụ dùng để xử lý các hành động bất đồng bộ trong Redux, nhưng chúng có một số sự khác biệt quan trọng trong cách sử dụng và cách tiếp cận.

**1. Redux Thunk**

**Redux Thunk** là một middleware của Redux cho phép bạn **dispatch các hàm (functions) thay vì chỉ dispatch các action objects**. Các hàm này có thể thực hiện các tác vụ bất đồng bộ như gọi API, sau đó dispatch các action sau khi hoàn thành. Redux Thunk yêu cầu bạn phải tự quản lý trạng thái pending, fulfilled, và rejected (hoặc tương tự) khi xử lý bất đồng bộ.

**Cách sử dụng Redux Thunk:**

* Bạn phải tự định nghĩa action types và các action creators.
* Trong action creator, bạn trả về một **hàm (function)** thay vì một **action object**.
* Bạn phải tự dispatch các action để cập nhật trạng thái (ví dụ: loading, success, failure).

**2. createAsyncThunk**

createAsyncThunk là một phần của Redux Toolkit, giúp bạn dễ dàng tạo ra các thunks bất đồng bộ mà không cần phải tự quản lý các action types, action creators, hay trạng thái như với Redux Thunk truyền thống. createAsyncThunk sẽ tự động tạo ra ba action type cho bạn: pending, fulfilled, và rejected. Bạn chỉ cần cung cấp logic bất đồng bộ (thường là API call) và Redux Toolkit sẽ lo việc dispatch các action này cho bạn.

**Cách sử dụng createAsyncThunk:**

* Bạn chỉ cần định nghĩa một hành động bất đồng bộ mà không cần phải tự quản lý action types.
* createAsyncThunk sẽ tự động xử lý trạng thái pending, fulfilled, và rejected khi bạn gọi API.
* Bạn chỉ cần cung cấp async function và payload sẽ được tự động cập nhật cho bạn.

|  |  |  |
| --- | --- | --- |
| Tiêu chí | Redux Thunk | createAsyncThunk (Redux Toolkit) |
| Quản lý Action Types | Bạn phải tự quản lý pending, fulfilled, rejected. | Tự động tạo ra các action types như pending, fulfilled, rejected. |
| Cấu trúc Action Creator | Bạn viết hàm action creator với logic bất đồng bộ và dispatch các action. | Chỉ cần cung cấp một async function, createAsyncThunk sẽ tự tạo action creators. |
| Code Setup | Phải viết thêm logic cho việc dispatch các trạng thái (loading, success, failure). | Tối giản hơn, chỉ cần cung cấp logic bất đồng bộ, Redux Toolkit lo phần còn lại. |
| Sử dụng với Redux Toolkit | Không cần, Redux Thunk là middleware riêng biệt. | Được tích hợp sẵn trong Redux Toolkit. |
| Đơn giản và dễ sử dụng | Có thể phức tạp hơn khi bạn cần phải xử lý nhiều trạng thái bất đồng bộ. | Cung cấp một cách tiếp cận rõ ràng và đơn giản cho các tác vụ bất đồng bộ. |

‘

**Khi nào nên sử dụng cái nào?**

* Sử dụng createAsyncThunk khi bạn dùng Redux Toolkit: Nếu bạn đang sử dụng Redux Toolkit trong dự án của mình, createAsyncThunk là lựa chọn tốt hơn vì nó đơn giản và tiết kiệm thời gian. Bạn chỉ cần cung cấp hàm bất đồng bộ và Redux Toolkit sẽ tự động tạo ra các hành động cần thiết và cập nhật state của bạn.
* Sử dụng Redux Thunk khi không sử dụng Redux Toolkit: Nếu bạn không sử dụng Redux Toolkit (ví dụ, bạn đang sử dụng Redux truyền thống), bạn sẽ phải sử dụng Redux Thunk. Trong trường hợp này, bạn sẽ cần tự quản lý các action types và trạng thái loading, success, failure.

**Kết luận**

* createAsyncThunk giúp bạn giảm bớt mã nguồn và tự động hóa việc xử lý các trạng thái bất đồng bộ. Đặc biệt là khi bạn đang làm việc với Redux Toolkit.
* Redux Thunk cho phép bạn kiểm soát chi tiết hơn nhưng yêu cầu bạn phải viết nhiều mã hơn để xử lý trạng thái và action types.

Vì vậy, nếu bạn sử dụng Redux Toolkit, createAsyncThunk là cách nhanh chóng và dễ dàng nhất để xử lý các hành động bất đồng bộ.