p

Contents

[I. ReactJS 3](#_Toc117801001)

[1. JSX 3](#_Toc117801002)

[a) Tại sao lại nên dùng JSX 4](#_Toc117801003)

[2. Components 5](#_Toc117801004)

[3. Lifecycle 6](#_Toc117801005)

[a) Mounting 6](#_Toc117801006)

[b) Updating 6](#_Toc117801007)

[c) Unmounting 6](#_Toc117801008)

[4. Props và State 7](#_Toc117801009)

[a) Props là gì 7](#_Toc117801010)

[b) State là gì 7](#_Toc117801011)

[5. Hook 8](#_Toc117801012)

[a) useState 8](#_Toc117801013)

[b) useEffect 8](#_Toc117801014)

[c) useMemo 10](#_Toc117801015)

[d) useCallback 10](#_Toc117801016)

[e) useContext 11](#_Toc117801017)

[f) Các hook khác và custom hook 12](#_Toc117801018)

[II. Redux 14](#_Toc117801019)

[1. Action 15](#_Toc117801020)

[2. Reducers 16](#_Toc117801021)

[3. Store 17](#_Toc117801022)

[4. Sử dụng Redux 17](#_Toc117801023)

[5. Redux-toolkit 17](#_Toc117801024)

[a) configureStore 17](#_Toc117801025)

[b) createSlice 18](#_Toc117801026)

[III. Typecript 19](#_Toc117801027)

[1. Ưu điểm của Typescript: 19](#_Toc117801028)

[2. Basic Types: 19](#_Toc117801029)

[3. Function 21](#_Toc117801030)

[4. Type aliases 22](#_Toc117801031)

[5. Interface 22](#_Toc117801032)

[6. Types vs Interfaces 23](#_Toc117801033)

[7. Generics 25](#_Toc117801034)

[IV. Antd 30](#_Toc117801035)

[1. Antd là gì 30](#_Toc117801036)

[2. Cách sử dụng 31](#_Toc117801037)

# ReactJS

ReactJS là một opensource được phát triển bởi Facebook, ra mắt vào năm 2013, bản thân nó là một thư viện JavaScript chuyên giúp các nhà phát triển xây dựng giao diện người dùng hay UI, Nó được phân loại thành kiểu “V” trong mô hình **MVC** (Model-View-Controller). Một trong những điểm nổi bật nhất của ReactJS đó là việc render dữ liệu không chỉ thực hiện được trên tầng Server mà còn ở dưới Client nữa.

React được sử dụng bởi hàng trăm công ty lớn trên thế giới, bao gồm Netflix, Airbnb, American Express, Facebook, WhatsApp, eBay, và Instagram.

Hơn nữa, để tăng tốc quá trình phát triển và giảm thiểu những rủi ro có thể xảy ra trong khi coding, React còn cung cấp cho chúng ta khả năng Reusable Code (tái sử dụng code)

## JSX

JSX (nói ngắn gọn cho JavaScript extension)  là một cú pháp mở rộng cho JavaScript. Nó giúp chúng ta dễ dàng thay đổi cây DOM bằng các HTML-style code đơn giản.

JSX = Javascript + XML. Nó transform cú pháp dạng gần như XML về thành Javascript. Giúp người lập trình có thể code ReactJS bằng cú pháp của XML thay vì sử dụng Javascript. Các XML elements, attributes và children được chuyển đổi thành các đối số truyền vào React.createElement.

XML là viết tắt của từ eXtensible Markup Language, hay còn gọi là ngôn ngữ đánh dấu mở rộng do W3C đề nghị với mục đích tạo ra các ngôn ngữ đánh dấu khác. Đây là một tập hợp con đơn giản có thể mô tả nhiều loại dữ liệu khác nhau nên rất hữu ích trong việc chia sẻ dữ liệu giữa các hệ thống. Điển hình nhất là ngôn ngữ đánh dấu siêu văn bản HTML sử dụng cú pháp của XML để tạo nên và nó có các bộ phần tử và thuộc tính không mềm dẻo nên chỉ có tác dụng trong việc trình bày dữ liệu trên trình duyệt Browser.

Cú pháp của tài liệu XML XML được xây dựng dựa vào cấu trúc NODE lồng nhau, mỗi node sẽ có một thẻ mở và một thẻ đóng như sau:

<nodename>nội dung</nodename>

Trong đó:

* <nodename> là thẻ mở, tên của thẻ này do bạn tự định nghĩa.
* </nodename> là thẻ đóng, tên của thẻ này phải trùng với tên của thẻ mở.
* content là nội dung của thẻ này

Cú pháp của JSX cũng tương tự như XML.

* Ta có thẻ mở tag:

<JSXElementName JSXAttributesopt>

* Đóng tag:

</JSXElementName>

* Ở đây lưu ý tên của thẻ mở tag và đống tag phải giống nhau. ví dụ :

<MyButton color="blue" shadowSize={2}>

Click Me

</MyButton>

* Ngoài ra JSX cũng có SelfClosingElement:

<JSXElementName JSXAttributes/>

* ví dụ:

<div className="sidebar" />

### Tại sao lại nên dùng JSX

Việc sử dụng JSX trong ReactJS là không bắt buộc. Bạn có thể sử dụng chỉ JS thuần thôi. Thay vì tách biệt các công nghệ một cách giả tạo bằng cách đưa định nghĩa giao diện và logic vào những tệp khác nhau, React [tách bạch mối quan hệ](https://en.wikipedia.org/wiki/Separation_of_concerns) bằng những đơn vị rời rạc gọi là “components” chứa cả hai cái trên. Nhưng có rất nhiều lý do cho việc nên sử dụng JSX trong ReactJS đấy.

* Thứ nhất, JSX với cú pháp gần giống XML, cấu trúc cây khi biểu thị các attributes, điều đó giúp ta dễ dàng định nghĩa, quản lý được các component phức tạp, thay vì việc phải định nghĩa và gọi ra nhiều hàm hoặc object trong javascript. Khi nhìn vào cấu trúc đó cũng dễ dàng đọc hiểu được ý nghĩa của các component. Code JSX ngắn hơn, dễ hiểu hơn code JS.
* Thứ 2, JSX không làm thay đổi ngữ nghĩa của Javascript
* Thứ 3, với cách viết gần với các thẻ HTML, nó giúp những developers thông thường (ví dụ như các designer) có thể hiểu được một cách dễ dàng, từ đó có thể viết hoặc sửa code mà không gặp nhiều khó khăn. Ví du với đoạn code JSX như sau:

const myelement = <h1>I Love JSX!</h1>;

ReactDOM.render(myelement, document.getElementById('root'));

* Không có JSX, đoạn code sẽ như sau:

const myelement = React.createElement('h1', {}, 'I do not use JSX!');

ReactDOM.render(myelement, document.getElementById('root'));

## Components

Components cho phép bạn chia UI thành các phần độc lập, có thể tái sử dụng, và hoàn toàn tách biệt nhau. Về mặt khái niệm, components cũng giống như các hàm Javascript. Chúng nhận vào bất kì đầu vào nào (còn được gọi là “props”) và trả về các React elements mô tả những gì sẽ xuất hiện trên màn hình.

Có 2 kiểu components 1 là function components 2 là class components nhưng chúng ta chỉ bàn về function components ở đây.

Cách đơn giản nhất để định nghĩa một component đó là viết một hàm JavaScript:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

Hàm này là một React component hợp lệ về nó nhận đầu vào là một tham số object “props” (viết tắt của properties) với dữ liệu và trả về một React element. Chúng ta gọi các components này là “function components” vì chúng là các hàm JavaScript theo đúng nghĩa đen.

Component có thể sử dụng nhưng cái HTML tag

function App() {

return (

<div>

<Welcome name="Sara" />

);

}

**Chú ý:** Luôn luôn bắt đầu tên của component bằng chữ in hoa.

React xử lí các components bắt đầu với chữ thường giống như các DOM tags. Ví dụ, <div /> biểu diễn HTML div tag, nhưng <Welcome /> biểu diễn một component và yêu cầu Welcome nằm trong scope.

Các components có thể tham chiếu đến các components khác tại đầu ra của chúng. Điều này cho phép chúng ta sử dụng cùng một component abstraction cho mọi mức độ chi tiết. Một button, form, dialog, màn hình: trong React apps, chúng đều được hiển thị như là các components.

Thông thường, các React apps mới tạo sẽ có một App component ở tầng cao nhất. Thế nhưng, nếu bạn tích hợp React vào ứng dụng hiện có, bạn có thể bắt đầu bằng cách tiếp cận bottom-up với một component nhỏ như là Button và dần dần đi lên các tầng trên cùng của cây kế thừa giao diện.

Đừng ngại ngần việc tách components thành các components nhỏ hơn. Chia các components ngay từ đầu là một công việc không đơn giản, nhưng bù lại chúng ta sẽ có được một tập hợp các components có thể tái sử dụng trong các ứng dụng lớn hơn khác. Một nguyên tắc quan trọng đó là nếu một phần UI của bạn được sử dụng lại nhiều lần

## Lifecycle

Lifecycle là vòng đời của 1 component react. Chúng ta có thể tương tác lifecycle của 1 component thông qua các giai đoạn của lifecycle:

Có 3 giai đoạn chính **Mounting**, **Updating**, và **Unmounting**.

### Mounting

Đây là giai đoạn đầu tiên của 1 component đó là khi component được khởi tạo

### **Updating**

Đaya là giai đoạn sau khi component được khởi tạo. mà được cập nhật mỗi khi state hoặc props thay đổi

### **Unmounting**

Đây là giai đoạn khi component được xóa khởi DOM

## Props và State

### Props là gì

Props là 1 từ viết ngắn gọn của properties. props là 1 đối tượng, nó lưu trữ các giá trị của các attribute (thuộc tính) của một thẻ (Tag).Là cách mà component có thể nhận được các giá trị của thuộc tính truyền vào từ bên ngoài vào, và là cách mà các component có thể nói chuyện với nhau.

Ví dụ :

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

const element = <Welcome name="ReactJS" />;

ReactDOM.render(

element,

document.getElementById('root')

);

Trong đó, name ở {props.name} là property, Welcome là Component. Mỗi property của Component sẽ tương ứng với 1 attribute của thẻ, giá trị của attribute sẽ được truyền vào property của Component.

* Chúng ta sử dụng props để truyền gửi dữ liệu đến các component.
* Mọi component được gọi là hàm javascript thuần khiết (pure function).
* props tương ứng với tham số của pure function javascript .
* Không thể thay đổi dữ liệu của props .

Dữ liệu của của React là một chiều có nghĩa là nó chỉ đi từ component cha xuống component con

### **State là gì**

Cũng giống như props , state cũng lưu trữ thông tin về component, nhưng là lưu trữ dữ liệu động của một component.

State là dữ liệu động , nó cho phép một component theo dõi thông tin thay đổi giữa các kết xuất (render) và làm cho nó có thể tương tác.

State chỉ có thể được sử dụng ở trong một component sinh ra nó.

Nếu dự đoán được một component cần quản lý state, thì nó nên được tạo ở trong một class component chứ không phải là một function component. Vì function components là stateless component có nghĩa là nó không có state để lưu trữ thông tin, thay vào đó nó sử dụng các hook để lưu giữ thông tin.

## Hook

Hooks chính thức được giới thiệu trong phiên bản React 16.8. Nó cho phép chúng ta sử dụng state và các tính năng khác của React mà không phải dùng đến Class

Điều này có nghĩa là từ phiên bản 16.8 trở đi, chúng ta đã có thể sử dụng state trong stateless (functional) component

Hook là 1 function có thể gắn (hook) vào 1 function component để giúp component có thể quản lý các state hay các tính năng vòng đời của 1 component

Hook không hoạt động được trong class component

### **useState**

Hàm useState nhận đầu vào là giá trị khởi tạo của 1 state và trả ra 1 mảng gồm có 2 phần tử, phần tử đầu tiên là state hiện tại, phần tử thứ 2 là 1 function dùng để update state. Ví dụ:

const [isLoading, setLoading] = useState(false);

onClick() {

setLoading(true)

}

Trong đó:

* isLoading là state lưu trữ giá trị hiện tại
* setLoading là hàm để thay đổi giá trị của isLoading
* false là giá trị khởi tạo của state

### **useEffect**

Hàm useEffect dùng để xử lý các side effects. Nó thay thế các hàm cũ trong lifecycle.

Hàm useEffect nhận đầu vào gồm 2 tham số:

Tham số 1 là 1 hàm sẽ được chạy khi các giá trị trong chuỗi ở hàm thứ 2 thay đổi. trong hàm này nếu bạn trả về 1 hàm thì hàm trả về này sẽ được chạy khi component được update

Tham số 2 là 1 chuỗi các biến hoặc hàm, khi các các giá trị này thay đổi hàm ở tham số 1 sẽ chạy

 Ví dụ:

import { callApi } from './actions'

const App = ({ callApi, data }) => {

const callApiWhenStart = () => {

callApi('some\_payload\_')

}

useEffect(callApiWhenStart,[])

return(

<div>

{data.map(item => {// do something })}

</div>

)

}

export default App

Trong đó callApiWhenStart là hàm sẽ chạy khi 1 trong các phần tử của chuỗi ở tham số 2 thay đổi, tuy nhiên vì tham số 2 ở đây là chuỗi rỗng nên hàm callApiWhenStart sẽ chỉ chạy 1 lần khi component được tạo lần đầu tiên

Ở 1 ví dụ khác:

useEffect(

() => {

const subscription = props.source.subscribe();

return () => {

subscription.unsubscribe();

};

},

[props.source], // giá trị được subcrive

);

Trong đó mỗi khi props.source thay đổi giá trị thì hàm ở tham số 1 sẽ chạy lại và trước khi hàm ở tham số 1 sẽ chạy lại hàm return sẽ chạy để dọn dẹp những dữ liệu của hàm ở tham số 1 để tránh tốn bộ nhớ

### useMemo

useMemo giúp ta kiểm soát việc được render dư thừa của các component con bằng cách truyền vào 1 tham số thứ 2 thì chỉ khi tham số này thay đổi thì thằng useMemo mới được thực thi. Ví dụ:

const NotUsingMemo = ({ products }) => {

const soldoutProducts = products.filter(x => x.isSoldout === true); };

const UsingMemo = ({ products }) => {

const soldoutProducts = useMemo(

() => products.filter(x => x.isSoldout === true), [products]

);

};

Trong đó ở component NotUsingMemo mỗi khi component được render thì soldoutProducts sẽ được thực khi lại. Điều này dẫn đến việc phải tính toán lại cả những trường hợp đầu vào và đầu ra không thay đổi. Nếu việc tính toán là 1 hàm nặng tốn nhiều thời gian vậy sẽ gay ảnh hưởng đến trải nghiệp người dùng

Trong khi đó ở component UsingMemo soldoutProducts sẽ chỉ được tính toán lại nếu như products (tham số truyền vào ở tham số thứ 2 của hàm useMemo) thay đổi. Nó giúp ta loại bỏ được các trường hợp đầu vào không thay đổi, giúp giảm tải việc tính toán

### useCallback

useCallback có nhiệm vụ tương tự như useMemo nhưng khác ở chỗ function truyền vào useMemo bắt buộc phải ở trong quá trình render trong khi đối với useCallback đó lại là function callback của 1 event nào đó như là onClick chẳng hạn.

### useContext

Như đã nói ở trên dòng dữ liệu trong react chỉ đi 1 chiều từ cha xuống con vậy làm sao để truyền props từ con lên cha hoặc giữa các component con với nhau. Để giải quyết được vấn đề này, bạn có thể tạo một context để thêm những shared state và sau đó component nào cần thì bạn gọi context đó ra xài, không cần phải truyền từ cha xuống con nữa. Nó cũng giống như biến global được chia sẻ và sử dụng ở nhiều nơi, nhưng context được quản lí tốt hơn để maintain code dễ hơn.

Ví dụ sử dụng context:

import React from 'react';

const ExampleContext = React.createContext();

const App = () => {

return (

<ExampleContext.Provider value={{ color: 'red' }}>

<div className='App'>

<ChildComponent />

</div>

</ExampleContext.Provider>

);

};

const ChildComponent = () => {

const { color } = React.useContext(ExampleContext);

return <p style={{ color }}>This text is {color}</p>;

};

export default App;

Đầu tiên chúng ta sẽ định nghĩa 1 context:

const ExampleContext = React.createContext();

Sau đó bao bọc toàn bộ thành phần DOM của component bằng thẻ Provider, đồng thời truyền giá trị mà mình muốn chia sẻ đến các component khác

<ExampleContext.Provider value={{ color: 'red' }}>

<div className="App">

<ChildComponent />

</div>

</ExampleContext.Provider>

Như vậy là chúng ta đã có thể sử dụng context đó trong các component con thông qua useContext:

const ChildComponent = () => {

const { color } = React.useContext(ExampleContext);

return <p style={{ color }}>This text is {color}</p>;

};

Ngoài ra value của hàm Provider còn có thể chưa các function

### Các hook khác và custom hook

Ngoài ra chúng ta còn rất nhiều cái hook khác được định nghĩa sẳn bởi React mà các bạn có thể đọc thêm ở trên trang chủ của React hoặc chúng ta có thể tự viết hook cho mình trong trường hợp cái logic đó lập lại ở các component khác nhau. Các hàm hook luôn được bắt đầu với use (vd: useFetch). Ví dụ:

import { useState, useEffect } from "react";

const useFetch = (url) => {

const [data, setData] = useState(null);

useEffect(() => {

fetch(url)

.then((res) => res.json())

.then((data) => setData(data));

}, [url]);

return [data];

};

export default useFetch;

Trong ví dụ trên ta tạo sử dụng hook useFetch để lấy dữ liệu từ data. Hàm useFetch nhận vào url và trả về data và sẽ gọi api mỗi khi url thay đổi (ứng dụng useEffect đã giới thiệu ở trên). Sau đó chúng ta có thể sử dụng useFetch như các hook khác

import useFetch from "./useFetch";

const Home = () => {

const [data] = useFetch("https://jsonplaceholder.typicode.com/todos");

return (

<>

{data &&

data.map((item) => {

return <p key={item.id}>{item.title}</p>;

})}

</>

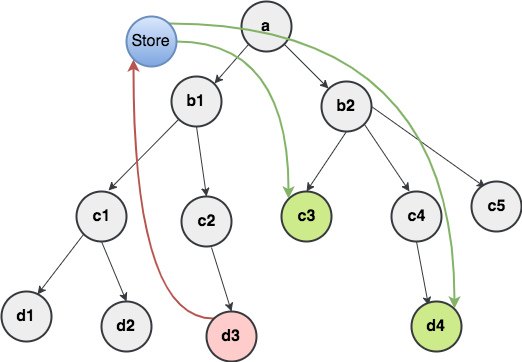
);

};

# Redux

Redux là 1 thư viện Javascript để quản lý state của ứng dụng, thường được sử dụng với javascript framework như React.

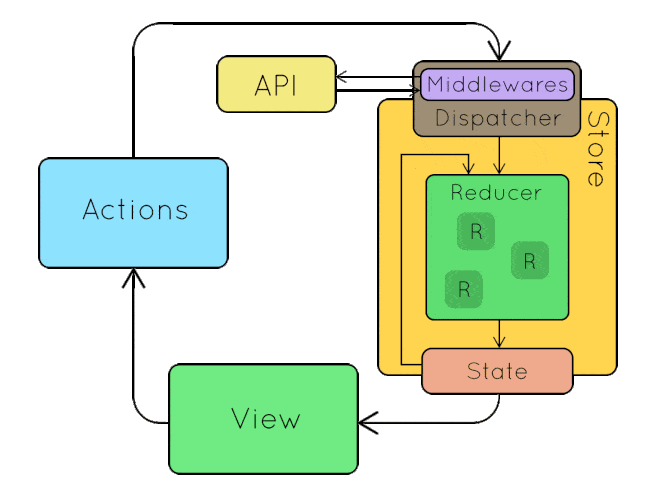
Như đã đề cập bên trên dữ liệu của react chỉ truyền 1 chiều từ cha cho con. Việc truyền dữ liệu giữa con lên cha hoặc giữ các component đồng cấp rất phức tạp. Chúng ta có thể dùng useContext đã đề cập ở trên tuy nhiên useContext sẽ render lại tất cả các component nằm trong provider khi value thay đổi điều này ổn với các ứng dụng nhỏ tuy nhiên với các ứng dụng lớn việc này sẽ giảm hiện xuất của trang web. Do đó redux là 1 giải pháp quản lý state được nhiều người sử dụng khi sử dụng React. Trái với useContext, khi sử dụng Redux các component chỉ render lại những component thay đổi props hoặc state tuy nhiên việc setup Redux đòi hỏi nhiều công sức hơn là useContext.



Hình 1 Mô hình hoạt động của Redux

Cái cách mà Redux hoạt động là khá đơn giản. Nó có 1 store lưu trữ toàn bộ state của app. Mỗi component có thể access trực tiếp đến state được lưu trữ thay vì phải truyền props từ component này đến component khác.

Cơ chế hoạt động của nó được tóm gọn trong 1 sơ đồ đơn giản:



Hình 2 Sơ đồ hoạc động của Redux

Khi người dùng tương tác với UI sẽ tạo ra 1 action action này sẽ được dispatcher gửi đến reducer (các action bất đồng bộ ví dụ như gọi api sẽ được sử lý bởi các middleware tại bước này). Reducer sẽ nhận thông tin từ action và state hiện tại và sử lý sao đó trả về một state mới. Khi state thay đổi Ui cũng sẽ cập nhật theo.

Thành phần chính của Redux gồm: Actions, Store, Reducers.

## Action

Actions đơn giản là các events. Chúng là cách mà chúng ta send data từ app đến Redux store. Những data này có thể là từ sự tương tác của user vs app, API calls hoặc cũng có thể là từ form submission.

Actions được gửi bằng cách sử dụng store.dispatch() method, chúng phải có một type property để biểu lộ loại action để thực hiện. Chúng cũng phải có một payload chứa thông tin. Actions được tạo thông qua một action creator. Ví dụ:

const setLoginStatus = (name, password) => {

return {

type: "LOGIN",

payload: {

username: "foo",

password: "bar"

}

}

}

Trong đó setLoginStatus là tên của action (function) sẽ được gọi trong hàm dispatch. setLoginStatus trả về 1 object gồm type để reducer nhận biết phải thực hiện hành động nào và payload để truyền data xuống reducer

## **Reducers**

Reducers là các function nguyên thủy chúng lấy state hiện tại của app, thực hiện một action và trả về một state mới. Những states này được lưu như những objects và chúng định rõ cách state của một ứng dụng thay đổi trong việc phản hồi một action được gửi đến store. 1 reducer tương đương với 1 state nhưng kèm theo các mô tả state sẽ thay đổi như thế nào khi các action khác nhau được gọi. Ví dụ:

export default function counterApp (state = initialState, action) {

switch (action.type) {

case INCREASE:

return {

increase: ++state.increase,

decrease: state.decrease

}

case DECREASE:

return {

increase: state.increase,

decrease: ++state.decrease

}

default:

return state

}

}

Trong đó counterApp là tên của reducer. Trong 1 ứng dụng có thể có nhiều reducer. Mỗi reducer có 1 tên khác nhau. Reducer nhận vào 2 tham số là state với giá trị default là initialState (giá trị khởi tạo của state) và 1 action

Reducer sẽ dựa theo action type để thay đổi state với payload (nếu có) và trả về state mới. Lưu ý state mới phải là object mới nếu không Redux sẽ không thể cập nhật lại UI

## Store

Store là nơi lưu trạng thái của ứng dụng và chỉ có duy nhất một Store trong bất kỳ một chương trình ứng dụng Redux nào. Nhiệm vụ của store chính là quản lý, access các state được lưu, update state thông qua dispatch, cho phép truy cập state thông qua Getstate và đăng ký hoặc hủy đăng ký các listeners thông qua helper methods.

## Sử dụng Redux

Sử dụng useSelector của react-redux để lấy state counter từ store.

Sử dụng useDispatch để trả về function dispatch, truyền increment và decrement vào dispatch để gọi 2 action này

## **Redux-toolkit**

Như đã trình bày ở trên việc setup redux khá phức tạp và tốn thời gian. Redux-toolkit là một package được sinh ra nhằm giải quyết phần lớn những vấn đề kể trên, được phát triển bởi chính chủ reduxjs team, giúp chúng ta viết code redux nhanh gọn, hoàn chỉnh theo một quy chuẩn thống nhất.

### configureStore

Hàm configureStore là 1 hàm của redux-toolkit giúp chung ta đơn giản việc setup store. Như ví dụ ở trên, chúng ta khởi tạo một store bằng hàm createStore của redux với tham số nhận vào là một reducer, configureStore sẽ làm điều tương tự như vậy, chúng ta sẽ khởi tạo store theo phương pháp sau

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

const store = configureStore({

reducer: rootReducer

})

Về cú pháp thì không khác nhau nhiều. Tuy nhiên thay vì chỉ khởi tạo một store đơn thuần, configureStore sẽ mặc định thiết lập cho phép sử dụng redux devtool để debug và theo dõi quá trình cập nhật state cũng như thiết lập sẵn một số middleware giúp thực hiện các action bất đồng bộ.

### createSlice

createSlide là hàm của redux-toolkit giúp chúng ta định nghĩa 1 reducer và các action liên quan đến nó. Ví dụ:

const initialState = {  
 value: 0,  
}  
  
export const counterSlice = createSlice({  
 name: 'counter',  
 initialState,  
 reducers: {  
 increment: (state) => {  
 state.value += 1  
 },  
 decrement: (state) => {  
 state.value -= 1  
 },  
 incrementByAmount: (state, action) => {  
 state.value += action.payload  
 },  
 },  
})  
  
export const { increment, decrement, incrementByAmount } = counterSlice.actions  
  
export default counterSlice.reducer

createSlice nhận vào 1 object gồm:

* name: Tên của reducer
* initialState: state khởi tạo của reducer
* reducer: là 1 object định nghĩa các hàm quy định reducer sẽ cập nhật state như thế nào. Bên cạnh đó như đã nhắc ở trên reducer phải trả về 1 object mới tuy nhiên redux-toolkit đã đơn giản hóa việc đó chúng ta có thể thực hiện tính toán ngay trên object cũ

createSlide trả về 1 object chứa actions và reducer. Chúng ta có thể export các action và reducer để sử dụng. Như vậy chỉ 1 hàm đơn giản đã giúp chúng ta tiết kiệm rất nhiều thời gian bằng cách gọp bước tạo action và tạo reducer vào làm 1 .

# Typecript

TypeScript là một phiên bản cao hơn của JavaScript, được thiết kế để xây dựng các ứng dụng lớn và phức tạp.  Nó kế thừa nhiều khái niệm từ Java và C#, TypeScript là ngôn ngữ tĩnh (Static typed) có nghĩa là nó nghiêm ngặt và có trật tự trái ngược với free-type. Nó còn được bổ sung thêm lớp hướng đối tượng mà điều này không có ở Javascript. Với TypeScript, ta có thể bê nguyên xi code JavaScript vào trong cùng một file và chạy cùng nhau bình thường, bởi vì TypeScript duy trì cú pháp của JavaScript và mở rộng nó bằng một loạt tính năng mới. Nhờ đó mà hiệu năng làm việc được tăng lên đáng kể.

## **Ưu điểm của Typescript:**

* Dễ dàng hơn trong phát triển các dự án lớn, được hỗ trợ bởi các Javascript Framework lớn.
* Hầu hết các cú pháp hướng đối tượng đều được hỗ trợ bởi Typescript như kế thừa, đóng gói, constructor, abstract, interface, implement, override…v.v
* Cách tổ chức code rõ ràng hơn, hỗ trợ cơ chế giúp kiến trúc hệ thống code hướng module, hỗ trợ namespace, giúp xây dựng các hệ thống lớn nơi mà nhiều lập trình viên có thể làm việc cùng nhau một cách dễ dàng hơn.
* Hỗ trợ các tính năng mới nhất của Javascript. TypeScript luôn đảm bảo việc sử dụng đầy đủ các kỹ thuật mới nhất của Javascript, ví dụ như version hiện tại là ECMAScript 2015 (ES6).
* Một lợi thế của Typescript nữa là mã nguồn mở vì vậy nó miễn phí và có cộng đồng hỗ trợ rất lớn.
* Với static typing, code viết bằng TypeScript dễ dự đoán hơn, và dễ debug hơn.

## Basic Types:

Trong TypeScript chia ra làm 7 loại cơ bản, bao gồm: boolean, number, string, array, enum, any, void.

* Boolean:

var isDone: boolean = true;

* String:

var name: boolean = "nguyen thi A";

* Number:

var height: number = 8;

* Array : có 2 kiểu khai báo tương đương với nhau trong TypeScript

1: var list: boolean[] = [true, false];

2: var isDone: Array<boolean> = [true, false];

* Enum: khi khai báo enum một cách thông thường các phần tử sẽ được đánh số từ 0 và tăng dần

enum Color{Red, Green, Blue};

var c: Color = Color.Green

var colorName = Color[1] // kết quả sẽ là Green

* Khi muốn phần tử đầu tiên là 1 chứ không phải là 0 như mặc định thì cần khai báo như sau:

enum Color{Red = 1, Green, Blue};

var c: Color = Color.Green

var colorName = Color[1] // kết quả sẽ là Red

* Any: Any là một kiểu mà bạn có thể gán bất kỳ kiểu nào cho nó.

var notSure: any = 4;

notSure = "maybe a string instead";

notSure = false; // khai báo này hoàn toàn được chấp nhận.

// nếu notSure ban đầu khai báo và number thì

// tại đây chắc chắn sẽ có lỗi

var list:any[] = [1, true, "free"]; // nếu sử dụng var list:number[] thì

// tất cả các phần tử trong list sẽ phải là kiểu number

list[1] = 100;

* Void: Cũng giống như any nhưng void được sử dụng là đầu ra của hàm.

function warnUser(): void {

alert("This is my warning message");

}

## Function

TypeScript có khai báo function giống như javaScript. Nhưng khi khai báo function TypeScript còn hỗ chợ việc khai báo với các kiểu trả ra của function và cũng như kiểu đầu vào của dữ liệu. Không những thế khi sử dụng TypeScript ta có thể khai báo giá trị mặc định của đầu vào ngay khi khai báo function, điều mà JavaScript không có. Không dừng lại ở đó typeScript còn hỗ chợ việc bỏ qua nhập một hoặc vài đầu vào.

Ví dụ:

function buildName(firstName: string, lastName?: string):string {

if (lastName)

return firstName + " " + lastName;

else

return firstName;

}

Trong đó

* “firstName: string”: Khai báo kiểu dữ liệu string cho firstName
* “lastName?: string”: Khai báo kiểu dữ liệu string lastName tuy nhiên lastName có thể là undefined
* “:string”: là kiểu trả về của function

## Type aliases

Trong TypeScript, có rất nhiều types cơ bản như là number, string,... Ngoài ra, trong TypeScript chúng ta có các types nâng cao và các types nâng cao này được gọi là type aliases. Với type aliases, chúng ta có thể tạo tên mới cho một type nhưng chúng ta không define một type mới. Từ khóa type trong TypeScript là một cách cung cấp type aliases cho các variables, objects và functions của chúng ta. Các type aliases này mô tả dữ liệu của chúng ta trông như thế nào. Để mô tả loại dữ liệu của mình trông như thế nào chúng ta sử dụng các type cơ bản (string, number,...) hoặc bằng cách tạo tạo các type tùy chỉnh của chúng ta.

Ví dụ:

// type của Year là một number

type Year = number;

// Biến currentYear là thuộc type Year và phải là number

let currentYear: Year = 2021;

// custom Person object type

type Person = {

name: String;

gender: String;

};

## Interface

Interface trong typescript cho phép bạn định nghĩ thuộc tính là gì và phương thức là gì mà đối tượng cần để được implement. Nếu đối tượng tuân thủ đúng khuôn mẫu interface thì đối tượng đã implement interface ấy sẽ được thực thi đúng. Nếu interface không được thực thi đúng thì typescript sẽ phát sinh lỗi ngay lập tức.

interface Person {

name: string

age: number

}

const viblo: Person = {

name: "Viblo",

age: 31

};

## Types vs Interfaces

* Có thể merge interfaces, types thì không

interface Person {

name: string

}

interface Person {

age: number

}

const viblo: Person = {

name: "Viblo",

age: 31

}

* Type aliases có thể sử dụng computed properties:

Từ khóa in có thể được sử dụng để iterate tất cả các item bên trong một tập hợp keys. Chúng ta có thể sử dụng tính năng này để tạo mapped types.

Ví dụ sử dụng type aliases

type keys = "firstname" | "lastname"

type Person = {

[key in keys]: string

}

const viblo: Person = {

firstname: "Viblo",

lastname: "blog"

}

* Extend và implements:

Trong TypeScript, chúng ta dễ dàng extends và implements interfaces. Nhưng không thể với types Với interface để kế thừa chúng ta sử dụng extends:

interface Animal {

name: string

}

interface Bear extends Animal {

honey: boolean

}

* Intersection

Intersection cho phép chúng ta kết hợp nhiều types thành một types duy nhất. Để tạo một intersection types chúng ta dùng &:

type Name = {

name: string

};

type Age = {

age: number

};

type Person = Name & Age;

Chúng ta cũng có thể làm điều này với interface:

interface Name {

name: “string”

};

interface Age {

age: number

};

type Person = Name & Age;

## Generics

Generic type trong TypeScript cho phép bạn viết các function, class và interface có thể tái sử dụng và tổng quát hóa. Trong hướng dẫn này, bạn sẽ tập trung vào việc tạo các hàm generic (generic function).

Giả sử ta gặp trường hợp chúng ta phải định nghĩa 1 function trả về ngẫu nhiên 1 phần tử trong 1 mảng. Nếu chúng ta biết mảng vào sẽ là mảng number thì chúng ta có function:

function getRandomNumberElement(items: number[]): number {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

Tuy nhiên sẽ ra sao nếu chuỗi vào có thể là mảng number hoặc là mảng string. Trong trường hợp này chúng ta có thể tạo thêm 1 hàm

function getRandomStringElement(items: string[]): string {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

Vậy nếu chuỗi vào có thể là bất cứ type nào thì sao? Chúng ta không thể tạo ra tất cả các hàm cho mỗi type được. Vậy phải làm sao đây? Bạn có thể dùng type any tuy nhiên nếu như vậy bạn sẽ không xác định được type trả về hay nói cách khác là nó không an toàn. Do đó typescript cho chúng ta 1 giải pháp cho nhưng trường hợp như vậy đó là generic type.

Chúng ta có thể định nghĩa 1 hàm generic như sau:

function getRandomElement<T>(items: T[]): T {

let randomIndex = Math.floor(Math.random() \* items.length);

return items[randomIndex];

}

Hàm này sử dụng biến kiểu T. Kiểu T cho phép bạn chỉ định kiểu dữ liệu tại thời điểm gọi hàm. Ngoài ra, hàm sử dụng biến kiểu T làm kiểu trả về của nó.

Hàm getRandomElement() là hàm generic bởi vì nó có thể làm việc với bất kỳ kiểu dữ liệu nào, ví dụ: chuỗi, số, đối tượng, ...

Theo quy ước, chúng ta sử dụng ký tự T làm kiểu dữ liệu của biến. Tuy nhiên, bạn có thể sử dụng các chữ khác như A, B C, ...

Sau đây là cách sử dụng hàm getRandomElement() với một mảng số:

let numbers = [1, 5, 7, 4, 2, 9];

let randomEle = getRandomElement<number>(numbers);

console.log(randomEle);

Ví dụ này chỉ định rõ kiểu number cho T khi gọi hàm getRandomElement().

Trong thực tế, bạn sẽ sử dụng kiểu suy luận cho đối số. Có nghĩa là bạn để trình biên dịch TypeScript tự xác định kiểu dữ liệu của T dựa trên loại đối số mà bạn truyền vào, như thế này:

let numbers = [1, 5, 7, 4, 2, 9];

let randomEle = getRandomElement(numbers);

console.log(randomEle);

Trong ví dụ này, chúng tôi đã không chỉ định kiểu number một cách rõ ràng khi gọi hàm getRandomElement(). Trình biên dịch sẽ xác định kiểu dữ liệu của T dựa vào kiểu dữ liệu của tham số truyền vào.

Bây giờ, hàm getRandomElement() cũng là kiểu an toàn. Ví dụ: nếu bạn gán giá trị trả về cho một biến kiểu chuỗi, bạn sẽ gặp lỗi:

let numbers = [1, 5, 7, 4, 2, 9];

let returnElem: string;

returnElem = getRandomElement(numbers); // compiler error

Hàm generic có nhiều kiểu dữ liệu trong TypeScript

Phần sau minh họa cách tạo một hàm generic với hai biến kiểu U và V:

function merge<U, V>(obj1: U, obj2: V) {

return {

...obj1,

...obj2

};

}

Hàm merge() kết hợp hai đối tượng với kiểu U và V. Nó kết hợp các thuộc tính của hai đối tượng thành một đối tượng duy nhất.

Kiểu suy luận suy ra giá trị trả về của hàm merge() là kiểu giao của U và V, là U & V.

Sau đây minh họa cách sử dụng hàm merge() để hợp nhất hai đối tượng:

let result = merge(

{ name: 'John' },

{ jobTitle: 'Frontend Developer' }

);

console.log(result);

Đầu ra:

{ name: 'John', jobTitle: 'Frontend Developer' }

Tuy nhiên trong ví dụ trên Hàm merge() mong đợi đầu vào là hai đối tượng. Tuy nhiên, nó không ngăn bạn truyền vào tham số như thế này:

let person = merge(

{ name: 'John' },

25

);

console.log(person);

Đầu ra:

{ name: 'John' }

Code vẫn chạy mà không có một lỗi nào, tuy nhiên đó không phải là điều chúng ta mong muốn. Chúng ta muốn đầu vào của hàm phải là 2 object vì vậy chúng ta có thể gàng buộc generic type là là type mở trọng của object với từ khóa extends.

function merge<U extends object, V extends object>(obj1: U, obj2: V) {

return {

...obj1,

...obj2

};

}

Điều này dẫn đến đoạn code dưới đây lỗi và đó là điều chúng ta mong muốn.

let person = merge(

{ name: 'John' },

25

);

TypeScript cho phép bạn khai báo một tham số có kiểu dữ liệu bị ràng buộc bởi kiểu dữ liệu của một tham số khác.

Hàm prop() sau chấp nhận hai tham số là một đối tượng và một tên thuộc tính. Nó trả về giá trị của thuộc tính.

function prop<T, K>(obj: T, key: K) {

return obj[key];

}

Trình biên dịch gặp lỗi sau:

Type 'K' cannot be used to index type 'T'.

Để khắc phục lỗi này, bạn thêm ràng buộc cho kiểu dữ liệu K để đảm bảo rằng nó là một khóa của kiểu dữ liệu T như sau:

function prop<T, K extends keyof T>(obj: T, key: K) {

return obj[key];

}

Nếu bạn truyền vào hàm prop một tên thuộc tính tồn tại trên obj, trình biên dịch sẽ không báo lỗi. Ví dụ:

let str = prop({ name: 'John' }, 'name');

console.log(str);

Đầu ra:

John

Tuy nhiên, nếu bạn truyền một khóa không tồn tại trên đối số đầu tiên, trình biên dịch sẽ phát ra lỗi:

let str = prop({ name: 'John' }, 'age');

Lỗi:

Argument of type '"age"' is not assignable to parameter of type '"name"'.

# Antd

## Antd là gì

Ant là tập hợp các components của React được xây dựng theo chuẩn thiết kế của Ant UED Team. Tương tự như chuẩn Material Design, Ant cung cấp hầu hết các component thông dụng trong ứng dụng web hiện đại, như Layout, Button, Icon, DatePicket, v.v…Bên cạnh đó Ant cũng có những component riêng thú vị, như LocaleProvider cho phép bạn thay đổi ngôn ngữ trên toàn ứng dụng.

Có thể coi Ant Design cho React là tập hợp của hầu hết các thư viện về React. Nó đáp ứng được hầu hết các yêu cầu của project của bạn mà ban không phải cài thêm bất cứ thư viện nào nữa. Dưới đây là danh sách các component mà nó cung cấp:

* General: Button, Icon
* Layout: Grid, Layout
* Navigation: Affix, Breadcrumb, Dropdown, Menu, Pagination, Steps
* Data Entry: AutoComplete, Checkbox, Cascader, DatePicker, Form, InputNumber, Input, Mention, Rate, Radio, Switch, Slider, Select, TreeSelect, Transfer, TimePicker, Upload
* Data Display: Avatar, Badge, Collapse, Carousel, Card, Calendar, List, Popover, Tree, Tooltip, Timeline, Tag, Tabs, Table
* Feedback: Alert, Drawer, Modal, Message, Notification, Progress, Popconfirm, Spin, Skeleton
* Other: Anchor, BackTop, Divider, LocaleProvider

## Cách sử dụng

Import css được antd sử dụng (chỉ cần import 1 lần duy nhất ở file index.js):

import 'antd/dist/antd.css';

Import component mà ta muốn sử dụng:

import { Button } from 'antd';

Sử dụng nhưng các component khác

<Button>Default Button</Button>

Đọc thêm về tất cả các component được thư viện cung cấp tại [trang chủ của antd](https://ant.design/components/overview/).