**React JS**

**1. Context API**

* React Context tồn tại để bạn không cần truyền dữ liệu một cách thủ công bằng việc sử dụng props ở tất cả các cấp của component. Context chia sử dữ liệu cho nhiều các component khác nhau. Việc truyền dữ liệu từ component cha xuống component con thông qua props là tương đối dài dòng và khó kiểm sóat so với việc sử dụng Context API. Bằng việc sử dụng Context API, chúng ta không còn cần phải truyền các dữ liệu muốn chia sẻ với nhau thông qua việc dùng props.
* Khi nào thì dùng Context API: Những thứ thuộc về context là dữ liệu được coi là global như thông tin người dùng, hay thông tin giỏ hàng... Vậy các lý do khác nhau cho việc sử dụng context:
  + Dữ liêụ là cần thiết ở nhiều nơi: dữ liệu cần được sử dụng bởi nhiều component như chủ đề, người dùng, giỏ hàng...
  + Truyền dữ liệu qua nhiều component: sử dụng context trong trường hợp này là tốt hơn khi bạn muốn chuyển 1 giá trị props thông qua nhiều component.
* Context API trong ReactJS:
  + React.createContext là hàm tạo một đối tượng Ngữ cảnh. Khi React hiển thị một thành phần đăng ký đối tượng Context này, nó sẽ đọc giá trị ngữ cảnh hiện tại từ kết quả khớp gần nhất Providerphía trên nó trong cây.
  + Context.Provider mọi đối tượng Context đều đi kèm với một thành phần Provider React cho phép các thành phần tiêu thụ đăng ký các thay đổi ngữ cảnh.
  + Class.contextType: Các contextType tài sản trên một lớp có thể được gán một đối tượng Context tạo ra bởi React.createContext(). Sử dụng thuộc tính này cho phép bạn sử dụng giá trị hiện tại gần nhất của loại Ngữ cảnh đó bằng cách sử dụng this.context. Bạn có thể tham chiếu điều này trong bất kỳ phương thức vòng đời nào bao gồm cả hàm kết xuất.
  + Context.Consumer: Một thành phần React đăng ký thay đổi ngữ cảnh. Sử dụng thành phần này cho phép bạn đăng ký một ngữ cảnh trong một thành phần chức năng .
  + Context.displayName: Đối tượng ngữ cảnh chấp nhận một thuộc tính displayNamechuỗi. React DevTools sử dụng chuỗi này để xác định những gì sẽ hiển thị cho ngữ cảnh.
* Xây dựng Context API bao gồm:
  + context: đối tượng context là một đối tượng lưu giữ giá trị context hiện tại và có thể được đăng ký.
  + provider: là một componet của React cung cấp giá trị, nó lấy từ đối tượng context.
  + consumer: là một component có thể sử dụng giá trị của provider và có thể hiển thị giá trị.
* Ví dụ:
  + ThemeContextProvider.js

import React, { Component, createContext } from 'react';

export const ThemeContext = createContext()

class ThemeContextProvider extends Component {

    constructor(props) {

        super(props);

        this.state = { theme: 'day' };

    }

    changeTheme = () => {

        const { theme } = this.state;

        if (theme === 'day') {

            this.setState({

                theme: 'night'

            })

        } else {

            this.setState({

                theme: 'day'

            })

        }

    }

    render() {

        const ctx = { theme: this.state.theme, changeTheme: this.changeTheme }

        return (

            <ThemeContext.Provider value={ctx}>

                {this.props.children}

            </ThemeContext.Provider>

        );

    }

}

export default ThemeContextProvider;

* + ThemeChange.js

import React, { Component } from 'react';

import { ThemeContext } from '../context/ThemeContextProvider';

class ThemeChange extends Component {

    static contextType = ThemeContext

    render() {

        const { theme, changeTheme } = this.context

        const color = theme === 'day' ? 'black' : 'red'

        return (

            <div style={{ height: 300, background: color }}>

                <button onClick={changeTheme}>Change Theme</button>

            </div>

        );

    }

}

export default ThemeChange;

* + App.js

import React, { Component } from 'react';

import ThemeContextProvider from './context/ThemeContextProvider';

import ThemeChange from './component/ThemeChange';

class App extends Component {

  render() {

    return (

      <ThemeContextProvider>

        <div>

          <ThemeChange></ThemeChange>

        </div>

      </ThemeContextProvider>

    );

  }

}

export default App;

**2. Hooks**

* React hook là gì?
  + Hooks là một bổ sung mới trong React 16.8.
  + Hooks là những hàm cho phép bạn “kết nối” React state và lifecycle vào các components sử dụng hàm.
  + Với Hooks bạn có thể sử dụng state và lifecycles mà không cần dùng ES6 Class
* React Hooks được sinh ra với mong muốn giải quyết những vấn đề:
  + “Wrapper hell” các component được lồng (nested) vào nhau nhiều tạo ra một DOM tree phức tạp.
  + Component quá lớn.
  + Sự rắc rối của Lifecycles trong class
* Lợi ích của hook
  + Khiến các component trở nên gọn nhẹ hơn
  + Giảm đáng kể số lượng code, dễ tiếp cận
  + Cho phép chúng ta sử dụng state ngay trong function component
* State Hook: 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 (giống như hàm setState cũ vậy). Ví dụ:
  + Ngày trước dùng Class thì viết như này:

constructor(props) {

    super(props);

    this.state = { isLoading: false }

 }

 onClick() {

     this.setState({

         isLoading: true,

     })

 }

* + Còn bây giờ thì chỉ cần viết ngắn gọn như này:

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

onClick() {

     setLoading(true)

 }

* useEffect: useEffect thêm khả năng để thực hiện side effects từ các components dạng hàm. Nó phục vụ cùng mục đích như componentDidMount, componentDidUpdate, và componentWillUnmount trong React classes, nhưng thống nhất lại trong một API duy nhất. Giống như useState, bạn có thể sử dụng nhiều hơn một effect trong một component. Ví dụ:

function FriendStatusWithCounter(props) {

    const [count, setCount] = useState(0);

    useEffect(() => {

      document.title = `Bạn đã bấm ${count} lần`;

    });

    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {

      ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

      return () => {

        ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

      };

    });

    function handleStatusChange(status) {

      setIsOnline(status.isOnline);

    }

* Quy tắc của Hooks, hooks là các hàm Javascript, nhưng nó bắt buộc thêm hai quy tắc: :
  + Chỉ gọi Hooks trên cùng. Không gọi Hooks bên trong vòng lặp, điều kiện, hoặc các hàm lồng nhau.
  + Chỉ gọi Hooks từ các React components dạng hàm. Không gọi Hooks từ hàm JavaScript bình thường.
* Additional Hooks: Thực tế khi sử dụng useState thì nó sẽ trả về 1 phiên bản đơn giản của useReducer, vậy nên chúng ta có thể coi useReducer như một phiên bản nâng cao hơn dùng để thay thế cho việc sử dụng useState. Nếu đã làm việc với React-Redux thì chắc hẳn bạn sẽ dễ dàng nhận ra flow quen thuộc này phải không nào. Giống như reducer trong Redux thì useReducer cũng nhận vào một reducer dạng (state, action) và trả ra một newState. Khi sử dụng chúng ta sẽ nhận được một cặp bao gồm current state và dispatch function. Ví dụ:

const initialState = {count: 0}

function reducer(state, action) {

  const [count, setCount] = useState(0);

  switch (action.type) {

    case 'INCREMENT':

      return setCount( count + 1);

    case 'DECREMENT':

      return setCount( count - 1);

    default:

      throw new Error();

  }

}

function Counter() {

  const [state, dispatch] = useReducer(reducer, initialState)

  return (

    <>

      <StyledLogo src={logo} count={count}/>

      <Count count={count} />

      <div style={{display: 'flex'}}>

        <button onClick={() => dispatch({type: 'DECREMENT'})}> - </button>

        <button onClick={() => dispatch({type: 'INCREMENT'})}> + </button>

      </div>

    </>

  )

* useMemo giúp ta kiểm soát việc được render dư thừa của các component con, nó khá giống với hàm shouldComponentUpdate trong LifeCycle. 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:
  + Không sử dụng useMemo:

const NotUsingMemo = ({ products }) => {

    const soldoutProducts = products.filter(x => x.isSoldout === true); // soldoutProducts sẽ luôn luôn thực thi mỗi khi NotUsingMemo được re-render

  };

* + Có sử dụng useMemo:

const UsingMemo = ({ products }) => {

    const soldoutProducts = useMemo(

      () => products.filter(x => x.isSoldout === true), // / soldoutProducts sẽ chỉ thực thi khi props products thay đổi

      [products] // watch products

    );

  };

* 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. Ví dụ:

const App = () => {

    const [text, setText] = React.useState('');

    return (

      <>

        <input type="text" value={text} onChange={e => setText(e.target.value)} />

        <Wrap />

      </>

    );

  };

  const Wrap = () => {

    const [isChecked, setIsChecked] = React.useState(false);

    const toggleChecked = useCallback(() => setIsChecked(!isChecked), [

      isChecked

    ]);

    return <Checkbox value={isChecked} onClick={toggleChecked} />;

  };

  const Checkbox = React.memo(({ value, onClick }) => {

    console.log('Checkbox is renderd!');

    return (

      <div style={{ cursor: 'pointer' }} onClick={onClick}>

        {value ? '☑' : '□'}

      </div>

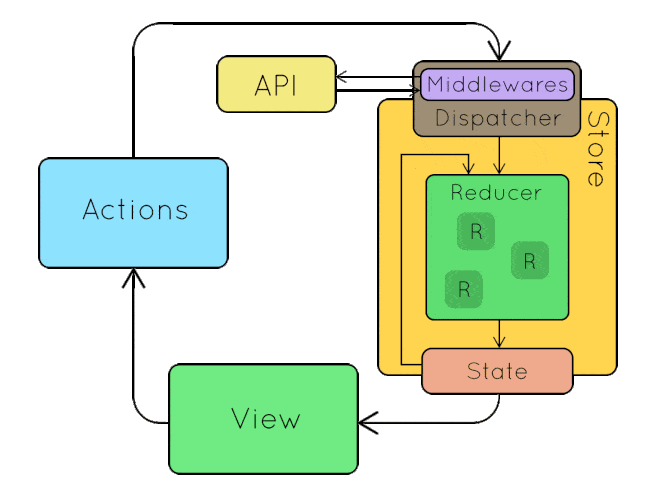
    );

  });

Trong ví dụ trên ta sử dụng useCallback cho sự kiện onClick, điều này có nghĩa là việc thay đổi giá trị text trong ô Input sẽ không làm component Checkbox bị re-render.

**3. Redux**

* Redux là một vùng chứa trạng thái có thể dự đoán được cho các ứng dụng JavaScript. Sử dụng kiến trúc uni-directional data flow
* Redux sẽ hoạt động dựa vào 3 thành phần cơ bản là: Actions, Reducers và Store:
  + Actions là các events và chúng là cách cần thiết mà bạn cần làm để send data từ app đến Redux store. Các data xuất hiện thông qua các tương tác của user hoặc qua app, API call hoặc từ form submission.
  + Reducers là những function dạng nguyên thủy và chúng thường lấy state hiện tại của app. Từ đó, thực hiện một action rồi trả về dưới dạng một state mới. Các states này sẽ được lưu trữ như objects và chúng sẽ định rõ các state của ứng dụng thay đổi khi phản hồi một action gửi đến store.
  + Store là loại lưu trạng thái ứng dụng và là duy nhất trong bất kỳ ứng dụng Redux nào. Người dùng có thể access các state đã được lưu, update hoặc đăng ký cũng như hủy đăng ký các listeners thông qua helper methods.
* Quá trình xây dựng Redux các chuyên gia đã dựa vào 3 nguyên lý cơ bản sau:
  + Sử dụng nguồn dữ liệu đáng tin cậy duy nhất. Các State của tất cả ứng dụng đều nằm trong một object tree nằm trong một Store duy nhất.
  + Chỉ được phép đọc trạng thái: Có nghĩa rằng, để thay đổi State của ứng dụng thì chỉ có cách duy nhất là phát một Action.
  + Sử dụng hàm thuần túy để thay đổi. Với mục đích chỉ ra cách State được biến đổi từ Action. Vì thế, chúng ta cần sử dụng các pure function gọi là Reducer.



* Cài đặt Redux: Sau khi khởi tạo môt dự án ReactJS, để có thể sử dụng Redux chúng ta cần phải cài đặt 2 module là redux và react-redux bằng cách sử dụng npm: npm install redux react-redux –save
* Redux Thunk là một Middleware có thể cho phép người dùng viết các Action trả về một function. Thay vì phải sử dụng một plain javascript object bằng cách trì hoãn quá trình đưa action đến reducer. Ngoài ra, Redux Thunk còn được sử dụng nhằm mục đích xử lý các logic bất đồng bộ phức tạp. Những đồng bộ này cần truy cập đến store hoặc lấy dữ liệu như Ajax request.
* Redux Persist là dạng gói tự động hóa cho quy trình duy trì trạng thái từ cửa hàng Redux của bạn đến với bộ nhớ của thiết bị cục bộ. Việc sử dụng Redux Persist sẽ giúp bạn thực hiện công việc hoàn toàn tự động mà chỉ cần lượng nhỏ bản ghi sẵn cho quá trình khởi tạo. Gói tự động này hoạt động rất hiệu quả và sở hữu nhiều bộ giảm được thiết kế vô cùng tốt. Ngoài ra, nó còn có thể giúp cho các bộ giảm bớt chi tiết khi cần thiết hoặc khi ứng dụng đang ngày càng phức tạp. Nhờ vậy mà quy trình quản lý cửa hàng Redux sẽ trở nên đơn giản mà hiệu quả hơn.