# Hook

# Hook

- Hook đã được thêm vào React trong phiên bản 16.8
- Hook cho phép các function component có quyền truy cập vào trạng thái và các tính năng React khác. Vì lý do này, các Class component thường không còn cần thiết nữa.

# Hook

1. useState
2. useEffect
3. useContext
4. useRef
5. useReducer
6. useCallback
7. useMemo

# useState

- **useState** là một hook trong React, cho phép quản lý trạng thái (state) trong các functional components.
- Trước khi có hooks, trạng thái chỉ có thể được quản lý trong các class components. Với useState, bạn có thể sử dụng state một cách đơn giản hơn mà không cần phải chuyển sang class components.

- Cú pháp cơ bản của useState:

```
const [state, setState] = useState(initialValue);
```

- **state**: Biến đại diện cho trạng thái hiện tại.
- **setState**: Hàm dùng để cập nhật giá trị mới cho trạng thái.
- **initialValue**: Giá trị khởi tạo cho trạng thái.

# useState

```jsx
import React, { useState } from 'react';

function Dem() {
  const [dem, setDem] = useState(0);

  return (
    <div>
      <p>Bạn đã bấm {dem} lần</p>
      <button onClick={() => setDem(count + 1)}>
        Bấm tôi
      </button>
    </div>
  );
}

export default Dem;
```

# useEffect

- **useEffect** là một hook trong React cho phép thực hiện các tác vụ phụ (side effects) trong các functional components, như gọi API, tương tác với DOM, hoặc đăng ký/huỷ đăng ký sự kiện.

# useEffect

- Trước khi có hook, các tác vụ này thường được thực hiện trong các phương thức vòng đời (lifecycle methods) của class components, như componentDidMount, componentDidUpdate, và componentWillUnmount.

# useEffect

```
useEffect(() => {
  // Tác vụ phụ (side effect) được thực hiện ở đây

  return () => {
    // Cleanup (dọn dẹp), nếu cần, sẽ được thực hiện ở đây
  };
}, [dependencies]);
```

- Trong đó:
  - Hàm đầu tiên là nơi thực hiện tác vụ phụ.
  - Hàm trả về (không bắt buộc) là nơi dọn dẹp trước khi component bị gỡ bỏ hoặc trước khi effect tiếp theo chạy.
  - Mảng dependencies (có thể rỗng) xác định khi nào effect sẽ chạy. Nếu một giá trị trong mảng thay đổi, effect sẽ được thực thi lại.

# useEffect

- Nếu dependencies không có thì useEffect sẽ chạy mỗi lần khi hàm được render lại.
- Nếu dependencies là một mảng rỗng thì useEffect sẽ chỉ chạy một lần duy nhất.
- Nếu dependencies có biến thì useEffect sẽ chạy lần đầu và chạy bất cứ khi nào biến thay đổi giá trị

# useEffect

```jsx
import React, { useState, useEffect } from "react"
function App() {

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

  useEffect(() => {
    console.log(`Component render! Count: ${count}`);
  });

  return (
    <div>
      <p>Giá trị đếm: {count}</p>
      <button onClick={() => setCount(count + 1)}>
        Tăng Count
      </button>
    </div>
  );
}
export default App
```

# useEffect

```
import React, { useState, useEffect } from "react"
function App() {

  const [data, setData] = useState([]);
  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));
  }, []); // Chỉ chạy một lần sau khi component được render

  return (
    <div>
      <ul>
        {data.map(item => (
          <li key={item.id}>{item.name}</li>
        ))}
      </ul>
    </div>
  );
}
export default App
```

# useContext

- useContext là một hook trong React, giúp truy cập vào giá trị của một context trong cây component mà không cần phải truyền props từ component cha xuống con qua nhiều cấp.

# useContext

- Context được dùng để chia sẻ dữ liệu giữa các component mà không phải truyền props qua nhiều cấp trung gian, rất hữu ích khi bạn có dữ liệu dùng chung như theme, thông tin người dùng, hoặc ngôn ngữ.

# useContext

```
const value = useContext(MyContext);
```

- MyContext: Đây là context đã tạo bằng React.createContext().
- value: Giá trị của context sẽ được truy cập trong component.

# useContext

- Bước 1: Tạo một context
- Tạo context bằng cách sử dụng React.createContext()

```
const ThemeContext = React.createContext('light');
```

- Bước 2: Cung cấp (provide) giá trị của context
- Sử dụng component ThemeContext.Provider để cung cấp giá trị của context cho các component con.

```
function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}
```

# useContext

- Bước 3: Sử dụng useContext trong các component con
- Các component con có thể truy cập vào giá trị của context mà không cần phải nhận nó qua props. Dưới đây là cách sử dụng useContext để lấy giá trị từ ThemeContext.

# useContext

```
function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext); // Truy cập giá trị của context
  return <button style={{ background: theme === 'dark' ? 'black' : 'white' }}>Nút có theme</button>;
}
```

# useContext

- useContext thường được sử dụng khi bạn có dữ liệu cần chia sẻ cho nhiều component mà không muốn truyền props qua nhiều cấp. Ví dụ:
  - Quản lý chủ đề (theme) của ứng dụng.
  - Thông tin người dùng đăng nhập.
  - Dữ liệu cấu hình hoặc ngôn ngữ trong ứng dụng đa ngôn ngữ.

# useRef

- useRef là một hook trong React, cho phép bạn tạo ra một đối tượng "tham chiếu" (ref) có thể giữ giá trị và không bị thay đổi qua các lần render của component.

# useRef

- Một trong những công dụng phổ biến nhất của useRef là truy cập trực tiếp vào các phần tử DOM, nhưng nó cũng có thể được sử dụng để giữ bất kỳ giá trị nào mà không cần component phải render lại khi giá trị đó thay đổi.

# useRef

```
const giaTHT = useRef(initialValue);
```

- initialValue: Giá trị khởi tạo cho ref. Giá trị này sẽ được lưu trong thuộc tính .current của đối tượng ref.
- giaTHT.current: Đây là nơi lưu trữ giá trị hiện tại của ref.

- Công dụng của useRef:
  - Truy cập trực tiếp vào DOM: có thể sử dụng useRef để tham chiếu tới một phần tử DOM và thực hiện các tác vụ như lấy giá trị của input, cuộn, hoặc focus.
  - Lưu giá trị không cần render lại: useRef giữ giá trị qua các lần render mà không gây render lại component khi giá trị thay đổi (khác với useState, khi thay đổi sẽ render lại component).

# useRef

```jsx
import React, { useState, useEffect, useRef } from "react"

function App() {

  const inputRef = useRef(null);

  useEffect(() => {
    // Focus vào input khi component được render lần đầu
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} placeholder="Focus vào đây khi render" />;
}

export default App
```

# useReducer

- useReducer là một hook trong React giúp quản lý state phức tạp bằng cách sử dụng một "reducer" (giống như trong Redux).

# useReducer

- Thay vì chỉ sử dụng useState cho việc quản lý trạng thái đơn giản, có thể sử dụng useReducer khi cần quản lý các logic cập nhật phức tạp hơn, đặc biệt khi có nhiều trạng thái liên quan hoặc các hành động liên tiếp cần xử lý.

# useReducer

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

- **reducer**: Là một hàm xác định cách trạng thái được cập nhật dựa trên hành động (action).
- **initialState**: Trạng thái khởi tạo.
- **state**: Trạng thái hiện tại (giống như giá trị từ useState).
- **dispatch**: Hàm dùng để gửi (dispatch) một hành động (action), yêu cầu reducer cập nhật state.

# useReducer

- **reducer** là một hàm nhận vào hai tham số: state hiện tại và action. Nó sẽ trả về state mới dựa trên hành động đã nhận.

```
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      return state;
  }
}
```

## Sử dụng useReducer khi

- Khi có logic cập nhật state phức tạp, đặc biệt là khi có nhiều trạng thái liên quan.
- Khi cần quản lý các state có nhiều bước hoặc điều kiện khác nhau.
- Khi muốn tổ chức mã nguồn gọn gàng hơn với các hành động rõ ràng và dễ hiểu.

# useReducer

```
import React, { useReducer } from 'react';

// Hàm reducer để cập nhật state dựa trên action
function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    case 'reset':
      return { count: 0 };
    default:
      return state;
  }
}
```

# useReducer

```jsx
function Counter() {
  // Khởi tạo state với giá trị { count: 0 }
  const [state, dispatch] = useReducer(reducer, { count: 0 });

  return (
    <div>
      <p>Giá trị đếm: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Tăng</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Giảm</button>
      <button onClick={() => dispatch({ type: 'reset' })}>Reset</button>
    </div>
  );
}

export default Counter;
```

# useCallback

- useCallback là một hook trong React giúp tối ưu hóa hiệu suất bằng cách ghi nhớ (memoize) một hàm. Nó trả về một phiên bản đã ghi nhớ của hàm mà chỉ thay đổi khi một trong các dependency (phụ thuộc) thay đổi. Điều này rất hữu ích khi bạn truyền các hàm vào component con hoặc sử dụng trong các hook như useEffect, giúp tránh việc tạo lại hàm không cần thiết sau mỗi lần render.

# useCallback

```
const memoizedCallback = useCallback(() => {
    // Hàm logic
  }, [dependencies]);
```

- Hàm logic: Là hàm mà muốn ghi nhớ.
- dependencies: Là mảng các giá trị phụ thuộc. Nếu một trong các giá trị này thay đổi, hàm sẽ được tạo lại; nếu không, React sẽ sử dụng lại phiên bản đã ghi nhớ trước đó.

# useCallback

```jsx
import React, { useState, useCallback } from 'react';

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

  // Sử dụng useCallback để ghi nhớ hàm tăng count
  const tang = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <ComponentCon onIncrement={tang} />
    </div>
  );
}
```

# useCallback

```
function ComponentCon({ skTang }) {

  return <button onClick={skTang}>Tăng Count</button>;
}

export default ComponentCha;
```

# useMemo

- useMemo là một hook trong React giúp tối ưu hóa hiệu suất bằng cách ghi nhớ (memoize) kết quả của một hàm tính toán phức tạp và chỉ tính lại khi các giá trị phụ thuộc (dependencies) thay đổi. Điều này giúp tránh việc thực hiện lại các tính toán tốn kém mỗi khi component re-render, nếu các giá trị đầu vào cho phép không thay đổi.

# useMemo

- Sử dụng useMemo khi
  - Khi có tính toán phức tạp hoặc tốn tài nguyên: Nếu bạn có logic tính toán trong component mà tốn nhiều thời gian hoặc tài nguyên, useMemo có thể giúp tối ưu hóa bằng cách chỉ thực hiện tính toán lại khi cần thiết.
  - Khi component render lại nhiều lần không cần thiết: Dùng useMemo để tránh việc tính toán lại các giá trị không thay đổi mỗi lần component render.

# useMemo

```
import React, { useState, useMemo } from 'react';

function SortedList({ list }) {
  const [filter, setFilter] = useState('');

  // Sử dụng useMemo để sắp xếp danh sách chỉ khi list thay đổi
  const sortedList = useMemo(() => {
    console.log('Sắp xếp danh sách...');
    return list.sort((a, b) => a.localeCompare(b));
  }, [list]);

  // Lọc danh sách dựa trên filter
  const filteredList = sortedList.filter((item) =>
    item.toLowerCase().includes(filter.toLowerCase())
  );
```

# useMemo

```
  return (
    <div>
      <input
        value={filter}
        onChange={(e) => setFilter(e.target.value)}
        placeholder="Lọc danh sách..."
      />
      <ul>
        {filteredList.map((item) => (
          <li key={item}>{item}</li>
        ))}
      </ul>
    </div>
  );
}

export default SortedList;
```