



# Lesson 8: Side effect with useEffect

🕒 Created March 22, 2022 9:55 AM

🏷 Tags Empty

🔗 Related to Untitle... Empty

💡 Hiểu rõ side effect là gì và cách xử lý chúng trong React

## 1. Side Effect là gì

Nhiệm vụ của các component và React chủ yếu là tạo ra giao diện người dùng. React đảm bảo việc thay đổi các dữ liệu trên màn hình dựa vào dữ liệu và các tương tác với ứng dụng từ phía người dùng.

Tuy nhiên có rất nhiều những tương tác từ ứng dụng web tới nhiều thành phần khác nhau. Một vài ví dụ có thể kể ra là:

- Xử lý HTTP request, response
- Tương tác với Local Storage, Session Storage.
- Xử lý timer ( `setTimeout` , `setInterval` )
- Thực hiện subscribes / unsubscribes các dịch vụ bên ngoài.
- Tương tác trực tiếp với DOM

Những logic được thực thi bên ngoài phạm vi của component đều được gọi là các "side effect". Các side effect có thể cập nhật lại giao diện ứng dụng ở một thời điểm khác nhau.

Xét một ví dụ đơn giản như sau:

1. Người dùng tiến hành đăng nhập. Khi click vào button "Login", tiến hành gửi thông tin user lên server.
2. Nếu người dùng nhập chính xác, tiến hành điều hướng về trang chủ.
3. Nếu người dùng nhập không chính xác, tiến hành hiển thị lỗi yêu cầu người dùng nhập lại.

Ở trong ví dụ trên, bước 1 là công việc của component. Tuy nhiên, bước 2 hoặc bước 3 sẽ được thực hiện dựa vào thông tin kết quả trả về. Đó có thể coi là các side effect, khi việc cập nhật giao diện ứng dụng sẽ phụ thuộc vào kết quả từ bên ngoài.

💡 Side effect là một khái niệm trong các ứng dụng nói chung, không chỉ riêng với React. Mỗi một công cụ sẽ có những cách khác nhau để xử lý các side effect. Vì vậy, việc hiểu rõ side effect là một điều quan trọng trong việc xây dựng các ứng dụng hiện đại.

## 2. Side effect cơ bản với `useEffect`

Để xử lý các logic bên ngoài component, React cung cấp một function hook đặc biệt là `useEffect` . Cú pháp cơ bản của `useEffect` như sau:

```
useEffect(() => { /* effect here */ })
```

Xét một ví dụ sau:

```
import {useState, useEffect} from 'react' const App = () => { const [count,
setCount] = useState(0) useEffect(() => { document.title = `You clicked
${count} times`; }) return ( <div> <p>You clicked {count} times</p> <button
onClick={() => setCount(count + 1)}> Click me </button> </div> ); }
```



#### `useEffect` thực hiện việc gì?

Với việc sử dụng `useEffect` hooks, chúng ta đã nói với React rằng hãy thực thi thêm công việc này sau khi component được render. React sẽ ghi nhớ function chúng ta đã truyền vào và gọi lại nó sau khi DOM được cập nhật.



#### Tại sao `useEffect` được gọi bên trong component?

Đặt `useEffect` vào bên trong component sẽ giúp effect truy cập được tới các giá trị state và props bên trong component.



#### `useEffect` sẽ chạy sau mỗi lần render?

Đúng. Mặc định, các `useEffect` sẽ luôn chạy sau lần render đầu tiên và sau mỗi lần giao diện được cập nhật. Chúng ta sẽ cùng học cách tùy chỉnh nó ở phần kế tiếp.

## 3. Side effect với “cleanup” function

Trong nhiều trường hợp, ví dụ như subscribe một dịch vụ, chúng ta sẽ cần phải thực hiện việc ngắt kết nối để tránh hiện tượng thất thoát bộ nhớ. `useEffect` có cơ chế để giải quyết vấn đề trên:

```
import {useEffect} from 'react' const App = () => { useEffect(() => { const
handleScroll = () => { const position = document.documentElement.scrollTop;
console.log("scrolling position: ", position); }
document.addEventListener("scroll", handleScroll) return () => {
document.removeEventListener("scroll", handleScroll) } }) return
<div>Hello</div> }
```

Nếu `useEffect` trả về kết quả là một function, function đó sẽ được gọi mỗi khi React nhận thấy đó là lúc để “dọn dẹp” effect đó.



#### Tại sao lại là trả về một function thay vì tạo ra một effect mới?

Đây là cơ chế để thực hiện việc cleanup cho effect. Việc này sẽ giúp cho logic của một effect được đứng cùng nhau trong source code, vì chúng là các thành phần của chung một effect.



### Khi nào React thực hiện việc cleanup effect?

React thực hiện việc cleanup khi component được gỡ khỏi màn hình. Tuy nhiên, các effect được chạy sau mỗi lần render. Vì vậy, **React cũng sẽ thực hiện việc cleanup với mỗi lần render**. Mục đích là để xoá effect từ lần render trước đó trước khi thực hiện effect cho lần tới. Việc này cũng sẽ tạo ra vấn đề với hiệu năng của ứng dụng.

## 4. Các tips với `useEffect`

### Sử dụng nhiều `useEffect` để chia nhỏ các vấn đề khác nhau

Trong một component, chúng ta có thể sử dụng nhiều effect cho các vấn đề khác nhau. Việc chia nhỏ `useEffect` này sẽ giúp việc tổ chức code bên trong một component **theo chức năng** của nó. Code sẽ dễ bảo trì hơn.

```
import {useState, useEffect} from 'react' const App = () => { const [count, setCount] = useState(0) // effect for update title useEffect(() => { document.title = `You clicked ${count} times`; }) // effect for scrolling useEffect(() => { document.addEventListener("scroll", () => { const position = document.documentElement.scrollTop; console.log("scrolling position: ", position); }) }) return ( <div> <p>You clicked {count} times</p> <button onClick={() => setCount(count + 1)}> Click me </button> </div> ); }
```

### Tối ưu hoá hiệu năng của component với việc bỏ qua các effect.

Việc cleanup effect và áp dụng effect sau mỗi lần render sẽ tạo các vấn đề liên quan tới hiệu năng của ứng dụng, khi việc thay đổi là quá thường xuyên. Để tránh việc làm không cần thiết này. Chúng ta có thể nói với React để nó bỏ qua các lần chạy effect nếu như một vài giá trị nhất định không thay đổi. Việc này được thực hiện bằng cách **truyền thêm cho `useEffect` một tham số thứ 2 có giá trị là một array**. Array này thường được gọi là “dependencies array” hoặc ngắn gọn là “deps array”.

```
import {useState, useEffect} from 'react' const App = () => { const [count, setCount] = useState(0) const [visible, setVisible] = useState(false) // effect for update title useEffect(() => { console.log("count changed") document.title = `You clicked ${count} times`; }, [count]) return ( <div> <p>You clicked {count} times</p> <button onClick={() => setCount(count + 1)}> Click me </button><div> {visible && <p>This is hidden content</p>} <button onClick={() => setVisible(!visible)}>Change visible</button> </div> </div> ); }
```


Với ví dụ trên, chúng ta đã truyền `[count]` là tham số thứ 2 của `useEffect`. Điều này có nghĩa là: khi giá trị của `count` = `0`, component được render với giá trị count tương ứng, React tiến hành so sánh giá trị cũ `[0]` và giá trị mới `[0]` của `count`. Vì tất cả các phần tử đều bằng nhau giữa 2 lần render ( `0 === 0` ), React sẽ bỏ qua effect của lần render đó.

Nếu giá trị `count` khác đi, do kết quả cũ và mới đã có sự khác nhau, React sẽ tiến hành thực thi effect đó như bình thường.

Khi sử dụng cách tối ưu hoá này cần lưu ý:

- Deps array cần bao gồm tất cả các giá trị nằm trong component có thể thay đổi theo các lần render: state, props và các giá trị tính toán khác. Nếu không, giá trị bên trong effect sẽ là giá trị của lần render cũ đã là lỗi thời.

- Deps array có thể nhận vào giá trị là một empty array. Với việc này, giá trị của deps array không bị thay đổi theo thời gian. Điều này nghĩa là effect đó sẽ chỉ được chạy một lần duy nhất và không bao giờ được chạy lại.

 Các function “setState” không cần thiết phải được liệt kê bên trong deps array của `useEffect` . Tuy nhiên các state hoặc props cần phải được liệt kê đầy đủ. Nếu muốn cập nhật giá trị state bên trong `useEffect` , chúng ta có thể sử dụng cú pháp `setState(prev => { return newState})` để tránh việc phải liệt kê các state bên trong deps array, có thể làm component bị rơi vào trường hợp lặp vô tận.

## 5. Đừng “nói dối” về `deps` array

Có một điều cần phải ghi nhớ khi sử dụng `useEffect` : tất cả các giá trị bên trong component được sử dụng bên trong effect cần phải được liệt kê bên trong deps array, bao gồm cả state, props, function, ...

Xét ví dụ sau:

```
import {useState, useEffect} from 'react' const Counter = () => { const [count, setCount] = useState(0) useEffect(() => { const timer = setInterval(() => { console.log("update count") setCount(count + 1) }, 1000) return () => { clearInterval(timer) } }, []) return <div>{count}</div> }
```

Tăng giá trị `count` lên một đơn vị sau mỗi giây. Tuy nhiên output lại không được như mong đợi

Với ứng dụng trên, ta muốn cập nhật giá trị của `count` lên một đơn vị sau mỗi giây. Tuy nhiên, giá trị `count` sẽ chỉ được cập nhật một lần duy nhất: từ 0 trở thành 1, nhưng ta vẫn có thể thấy dòng chữ `update count` được xuất hiện đều đặn sau mỗi giây trên màn hình console.

Vấn đề là gì? Tại sao giá trị `count` không được cập nhật như dự đoán?

```
// Lần render đầu tiên, giá trị count = 0 const Counter = () => { // ...
useEffect( // Effect của lần render đầu tiên () => { const id = setInterval(()
=> { setCount(0 + 1); // Luôn luôn là setCount(1) }, 1000); return () =>
clearInterval(id); }, []) // Không bao giờ chạy lại ); // ... } // Những lần
render tiếp theo, count = 1 const Counter = () => { // ...
useEffect( // Effect này không bao giờ được chạy // Vì chúng ta đã "nói dối" về deps () => {
const id = setInterval(() => { setCount(1 + 1); }, 1000); return () =>
clearInterval(id); }, []) ); // ... }
```

Giải thích 2 lần render của `Counter`

Việc cập nhật lại state khiến React thực hiện tính toán lại component. Việc tính toán đơn giản là “chạy lại” function component đó với những giá trị state được cập nhật. Nghĩa là lần render đầu tiên, component `Counter` được chạy với giá trị `count` là 0, lần render tiếp theo, `Counter` được chạy lại với giá trị `count` là 1.

Tuy nhiên, `useEffect` có dependencies là một mảng rỗng. Điều này khiến cho React không phát hiện được sự khác nhau của deps array giữa 2 lần render. Và nó đã **bỏ qua** việc cập nhật effect của các lần render tiếp theo. Function effect không được cập nhật với giá trị `count` mới, khiến cho luôn luôn lấy giá trị `count` của lần render trước đó. Nói cách khác, effect luôn luôn chạy lại `setCount(0 + 1)` sau mỗi giây.

Để giải quyết vấn đề này, ta luôn luôn cần “trung thực” về deps array của effect. Hãy luôn luôn liệt kê đầy đủ các giá trị bên trong component được sử dụng bên trong effect với deps, để React luôn luôn cập nhật lại effect của chúng ta mỗi khi component có sự thay đổi.

```
import {useState, useEffect} from 'react' const Counter = () => { const [count, setCount] = useState(0) useEffect(() => { const timer = setInterval(() => { console.log("update count") setCount(count + 1) }, 1000) return () => { clearInterval(timer) } }, [count]) return <div>{count}</div> }
```

Luôn trung thực với deps array

Với trường hợp đặc biệt của state, nếu như chúng ta cần cập nhật giá trị của state bên trong các effect, ta có thể sử dụng cú pháp cập nhật state dựa vào giá trị cũ của chúng. Điều này có thể giúp chúng ta tránh việc phải liệt kê các giá trị state bên trong deps array.

```
import {useState, useEffect} from 'react' const Counter = () => { const [count, setCount] = useState(0) useEffect(() => { const timer = setInterval(() => { console.log("update count") setCount((prev) => { return prev + 1} ) }, 1000) return () => { clearInterval(timer) } }, []) return <div>{count}</div> }
```

Sử dụng cú pháp khác để cập nhật state. Tránh sử dụng state bên trong effect.



Ngoài ra, trong thực tế, với các state, người ta cũng có thể sử dụng một hooks đặc biệt là `useReducer` để cập nhật các giá trị state thông qua các "action". Tìm hiểu thêm về hook này ở đây: <https://reactjs.org/docs/hooks-reference.html>

## 6. (Nâng cao) “Đồng bộ hoá”, không phải “vòng đời”



Phần này sử dụng để giúp người đọc đã biết về khái niệm lifecycle trong React đang cố gắng nghĩ `useEffect` tương đương với lifecycle methods nào

“*Unlearn what you have learned.*” — Yoda

Có rất nhiều các bài báo, blog trên internet hướng dẫn chúng ta sử dụng `useEffect` thay thế cho lifecycle methods. Điều này không thực sự tương đương. Tiếp tục tìm hiểu theo hướng đó có thể dẫn tới những cách sử dụng sai lầm với effect. Chúng ta cần tư duy “think in effects”, mô hình của nó hướng tới sự “đồng bộ hoá”, thay vì phản hồi lại với các sự kiện trong lifecycle.

ReactJS là một thư viện UI tuyệt vời, giúp bạn “đồng bộ hoá” source code của bạn với DOM tree, là những gì chúng ta nhìn thấy trên màn hình. Code của bạn thay đổi, giao diện của bạn cập nhật theo. Hoàn toàn không có sự phân biệt “mounting” và “updating”

Bạn cũng nên có tư tưởng như vậy khi nghĩ về `useEffect`, nhưng ở level component.

`useEffect` cho phép chúng ta đồng bộ những thứ nằm bên ngoài cây component React, dựa vào state và props. State, props của bạn thay đổi, effect được thực thi theo.

Điều này khác so với mô hình mount/update/unmount đã quen thuộc với các lifecycle. Nếu như chúng ta viết các effects có cách xử lý khác nhau dựa vào đó có phải là lần render đầu tiên của nó hay không, chúng ta đang bơi ngược dòng nước.


`useEffect` cho chúng ta một cơ hội để có thể nhóm các đoạn code có cùng một nghiệp vụ lại với nhau. Đối với lifecycle methods, những logic đó sẽ nằm rải rác khắp các lifecycle methods.

```
import { useState, useEffect, useRef } from "react"; function Counter() { const [count, setCount] = useState(0); useEffect(() => { const timer = setInterval(() => { setCount(count + 1); }, 1000); return () => { clearInterval(timer) } }; }, []); return <h1>{count}</h1>; }
```

Các logic liên quan nằm trong một effect duy nhất

```
import {Component} from 'react' class Counter extends Component { state =
{count: 1} timer = 0 componentDidMount() { this.timer = setInterval(() => {
this.setState({ count: this.state.count + 1 }) }, 1000) }
componentWillUnmount() { clearInterval(this.timer) } render() { return ( <h1>
{this.state.count}</h1> ) } }
```

Logic phải nằm bên trong 2 life cycle methods là `componentDidMount` và `componentWillUnmount` để có thể đạt được hiệu ứng tương tự như với `useEffect`

 Đọc thêm về `useEffect` ở đây: <https://overreacted.io/a-complete-guide-to-useeffect/>