React Tutorial

Save **UseEffect** For **Rainy** Days!

Otherwise, beware...

How many renders? #1

You've likely run into this situation before, many of us do. Here's what happens: when you click the button, it runs a function that updates the state. Then, you've used **useEffect** to watch for changes and trigger onChange. If you guessed it renders **twice**, you're right! This is a common mistake with useEffect.

```jsx
const { useState, useEffect } = require("react");

const ChildComponent = ({ onClicked }) => {
  const [isActive, setIsActive] = useState(false);

  useEffect(() => {
    onClicked(isActive);
  }, [isActive]);

  const handleClick = () => {
    setIsActive(!isActive);
  };

  return <button onClick={handleClick}>Click Me!</button>;
};

const ParentComponent = () => {
  const [status, setStatus] = useState(false);
  return <ChildComponent onClicked={(status) => setStatus(status)} />;
};
```

Fix is here >

What you can do is wrap everything and handle all updates directly inside the event when the action happens. This way, you can manage the situation efficiently without needing to use useEffect, making the code simpler and preventing extra renders.

```jsx
const ChildComponent = ({ onClicked }) => {
  const [isActive, setIsActive] = useState(false);

  const handleClick = () => {
    const status_ = !isActive;
    setIsActive(status_);
    onClicked(status_);
  };

  return <button onClick={handleClick}>Click Me!</button>;
};

const ParentComponent = () => {
  const [status, setStatus] = useState(false);
  return <ChildComponent onClicked={(status) => setStatus(status)} />;
};
```

Does Your Code Run Once in React? #2

In React, useEffect may run multiple times in development mode or with future **Concurrent Mode, even with an empty dependency array**. This can cause issues for one-time logic, like initializing an authentication provider. To make sure the logic runs only once, additional safeguards are needed.

```
1  import React, { useEffect } from "react";
2
3  const ChildComponent = () => {
4    useEffect(() => {
5      // logic goes here that need to run only once
6    }, []);
7
8    return <div>ChildComponent</div>;
9  };
10
11 export default ChildComponent;
```

Fix is here >

Approach 1

This top-level flag approach introduces a top-level variable **didInit** outside the component. It make sure, that the logic runs only once per app load, regardless of whether the component is re-mounted or strict mode is enabled.

```jsx
import React, { useEffect } from "react";

// top level state management...
let didInit = false;

const ChildComponent = () => {
  useEffect(() => {
    if (!didInit) {
      // only run when app loads
      didInit = true;
    }
  }, []);

  return <div>ChildComponent</div>;
};

export default ChildComponent;
```

Fix is here >

Approach 2

Here, the logic runs outside of the component altogether, ensuring it runs only once before the app renders. This is a more straightforward approach, which make sure you avoid the complications of React's re-mount behavior entirely.

```
const ChildComponent = () => {
  useEffect(() => {
    if(typeof window !== 'undefined'){
        // only run when app loads
    }
  }, []);

  return <div>ChildComponent</div>;
};

export default ChildComponent;
```

Should data flow in the correct way? #3

Data should flow from parent to child components for easier maintenance and debugging. Reversing this flow makes the app harder to manage and leads to unpredictable behavior as it grows. Here's an example of bad code where parents rely on children.

```
const { useState, useEffect } = require("react");

const ChildComponent = ({ onData }) => {
  const data = useGetData();

  useEffect(() => {
    onData(data);
  }, [data]);

  return <p>{data}</p>;
};

const ParentComponent = () => {
  const [fetchedData, setFetchedData] = useState("");
  return <ChildComponent onData={(data) => setFetchedData(data)}
};
```

Fix is here >

You can fetch data in the parent component and pass it down to child components, following proper data flow practices. This avoids using unnecessary **useEffect** and extra **useState** in your code. As a result, it's now easier to maintain and track changes.

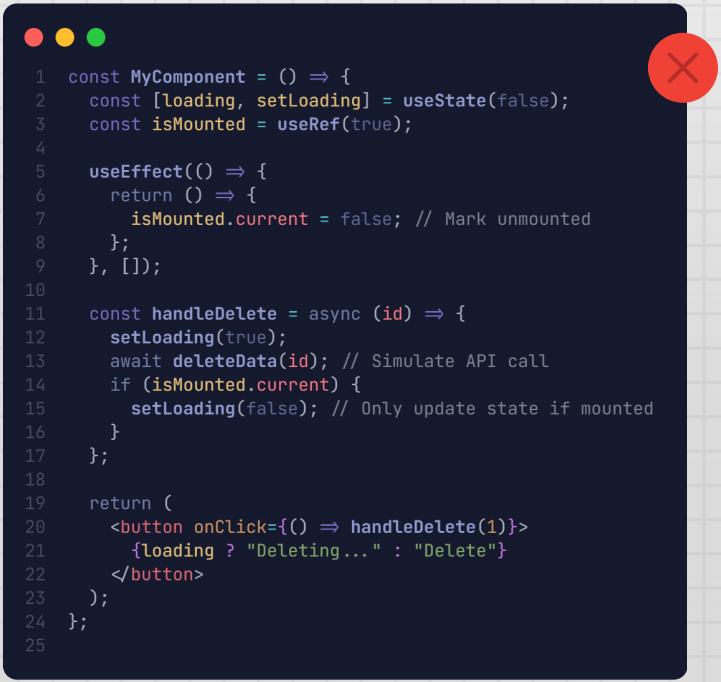
```
const ChildComponent = ({ data }) => {
  return <p>{data}</p>;
};

const ParentComponent = () => {
  const data = useGetData();
  return <ChildComponent data={data} />;
};
```

Is It Safe to Update State in Unmounted React Components?

#4

Yes. In this bad example, useEffect and useRef are used to track if the component is mounted, adding unnecessary complexity. This makes the code harder to understand. It's not needed since React can handle state updates safely.



```jsx
const MyComponent = () => {
  const [loading, setLoading] = useState(false);
  const isMounted = useRef(true);

  useEffect(() => {
    return () => {
      isMounted.current = false; // Mark unmounted
    };
  }, []);

  const handleDelete = async (id) => {
    setLoading(true);
    await deleteData(id); // Simulate API call
    if (isMounted.current) {
      setLoading(false); // Only update state if mounted
    }
  };

  return (
    <button onClick={() => handleDelete(1)}>
      {loading ? "Deleting..." : "Delete"}
    </button>
  );
};
```

Fix is here >

In the better approach, we rely on React to handle state updates, even if the component unmounts, without extra checks. The bad code complicates things by tracking the mount status, while the good code keeps it simple and clean. Trusting React makes the code easier to read.

```
const MyComponent = () => {
  const [loading, setLoading] = useState(false);

  const handleDelete = async (id) => {
    setLoading(true);
    await deleteData(id); // Simulate API call
    setLoading(false); // Safe to update, React manages it
  };

  return (
    <button onClick={() => handleDelete(1)}>
      {loading ? "Deleting..." : "Delete"}
    </button>
  );
};
```

Conclusion

- Lift state up to the parent instead of using callbacks for passing data up, keeping the data flow clean.
- Use useEffect for fetching or calculating data, but always handle cleanup to avoid issues like race conditions.
- Set local state in useEffect only when necessary to avoid extra renders.
- Avoid overusing useEffect; if you can calculate during render, do it.
- Don't fear state updates in unmounted components; working around it can make your code unnecessarily complex.

When do you really need useEffect?

- Use useEffect to run code after the component renders.
- Use it to fetch data, update the page, or set up listeners.
- Run code only when certain values change by adding them to the effect's list (dependencies).
- Clean up things like timers or listeners when the component is removed.

Example scenarios

- Run one-time logic during the component's lifetime, not the app's lifetime.
- Optimize performance when other solutions (like pagination or virtualization) aren't possible.
- Handle open connections or subscriptions that need cleanup when the component unmounts.
- When use WebSocket, analytics, or third-party APIs.

Follow For More !

Learn Together, Grow Together

Dinuka Nilupul

@dinukanilupul