4 common React error messages and how to address them

This article will help you understand these details by going over some of the most common React error messages and explaining what they mean, what their consequences are, and how to fix them.

We'll be covering the following error messages:

- Can't perform a React state update on an unmounted component
- Too many re-renders. React limits the number of renders to prevent an infinite loop
- Objects are not valid as a React child / Functions are not valid as a React child
- Adjacent JSX elements must be wrapped in an enclosing tag

- **Can’t perform a React state update on an unmounted component:**

  When dealing with async data or logic flows in your components, you may encounter a runtime error in your browser’s console telling you that you can’t perform a state update on a component that is already unmounted. The issue is that somewhere in your components tree, a state update is triggered onto a component that is already unmounted.

```
const Component = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetchAsyncData().then((data) => setData(data));
  }, []);

  // ...
};
```

This is caused by a state update that is dependent on an async request. The async request starts somewhere in the lifecycle of a component (such as inside a `useEffect` Hook) but takes a while to complete.

Meanwhile, the component has already been unmounted (due to e.g. user interactions), but the original async request still finishes — because it’s not connected to the React lifecycle — and triggers a state update to the component. The error is triggered here because the component doesn’t exist anymore.

How to address this

There are several ways to address this, all of which boil down to two different concepts. First, it’s possible to keep track of whether the component is mounted, and we can perform actions based on that.

While this works, it’s not recommended. The problem with this method is that it unnecessarily keeps a reference of unmounted components around, which causes memory leaks and performance issues.

```
const Component = () => {
  const [data, setData] = useState(null);
  const isMounted = useRef(true);

  useEffect(() => {
    fetchAsyncData().then(data => {
      if(isMounted.current) {
        setData(data);
      }
    });

    return () => {
      isMounted.current = false;
    };
```

The second — and preferred — way is to cancel the async request when the component unmounts. Some async request libraries will already have a mechanism in place to cancel such a request. If so, it’s as straightforward as cancelling the requst during the cleanup callback of the useEffect Hook.

If you're not using such a library, you could achieve the same using AbortController. The only downsides to these cancel methods are that they are fully reliant on a library's implementation or browser support.

```
const Component = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    const controller = new AbortController();
    fetch(url, { signal: controller.signal }).then((data) => setData(data));
    return () => {
      controller.abort();
    }
  }, []);

  // ...
};
```

- **Too many re-renders. React limits the number of renders to prevent an infinite loop:**

  Infinite loops are the bane of every developer's existence and React developers are not an exception to this rule. Luckily, React does a very nice job of detecting them and warning you about it before your entire device becomes unresponsive.

  ### How to address this

  As the warning suggests, the problem is that your component is triggering too many re-renders. This happens when your component queues too many state updates in a very short amount of time. The most common culprits for causing infinite loops are:

  - Performing state updates directly in the render
  - Not providing a proper callback to an event handler

If you're running into this particular warning, make sure to check those two aspects of your component.

```
const Component = () => {
  const [count, setCount] = useState(0);

  setCount(count + 1); // State update in the render

  return (
    <div className="App">
      {/* onClick doesn't receive a proper callback */}
      <button onClick={setCount((prevCount) => prevCount + 1)}>
        Increment that counter
      </button>
    </div>
  );
}
```

- **Objects are not valid as a React child / Functions are not valid as a React child:**

In React, there are a lot of things that we can render to the DOM in our components. The choices are almost endless: all the HTML tags, any JSX element, any primitive JavaScript value, an array of the previous values, and even JavaScript expressions, as long as they evaluate to any of the previous values.

Despite that, unfortunately, React still doesn't accept everything that possibly exists as a React child. To be more specific, you can't render objects and functions to the DOM because these two data values will not evaluate to anything meaningful that React can render into the DOM. Therefore, any attempts to do so will result in React complaining about it in the form of the mentioned errors.

How to address this

If you're facing either of these errors, it's recommended to verify that the variables that you're rendering are the expected type. Most often, this issue is caused by rendering a child or variable in JSX, assuming it's a primitive value — but, in reality, it turns out to be an object or a function. As a prevention method, having a type system in place can significantly help.

```
const Component = ({ body }) => (
  <div>
    <h1>{/* */}</h1>
    {/* Have to be sure the `body` prop is a valid React child */}
    <div className="body">{body}</div>
  </div>
);
```

- **Adjacent JSX elements must be wrapped in an enclosing tag:**

One of React's biggest benefits is being able to construct an entire application by combining a lot of smaller components. Every component can define its piece of UI in the form of JSX that it should render, which ultimately contributes to the application's entire DOM structure.

```
const Component = () => (
  <div><NiceComponent /></div>
  <div><GoodComponent /></div>
);
```

Due to React's compounding nature, a common thing to try is returning two JSX elements in the root of a component that is only used inside another component. However, doing so will surprisingly present React developers with a warning telling them they have to wrap adjacent JSX elements in enclosing tags.

From the perspective of the average React developer, this component will only be used inside of another component. So, in their mental model, it makes perfect sense to return two elements from a component because the resulting DOM structure would be the same, no matter whether an outer element is defined in this component or the parent component.

However, React isn’t able to make this assumption. Potentially, this component could be used in the root and break the application, as it will result in an invalid DOM structure.

How to address this

React developers should always wrap multiple JSX elements returned from a component in an enclosing tag. This can be an element, a component, or React’s Fragment, if you’re sure that the component doesn’t require an outer element.