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:

- Warning: Each child in a list should have a unique key prop
- Prevent usage of Array index in keys
- React Hook "useXXX" is called conditionally. React Hooks must be called in the exact same order in every component render
- React Hook has a missing dependency: 'XXX'. Either include it or remove the dependency array

➢ Warning: Each child in a list should have a unique Key prop

```
import { Card } from "./Card";

const data = [
  { id: 1, text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit." },
  { id: 2, text: "Phasellus semper scelerisque leo at tempus." },
  { id: 3, text: "Duis aliquet sollicitudin neque," }
];

export default function App() {
  return (
    <div className="container">
      {data.map((content) => (
        <div className="card">
          <Card text={content.text} />
        </div>
      ))}
    </div>
```

One of the most common things in React development is taking the items of an array and using a component to render them based on the content of the item. Thanks to JSX, we can easily embed that logic into our component using an Array.map function and return the desired components from the callback.

However, it's also common to receive a React warning in your browser's console saying that every child in a list should have a unique key prop. You'll likely run into this warning several times before making it a habit to give each child a unique key prop, especially if you're less experienced with React. But how do you fix it before you've formed the habit?

How to address this:

As the warning indicates, you'll have to add a key prop to the most outer element of the JSX that you're returning from the map callback. However, there are several requirements for the key that you're gonna use. **The key should be:**

1. Either a string or a number
2. Unique to that particular item in the list
3. Representative of that item in the list across renders

```
export default function App() {
  return (
    <div className="container">
      {data.map((content) => (
        <div key={content.id} className="card">
          <Card text={content.text} />
        </div>
      ))}
    </div>
  );
}
```

Although your app won't crash if you don't adhere to these requirements, it can lead to some unexpected and often unwanted behavior. React uses these keys to determine which children in a list have changed, and use this information to determine which parts of the previous DOM can be reused and which it should recomputed when components are re-rendered. Therefore, it's always advisable to add these keys.

➢ Prevent usage of Array index in keys:

Building upon the previous warning, we're diving into the equally common ESLint warning regarding the same topic. This warning will often present after you've made a habit of including a key prop with the resulting JSX from a list.

```
import { Card } from "./Card";

// Notice that we don't include pre-generated identifiers anymore.
const data = [
  { text: "Lorem ipsum dolor sit amet, consectetur adipiscing elit." },
  { text: "Phasellus semper scelerisque leo at tempus." },
  { text: "Duis aliquet sollicitudin neque," }
];
```

```
export default function App() {
  return (
    <div className="container">
      {data.map((content, index) => (
        <div key={index} className="card">
          <Card text={content.text} />
        </div>
      ))}
    </div>
  );
}
```

Sometimes, you won't have a unique identifier attached to your data. An easy fix is to use the index of the current item in the list. However, the problem with using the item's index in the array as its key is that it's not representative of that particular item across renders.

Let’s say that we have a list with several items, and that the user interacts with them by removing the second item. For the first item, nothing has changed to its underlying DOM structure; this is reflected in its key, which stays the same, 0.
For the third item and beyond, their content hasn’t changed, so their underlying structure also shouldn’t change. However, the key prop from all the other items will change because the keys are based on the array index. React will assume that they’ve changed and recomputed their structure — unnecessarily. This negatively affects performance and can also lead to inconsistent and incorrect states.

How to address this

To solve this, it’s important to remember that keys don’t necessarily have to be identifiers. As long as they’re unique and representative of the resulting DOM structure, whatever key you want to use will work.

```
export default function App() {
  return (
    <div className="container">
      {data.map((content) => (
        <div key={content.text} className="card">{/* This is the best we can do,
          <Card text={content.text} />
        </div>
      ))}
    </div>
  );
```

- **React Hook "useXXX" is called conditionally. React Hooks must be called in the exact same order in every component render:**

We can optimize our code in different ways during development. One such thing you can do is make sure that certain code is only executed in the code branches where the code is necessary. Especially when dealing with code that is time or resource-heavy, this can make a world of difference in terms of performance.

```
const Toggle = () => {
  const [isOpen, setIsOpen] = useState(false);

  if (isOpen) {
    return <div>{/* ... */}</div>;
  }
  const openToggle = useCallback(() => setIsOpen(true), []);
  return <button onClick={openToggle}>{/* ... */}</button>;
};
```

This is necessary because, internally, React uses the order in which Hooks are called to keep track of their underlying states and preserve them between renders. If you mess with that order, React will, internally, not know which state matches with the Hook anymore. This causes major issues for React and can even result in bugs.

How to address this

React Hooks must be always called at the top level of components — and unconditionally. In practice, this often boils down to reserving the first section of a component for React Hook initializations.

```
const Toggle = () => {
  const [isOpen, setIsOpen] = useState(false);
  const openToggle = useCallback(() => setIsOpen(true), []);

  if (isOpen) {
    return <div>{/* ... */}</div>;
  }
  return <button onClick={openToggle}>{/* ... */}</button>;
};
```

- **React Hook has a missing dependency: 'XXX'. Either include it or remove the dependency array**

  An interesting aspect of React Hooks is the dependencies array. Almost every React Hook accepts a second argument in the form of an array, inside of which you're able to define the dependencies for the Hook. When any of the dependencies change, React will detect it and re-trigger the Hook.

  In their documentation, React recommends developers to always include all variables in the dependencies array if they're used in the Hook and affect the component's rendering when changed.

  ### How to address this

  To help with this, it's recommended to make use of the exhaustive-deps rule inside the react-hooks ESLint plugin. Activating it will warn you when any React Hook doesn't have all dependencies defined.

```
const Component = ({ value, onChange }) => {
  useEffect(() => {
    if (value) {
      onChange(value);
    }
  }, [value]); // `onChange` isn't included as a dependency here.

  // ...
}
```

  The reason you should be exhaustive with the dependencies array matters is related to the concept of closures and scopes in JavaScript. If the main callback of the React Hook uses variables outside its own scope, then it can only remember the version of those variables when it was executed.

But when those variables change, the closure of the callback can’t automatically pick up on those changed versions. This can lead to executing your React Hook code with outdated references of its dependencies, and result in different behavior than expected.

For this reason, it’s always recommended to be exhaustive with the dependencies array. Doing so addresses all possible issues with calling React Hooks this way, as it points React towards the variables to keep track of. When React detects changes in any of the variables, it will rerun the callback, allowing it to pick up on the changed versions of the dependencies and run as expected.