React mistakes to avoid for successful app development

React is a popular framework that is easy to learn but also easy to make mistakes when programming if you are not careful. Here are common React mistakes that you might make during app development.

React is a popular framework that is easy to learn but also easy to make mistakes when programming if you are not careful. Here are **common React mistakes** that you might make during app development.

Using the wrong type of callback function

Event handling is a common task in React, through JavaScript's powerful event listening feature. Maybe you want to change the color of the button click on mouseover or send form data to the server on submission. In both cases, you need to pass a callback function to the event to perform the desired response. This is where some React programmers often make mistakes.

For example:

```
export default function App() { ??function handleSubmit(e) { ????
e.preventDefault() ????console.log("Form submitted!") ??} ??
function print(number) { ????console.log("Print", number) ??} ??
function doubler(number) { ????return () => { ??????
console.log("Double", number * 2) ????} ??} ??return ( ????> ??????
{/* Code will go here */} ???? ??) }
```

Here you have 3 separate functions. The first two functions return no results, the third returns another function. You must remember that it will be the key to understanding what you will learn next.

Now, moving to JSX, starting with the first function, and the most common way is to pass a function as an event handler:

```
?? ??Submit
```

The example passes the name of the function to this event via the onSubmit attribute, so that React calls handleSubmit when you submit the form. Inside **handleSubmit** , you can access the event object, granting permissions to properties like **event.target.value** and methods like **event.preventDefault()** .

Another way to pass an event handler is to call it inside the callback function. You're essentially passing onClick a function that calls print() for you:

```
{[1, 5, 7].map((number) => { ??return ( ????
print(number)}> ??????Print {number} ????
??) })}
```

The above method is useful in cases where you want to pass local data to this function. Example passing each number to the print() function. If you used the first method, you cannot pass arguments to this function.

The third method is where a lot of programmers make mistakes. Remember that the double function returns another function:

```
function doubler(number) { ??return () => { ????
console.log("Double", number * 2) ??} }
```

Now if we used it in JSX like this:

```
{[1, 5, 7].map((number) => { ??return ( ????
doubler(number)}> ??????Double {number} ????
??) })}
```

Here, the function that you return from **double()** will be attached to **onClick** . It's basically the same as copying the returned function and pasting it inside **onClick** . In general, you should add the function as a variable (same way 1) and call the function inside **the callback** (method 2).

All 3 ways are valid because you are passing a function to this event. In React, you need to make sure you pass a function to the event. It can be a variable, a hard-coded (inline) function, or an object/function that returns another function.

Returns 0 during error checking

When exporting a conditional element in React, you can use if…else statements or short-circuiting techniques. Short-circuiting involves the use of the ampersand (&&) notation. If the condition before the ampersand evaluates to true, the browser runs the code following the &. Otherwise, the browser does not run the code.

Short-circuiting is a better technique thanks to the correct syntax, but it comes with a side effect that many programmers don't realize. This error is caused by not understanding exactly how JSX works with dummy

values.

Consider the following example:

```
export default function App() { ??const array = [1, 2, 3, 4] ??return (
???? ??????{array.length && ( ???????? ??????????
Array items: {array.join(", ")} ???????? ??????)} ???? ??) }
```

As long as the array is inside it, React will print each item on the page. **This is because the array.length** check is returning a true value. But, what happens when the array is empty? First, the following elements will appear on the page as you expect. However, you will see a weird zero appear on the screen.

Cause **array.length** returns 0. The value 0 is fake in JavaScript. And the problem here is that JSX outputs a zero on the screen. Other dummy values ??like null, false and undefined are not output. Sometimes this can lead to a bad user experience because zero is always present on the page. Sometimes 0 can be so small that you don't notice it.

The solution here is to make sure to only return null, undefined or false. You do this by explicitly checking for zero in the condition instead of relying on a dummy value:

```
export default function App() { ??const array = [1, 2, 3, 4] ??return (
???? ??????{array.length !== 0 && ( ???????? ??????????
Array items: {array.join(", ")} ???????? ??????)} ???? ??) }
```

Now the value 0 will not appear on the screen, even if the array is empty.

Status update error

When updating state in a React component, you need to do it properly to avoid unwanted effects. The worst error is that there is no error for you, making it difficult to debug and determine what the error is. Failure to update status also causes this effect.

This error stems from not understanding how to update the state while using the current state. Let's look at the sample code. For example:

```
export default function App() { ??
const [array, setArray] = useState([1, 2, 3]) ??
function addNumberToStart() { ????array.unshift(number) ????
setArray(array) ??} ??function addNumberToEnd() { ????
array.unshift(number) ????setArray(array) ??} ??return ( ????> ??????
{array.join(", ")} ??????
 ??????
 { ??????????addNumberToStart(0) ??????????addNumberToEnd(0) ??????????console.log(array) ????
 ???? ??) }
```

If you're running the above code, you'll see that both functions add 0 at the start and end of the array. However, it does not add 0 to the array printed on the page. You can repeatedly click this button, but the UI remains the same.

Cause in both functions you are changing state with **array.push()** . React explicitly warns that state must be immutable in React, meaning you cannot change it. React uses reference values ??with this state.

The solution here is to access the current state (currentArray), make a copy of the state and an update for that copy:

```
function addNumberToStart(number) { ??setArray((currentArray) => { ????
return [number, .currentArray] ??}) } function addNumberToStart(number) {
??setArray((currentArray) => { ????return [.currentArray, number] ??}) }
```

This is the correct method to update state in React. Now when you click the button, it adds zeros to the beginning and end of the array. But the most important here is that the updates will reflect immediately on the page.

Above are **the React programming mistakes to avoid** . Hope the article is useful to you.