

Handling multiple – states using “useState” hook

To handle multiple states using the `useState` hook in React, you can call the `useState` function multiple times, once for each state you want to manage. Here's an example:

```
import { useState } from 'react';

function MyComponent(props) {

  const [count, setCount] = useState(0);

  const [text, setText] = useState('');

  function handleIncrement() {

    setCount(count + 1);

  }

  function handleTextChange(event) {

    setText(event.target.value);

  }

  return (

    <div>

      <p>Count: {count}</p>

      <button onClick={handleIncrement}>Increment</button>

      <br />

      <input type="text" value={text} onChange={handleTextChange} />

      <p>Text: {text}</p>

    </div>

  );

}
```

In this example, we're managing two states using `useState`: `count` and `text`. We're also defining two functions: `handleIncrement`, which updates the `count` state when a button is clicked, and `handleTextChange`, which updates the `text` state when the text input changes.

By calling `useState` twice, we're creating two independent pieces of state that can be managed separately. We're also using destructuring to assign the current value of each state and its corresponding setter function to separate variables (`count` and `setCount`, and `text` and `setText`).

Overall, using multiple `useState` hooks can help you manage multiple pieces of state in a clean and organized way.

Other Alternative:

We can also manage multiple states using a single `useState` hook by passing an object as the initial state and using destructuring to access individual state variables and their corresponding update functions. Here's an example:

```
import { useState } from 'react';

function MyComponent(props) {

  const [state, setState] = useState(
    { count: 0, text: '' }
  );

  function handleIncrement() {

    setState(
      prevState => (
        { ...prevState, count: prevState.count + 1 }
      )
    );
  }

  function handleTextChange(event) {

    setState(
      prevState => (
        { ...prevState, text: event.target.value }
      )
    );
  }

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={handleIncrement}>Increment</button>
      <br />
    </div>
  );
}
```

```
<input type="text" value={state.text} onChange={handleTextChange} />

<p>Text: {state.text}</p>

</div>

);

}
```

In this example, we're still managing two pieces of state (`count` and `text`), but we're using a single `useState` hook to initialize both states as properties of an object (`state`). We're also using destructuring to access individual state variables (`count` and `text`) and their corresponding update functions (`setCount` and `setText`).

To update a piece of state, we're using the functional update form of `setCount`, which takes a callback function that receives the previous state as an argument and returns the new state. We're spreading the previous state using the spread operator (`...prevState`) to create a new object with all the previous state properties, and then updating the property we want to change (`count` or `text`) using object property shorthand.

Using a single `useState` hook to manage multiple pieces of state can be a convenient way to keep related state together and reduce boilerplate code. However, it can also make the code more complex and harder to read, especially if you have many pieces of state or complex state updates. So it's up to you to decide which approach works best for your specific use case.

Revision – Map Function in JS

The `map` function is a built-in method in JavaScript that allows you to apply a function to every element of an array and returns a new array with the results. The `map` function is one of the most useful and commonly used functions in JavaScript because it provides an easy way to transform and manipulate data in an array.

The syntax of the `map` function is as follows:

```
array.map(callback(element, index, array), thisArg)
```

Here, `array` is the array that you want to map, `callback` is the function that will be called on each element of the array, `element` is the current element being processed, `index` is the index of the current element, and `array` is the original array that is being mapped. `thisArg` is an optional parameter that refers to the `this` value that will be used when executing the callback function.

The `callback` function takes three arguments: `element`, `index`, and `array`. The `element` argument is the current element being processed, the `index` argument is the index of the current element, and the `array` argument is the original array that is being mapped. The `callback` function returns a new value that will be added to the new array that is being created.

Here is an example of using the `map` function:

```
const numbers = [1, 2, 3, 4, 5];

const doubledNumbers = numbers.map(number => number * 2);

console.log(doubledNumbers); // [2, 4, 6, 8, 10]
```

In this example, we have an array of numbers `[1, 2, 3, 4, 5]`. We use the `map` function to apply the `number * 2` function to every element of the array. The `map` function creates a new array called `doubledNumbers` with the results of the `number * 2` function applied to every element of the original array.

The `map` function is often used in combination with other JavaScript methods, such as `filter`, `reduce`, and `forEach`, to manipulate and transform data in an array. Here is an example of using the `map` function in combination with the `filter` function:

```
const people = [  
  { name: 'Alice', age: 20 },  
  { name: 'Bob', age: 30 },  
  { name: 'Charlie', age: 40 }  
];  
  
const names = people.filter(person => person.age >= 30).map(person => person.name);  
  
console.log(names); // ['Bob', 'Charlie']
```

In this example, we have an array of people with `name` and `age` properties. We use the `filter` function to create a new array of people who are over the age of 30. We then use the `map` function to create a new array of just the names of those people.

In conclusion, the `map` function is a powerful tool for transforming and manipulating data in JavaScript arrays. It allows you to apply a function to every element of an array and create a new array with the results. The `map` function is often used in combination with other JavaScript methods to create complex data transformations.

Important Difference

The main difference between `onClick={() => setCategory(data.title)}` and `onClick={setCategory(data.title)}` is when the `setCategory` function is called.

In the first case, `onClick={() => setCategory(data.title)}`, an arrow function is used as a callback function for the `onClick` event handler. This means that when the element is clicked, the arrow function will be executed and then call the `setCategory` function with the `data.title` argument. This is useful when you need to perform some additional logic or calculations before calling the `setCategory` function.

In the second case, `onClick={setCategory(data.title)}`, the `setCategory` function is called immediately when the component is rendered, and the return value of the `setCategory` function is assigned to the `onClick` event handler. This is not desirable because it will call the `setCategory` function on every render, which can lead to unnecessary re-renders and performance issues.

Therefore, the correct way to pass a function with arguments to an `onClick` event handler in React is to use the first approach with an arrow function as follows: `onClick={() => setCategory(data.title)}`. This way, the `setCategory` function will only be called when the element is clicked, and not on every render.

